Decoupled Vector-Fetch Architecture with a Scalarizing Compiler

by

Yunsup Lee

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Krste Asanović, Chair
Professor David A. Patterson
Professor Borivoje Nikolić
Professor Paul K. Wright

Spring 2016

Decoupled Vector-Fetch Architecture with a Scalarizing Compiler

Abstract

Decoupled Vector-Fetch Architecture with a Scalarizing Compiler

by

Yunsup Lee

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Krste Asanović, Chair

As we approach the end of conventional technology scaling, computer architects are forced to incorporate specialized and heterogeneous accelerators into general-purpose processors for greater energy efficiency. Among the prominent accelerators that have recently become more popular are data-parallel processing units, such as classic vector units, SIMD units, and graphics processing units (GPUs). Surveying a wide range of data-parallel architectures and their parallel programming models and compilers reveals an opportunity to construct a new data-parallel machine that is highly performant and efficient, yet a favorable compiler target that maintains the same level of programmability as the others.

In this thesis, I present the Hwacha decoupled vector-fetch architecture as the basis of a new data-parallel machine. I reason through the design decisions while describing its programming model, microarchitecture, and LLVM-based scalarizing compiler that efficiently maps OpenCL kernels to the architecture. The Hwacha vector unit is implemented in Chisel as an accelerator attached to a RISC-V Rocket control processor within the open-source Rocket Chip SoC generator. Using complete VLSI implementations of Hwacha, including a cache-coherent memory hierarchy in a commercial 28 nm process and simulated LPDDR3 DRAM modules, I quantify the area, performance, and energy consumption of the Hwacha accelerator. These numbers are then validated against an ARM Mali-T628 MP6 GPU, also built in a 28 nm process, using a set of OpenCL microbenchmarks compiled from the same source code with our custom compiler and ARM's stock OpenCL compiler.

To my loving wife Soyoung and my family.

Contents

List of Figures

List of Tables

Acknowledgments

Looking back at all the years I have been at U.C. Berkeley, I am grateful to have had a chance to work with so many talented colleagues and friends.

First and foremost, I would like to thank my advisor Krste Asanović who has been a true mentor, a role model, and a passionate teacher. Thanks for supporting me throughout various projects and shaping up my knowledge and understanding of computer architecture and research. Thanks for your advice, encouragement, and exceptional patience throughout the long journey of figuring out how to build computer systems right. I would also like to thank the rest of my thesis committee, Dave Patterson, Bora Nikolić, and Paul Wright for their valuable feedback. Sadly, committee member David Wessel passed away, the loss of a mentor and a friend.

Special thanks to Andrew Waterman, who co-designed the RISC-V ISA among many other things. I have truly enjoyed our late-night conversations, debates, and arguments—I believe they have improved the work we have done together. Thanks to other RISC-V instigators, Krste and Dave—I look forward to seeing RISC-V evolve into something even bigger.

Thanks to the members of the Hwacha team at U.C. Berkeley for helping to create a new data-parallel machine and a compiler that goes with it. Particular thanks to Albert Ou, Colin Schmidt, and Sagar Karandikar for making it happen. Without your help, the Hwacha vector-fetch architecture and this thesis would not have existed in their current form. Thanks to the rest of the Hwacha team including Henry Cook, Andrew Waterman, Palmer Dabbelt, Howard Mao, John Hauser, Huy Vo, Stephen Twigg, and Quan Nguyen. Section 1.6 discusses in more detail how the members of the Hwacha project contributed to this thesis.

Thanks to the members of the EOS team at U.C. Berkeley and MIT for helping to design and fabricate working 45 nm RISC-V prototypes. Particular thanks to Chen Sun and Rimas Avižienis, who taught me how to whisper into the ear of ECAD tools. Chen, I will never forget the night at BWRC when we got the EOS22 chip to execute the *Hello World!* program using our silicon photonics links. Thanks to the rest of the EOS team including Michael Georgas, Yu-Hsin Chen, Rajeev Ram, and Vladimir Stojanović.

Thanks to the members of the Raven team at U.C. Berkeley for helping to design and fabricate working 28 nm RISC-V prototypes. Particular thanks to Brian Zimmer, who was there starting from the very first Raven tapeout. Thanks to the rest of the Raven team including Jaehwa Kwak, Ružica Jevtić, Hanh-Phuc Le, Ben Keller, Stevo Bailey, Pi-Feng Chiu, Alberto Puggelli, Milovan Blagojević, Brian Richards, Elad Alon, and Bora Nikolić. Brian, Alberto, I will never forget the night at BWRC when we got Linux to boot on the Raven3 chip.

Thanks to the members of the architecture research team at NVIDIA for helping to develop scalarization and predication ideas. Particular thanks to Ronny Krashinsky, Vinod Grover, Mark Stephenson, and Steve Keckler who helped me hack the production CUDA compiler for three years. Thanks to the rest of the architecture research team including James Balfour, Brucek Khailany, Daniel Johnson, Siva Hari, and Bill Dally.

Thanks to the members of the Chisel team at U.C. Berkeley for spearheading the custom HDL effort. Particular thanks to Jonathan Bachrach and Stephen Twigg for putting a lot of effort into making it all work. Thanks to the rest of the Chisel team including Jim Lawson.

Thanks to the members of the Maven team at U.C. Berkeley for helping to design the Maven vector-thread architecture and evaluate it. Particular thanks to Christopher Batten, who has been a great mentor and teacher, and Rimas Avižienis, who has always found a way to workaround the nastiest problems I have given up on. Thanks to the rest of the Maven team including Alex Bishara, Richard Xia, and Chris Celio.

Thanks to the members of the RAMP Gold team at U.C. Berkeley for showing me what it takes to build working things. Particular thanks to Zhangxi Tan, who always told me to *make it work!* I can now say with more confidence, *I sure did.*

Thanks to my fellow graduate students, system administrators, and administrative staff at U.C. Berkeley who have enhanced my graduate school experience and taught me a great deal. Particular thanks to Scott Beamer, Kostadin Ilov, Jon Kuroda, Roxana Infante, Tamille Johnson, and the rest of my research group.

Thanks to my parents, Sangkook Lee and Kueyoung Lee, for guiding me through my life, and being supportive throughout graduate school even from 5,000 miles away. Thanks to my wife, Soyoung, for her unending patience, love, and support for everything in my life. I could not have done it without you.

Chapter 1

Introduction

It is truly an interesting time to be a computer architect, as we hit the inflection point at which old conventional wisdom in computer architecture breaks down. For the past 50 years, Moore's law [87] in conjunction with Dennard scaling [35] have given computer architects 2× more transistors every 18 months in successive process nodes that fit in a similar power budget while increasing clock frequency. That meant the main job of a computer architect was to design a computer that delivered twice the performance compared to the previous generation by using those extra faster transistors. However, in an era where transistor scaling and Dennard scaling is slowing down or arguably grinding to a halt due to physical limits and leakage concerns, the job of a computer architect is radically different [124]. Power and energy efficiency are the most critical aspects to consider when designing a computer. Architects also need to carefully budget their transistors to deliver more performance and energy efficiency, as the cost of transistors is increasing as we move to the more advanced FinFET process nodes. Yet, users' imagination is unlimited, creating new useful applications with endlessly growing compute demands.

This thesis first surveys a wide range of data-parallel architectures—a set of machines designed by other computer architects to address the problem stated above—and analyzes pros and cons of their assembly programming models, architectural features, and compiler support. Based on these observations, this thesis then presents the new *Hwacha decoupled vector-fetch architecture*, an attempt to build a highly performant and efficient data-parallel machine that maintains the same level of programmability as others, by pushing some complexity into the *scalarizing compiler* and therefore simplifying the underlying hardware.

This chapter expands on the current trends in computer architecture, how computer architects are responding to them, and the rise of programmable data-parallel architectures, before outlining the thesis contributions and overview.

1.1 Upheaval in Computer Design

In 1965, Gordon Moore made an observation after looking at a few years of data that the number of transistors per integrated circuit doubled every year, and if that trend was extrapolated, we would

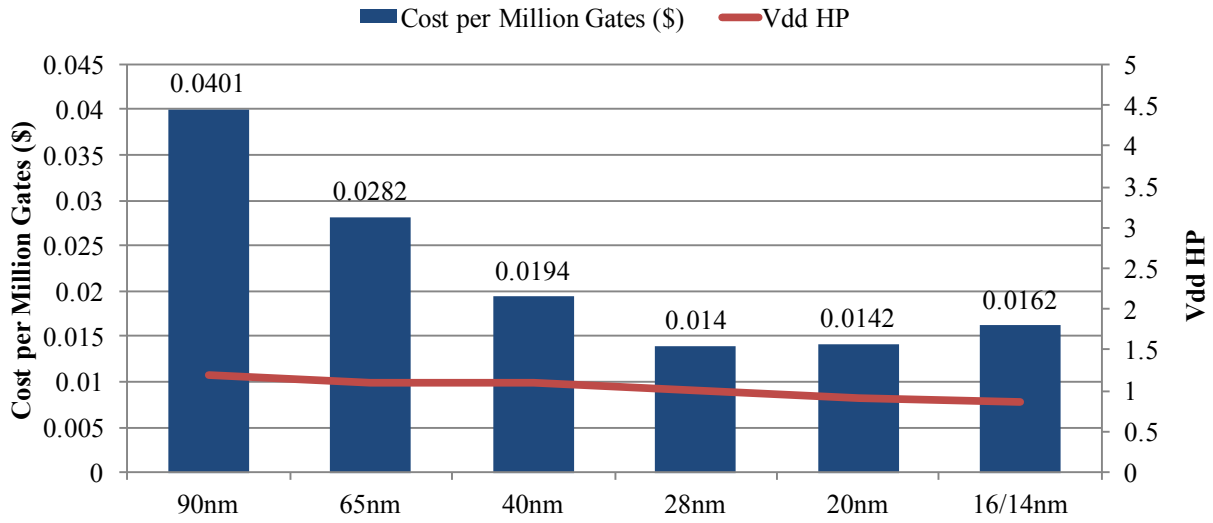


Figure 1.1: Trends in Cost per Gate and Nominal Vdd for Advanced HP (High-Performance) Process Nodes – Cost per gate data from [59] and nominal Vdd data from International Technology Roadmap of Semiconductors (ITRS).

be able to integrate 65,000 transistors on one chip by 1975 [87]. In 1974, Robert Dennard also made an important observation that as the technology scales with a factor of $1/\kappa$, the quantity of transistors (Q) increases by a factor of κ^2, the speed of the transistor (F) increases by a factor of κ, the gate capacitance (C) shrinks by a factor of $1/\kappa$, and assuming that the voltage (V) also scales by $1/\kappa$, the power density QCV^2F would remain constant [35]. These observations have later been coined as "Moore's Law" and "Dennard Scaling". Coupled together, they meant that computer architects would get twice more transistors to use at the same power budget: a golden era where performance doubled every 12–18 months thanks to higher clock frequencies, bigger caches, more functional units, more on-die integration, more aggressive speculation with better branch prediction, larger issue windows and larger reorder buffers.

However, as Gordon Moore said in 2003, no exponential is forever [88]. As shown in Figure 1.1, around 2004 when the 90 nm process node was introduced, Dennard scaling unexpectedly hit a wall in which we could no longer reduce the supply voltage by a factor of $1/\kappa$ due to increasing leakage power. As a result, the power density went up by a factor of κ^2 every process node rather than staying constant. This meant that computer architects would still get twice more transistors to use every successive process node, but they could not switch all the extra transistors at full frequency at the same time: the *dark silicon era*, a term later coined by ARM's CTO Mike Muller in 2009 [86], had arrived. Industry responded to the dark silicon era with a new roadmap of multicore designs around 2005 that doubled the number of cores on a chip each process generation while making individual cores simpler [15].

Despite skepticism, Moore's law has kept up in the last decade: five successive process generations were introduced since the 90 nm node from 2004 with advances in technology such as optical

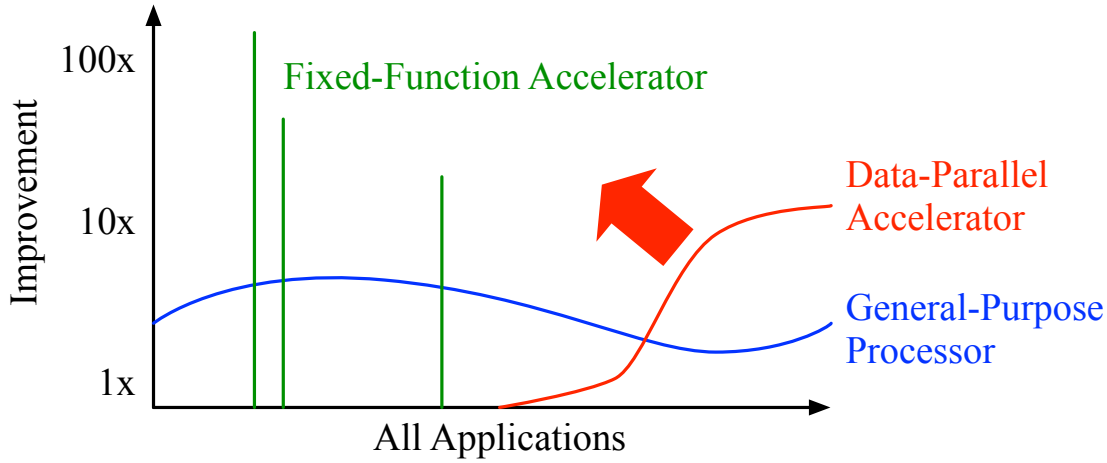


Figure 1.2: The Application Space – All applications are shown on the X-axis, and the improvement in terms of performance and energy efficiency by running an application on a particular architecture is shown on the Y-axis.

proximity correction (OPC), multi-patterning, FinFET transistors, and EUV ultraviolet lithography. However, as shown in Figure 1.1, the cost per million gates has gone up starting from the 28 nm process and beyond. This means that computer architects can still get more transistors, but they are not only dark but also no longer "cheap": as we approach the *terminal era* of Moore's law, transistors need to be carefully budgeted. It is also unclear when the actual doubling will stop since we are close to counting the number of silicon atoms across the gate of a transistor. It may be the case that 3D stacking of chips or a new process technology could keep the transistor doubling on track, however, it seems unlikely that a general solution is imminent that will be applicable across the board.

1.2 Specialization To the Rescue

So what are the implications in designing computers given the trends in Section 1.1? It means that power and energy efficiency are first-class design constraints that shape the primary design decisions in architecting a computer system. In particular, computer architects are motivated to incorporate specialized and heterogeneous accelerators into general-purpose processors.

When it comes to specialization, there are two schools of thought. Figure 1.2 helps describe both approaches. All applications that run on computer systems are shown on the X-axis, while the improvement running a given application on a given architecture over running it on a simple baseline processor in terms of performance and energy efficiency is shown on the Y-axis. For example, the blue line shows the improvement of running all applications on a general-purpose out-of-order processor. As architectural features of an out-of-order processor generally improve performance across the board, the blue line smoothly spreads across the spectrum of applications.

The blue line has ups and downs as the out-of-order features help some applications more than others, however, the variance is not too high.

One school of thought is to deploy hundreds of fixed-function accelerators that provide *focused specialization*, which individually target 100× improvement over the baseline or more [50]. These accelerators target a very limited set of applications but give large improvements, and therefore are shown as green lines in Figure 1.2 that resemble impulse responses.

The other school of thought is to deploy a few flexible accelerators that provide *general-purpose specialization*, which target somewhere around the 10× improvement range over the baseline but cover a wider range of applications. Floating-point units and data-parallel accelerators are good examples that help performance and energy efficiency while not changing the programming model, and interacting amicably with the operating system and virtualization. The red line in Figure 1.2 depicts an example data-parallel accelerator. Generally speaking, it improves performance and energy efficiency on a much wider range of applications than a fixed-function accelerator, however, the gain is lower. Since not all applications are able to take advantage of the accelerator, the red line does not spread across the entire spectrum of applications.

Both approaches have their own set of tradeoffs. It is up to the computer architect to make the decision on which approach to take to solve for a given problem. The former approach can dramatically improve performance and energy efficiency, however, more area is spent on implementing a wide range of accelerator as individual accelerators can only cover a very narrow subset of applications. Power- and clock-gating circuitry become more complicated in order to keep individual accelerators from consuming energy when they are not in-use. A bigger die area also means that more energy is spent on inter-accelerator data movement. Backwards compatibility becomes problematic as the number of accelerators grow. The latter approach is more likely to end up with a smaller die area as fewer accelerators are implemented in total, sidestepping some drawbacks of the former approach related to using up more area. The latter approach, however, might not be able to deliver the same efficiency gains of the former as it is less specialized to target a wider subset of applications. Also, the area of an individual accelerator that provides general-purpose specialization might turn out to be bigger than an individual fixed-function accelerator, therefore spending more energy on intra-accelerator data movement.

1.3 Rise of Programmable Data-Parallel Architectures

Given the current trends and constraints, especially the fact that transistors and die area are not totally free, we prefer general-purpose specialization over deploying hundreds of fixed-function accelerators, which are specialized for very narrow tasks. We believe this is why programmable data-parallel processing units, such as classic vector units, packed-SIMD units, and graphics processing units (GPUs) have become popular.

Accepting the benefits of programmable data-parallel architectures, a very important question arises for those computer architects: what architectural features should be added in order to improve performance and energy efficiency on existing applications as well as programmability to cover a wider range of applications? In other words, what can an architect do to push the red line

in Figure 1.2 towards the red arrow (upper left direction), while not hampering improvements on existing applications (not pushing the tail of the red line back down)? Indeed, in the last decade, most of the research and innovation related to data-parallel architectures were geared towards that direction. For example, GPUs added texture caches, transcendental functional units, and hardware divergence stacks to support applications with irregular control flow. Texture caches and transcendental functional units improve performance and energy efficiency on existing graphics applications, while hardware divergence stacks widen the range of applications that can run on a GPU.

This thesis documents our journey in answering the key question stated above. By surveying a wide range of data-parallel architectures, we look into the pros and cons of new architectural features that were added, and wonder whether there are simpler yet more efficient ways to support them, and also whether there are new ideas that have potential to further improve performance, energy efficiency, and programmability.

1.4 The Maven Project

The earlier Maven project was our first attempt to build a highly performant and efficient data-parallel architecture that was flexible enough to support a wide spectrum of applications. Throughout the project, we explored the tradeoffs between programmability and efficiency of three different data-parallel architectures: MIMD, traditional vector, and the newly-proposed Maven vector-thread architecture. The Maven vector-thread architecture was a hybrid architecture between traditional vector and GPU [81] to maintain the efficiency of a traditional vector machine while providing the flexibility of a GPU machine by pushing the divergence management burden onto the hardware.

We implemented a flexible microarchitecture that was capable of instantiating the MIMD architecture, the traditional vector architecture, and the Maven vector-thread architecture from the same RTL code base. We pushed hundreds of designs with different parameters through the VLSI flow using the TSMC 65 nm process node to get accurate area and cycle time numbers. We ran compiled microbenchmarks and application kernels on the gate-level simulator to get accurate performance and energy numbers. These numbers were then compared to quantify tradeoffs between programmability (how easy is it to write software for the architecture) and efficiency (energy/task and tasks/second/area).

The details of the project are published in Christopher Batten's PhD thesis [21], the ISCA conference paper [72], my master thesis [70], and the TOCS journal paper [71]. The takeaway points from the project are summarized here. Successes of the project include showing that vector-based architectures (such as the Cray-1 vector machine [109] built by Seymour Cray in the 1970s) are indeed more area and energy efficient than other data-parallel architectures. We also show that decoupling—which enables non-speculative prefetching—is an efficient way to hide memory latency. Banked vector register files are proven to be area efficient. Weaknesses of the Maven project were that the hardware divergence management was tricky to get right: we have spent more than half of our debugging cycles on getting the divergence management hardware to function

correctly. We have confirmed that lack of scalar registers has negative consequences in terms of performance and energy efficiency. Also, multi-ported flip-flop-based register files are proven to be area inefficient.

These observations heavily influenced the research in this thesis. We carefully examine other divergence management architectures (see Chapter 2), and ways to integrate scalar resources into the microarchitecture. We also explore the best machine organization to incorporate a banked register file built with area-efficient SRAM macros.

During the Maven project, we glossed over details of popular GPU architectures, due to time and resource constraints. This led me to pursue an internship at NVIDIA after the Maven project was over in 2011. I ended up working part-time at NVIDIA until 2014, where I got a chance to take a fresh look at the compiler-architecture boundary for the GPU architecture, and argue for a different split where the compiler takes more burden off the hardware to make it both simpler and more efficient. The main results are summarized in Chapter 3–5, in which we argue for scalarizing compilers.

1.5 Thesis Contributions and Overview

This thesis makes the following contributions:

- **Survey of Data-Parallel Architectures on their Assembly Programming Models, Architectural Features, and Compiler Support –** Chapter 2 starts out by providing an overview of the most popular data-parallel programming models (implicitly parallel autovectorization and explicitly parallel Single-Program Multiple-Data (SPMD) programming models like CUDA [91], and OpenCL [125]). We then walk through a range of data-parallel architectures (packed-SIMD, GPU/SIMT, traditional vector [109, 13], vector-thread [66, 21, 71]) and present their assembly programming models, the abstract low-level software interface that gives programmers or compiler writers an idea of how code executes on the machine. We also discuss how different data-parallel architectures support irregular control flow present in data-parallel programming models—the way to manage divergence is often the single most important design decision that shapes the architecture.

- **Scalarizing Compilers –** Chapter 3 presents the overheads of the SPMD programming model, and proposes an alternate way of managing the execution of SPMD programs on data-parallel architectures: let the scalarizing compiler automatically *scalarize* and *predicate* the explicitly parallel SPMD program, so that the generated code can run on simpler yet efficient data-parallel hardware. An easy way to reason about *scalarizing compilers* is that they are the opposite of *vectorizing compilers*. Vectorizing compilers automatically convert parts of a single-threaded program to run on a data-parallel unit, while scalarizing compilers automatically convert parts of an explicitly parallel program to run on a scalar processor. Scalarizing compilers also make transformations to the code to efficiently map the parallel portion down to the data-parallel unit. Chapter 4 describes the compiler foundation for the scalarization compiler pass, discusses the details of implementing the compiler pass in a

production CUDA compiler, and presents the evaluation results. Similarly, Chapter 5 walks through the compiler foundation, implementation, and evaluation results for the predication compiler pass.

- **The Hwacha Vector-Fetch Architecture –** With the scalarizing compiler mentioned above, this thesis shows that traditional vector-like architectures can maintain the same level of programmability as other data-parallel architectures while being highly performant, efficient, yet a favorable compiler target. With that in mind, Chapter 6 introduces the Hwacha vector-fetch architecture as the basis of a new data-parallel machine. We also present its assembly programming model and architectural features, and reason through the design decisions. Later chapters describe the instruction set architecture (Chapter 7), microarchitecture (Chapter 8), implementation and evaluation results (Chapter 9) of the new Hwacha vector machine.

1.6 Collaboration, Previous Publications, and Funding

As with all large systems projects, this thesis describes work that was performed as part of a group effort. Many people have made contributions to the Hwacha project. The Hwacha vector accelerator was developed by myself, Albert Ou, Colin Schmidt, Sagar Karandikar, Krste Asanović, and others from 2011 through 2016. As the lead architect of Hwacha, I directed the development and evaluation of the ISA, architecture, microarchitecture, RTL, compiler, verification framework, microbenchmarks, and application kernels. Albert Ou was primarily responsible for the RTL implementation of the Vector Memory Unit (VMU) and mixed-precision extensions. Colin Schmidt took the lead on the definition of the Hwacha ISA, RTL implementation of the scalar unit, C++ functional ISA simulator, vector torture test generator, Hwacha extensions to the GNU toolchain port, and the OpenCL compiler and benchmark suite. Sagar Karandikar took the lead on the barcrawl tool for design-space exploration, VLSI floorplanning, RTL implementation of the Vector Runahead Unit (VRU), ARM Mali-T628 MP6 GPU evaluation, and the assembly microbenchmark suite. Palmer Dabbelt took the lead on the physical design flow and post-PAR gate-level simulation in the 28 nm process technology. Henry Cook took the lead on the RTL implementation of the uncore components, including the L2 cache and the TileLink cache coherence protocol. Howard Mao took the lead on dual LPDDR3 memory channel support and provided critical fixes for the outer memory system. Andrew Waterman took the lead on the definition of the RISC-V ISA, the RISC-V GNU toolchain port, and the RTL implementation of the Rocket core. Andrew also helped to define the Hwacha ISA. John Hauser took the lead on developing the hardware floating-point units. Many others contributed to the surrounding infrastructure, such as the Rocket Chip SoC generator. Huy Vo, Stephen Twigg, and Quan Nguyen contributed to older versions of Hwacha. Finally, Krste Asanović was integral in all aspects of the project.

Some of the content in this thesis is adapted from previous publications, including: “Convergence and Scalarization for Data-Parallel Architectures” from CGO, 2013 [74], “Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures” from MICRO, 2014 [73], “A 45nm 1.3GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector

Accelerators" from ESSCIRC, 2014 [78], "A Case for MVPs: Mixed-Precision Vector Processors" from PRISM, 2014 [100], "A Case for OS-Friendly Hardware Accelerators" from WIVOSCA, 2013 [127], and technical reports, including: "The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1" [77], "The Hwacha Microarchitecture Manual, Version 3.8.1" [75], "Hwacha Preliminary Evaluation Results, Version 3.8.1" [76]. Permission to use any material from above publications have been received from all co-authors in writing (via email).

This work has been partially funded by the following sponsors.

- **Par Lab**: Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates: Nokia, NVIDIA, Oracle, and Samsung.
- **Silicon Photonics**: DARPA POEM program, Award HR0011-11-C-0100.
- **ASPIRE Lab**: DARPA PERFECT program, Award HR0011-12-2-0016. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support came from ASPIRE Lab industrial sponsors and affiliates: Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung.
- **NVIDIA graduate fellowship**

Chapter 2

Background

Data-parallel architectures have long been known to provide greater performance and energy efficiency than general-purpose architectures for applications with abundant data-level parallelism. This chapter provides background on a wide range of data-parallel architectures such as Cray-1-like vector machines [109, 71], Intel packed-SIMD units [105, 82], NVIDIA and AMD graphics processing units (GPUs) [81, 97, 96, 5], and Intel MIC accelerators [110]. We mainly discuss how they get programmed, and how the code gets executed on the machine. Section 2.1 first presents two of the most popular data-parallel programming models—autovectorization and the *Single-Program Multiple-Data* (SPMD) model. Section 2.2 then pops down one level and describes assembly programming models for these data-parallel architectures, the abstract low-level software interface that gives programmers or compiler writers an idea on how code executes on the machine. Section 2.3 goes into more detail on how these data-parallel architectures support irregular control flow present in the programming models described above. Divergence management is of particular importance, as it often ends up dictating how a data-parallel machine is organized. Section 2.4 summarizes and gives an overview of the rest of the thesis.

2.1 Data-Parallel Programming Languages

Programming data-parallel systems is inherently challenging, and over decades of research and development only a few models have attained broad success. Implicitly parallel autovectorization approaches were popular with early vector machines, while explicitly parallel *Single-Program Multiple-Data* (SPMD) accelerator languages like CUDA [91] and OpenCL [125] have proven to be accessible and productive for newer GPUs and SIMD extensions. Figure 2.1 uses the SGEMM kernel, which multiplies 32 by 32 single-precision floating-point matrices (**C = A**$\times$**B**), as an example to describe the similarities and differences of both programming models.

The SGEMM kernel is written as a triply nested loop in Figure 2.1a. In the program, the outer two loops `i` and `j` (lines 2 and 3) iterate through all elements in the C matrix, and perform a dot product of a row in matrix A and a column in matrix B. The same SGEMM kernel is written in the SPMD programming model in Figure 2.1c. The program is organized into a 2-D thread launch

```
1 float C[32][32], A[32][32], B[32][32]; // assume C is zeroed
2
3 for (int i = 0; i < 32; ++i)
4   for (int j = 0; j < 32; ++j)
5     for (int k = 0; k < 32; ++k)
6       C[i][j] += A[i][k] * B[k][j];
```

(a) Triply Nested Loop

```
1 float C[32][32], A[32][32], B[32][32]; // assume C is zeroed
2
3 for (int i = 0; i < 32; ++i)
4   for (int k = 0; k < 32; ++k)
5     for (int j = 0; j < 32; ++j)
6       C[i][j] += A[i][k] * B[k][j];
```

(b) Interchanged Triply Nested Loop that is Vectorizable

```
1 float C[32][32], A[32][32], B[32][32]; // assume C is zeroed
2
3 sgemm<<<32, 32>>>(C, A, B);
4
5 void sgemm(float** C, float** A, float** B)
6 {
7   for (int k = 0; k < 32; ++k)
8     C[tid.y][tid.x] += A[tid.y][k] * B[k][tid.x];
9 }
```

(c) SPMD Model

Figure 2.1: Single-Precision Matrix Multiplication Kernel (SGEMM) Written in Two Parallel Programming Languages – (a) triply nested loop, (b) interchanged triply nested loop that is vectorizable, and (c) SPMD model.

(line 3) and an *kernel* function (lines 5–9) that all threads execute. 32 threads are launched in both X and Y dimensions (1024 threads are launched in total), and they mimic the outer two loops `i` and `j` from Figure 2.1a. Each thread has access to the loop variables `i` and `j` through `tid.y` and `tid.x`, and is responsible for calculating the result for an element in the C matrix.

The autovectorization programming model lets the programmer write sequential loops to express data-parallel computation. It is the vectorizing compiler's job to extract the parallelism out of a single-threaded program, hence we refer to it as the implicitly parallel programming model. The vectorizing compiler first analyzes loop dependence of the nested loop, interchanges the loop ordering based on the dependence information so that the innermost loop is vectorizable (i.e., the loop does not carry a dependence that prevents from running it in parallel) and the outermost loop is parallelizable, and finally performs the vectorizing transformation on the code [1, 3, 4]. Bernstein's conditions for parallel execution give the formal definition of dependence [23]. There are three types of dependences among two statements that access the same memory location, which are identical to the data hazards that show up in a processor pipeline [55]—true dependence (a read-after-write or RAW hazard in a processor pipeline), anti-dependence (a WAR hazard), and output dependence (a WAW hazard). Testing loop dependence based on the dependence information above turns out to be an NP-complete problem [49], so conservative heuristics are often used as approximations such as the Banerjee's test [20] and the more recent polyhedral models [25].

A vectorizing compiler would take the triply nested loop in Figure 2.1a, analyze the loop dependence, and figure out that the inner loop (`k` loop) cannot be vectorized (i.e., all iterations must execute sequentially), since it writes and reads the same memory location (`C[i][j]`) every iteration. The compiler will then interchange the `k` loop and the `j` loop as shown in Figure 2.1b, as it maximizes profitability: the resulting triply nested loop not only exposes unit-strided vector memory operations in the innermost loop, but also allows the outermost loop to be executed in parallel. Finally, the compiler will vectorize the innermost `j` loop, and generate code with scalar instructions and vector instructions. The compiler leaves loop bookkeeping, address calculation, shared data across all elements in a vector on the scalar control processor, and only maps the data-parallel computation onto the vector unit. The compiler will also generate unit-strided vector loads and stores for efficient data movement and vector-scalar instructions to minimize data replication. If the compiler fails to vectorize the code for whatever reason, it can always fall back and map the code onto the scalar processor.

SPMD accelerator languages express data-parallel computation in the form of multithreaded kernels, hence we refer to it as the explicitly parallel programming model. Inside a kernel, the programmer writes code for a single thread. A thread typically processes a small amount of data. For example, a thread might compute the color of a single pixel in a graphics application. The programmer expresses parallel computation with explicit data-parallel kernel invocations that direct a group of threads to execute the kernel code. In CUDA, these thread groups are termed *cooperative thread arrays* (CTAs). A CTA may have up to 1024 threads. This approach allows the SPMD compiler to use a fairly conventional thread compilation model, while pushing most of the burden onto the hardware to figure out an efficient way of executing the threaded code.

Explicitly data-parallel languages map naturally to highly multithreaded architectures, such as GPUs and other multicore accelerators. These throughput architectures leverage parallelism spa-

tially to execute computations at a high rate across many datapaths and cores. They also leverage parallelism temporally to saturate high-bandwidth memory systems. The interleaved execution of multiple threads essentially hides hardware latencies from each individual thread. This approach simplifies the programming model since the code written for an individual thread can simply access data and operate on it, without great concern for the access latency.

SPMD programming models also implicitly expose locality, which improves efficiency. GPUs use a *Single-Instruction Multiple-Thread* (SIMT) architecture that executes an instruction on parallel datapaths for many threads at the same time, for example 32 threads in *warps* using NVIDIA terminology, or 64 threads in *wavefronts* using AMD terminology. A warp may issue in a single cycle if the datapath width matches the warp width, or it may be sequenced over several cycles on a narrower datapath. Similar to *Single-Instruction Multiple-Data* (SIMD) architectures, SIMT architectures use this organization to amortize the instruction fetch and other control overheads associated with executing instructions. SIMT architectures also derive efficiency from data locality for the common case when the threads in a warp access neighboring data elements. To exploit this locality, SIMT architectures use *dynamic address coalescing* to turn individual element accesses into wide block accesses that the memory system can process more efficiently, for example with only a single cache tag check.

Although the SPMD programming model is simple for the programmer, it can introduce many hidden overheads. The programmers cannot express the scalar parts of the computation, since they only specify the kernel function and the grid dimensions for thread launch. As a result, there are instructions and data that are redundant across all threads executing a kernel. Also, SPMD programs tend to have substantial and complex per-thread control flow, extending beyond simple if-then-else clauses to nested loops and function calls. Section 2.3 describes how different data-parallel architectures manage divergence (i.e., support irregular control flow present in SPMD programming models) in more detail. Chapter 3 describes the overheads of the SPMD programming model in more detail, and how *scalarizing compilers* alleviate these overheads while retaining the identical threaded SPMD programming model.

2.2 Assembly Programming Models of Data-Parallel Architectures

This section introduces assembly programming models of various data-parallel architectures, such as packed-SIMD, GPU/SIMT, traditional vector, and vector-thread architectures. The assembly programming model is the abstract low-level software interface that gives programmers or compiler writers an idea on how the code executes on each data-parallel machine. The compiler will typically take the high-level program written in a parallel programming language introduced in the previous section, and generate assembly code for a given data-parallel architecture described below.

As a running example, we use a conditionalized SAXPY kernel (CSAXPY), which performs single-precision **a·X plus Y** conditionally. Figure 2.2 shows CSAXPY expressed in both a vec-

```
1 void csaxpy(size_t n, bool cond[], float a, float x[], float y[])
2 {
3   for (size_t i = 0; i < n; ++i)
4     if (cond[i])
5       y[i] = a*x[i] + y[i];
6 }
```

(a) Vectorizable Loop

```
1 csaxpy_spmd<<<((n-1)/32+1)*32>>>;
2
3 void csaxpy_spmd(size_t n, bool cond[], float a, float x[], float y[])
4 {
5   if (tid.x < n)
6     if (cond[tid.x])
7       y[tid.x] = a*x[tid.x] + y[tid.x];
8 }
```

(b) SPMD Model

Figure 2.2: Conditional SAXPY Kernel Written in C – (a) vectorizable loop, and (b) SPMD model.

torizable loop and as a SPMD kernel. CSAXPY takes as input an array of conditions, a scalar **a**, and vectors **x** and **y**; it computes **y += ax** for the elements for which the condition is true. In Figure 2.2a, the computation is written as a sequential for loop (lines 3–5). In Figure 2.2b, the computation is written as a 1-D thread launch (line 1), and a kernel function `saxpy_spmd` (lines 3–8). Note, the number of launched threads has to be quantized to a multiple of the warp size (32 for NVIDIA GPUs), and as a result, the kernel function has to check whether the thread index (`tid.x`) is in bounds with the application vector length (`n`) with a conditional. Note, this bounds check is unnecessary when the computation is expressed with a for loop in Figure 2.2a.

Packed-SIMD Assembly Programming Model

Figure 2.3a shows the CSAXPY kernel mapped to a hypothetical packed-SIMD architecture with predication support, similar to Intel's SSE and AVX extensions. Predication support is quite uncommon; Intel's AVX architecture, for example, only supports predication as of 2015, and then only in its Xeon line of server processors. The example SIMD architecture has 128-bit registers, each partitioned into four 32-bit fields. As with other packed-SIMD machines, ours cannot mix scalar and vector operands, so the code begins by filling a SIMD register with copies of **a** (line 4). To map a long vector computation to this architecture, the compiler generates a *stripmine loop*, each iteration of which processes one four-element vector (lines 5–15). In this example, the stripmine loop consists of a load from the conditions vector (line 6), which in turn is used to set a predicate register (line 7). The next four instructions (lines 8–11), which correspond to the body

```
1  csaxpy_simd_with_predication:
2        slli    a0, a0, 2
3        add     a0, a0, a3
4        vsplat4 vv0, a2
5  stripmine_loop:
6        vlb4    vv1, (a1)
7        vcmpez4 vp0, vv1
8  !vp0  vlw4    vv1, (a3)
9  !vp0  vlw4    vv2, (a4)
10 !vp0  vfma4   vv1, vv0, vv1, vv2
11 !vp0  vsw4    vv1, (a4)
12       addi    a1, a1, 4
13       addi    a3, a3, 16
14       addi    a4, a4, 16
15       bleu    a3, a0, stripmine_loop
16 # handle edge cases
17 # when (n % 4) != 0 ...
18       ret
```

(a) With Predication

```
1  csaxpy_simd_without_predication:
2    slli    a0, a0, 2
3    add     a0, a0, a3
4    vsplat4 vv0, a2
5  stripmine_loop:
6    vlb4    vv1, (a1)
7    vcmpez4 vv3, vv1
8    vlw4    vv1, (a3)
9    vlw4    vv2, (a4)
10   vfma4   vv1, vv0, vv1, vv2
11   vblend4 vv1, vv3, vv1, vv2
12   vsw4    vv1, (a4)
13   addi    a1, a1, 4
14   addi    a3, a3, 16
15   addi    a4, a4, 16
16   bleu    a3, a0, stripmine_loop
17 # handle edge cases
18 # when (n % 4) != 0 ...
19   ret
```

(b) Without Predication

Figure 2.3: CSAXPY Kernel Mapped to the Packed-SIMD Assembly Programming Model – In all pseudo-assembly examples presented in this section, `a0` holds variable `n`, `a1` holds pointer `cond`, `a2` holds scalar `a`, `a3` holds pointer `x`, and `a4` holds pointer `y`. The `vblend4 d,m,s,t` instruction computes `d=s&m|t&~m` and implements a select function.

of the *if*-statement (`y[i]=a*x[i]+y[i]`) in Figure 2.2a, are masked by the predicate register. Finally, the address registers are incremented by the SIMD width (lines 12–14), and the stripmine loop is repeated until the computation is finished—almost. Since the loop handles four elements at a time, extra code is needed to handle up to three *fringe* elements (lines 16–17). For brevity, we omitted this code; in this case, it suffices to duplicate the loop body, predicating all of the instructions on whether their index is less than **n**.

Figure 2.3b shows CSAXPY kernel mapped to a similar packed-SIMD architecture without predication support. The compare instruction sets up the mask in a vector register instead (line 7). The bulk of the computation is done regardless of the condition (lines 8–10). The `vblend4` selects the new value or the old value depending on the condition (line 11). The result is then stored back out to memory (line 12). The fringe case is handled on the scalar processor due to lack of predication.

The most important drawback to packed-SIMD architectures lurks in the assembly code: the SIMD width is expressly encoded in the instruction opcodes and memory addressing code. When the architects of such an ISA wish to increase performance by widening the vectors, they must add a new set of instructions to process these vectors. This consumes substantial opcode space:

for example, Intel's newest AVX instructions are as long as 11 bytes. Worse, application code cannot automatically leverage the widened vectors. In order to take advantage of them, application code must be recompiled. Conversely, code compiled for wider SIMD registers fails to execute on older machines with narrower ones. As we later show, this complexity is merely an artifact of poor design.

GPU/SIMT Assembly Programming Model

Figure 2.4 shows the same code mapped to a hypothetical SIMT architecture, akin to an NVIDIA GPU. The SIMT architecture exposes the data-parallel execution resources as multiple threads of execution; each thread executes one element of the vector. The microarchitecture fetches an instruction once but then executes it on many threads simultaneously using parallel datapaths. Therefore, a scalar instruction shown in Figure 2.4 executes like a vector instruction.

One inefficiency of this approach is immediately evident: the first action each thread takes is to determine whether it is within bounds, so that it can conditionally perform no useful work (line 3). Section 2.3 details how the microarchitecture manages divergence at control conditions such as if-then-else statements, loops, and function calls, as each thread may execute control flow independently.

Another inefficiency results from the duplication of scalar computation: despite the unit-stride access pattern, each thread explicitly computes its own addresses (lines 4, 7–9). The SIMD architecture, in contrast, amortized this work over the SIMD width, as the address bookkeeping is only done once per vector on the scalar processor (lines 12–14 in Figure 2.3a). Moreover, massive repli-

```
1  csaxpy_simt:
2    mv    t0, tid
3    bgeu  t0, a0, skip
4    add   t1, a1, t0
5    lb    t1, (t1)
6    beqz  t1, skip
7    slli  t0, t0, 2
8    add   a3, a3, t0
9    add   a4, a4, t0
10   lw    t1, (a3)
11   lw    t2, (a4)
12   fma   t0, a2, t1, t2
13   sw    t0, (a4)
14 skip:
15   stop
```

Figure 2.4: CSAXPY Kernel Mapped to the GPU/SIMT Assembly Programming Model – The SPMD kernel launch code, which runs on the host processor, is omitted for brevity. This example only shows the assembly code for the kernel function.

cation of scalar operands reduces the effective utilization of register file resources: each thread has its own copy of the three array base addresses (registers `a1`, `a3`, and `a4`) and the scalar **a** (register `a2`). This represents a threefold increase over the fundamental architectural state.

Traditional Vector Assembly Programming Model

Packed-SIMD and GPU/SIMT architectures have a disjoint set of drawbacks: the main limitation of the former is the static encoding of the vector length, whereas the primary drawback of the latter is the lack of scalar processing. One can imagine an architecture that has the scalar support of the former and the dynamism of the latter. In fact, it has existed for over 40 years, in the form of the traditional vector machine, embodied by the Cray-1. The key feature of this architecture is the *vector length register* (VLR), which represents the number of vector elements that will be processed by the vector instructions, up to the hardware vector length (HVL). As shown in Figure 2.5, software manipulates the VLR with a `vsetvl` instruction (line 3), which requests a certain application vector length (AVL); the vector unit responds with the smaller of the AVL and the HVL [13].

As with packed-SIMD architectures, a stripmine loop iterates until the application vector has been completely processed. But, as Figure 2.5 shows, the difference lies in the manipulation of the VLR at the head of every loop iteration (line 3). The primary benefits of this architecture follow directly from this code generation strategy. Most importantly, the scalar software is completely oblivious to the hardware vector length: the same code executes correctly and with maximal effi-

```
1  csaxpy_tvec:
2  stripmine_loop:
3       vsetvl  t0, a0
4       vlb     vv0, (a1)
5       vcmpez  vp0, vv0
6  !vp0 vlw     vv0, (a3)
7  !vp0 vlw     vv1, (a4)
8  !vp0 vfma    vv0, vv0, a2, vv1
9  !vp0 vsw     vv0, (a4)
10      add     a1, a1, t0
11      slli    t1, t0, 2
12      add     a3, a3, t1
13      add     a4, a4, t1
14      sub     a0, a0, t0
15      bnez    a0, stripmine_loop
16      ret
```

Figure 2.5: CSAXPY Kernel Mapped to the Traditional Vector Assembly Programming Model – The overall structure looks similar to Figure 2.3a, however, the vector length is not exposed in the vector instruction.

ciency on machines with any HVL. Second, there is no fringe code: on the final trip through the loop, the VLR is simply set to the length of the fringe.

The advantages of traditional vector architectures over the GPU/SIMT approach are owed to the coupled scalar control processor. There is only one copy of the array pointers and of the scalar **a**. The address computation instructions execute only once per stripmine loop iteration, rather than once per element, effectively amortizing their cost by a factor of the HVL.

Vector-Thread Assembly Programming Model

The vector-thread assembly programming model loosely follows the traditional vector assembly programming model. The vector memory instructions are left in the stripmine loop, while the vector arithmetic instructions are hoisted out into a separate vector-fetch block.

Figure 2.6 shows the same CSAXPY code mapped to the Maven vector-thread architecture [71]. The stripmine loop structure is unchanged (lines 2–15). The `mtvtu` instruction moves the scalar **a** to a vector register. Vector memory operations (lines 5–7, 9) are left in the stripmine loop, while the vector arithmetic operations are hoisted out into a separate vector-fetch block (lines 18–22), and are connected via a *vector-fetch* instruction (line 8). The vector-fetch abstraction is that each

```
1  csaxpy_vt:
2  stripmine_loop:
3    vsetvl  t0, a0
4    mtvtu   vv0, a2
5    vlb     vv1, (a1)
6    vlw     vv2, (a3)
7    vlw     vv3, (a4)
8    vf      csaxpy_vt_vf
9    vsw     vv3, (a4)
10   add     a1, a1, t0
11   slli    t1, t0, 2
12   add     a3, a3, t1
13   add     a4, a4, t1
14   sub     a0, a0, t0
15   bnez    a0, stripmine_loop
16   ret
17
18 csaxpy_vt_vf:
19   beqz  v1, skip
20   fma   v3, v0, v2, v3
21 skip:
22   stop
```

Figure 2.6: CSAXPY Kernel Mapped to the Vector-Thread Assembly Programming Model – The `mtvtu` instruction stands for "move to vector-thread unit" and splats a scalar value to a vector register, similar to the `vsplat4` instruction used in Figure 2.3.

element in a vector will execute the scalar instructions in the vector-fetch block (lines 19–22) until it hits the `stop` instruction (line 22), similar to the GPU/SIMT assembly programming model shown in Figure 2.4. Note, the scalar control processor views a scalar register in a vector-fetch block as a vector register: register `v0` (line 20) is considered as vector register `vv0` (line 4) outside the vector-fetch block.

The vector-thread assembly programming model lets the Maven vector-thread processor queue up the vector arithmetic instructions only to be fetched and decoded after vector memory instructions have run ahead to prefetch the needed data. The microarchitecture implements a *Pending Vector Fragment Buffer* (PVFB) in hardware to implicitly manage divergence—a consequence of allowing branches in the assembly programming model (line 19). Similar to the GPU/SIMT assembly programming model, the Maven vector-thread assembly programming model precludes scalar operands and scalar computation to be expressed in a vector-fetch block, which affects the efficiency of the architecture. More details of the Maven vector-thread architecture can be found in [71].

The earlier Scale vector-thread architecture had a similar vector-thread assembly programming model, however, the clustering of the different functional units within the vector lane is exposed to the programmer or the compiler writer, complicating the assembly programming model [66, 53].

2.3 Divergence Management Schemes of Data-Parallel Architectures

This section provides an overview on how different data-parallel architectures support control flow present in the parallel programming languages discussed in Section 2.1. For the discussion, without any loss of generality, we use examples written in the SPMD programming model, where data-parallel computation is expressed in the form of multithreaded kernels. The programmer

```
1  kernel<<<32>>>(...);
2
3  void kernel(...) {
4    a = op0;
5    b = op1;
6    if (a < b) {
7      c = op2;
8    } else {
9      c = op3;
10   }
11   d = op4;
12 }
```

(a) If-Then-Else Statement

```
1  kernel<<<32>>>(...);
2
3  void kernel(...) {
4    bool done = false;
5    while (!done) {
6      a = op0;
7      b = op1;
8      done = a < b;
9    }
10   c = op2;
11 }
```

(b) While Loop

Figure 2.7: Control Flow Examples Written in the SPMD Programming Model – (a) if-then-else statement, and (b) while loop.

writes code both for a single thread and for an explicit kernel invocation that directs a group of threads to execute the kernel code in parallel.

To achieve an efficient mapping, threads are processed together in SIMD vectors, but orchestrating the execution of SPMD threads on wide SIMD datapaths is challenging. As each thread executes control flow independently, execution may *diverge* at control conditions such as if-then-else statements, loops, and function calls. The architecture must therefore track and sequence through the various control paths taken through the program by the different elements in a vector. This is generally done by selectively enabling a subset of threads in a vector while each control path is traversed. Because divergence leads to a loss of efficiency, *reconvergence* is another important component of divergence management on data-parallel architectures.

We explain next how different data-parallel architectures manage divergence and reconvergence with the if-then-else statement and the while loop example shown in Figure 2.7. Due to the SIMD execution nature of these architectures, the hardware must provide a mechanism to execute an instruction only for selected elements within a vector. Some architectures expose predication to the compiler, while others hide it from the compiler and manage divergence implicitly in hardware.

Vector-Like Machines

Compilers for vector machines manage divergence explicitly. Figure 2.8 illustrates how a vector machine would typically handle the control flow shown in Figure 2.7. The example shows a mixture of scalar and vector instructions, along with scalar and vector registers. To execute both sides of the if-then-else statement conditionally, the vector machine first writes the result of the `vslt` compare instruction into a vector predicate register `vp0`, then conditionally executes `vop2` under `vp0`. Similarly, `vop3` is executed under the negated condition `!vp0`. Vector register `vc` is partially updated from both sides of the execution paths. The diverged execution paths merge

```
1        va = vop0
2        vb = vop1
3        vp0 = vslt va, vb
4        s0 = vpopcnt vp0
5        branch.eqz s0, else
6   @vp0 vc = vop2
7  else:
8        s0 = vpopcnt !vp0
9        branch.eqz s0, ipdom
10 !@vp0 vc = vop3
11 ipdom:
12       vd = vop4
```

(a) If-Then-Else Statement

```
1       vp0 = true
2  loop:
3       s0 = vpopcnt vp0
4       branch.eqz s0, exit
5  @vp0 va = vop0
6  @vp0 vb = vop1
7  @vp0 vp1 = vslt va, vb
8       vp0 = vand vp0, !vp1
9       j loop
10 exit:
11      vc = vop2
```

(b) While Loop

Figure 2.8: Divergence Management on Vector Architectures – (a) if-then-else statement, and (b) while loop.

```
1         va = vop0
2         vb = vop1
3         vslt va, vb # vcc=va<vb
4         s0 = mov exec
5         exec = and s0, vcc
6         branch.execz else
7         vc = vop2
8   else:
9         exec = and s0, !vcc
10        branch.execz ipdom
11        vc = vop3
12  ipdom:
13        exec = mov s0
14        vd = vop4
```

(a) If-Then-Else Statement

```
1       s0 = mov exec
2   loop:
3       branch.execz exit
4       va = vop0
5       vb = vop1
6       vslt va, vb # vcc=va<vb
7       exec = and exec, !vcc
8       j loop
9   exit:
10      exec = mov s0
11      vc = vop2
```

(b) While Loop

Figure 2.9: Divergence Management on Vector Architectures with Implicit Predicates – (a) if-then-else statement, and (b) while loop. The `vcc` predicate register implicitly stores the result of the vector compare instruction (`vslt` above). The `exec` predicate register implicitly masks all instruction execution.

at the immediate post-dominator, where `vop4` is executed. The compiler statically encodes this information by emitting `vop4` under no predicate condition.

Several optimizations such as density-time execution and compress-expand transformations have been proposed [115] and evaluated [71] to save execution time of sparsely activated vector instructions. However, these optimizations cannot prevent vector instructions with an all-false predicate mask from being fetched and decoded. The compiler can optionally insert a check to test whether the predicate condition is null, meaning that instructions under that predicate condition are unnecessary. In Figure 2.8a, both conditionally executed paths are guarded with a dynamic check. A `vpopcnt` instruction, which writes a count of all `true` conditions in a vector predicate register to a scalar register, is used to count active elements. A scalar branch (`branch.eqz` instruction) checks whether the count is zero to jump around unnecessary work. These checks may not always turn out to be profitable, as the condition could truly be unbiased. For those cases, it would be better to schedule both sides of the execution paths simultaneously.

Figure 2.8b shows how while loops in SPMD programs are mapped to vector machines. Loop mask `vp0`, which keeps track of active elements executing the loop, is initialized to `true`. A `vpopcnt` instruction is combined with a branch-if-equals-zero instruction to test whether all elements have exited the loop. All instructions in the loop body (`vop0`, `vop1`, `vslt`) are predicated under the loop mask `vp0`, except the `vand` instruction, which updates the loop mask. The loop's backwards branch is implemented with an unconditional jump instruction (`j`).

Cray-1 vector processors, AMD GPUs, and Intel MIC accelerators execute in a similar manner as shown in Figure 2.9. The Cray-1 has one vector predicate register on which vector instructions

are implicitly predicated [109, 115], while the Intel MIC has 8 explicit vector predicate registers [57]. AMD GPUs use special `vcc` and `exec` predicate registers to hold vector comparison results and to implicitly mask instruction execution [5]. AMD GPUs also provide a form of escape hatch for complex irreducible control flow. Fork and join instructions are provided for managing divergence in these cases, and a stack of deferred paths is physically stored in the scalar register file [7].

Some vector machines lack vector predicate registers and instead have an instruction to conditionally move a word from a source register to a destination register. Packed-SIMD extensions in Intel desktop processors are a common example of this pattern. However, these approaches have limitations, including the exclusion of instructions with side-effects from poly-path execution [83]. Karrenberg and Hack [61, 60] propose compiler algorithms to map OpenCL kernels down to packed-SIMD units with explicit vector blend instructions.

NVIDIA Graphics Processing Units

Hardware Divergence Management. NVIDIA GPUs provide implicit divergence management in hardware. As shown in Figure 2.10, control flow is expressed with *thread branches* (`tbranch.eqz` and `tbranch.neqz` instructions) in the code. When threads in a warp branch in different directions, the warp diverges and is split into two subsets: one for branch taken and the other for branch not taken. Execution of one subset is deferred until the other subset has completed execution. A *divergence stack* is used to manage execution of deferred warp subsets. The compiler pushes a *reconvergence* point with the current active mask onto the divergence stack before the thread branch. The reconvergence point indicates where diverged threads are supposed to join.

In Figure 2.10a, the reconvergence point is the immediate post-dominator of the if-then-else statement. When the warp splits, the hardware picks one subset (`then` clause), and pushes the

```
1     a = op0
2     b = op1
3     p = slt a, b
4     push.stack reconverge
5     tbranch.eqz p, else
6     c = op2
7     pop.stack
8  else:
9     c = op3
10    pop.stack
11 reconverge:
12    d = op4
```

(a) If-Then-Else Statement

```
1     done = false
2     push.stack reconverge
3  loop:
4     tbranch.neqz done, exit
5     a = op0
6     b = op1
7     done = slt a, b
8     j loop
9  exit:
10    pop.stack
11 reconverge:
12    c = op2
```

(b) While Loop

Figure 2.10: Hardware Divergence Management on NVIDIA GPUs – (a) if-then-else statement, and (b) while loop.

```
1        a = op0
2        b = op1
3        p0 = slt a, b
4        cbranch.ifnone p0, else
5    @p0 c = op2
6  else:
7        cbranch.ifnone !p0, ipdom
8   @!p0 c = op3
9  ipdom:
10       d = op4
```

(a) If-Then-Else Statement

```
1       p0 = true
2  loop:
3       cbranch.ifnone p0, exit
4   @p0 a = op0
5   @p0 b = op1
6   @p0 p1 = slt a, b
7       p0 = and p0, !p1
8       j loop
9  exit:
10      c = op2
```

(b) While Loop

Figure 2.11: Software Divergence Management on NVIDIA GPUs – (a) if-then-else statement, and (b) while loop.

other (`else` clause) onto the stack. Then the hardware first executes `op2` under an updated active mask, which is now set to the active threads of the `then` clause. When the hardware executes the `pop.stack` operation, it discards the currently executing warp subset and picks the next warp subset on the top of the stack (`else` clause). When execution reaches the next `pop.stack` operation, it pops the PC for the reconvergence point. If all threads follow the same `then` or `else` path at the branch, the warp hasn't diverged, so no thread subset is pushed on the stack, and the only `pop.stack` operation pops the PC for the reconvergence point. Divergence and reconvergence nest hierarchically through the divergence stack.

The reconvergence point of a loop is the immediate post-dominator of all loop exits. Since there is only one exit in the example shown in Figure 2.10b, the exit block is the reconvergence point. The compiler similarly pushes the reconvergence point onto the stack and sequences the loop until all threads have exited the loop. All exited threads will pop a token off the stack, until eventually the reconverged PC with the original mask at the loop entry is recovered.

Software Divergence Management. NVIDIA GPUs with divergence stacks also provide support to manage divergence in software. The hardware provides predicate registers, native instructions with guard predicates, and *consensual branches* [92], where the branch is only taken when all threads in a warp have the same predicate value. Figure 2.11 shows how the compiler could manage divergence on NVIDIA hardware. Conditional execution is expressed with predicates, and consensual branches (`cbranch.ifnone` instruction) can be added to jump around unnecessary work (see Figure 2.11a), and also sequence a loop until all threads have exited (see Figure 2.11b).

Although consensual branches are implemented in current NVIDIA GPUs, they have not been publicly described except for in an issued US patent [92]. NVIDIA's thread-level predication and consensual branches are isomorphic to vector predication and the scalar branches on popcount values used in vector processors. This scheme is only used in a very limited fashion by the current NVIDIA backend compiler. The compiler selects candidate if-then-else regions using pattern

matching, and employs a simple heuristic to determine if predication will be advantageous compared to using the divergence stack.

2.4 Background Summary

This chapter provides background on data-parallel programming models, assembly programming models and divergence management architectures of a wide range of data-parallel architectures (such as packed-SIMD, GPU/SIMT, traditional vector [109, 13], vector-thread [66, 21, 71]), and establishes the common terminology that this thesis relies on for the rest of the chapters.

Chapters 3–5 present the overheads of the popular SPMD programming model, and propose how to alleviate these overheads with compiler technology. Based on these observations, Chapters 6–9 propose the new Hwacha vector-fetch data-parallel architecture. Chapter 10 concludes this thesis.

Chapter 3

Scalarizing Compilers

Over the past decade, the SPMD programming model has gained popularity among programmers over autovectorization. The recent proliferation of SPMD programming models is mainly linked to the rise of GPUs that provide teraflops of compute at a cheap price point thanks to the commoditized graphics cards market—programmers must use the SPMD programming model to tap into the cheap compute available on a GPU. A GPU architecture is able to substantially reduce the program counter and instruction fetch overheads of multithreading, but many hidden overheads of the SPMD programming model remain. Writing kernel code for a single thread is simple for the programmer and improves productivity, but with a conventional compiler this model can create a substantial amount of redundant work across threads. Also, a lot of burden is put on the underlying architecture to support the complex control flow found in SPMD programs, which extends beyond simple if-then-else clauses to nested loops and function calls. Section 3.1 first shows how a *scalarizing compiler* can alleviate them, before outlining the *scalarization* and *predication* compiler algorithms that constitute a scalarizing compiler in Sections 3.2 and 3.3, respectively.

3.1 Overheads of SPMD

The SPMD programming model may be accessible to the programmer, however, certain overheads of the programming model still remain even after the GPU architecture is able to amortize instruction fetch overhead by grouping threads and executing them in SIMD fashion. Using motivating examples, Section 3.2 points out redundancy across threads as one of the key inefficiencies of the SPMD programming model, while Section 3.3 describes the overhead of supporting complex control flow constructs in SPMD programming models.

This thesis proposes to alleviate these overheads with a scalarizing compiler. Figure 3.1 contrasts scalarizing compilers to vectorizing compilers. Vectorizing compilers automatically pick out parts that can execute in parallel from single-threaded code, and efficiently map them down to the data-parallel unit. While doing so, it leaves bookkeeping, address calculation, shared data on the scalar processor. Conversely, scalarizing compilers find the most efficient mapping of explicitly parallel programs (e.g., SPMD programs) down to the scalar processor and the data-parallel unit.

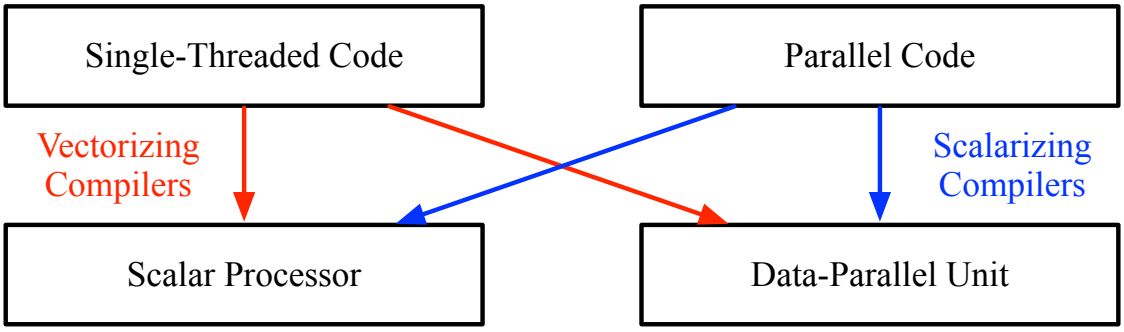


Figure 3.1: Vectorizing Compilers and Scalarizing Compilers – Single-threaded code examples include nested for loops, while parallel code examples include explicitly parallel SPMD programs.

This thesis focuses on scalarization and predication among many things a scalarizing compiler can do. The scalarization compiler pass statically analyzes SPMD programs to pick out redundant operands and instructions, and generates code with shared scalar registers and scalar instructions. Chapter 4 details the compiler foundation for scalarization, and presents our implementation and evaluation results. The predication compiler pass maps complex control flow found in SPMD programs down to predicated vector instructions with consensual branches. With predication, the underlying hardware does not need to rely on complicated hardware structures such as the divergence stack to manage control flow, and therefore can be simplified. Chapter 5 details the compiler foundation for predication, and presents our implementation and evaluation results.

3.2 Scalarization

Consider the simple FIR filter example shown in Figure 3.2a in which each thread computes one output element by convolving a range of `flen` input elements with an array of `flen` coefficients. The compiled SPMD code is shown in Figure 3.2b. Each thread maintains both a loop iteration count (`r7`) and a loop end count (`r3`) in registers and uses counter increment (line 21) and conditional branch instructions (line 25) to execute the loop. Thus, each thread executes a substantial amount of bookkeeping overhead in addition to the actual multiply-adds that perform useful work. Furthermore, most of the bookkeeping overhead is entirely redundant across threads. Each thread maintains identical loop counts, calculates the same branch conditions, replicates the same base addresses, and performs similar address math to retrieve data from structured arrays.

In addition to bookkeeping overheads, a SPMD program often has redundancy in the actual data operands accessed and computation performed by individual threads. The kernel code executed by each thread can be viewed as one iteration of an inner loop. A single-threaded encoding of the kernel often has "outer loop" data that could be accessed or computed once and then used many times. However in the SPMD program encoding, factoring out this redundant work is not as straightforward for a programmer or compiler. For example, in Figure 3.2b, each thread loads the same coefficients redundantly (lines 18, 23) and replicates their storage in private registers (`r5`).

```
1 fir_spmd<<<n>>>(...);
2
3 void fir_spmd(float* samples, float* coeffs, int flen, float* results)
4 {
5   float result = 0;
6   for (int i=0; i<flen; i++)
7     result += (coeffs[i] * samples[tid.x+i]);
8   results[tid.x] = result;
9 }
```

(a) SPMD Program

```
1 fir_spmd_conventional:
2 BB_1:
3   mov r9, r1;        # tid.x
4   ld.u64 r1, [4096]; # samples
5   ld.u64 r2, [4104]; # coeffs
6   ld.u32 r3, [4112]; # flen
7   ld.u64 r4, [4120]; # results
8   iset.s32.gt r5, r3, 0;
9   mov r6, 0;         # init result
10  @r5 bra BB_3;
11 BB_2:
12  bra BB_5;
13 BB_3:
14  shl r5, r9, 2;     # tidx * 4
15  iadd r1, r1, r5;   # sample addr gen
16  mov r7, 0;
17 BB_4:
18  ld.f32 r5, [r2];   # load coeff
19  ld.f32 r8, [r1];   # load sample
20  fma.f32 r6, r5, r8, r6; # fp mul add
21  iadd r7, r7, 1;    # loop bookkeeping
22  iadd r1, r1, 4;    # samples bookkeeping
23  iadd r2, r2, 4;    # coeffs bookkeeping
24  iset.s32.lt r5, r7, r3; # test loop break
25  @r5 bra BB_4;
26 BB_5:
27  shl r5, r9, 2;     # tidx * 4
28  iadd r4, r4, r5;   # result addr gen
29  st.f32 [r4], r6;   # store result
30  exit;
```

(b) Conventional Compiler Output

```
1 fir_spmd_scalarized:
2 BB_1:
3
4   @s ld.u64 s1, [4096];  # samples
5   @s ld.u64 s2, [4104];  # coeffs
6   @s ld.u32 s3, [4112];  # flen
7   @s ld.u64 s4, [4120];  # results
8   @s iset.s32.gt s5, s3, 0;
9      mov r6, 0;          # init result
10  @s @s5 bra BB_3;
11 BB_2:
12  @s bra BB_5;
13 BB_3:
14
15
16  @s mov s7, 0;
17 BB_4:
18  @s ld.f32 s5, [s2];    # load coeff
19     ldvec.f32 r8, [s1]; # load sample
20     fma.f32 r6, s5, r8, r6; # fp mul add
21  @s iadd s7, s7, 1;     # loop bookkeeping
22  @s iadd s1, s1, 4;     # samples bookkeeping
23  @s iadd s2, s2, 4;     # coeffs bookkeeping
24  @s iset.s32.lt s5, s7, s3; # test loop break
25  @s @s5 bra BB_4;
26 BB_5:
27
28
29     stvec.f32 [s4], r6; # store result
30     exit;
```

(c) Scalarizing Compiler Output

Figure 3.2: Simplified FIR Filter Code Example – (a) kernel code, (b) conventional compiler output, (c) scalarizing compiler output. Assume the number of threads launched (`n`) is a multiple of the warp size for simplicity. In the scalarized code, register specifiers which begin with `s` are scalar registers and `@s` is used to annotate scalar instructions. Register numbers are preserved between the conventional code and the scalarized code for clarity.

As another example shown in Figure 2.1c, a straightforward SPMD coding of matrix-multiply has each thread compute the dot-product of a shared vector (a row of the first matrix) with a private vector (a column of the second matrix). In this formulation, the load operations of the shared vector (`A[tid.y][k]`) are redundant across all threads.

Redundancy across threads is one of the key inefficiencies that a scalarizing compiler targets. Figure 3.2c shows the scalarized version of the same program. We describe the compiler algorithm required to generate this code in Chapter 4.

Our scalarization algorithm statically maps replicated operands to shared scalar registers. If we consider a single 32-thread warp executing the example in Figure 3.2, the conventionally compiled code would use 9 registers per thread. The scalarized code in comparison uses 2 private registers per thread and 6 shared registers per warp, 76% fewer registers per warp (70 vs. 288). In terms of dynamic register operands accessed, the conventionally compiled code reads 11 operands and writes 7 operands per thread per loop iteration. The scalarized code in comparison reads 2 private and 9 scalar operands per iteration and writes 2 private and 5 scalar operands per iteration. Since the scalar reads and writes only need to be performed once per warp, a 32-thread warp would read 79% fewer source operands (73 vs. 352) and write 69% fewer destination operands in total (69 vs. 224).

The scalarizing compiler algorithm also converts redundant instructions to scalar instructions. As described above, while conventional SIMT architectures factor out instruction fetch overheads across a warp, each thread still executes each operation. In Figure 3.2, the conventionally compiled code executes 8 operations per thread per loop iteration. The scalarized code executes only 7 scalar (including `ldvec` and `stvec`) operations and 1 regular thread operation per iteration. Since the scalar operations only execute once per warp, a 32-thread warp would execute 85% fewer operations with the scalarized code (39 vs. 256).

The scalarizing compiler algorithm also generates vector loads and stores for the input and output data that is accessed with unit-stride addressing across threads. These accesses are *coalesced statically* by the compiler, eliminating the need for dynamic coalescing. In the conventionally compiled code, a total of 64 unique addresses are generated per warp per loop iteration, compared to only 2 addresses per warp for the scalarized code.

In Chapter 4, we detail our compiler algorithm that scalarizes both thread registers and instructions, such that there is only one per warp (or wavefront) instead of one per thread. Our compiler uses two interlinked analyses to enable scalarization. The first, *convergence analysis* statically determines program points where the threads in a warp are guaranteed to be converged (i.e. no thread is following a divergent control-flow path). Convergence analysis is critical for scalarization, since the compiler can only scalarize regions that it can prove to be convergent. The second is *variance analysis*, which statically determines which program variables are guaranteed to have the same (or thread-invariant) value across the threads in a warp. In Section 4.1, we construct an intuitive argument for something that was not immediately apparent when we began this work: convergence and variance information can be usefully analyzed together in the same pass. In fact the two are inseparable in our implementation. We present an algorithm that iteratively analyzes and propagates convergence and variance information over a kernel's control dependence graph (CDG) [43, 133, 89]. Scalarization then uses this analysis to convert private thread registers into scalar registers

shared across the threads in a warp, and also converts thread instructions into scalar instructions that execute one operation per warp instead of one operation per thread. Using *affine analysis*, our compiler also generates vector loads and stores when the threads in a warp access sequential (unit-stride) data in memory. For example, a single vector load instruction can fetch a word from memory on behalf of each of the threads in the warp, placing the result in the same-named private register belonging to each thread.

In Section 4.2, we implement these compiler algorithms in an NVIDIA production compiler, and run the compiled code on our in-house simulator to get a detailed breakdown of microarchitectural events. Section 4.3 characterizes 23 benchmarks and finds that the compiler is able to keep warps converged for 66% of the total thread execution time on average. When augmented with simple *dynamic convergence preservation*, convergent execution can be maintained for up to 97% of total execution time. Scalarization reduces thread register usage by 20–33% on average depending on warp size, making it possible to either support more threads or reduce the register file size. Furthermore, 24–31% of dynamic instruction operands are scalars. On average our compiler scalarizes 23–29% of dynamically dispatched instructions, reduces memory address generation counts by 37–47%, and eliminates data access counts by 30–38%. These savings can provide proportional energy and performance gains.

Section 4.4 describes how compiler convergence analysis and scalarization are generally applicable to a variety of data-parallel architectures. We also discuss promising areas of future work in Section 4.5.

3.3 Predication

The SPMD programming model allows programmers to write arbitrary complex control flow. Figure 3.3 shows a code example with nested if-statements, a virtual function call, and a goto statement. A vectorizing compiler on a vector machine can always give up and run the complex control flow shown in Figure 3.3a on the scalar control processor. On the other hand, supporting complex control flow shown in Figure 3.3b for SPMD compilers is a functional requirement rather than an optional performance optimization, as the programming model is explicitly parallel—there is no equivalent fallback that allows running it sequentially on the scalar control processor. This is one of the reasons why GPU architectures end up implementing complicated divergence management schemes in hardware.

The overheads of the hardware divergence management scheme is the other key inefficiency that a scalarizing compiler targets. We describe the compiler algorithm required to map the complex control flow found in SPMD programs down to simple vector predication, therefore simplifying the underlying hardware in Chapter 5.

NVIDIA GPUs support the SPMD model directly in hardware (see Section 2.3 for more details) with a thread-level hardware ISA that includes thread-level branch instructions [81, 45]. This approach allows the compiler to use a fairly conventional thread compilation model, while pushing most of the divergence management burden onto hardware. Threads are grouped into warps, and when threads in a warp branch in different directions, the hardware chooses one path to continue

```
1  void complex_ctrlflow(...)
2  {
3    for (int i = 0; i < 32; ++i) {
4      if (a[i] < b[i]) {
5        if (f[i]) {
6          c[i]->vfunc();
7          goto SKIP;
8        }
9        d[i] = op;
10     }
11 SKIP:
12   }
13 }
```

(a) Vectorizable Loop

```
1  complex_ctrlflow_spmd<<<32>>>(...);
2
3  void complex_ctrlflow_spmd(...)
4  {
5    if (a[tid.x] < b[tid.x]) {
6      if (f[tid.x]) {
7        c[tid.x]->vfunc();
8        goto SKIP;
9      }
10     d[tid.x] = op;
11   }
12 SKIP:
13 }
```

(b) SPMD Model

Figure 3.3: Code Example with Complex Control Flow – (a) vectorizable loop, and (b) SPMD model. The loop body has a nested if-statement, a virtual function call, and a goto statement, which can execute in parallel.

executing while deferring the other path by pushing its program counter and thread mask onto a specialized divergence stack. Reconvergence is also managed through tokens pushed and popped on this stack by the compiler.

Vector-style architectures with a scalar+vector ISA employ a compiler-driven approach to divergence management [2, 101, 115, 112, 61, 60]. In this model, the vector unit cannot execute branch instructions. Instead, the compiler must explicitly use scalar branches to sequence through the various control paths for the elements or threads in a vector, while using vector predication to selectively enable or disable vector elements.

Although a wide range of software and hardware SPMD divergence management schemes are implemented in the field, software divergence management in particular has received relatively little attention from the academic research community. At first glance the topic may seem like a recasting of classic vectorization and predication issues, but the challenges are unique in the context of modern architecture for several reasons: (1) Unlike traditional vectorization, the parallelization of arbitrary thread programs is a functional requirement rather than an optional performance optimization; (2) The divergence management architecture must not only partially sequence all execution paths for correctness, but also reconverge threads from different execution paths for efficiency; (3) Traditional compiler algorithms for predication in serial processors are *thread-agnostic*, as they only need to consider optimizing the control flow for a single thread of execution. A data-parallel architecture on the other hand requires different *thread-aware* performance considerations; (4) GPUs and other multithreaded processors with a shared register pool are particularly sensitive to register pressure, as register count determines the number of threads that can execute concurrently. This constraint results in unique tradeoffs and optimization opportunities for the divergence management architecture. Finally (5), in SPMD programs, uniform control and

data operations can be *scalarized* to improve efficiency, a challenge that is related to but different than vectorization [74, 33, 61].

In Chapter 5, we further describe and analyze the design-space, tradeoffs, and unique challenges of SPMD divergence management architectures. In Section 5.1 we detail our thread-aware predication compiler algorithm for SPMD divergence management. We have developed optimizations including a *static branch-uniformity optimization* and a compiler-instigated *runtime branch-uniformity optimization* that eliminates unnecessary fetch and issue of predicate-false instructions. As described in Section 5.2, we use these algorithms to modify an NVIDIA production compiler to only use predication and uniform branches, eliminating all use of the hardware divergence stack.

In Section 5.3, we first characterize the control flow of a wide range of data-parallel applications. We then compare and analyze in detail the performance characteristics of software-based and hardware-based divergence management architectures on production GPU silicon. We describe conditions where software predication performs better, and other conditions where the hardware divergence stack performs better. We then discuss the tradeoffs in Section 5.4, and suggest promising areas of future work for further optimization of our software divergence management implementation in Section 5.5.

Chapter 4

Scalarization

This chapter discusses the details of the scalarization compiler algorithm. We assume a SPMD programming language that approximates NVIDIA's CUDA programming language and a SIMT architecture that closely resembles an NVIDIA GPU as the basis for the discussion. However, our compiler algorithms are applicable to other SPMD programming languages and data-parallel architectures with scalar execution resources such as the traditional vector machine. The compiler algorithm statically analyzes the program and scalarizes both thread registers and instructions, such that there is only one per warp instead of one per thread. We describe our compiler foundation in Section 4.1, discuss issues related to implementing our compiler algorithm within the NVIDIA production CUDA compiler in Section 4.2, and present evaluation results in Section 4.3. We then discuss how scalarization is generally applicable to other data-parallel architectures in Section 4.4, outline future research directions in Section 4.5, before summarizing in Section 4.6.

4.1 Compiler Foundation

To identify redundancy across multiple threads, the compiler must prove that a variable has a uniform value across all of the threads in a group. This process requires two key analyses. First, *convergence analysis* proves that the threads are in a converged state, meaning that all of the threads in the group are in the same point in the control-flow graph at the same time. This analysis builds on the CUDA kernel invocation model in which threads are launched in an initial convergent state. It also assumes convergence at `__syncthreads()` (i.e. barrier synchronization) calls, which are in effect programmer supplied assertions that threads are converged.

Second, *variance analysis* determines which variables in the converged threads have the same (uniform) value across all threads. This analysis builds on the semantics that kernel function call arguments are thread-invariant. Variance across threads originates with use of thread indices (e.g. `tid.x` in CUDA) and with volatile and atomic memory accesses. Our compiler uses data-flow and control-dependence analysis to determine which variables are not dependent on thread-specific values. Such variables can be converted safely from per-thread variables to per-warp scalar variables.

We implement the algorithms in the context of a production CUDA compiler, based on the LLVM infrastructure [69]. Our compiler algorithms are agnostic to the divergent execution models described in Section 2.3, and are generally applicable to data-parallel architectures with scalar execution resources.

Convergence Analysis

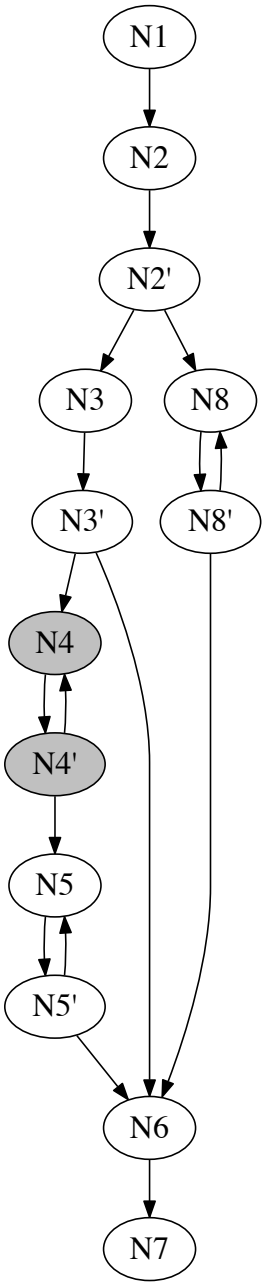
A program point is considered convergent if and only if a thread-group barrier placed at that point can never fail. This property implies that either all threads in the group will arrive at the barrier, or none of the threads will. Note that reconvergence points found by an immediate post-dominator scheme may not be considered convergent, since our definition of convergent implies that all threads are fully converged rather than a subset being partially converged. Convergence may be defined with respect to a particular group size such as CTAs or warps.

To perform convergence analysis, we leverage two data structures common to compilers. First, the control flow graph (CFG) represents the program as a graph of basic blocks (BBs) connected via control flow (branch, jump) edges [106]. Instructions unrelated to control are encapsulated within the basic blocks. Figure 4.1a shows an example CFG containing conditional branch points, loops, and merges. Second, we leverage a standard global data-flow representation such as static single assignment form (SSA) [34] and the control dependence graph (CDG) [43, 133] to identify basic blocks that are obviously convergent and determine a starting point for convergence analysis. Ferrante et al. [43] define control dependence as follows:

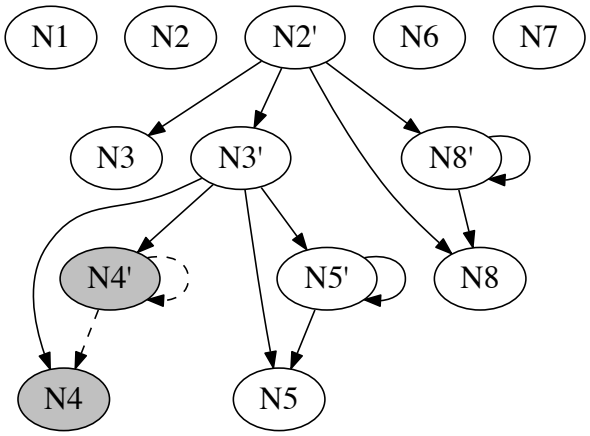
> **Definition:** If X and Y are basic blocks in a CFG, Y is control dependent on X (written $X \prec Y$) *iff*
>
> 1. there exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and
> 2. X is not post-dominated by Y.

Figure 4.1b shows the control-dependence relations in the CFG from Figure 4.1a.

The simplest approach to convergence analysis is to use the control flow structure of the kernel. Entry and exit blocks of a single-entry-single-exit (SESE) region have the same convergence properties. If the entry of an SESE region R is convergent then so is its exit. We then use the notion of regions and its characterization as described in [19], where two blocks of a CFG are in the same region if both nodes have identical control-dependence predecessors. Such nodes are termed *control-equivalent*. Since all threads of a warp (and a thread block) are convergent at the entry block to the kernel, all blocks that are control-equivalent to the entry block must be convergent since they execute under the same control condition. Because the entry block where all threads in the kernel start has no control-dependence predecessor, all basic blocks with no control dependence predecessors are marked as convergent. Using this simple notion of convergence, it is easy to see from Figure 4.1b that blocks $N1$, $N2$, $N2'$, $N6$, and $N7$ have no predecessors and therefore must be convergent.



(a) Control Flow Graph (CFG)



(b) Control Dependence Graph (CDG)

Figure 4.1: Example Control Flow and Dependence Graphs – (a) control flow graph, (b) control dependence graph.

Combined Convergence and Variance Analysis

Leveraging variance analysis [120], we extend the simple convergence analysis above to identify when basic blocks across the threads are guaranteed to depend on the same condition. The key insight is that a basic block is convergent if and only if it is transitively control dependent only on convergent blocks whose branch condition is thread-invariant (written $Tinv(block)$ below) and that the entry block of the kernel is always convergent. Any result of a thread-invariant instruction is uniform and is a candidate for scalarization.

$$\forall b\ b \prec x : convergent(b) \wedge Tinv(b) \Rightarrow convergent(x)$$

Alternatively, a basic block is divergent if it is transitively control dependent on a divergent block or it is transitively control dependent on a block with a thread-variant branch condition (written $Tvariant(block)$).

$$\exists b\ b \prec x : divergent(b) \vee Tvariant(b) \Rightarrow divergent(x)$$

Our algorithm exploits the latter characterization to mark blocks as divergent after initially assuming, optimistically, that all blocks are convergent. This approach fits well with our combined variance and convergence analysis which starts with optimistic assumptions about thread-variance.

Figure 4.2 describes our optimistic algorithm for variance and convergence analysis. The first step performs initializations as follows (Figure 4.2a):

1. Optimistically mark every basic block of the kernel as convergent.
2. Optimistically mark every instruction as thread-invariant.
3. Initialize a worklist of instructions with those that read the thread id register, perform an atomic action on shared memory, or access volatile memory.

The worklist always consists of currently known thread-variant instructions and is seeded with those instructions that cannot be proven to be thread-invariant. The second step performs a fixed-point loop in which each step removes an instruction from the worklist and performs the following actions until the worklist is empty (Figure 4.2b):

1. Mark the chosen instruction, i, as thread-variant, and
2. Add every thread-invariant data-flow successor instruction of i in the SSA graph to the worklist.
3. If instruction i is a conditional branch instruction, propagate divergence to all convergent blocks that are iteratively control dependent on i but do not contain a barrier instruction. Add every instruction in blocks that are newly marked as divergent to the worklist.

When the algorithm terminates, any blocks that are marked convergent must be so; and any instructions not visited and marked as thread invariant must be so as well.

```
worklist ← ∅
for bb ∈ blocks(kernel) do
    Conv(bb) ← True
    for instr ∈ instructions(bb) do
        Invariant(instr) ← True
        if instr reads thread id then
            worklist ← worklist ∪ {instr}
        end if
        if instr is an atomic instruction then
            worklist ← worklist ∪ {instr}
        end if
        if instr accesses volatile memory then
            worklist ← worklist ∪ {instr}
        end if
    end for
end for
```

(a) Initialization.

```
while worklist ≠ ∅ do
    instr ← POP(worklist)
    Invariant(instr) ← False
    for s ∈ DataFlowSucc(instr) do
        if Invariant(s) = True then
            worklist ← worklist ∪ {s}
        end if
    end for
    if instr is a conditional branch instruction then
        for bb ∈ IteratedControlDependenceSucc(instr) do
            if bb doesn't have a __syncthreads() call then
                if Conv(bb) = True then
                    Conv(bb) ← False
                    for i ∈ instructions(bb) do
                        worklist ← worklist ∪ {i}
                    end for
                end if
            end if
        end for
    end if
end while
```

(b) Analysis and propagation.

Figure 4.2: Combined Convergence and Variance Analysis – (a) initialization, and (b) analysis and propagation.

Analysis Example

To illustrate the algorithm, we use the flowgraph in Figure 4.1a. This example assumes that the branch condition in the basic block $N4'$ is thread-variant. The corresponding control dependencies are reflected in Figure 4.1b with dotted lines. Note that the $IteratedControlDependenceSucc$ of instructions in $N4'$ is $\{N4', N4\}$. The algorithm in Figure 4.2 will propagate divergence to the targets of these control dependencies transitively, illustrated in Figure 4.1b by marking dark divergent blocks $N4'$ and $N4$.

After the algorithm terminates, all the blocks which are not marked as divergent (not colored in Figure 4.1) are convergent. Block $N5$ is inferred as convergent simply because N5 is control independent of $N4'$ (in the control dependence graph), which means that all diverged threads must pass through $N5$.

Convergence of Warps that Exit Early

In many CUDA applications, threads in a CTA may exit early based on tests that check for the thread index. In the following, all threads with `tid.x` greater than 3 return and wait at the kernel exit for the rest of the warp to arrive.

```
kernel_spmd<<<n>>>(...);

void kernel_spmd(...)
{
  if (tid.x <= 3) {
    S1;
  }
}
```

The compiler can safely assume that S1 is convergent since the remaining threads are at the exit, and any scalar registers which are otherwise holding thread-invariant values are safe to initialize in statement S1. We extended our convergence analysis algorithm to not propagate divergence information across control-dependent successors of conditionals if the exit block of the kernel is control dependent on the variant condition.

Affine Analysis

Affine analysis is used to determine if thread-variant address operands of load and store instructions can be converted to vector load and store instructions. Vector load and store instructions access memory with a unit-stride address across successive threads. For example, given a load instruction `ld.type Rx,[addr]`, the compiler leverages both variance and scalar evolution analyses to determine if `addr` can be expressed as a simple linear expression of the following form `base + bitwidth(type) * tid.x`. Such load instructions can be transformed into a vector load

instruction `ldvec.type Rx,[base]`, in which register Rx in each thread of the warp is written with the respective value in the loaded vector corresponding to the thread's index.

Related Work

Our convergence analysis is based on the variance analysis described by Stratton et al. [120]. Their work identifies data accesses that are thread-invariant or will give the same value across the threads of a CTA. Their basic variance analysis was used to optimize CUDA programs when compiled to multicore CPUs. We extend their basic variance analysis algorithm to track not just thread-variant data, but also control divergence. We make optimistic assumptions about convergence and thread-invariant data and then track thread-variant information and divergent information together.

Coutinho et al. [33] describe what they call *divergence analysis*, which is also an extension of the approach of [120]. Their analysis finds divergent values, by first converting SSA information into gated single assignment [98], and then replacing control-flow merges with a predicate select operator. Their end algorithm is relatively simple because it can use data-flow analysis to propagate divergence information, but it requires a change of representation. However their divergent values are similar to thread-variant values as described in [120].

Collange [31] presents work with goals similar to ours, but uses an approach like that described in [33]. Collange does not use a gated representation but instead performs a symbolic analysis on a lattice of tags, which encodes and tracks alignment of various instruction operands. Coutinho et al. [33] and Collange [31] do not perform convergence analysis, which is important for exploiting scalar code generation.

Karrenberg and Hack [61] describe an analysis based on a data-flow lattice approach which is similar to our affine analysis. However, their analysis is geared towards vectorization, rather than scalarization. Also, their analysis does not use control dependence information, which is useful in our case to perform convergence analysis.

The ISPC language includes explicit uniform data types that allow a program to indicate scalar values in source code [103]. While this approach may be well matched to tightly coupled SIMD architectures, our approach relieves the programmer from this burden and uses the compiler to discover uniform values that a programmer may not be able to specify. Furthermore ISPC does not have any explicit notion of convergence.

Kerr et al. [63] implement a thread-invariant expression elimination pass, also based on [120]. The focus of their optimization pass is different than ours; they use common subexpression elimination on invariants after vectorization, whereas we allocate invariants to scalar register.

4.2 Implementation

We have implemented our compiler algorithms described in the previous section in a production CUDA LLVM compiler. We have modified the backend to target our own in-house simulator, since the actual GPU does not have any scalar execution resources.

The modified CUDA LLVM compiler first generates PTX instructions annotated with convergence information for each basic block, as well as variance and affine information for all registers. The PTX source code is processed by our backend compiler to target the RISC-like machine ISA shown in Figure 3.2. The convergence, variance, and affine information is used by the backend compiler to map invariant values to scalar registers, and to mark redundant instructions as scalar instructions. Instruction scheduling and register allocation are also performed in the backend.

We run the compiled code on our in-house simulator to get a detailed breakdown of instructions issued, operations executed, register reads and writes, memory address counts and data access counts. Our simulator runs one kernel (i.e. one grid invocation) at a time. We execute PTX source code on Ocelot [36] to obtain reference memory dumps before and after each kernel launch. The initial memory dump is used to populate the initial memory state of the simulator, and the post-kernel launch memory dump is used to verify the kernel execution. Each benchmark's composite kernel runs are summed together for all results presented in this chapter.

4.3 Evaluation

We evaluate our compiler using CUDA benchmarks from Rodinia [30] and Parboil [121]. Benchmarks in these two suites cover compute-intensive scientific domains including bioinformatics, image processing, medical imaging, graph algorithms, data mining, physical simulation, and pattern recognition. We reduced the input dataset sizes in some cases to make simulation time manageable. We also modified the source code to change texture references into global memory references, since our target abstract architecture lacks texture caches.

Convergence Analysis Results

The quality of convergence analysis is critical for scalarization, as the compiler can only scalarize regions that it can prove are convergent. Convergence analysis is also important for managing reconvergence in a stackless SIMT architecture later discussed in Section 4.5. Figure 4.3 shows the effectiveness of our compiler analyses. The benchmarks on the X-axis are sorted left-to-right in decreasing effectiveness of compiler convergence analysis. The Y-axis represents the total instructions dynamically dispatched for execution by the microarchitecture. For each benchmark, the left bar shows the breakdown of instructions proven convergent by different variants of the compiler. The right bar shows the fraction of instructions that could be proven convergent by a dynamic oracle. The fraction of the bars labeled *Diverged* could not be proven convergent at compile time.

Simple convergence analysis, which only looks at the shape of the control flow graph, can only keep thread execution convergent 32% of the time on average. By coupling convergence analysis with variance analysis, the compiler is able to determine cases where branch conditions are invariant across threads, increasing convergent execution to 57%. The exit optimization further increases convergence to 66%. We also show results for dynamic convergence preservation, a simple hardware mechanism that prevents warps from diverging when threads dynamically branch in the same direction (note that this mechanism does not imply a hardware divergence stack). The

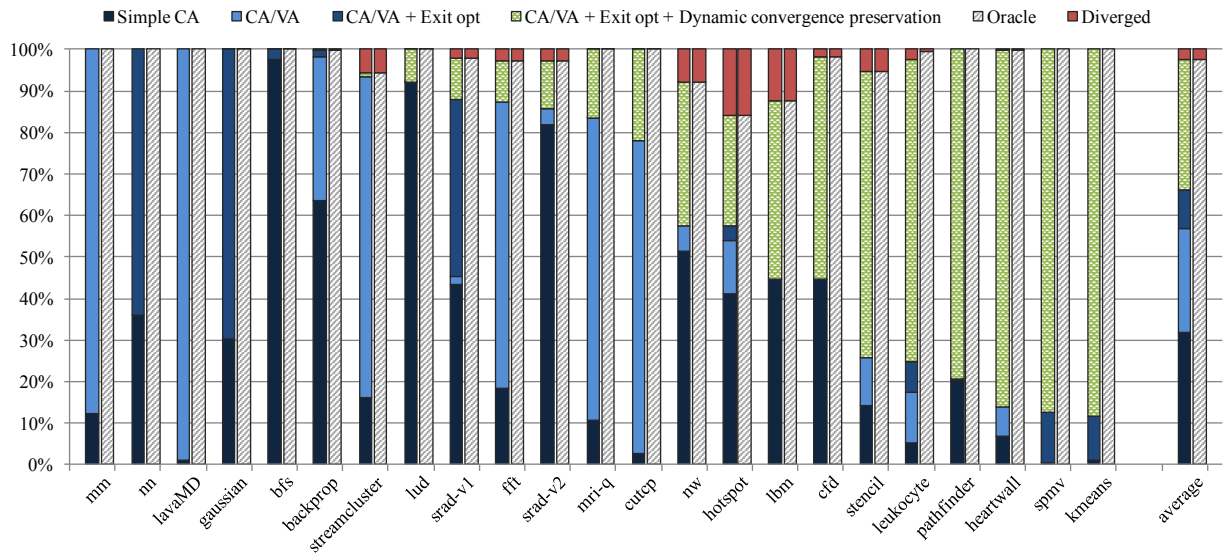


Figure 4.3: Effectiveness of Convergence Analysis with Warp Size of 4 Threads – The X-axis is sorted by the effectiveness of convergence analysis done by the compiler. CA=Convergence Analysis, VA=Variance Analysis.

dynamic convergence preservation statistics highlight the limits of the static compiler analysis by capturing the dynamic convergence behavior of the warps. Figure 4.3 shows that this hardware and software approach improves convergent execution to 97% on average with warp size of 4 threads. With a wider warp size, convergence identified by the compiler will remain the same, while convergence imposed by dynamic convergence preservation will tend to decrease because the likelihood of a warp to diverge increases with more threads.

We also performed a limit study where we use oracle knowledge to maximize convergence. Optimal alignment of convergent blocks and instructions can be reduced to the Multiple Longest Common Sequence (MLCS) problem [85]. We reduced the complexity of our MLCS implementation by leveraging the compiler's convergence analysis and only analyzing divergent regions. Oracle convergence analysis based on the dynamic instruction trace shows that the best possible schedule can keep thread execution convergent 97% of the time on average. Overall, oracular analysis is no better than the combination of our convergence/variance analysis with dynamic convergence preservation.

Scalarization Results

Figure 4.4a shows the breakdown of static instructions into scalarized and unscalarized (labeled *thread*), normalized to a baseline without scalarization (left bar in each group). On average, the compiler scalarizes 29% of static instructions. The total static instruction count increases by 2%, primarily due to instructions generated to calculate the base address of warp-sequential memory operations. Figure 4.4b shows the breakdown of register accesses into the same categories, relative

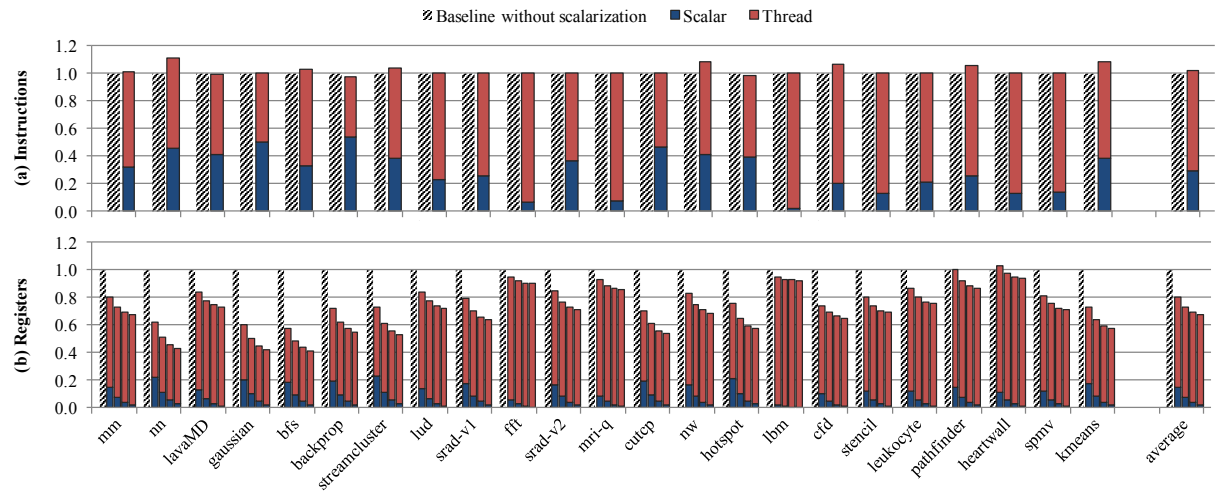


Figure 4.4: Static Scalarization Metrics – (a) instructions, (b) registers. Each group shows results for warp sizes of 4, 8, 16, and 32.

to the same baseline (left bar in each group), for warp sizes of 4, 8, 16, and 32. Scalarization reduces total register requirements by 20% with a warp size of 4, and up to 33% with a warp size of 32, as wider warp sizes amortize more redundancy from scalarized operations.

Figure 4.5 shows how scalarization affects dynamic instruction, register, and memory activity counts. In each graph, the left bar in each benchmark group is the baseline without scalarization, while the remaining bars show warp widths of 4, 8, 16, and 32 threads. We differentiate between the number of issued instructions (Figure 4.5a) and executed thread operations (Figure 4.5b), and each of these counts are broken down by source: scalars, statically converged warps, warps converged through dynamic convergence preservation, or diverged threads. Note that only one instruction is *issued* for all the threads in a warp when it is converged. *Operations executed* counts the total number of individual thread operations, regardless of convergence. Scalarization is subject to the effectiveness of convergence analysis (Figure 4.3), which is why benchmarks towards the left of Figure 4.5 have more scalar and statically converged warp instructions, and the benchmarks towards the right have more dynamically converged warp instructions.

In general, wider warp sizes decrease the instruction and operation counts because an instruction only issues once for all threads in a converged warp, and because scalar operations are only issued and executed once per warp. However, the number of diverged thread instructions increases with wider warps mainly because the likelihood of divergence increases with larger groupings of threads. This effect is apparent for `nw`, `hotspot`, `lbm`, and `stencil`. Still, since each converged warp instruction represents 4–32$\times$ more operations than a diverged thread instruction (following the warp size), the total operation count is always lower with scalarization than without. The savings range from 23–29% depending on warp size.

As expected, the other statistics of register read counts, register write counts, memory address lookups, and memory data accesses (Figures 4.5c-f) have roughly the same shape as the executed

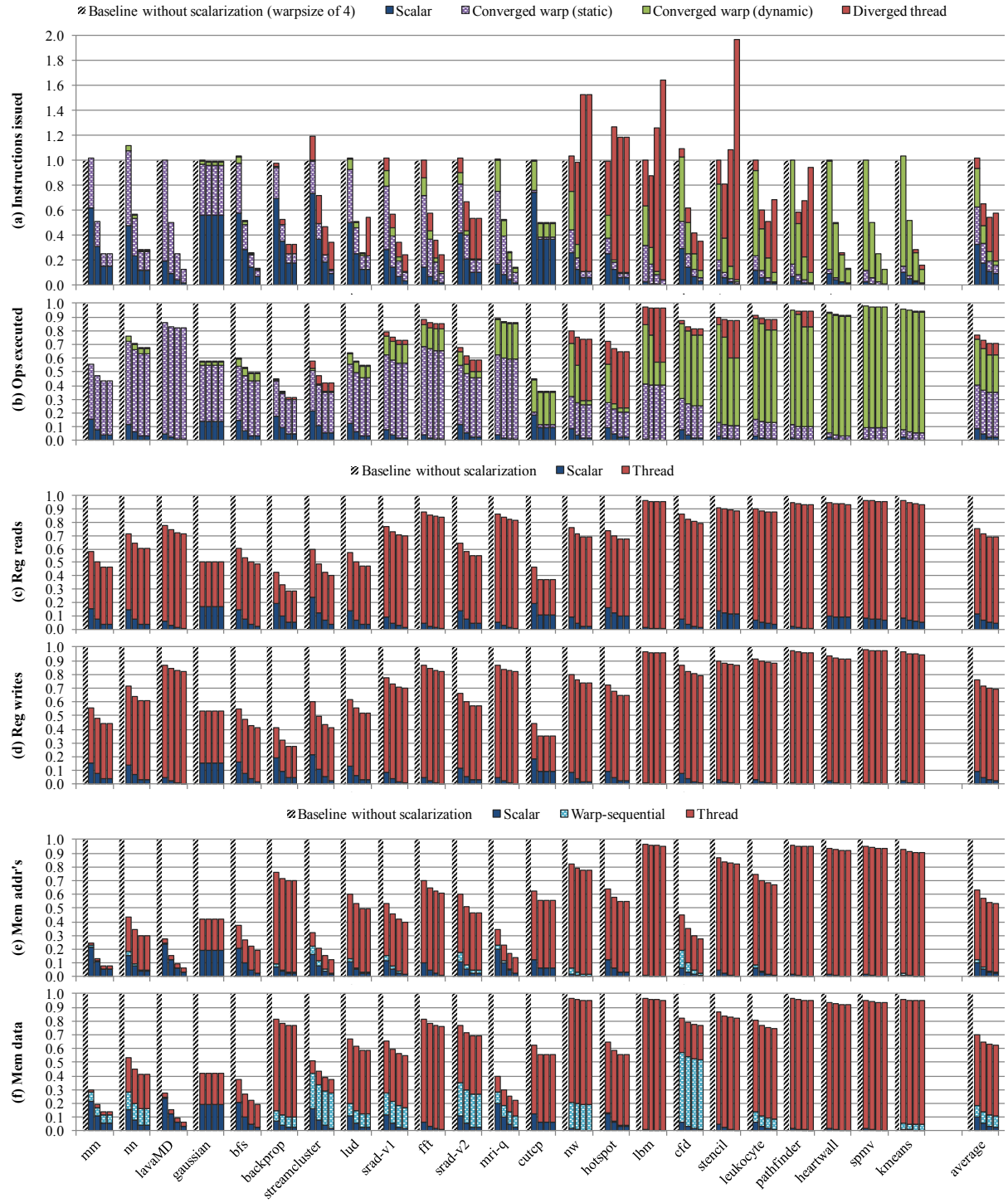


Figure 4.5: Dynamic Scalarization Metrics – (a) instructions issued, (b) operations executed, (c) register reads, (d) register writes, (e) memory addresses, (f) memory data. Each group shows results for warp sizes of 4, 8, 16, and 32.

thread operations. With a warp size of 4, register reads and writes are reduced by 24%, memory address counts going to the innermost cache are reduced by 37%, and the number of memory data elements accessed is reduced by 30%. With wider warp sizes, scalar register accesses and scalar memory operations are more effectively amortized, and warp-sequential memory address activity similarly decreases (though the data counts stay constant). With a warp size of 32, register reads and writes are reduced by 31%, memory address counts are reduced by 47%, and data access counts go down by 38%.

4.4 Discussion

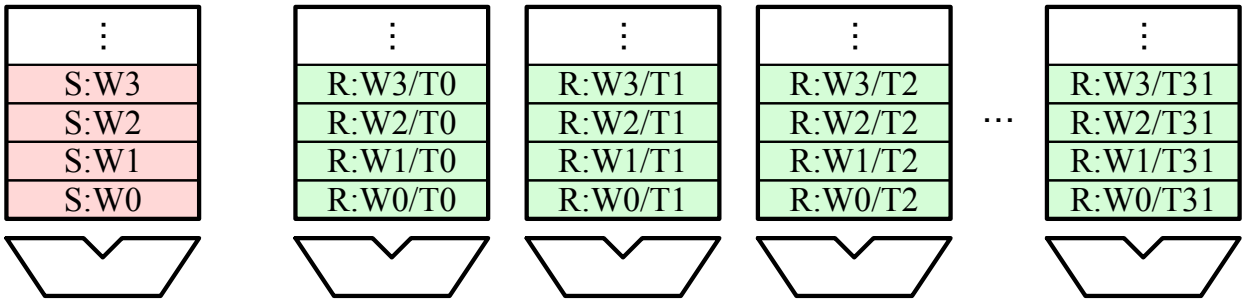
Compile-time convergence analysis and scalarization can improve efficiency and performance in various contexts. The algorithms in this thesis apply to existing processors, and they may also enable new hardware microarchitectures.

Scalarization in Data-Parallel Microarchitectures

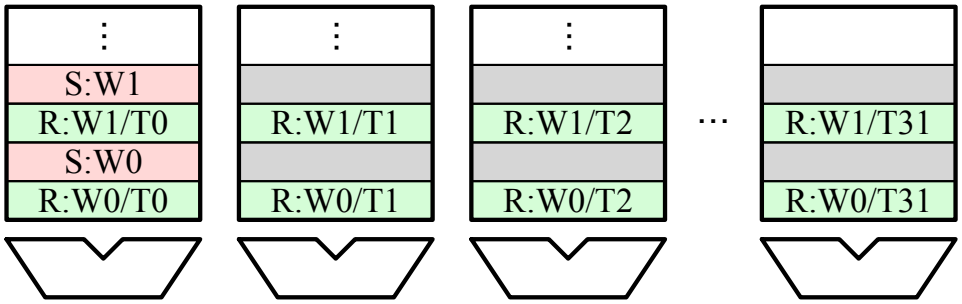
Figure 4.6 shows a range of data-parallel microarchitectures extended with scalarization support. The same instruction-set architecture that we target in Section 4.1 applies to all of these microarchitectures, as do the microarchitecture-neutral results in Section 4.3. Figure 4.6a shows a traditional vector microarchitecture extended with a scalar unit on the left. A warp is mapped across the vector lanes with one thread per lane. In contrast to the wide vector unit, the scalar unit has a 1-wide datapath with a scalar register file and resources to execute scalar instructions. Scalarization reduces overall register file capacity by eliminating redundant operand storage, or alternatively allows a register file of a given size to map more threads. Scalar instructions improve performance as they allow regular vector instructions to execute in parallel, and they reduce energy by eliminating replicated work. Figure 4.6b shows an alternative microarchitecture which executes scalar instructions on a single lane instead of a separate unit, thus avoiding the area overhead when scalarization is not used. This architecture still reduces energy by only activating one lane when executing scalar instructions, but it would not reduce register pressure or improve performance.

Scalarization may also help enable new microarchitectures with better divergent-thread performance. In a *spatial* vector microarchitecture, divergence can be a performance bottleneck since throughput and efficiency are halved each time the threads in a warp diverge [18]. With complete divergence, only one of the warp's threads executes instructions at a time. A potential solution is a *temporal* vector microarchitecture, as shown in Figure 4.6c. In temporal vector lanes fetch and execute instructions independently. A warp is mapped to a single lane, and the threads in a converged warp dispatch an instruction one after the other over successive cycles. In this way the temporal vector lane amortizes instruction overheads similar to a 1-lane vector machine [109]. When threads are diverged, on the other hand, instructions simply dispatch for a single cycle and the independent lanes essentially operate as a traditional multithreaded MIMD processor.

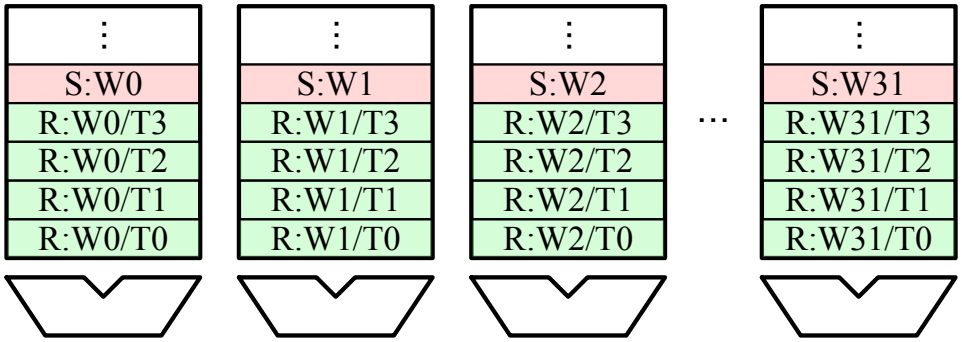
A temporal vector microarchitecture is a particularly good match for efficient scalarization. As shown in Figure 4.6c, the register file mappings are configured to allocate a set of scalar registers



(a) Spatial Vector with Scalar Unit



(b) Spatial Vector with Scalar Units in Lane 0



(c) Temporal Vector with Scalar Units in Each Lane

Figure 4.6: Data-Parallel Microarchitectures with Scalarization Support – (a) spatial vector with scalar unit, (b) spatial vector with scalar units in lane 0, and (c) temporal vector with scalar units in each lane.

for each warp. No physical partitioning of the register file is necessary (as in Figure 4.6a), and no register file slots are wasted (as in Figure 4.6b). Regular thread instructions can directly source operands from scalar registers instead of their usual private registers. The register file is only read once as the first thread dispatches, and the scalar operand is then held in a pipeline register as other threads dispatch. When a scalar instruction executes on a temporal vector architecture, it simply dispatches once for the warp instead once per thread. Note that there will be no additional ports added to the register file or the instruction cache if the microarchitecture continues to issue one instruction per cycle. Scalarization on a temporal vector architecture will improve both performance and energy, while requiring only minimal hardware modification and no separate scalar execution resources.

4.5 Future Research Directions

This section briefly describes some possible directions for future improvements with respect to scalarization.

Partial Scalarization. The compiler algorithm described in Section 4.1 only looks for uniform values across threads that are fully converged. This constraint makes the algorithm agnostic to how the underlying data-parallel architecture supports control flow—the only requirement is to keep all threads converged in regions where the compiler can statically prove so. We can augment the compiler algorithm with specific knowledge on how the underlying hardware handles control flow to enable *partial scalarization*, in which thread operations and register accesses across subsets of warps are scalarized. Assuming convergence in warps that exit early is a specific example of partial scalarization.

Dynamic Scalarization. Currently the compiler algorithm described in Section 4.1 can only scalarize parts that the compiler can statically prove thread-invariant. However, during runtime the thread operations or register accesses may happen to be uniform across a warp. We can architect a machine in which the operands are dynamically checked for their uniformity, and the resulting information is taken into consideration to reduce certain microarchitecture events. Kim et al. [64] takes advantage of not only the uniform values across a warp but also affine values to optimize the execution of arithmetic, branch, and memory instructions during runtime. Gilani et al. [47] adds a comparison stage to the execution units in order to detect uniform values, and when detected, the uniform value is sent to the scalar processor with separate execution resources.

Stackless SIMT. The divergence stacks used in current GPUs (Section 2.3) have several drawbacks. First, as mentioned above, execution efficiency drops each time the threads in a warp diverge. The warp itself executes all of its composite thread code paths serially, so no parallelism is possible between diverged threads in the same warp. Secondly, in current GPU compilers, achieving correct execution in the presence of unstructured control flow is a major challenge [134]. Finally, the divergence stack creates a pitfall where the SPMD model can break down: since the

"threads" in a warp do not truly execute asynchronously. Threads can only synchronize at warp (or CTA) granularity, and threads in the same warp are only able to communicate when their execution is convergent [94].

These drawbacks of SIMT could all be addressed by mapping SPMD kernels to a future *stackless temporal-SIMT* architecture. Similar to MIMD, such an architecture would provision a hardware PC per thread and allow diverged threads to truly execute independently. Compiler convergence analysis is an important enabler for this form of stackless SIMT since, without a stack, hardware has no way to reconverge diverged warps. To enable *compiler-managed reconvergence*, the architecture can provide a `syncwarp` instruction that acts as a barrier for currently executing threads. The compiler simply places a `syncwarp` in blocks it identifies as convergent, and as long as all threads make progress, the warp will eventually reconverge at this instruction. However, placing a `syncwarp` in every convergent block incurs an overhead that is often not necessary. We can place these synchronization operations only at those convergent blocks that have at least one divergent predecessor, thus ensuring that scalar registers are only written in convergent blocks guarded by `syncwarp` instructions. In the example shown in Figure 4.1a, node $N5$ must be guarded by a `syncwarp`.

4.6 Scalarization Summary

This chapter presents new compiler algorithms for thread convergence and variable variance analysis that elides redundant instructions and register accesses in threaded code through a technique called scalarization. Our compiler algorithms are extremely effective at identifying convergent execution even in programs with arbitrary control flow, identifying two thirds of the instructions captured by a dynamic oracle. Simple hardware mechanisms can boost this to 100%. The compile-time analysis leads to a reduction in operations executed and register accesses of 23–31% depending on warp size.

We anticipate additional optimizations, such as scalarizing across subsets of warps and dynamic scalarization where the microarchitecture scalarizes instructions and operands without compiler guidance, will provide even greater benefits.

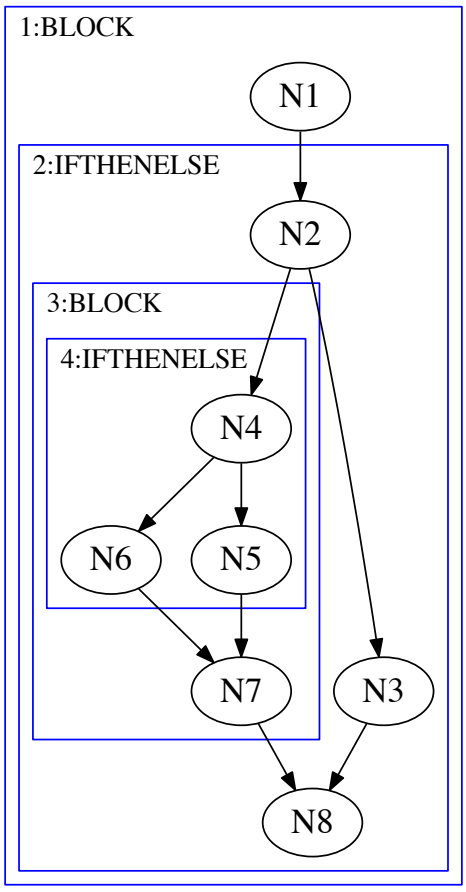
Chapter 5

Predication

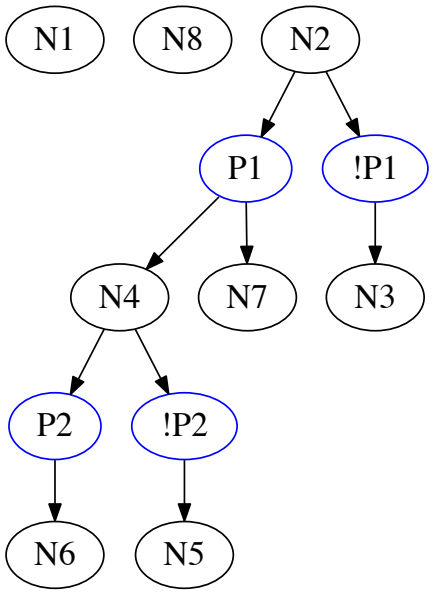
This chapter discusses the details of the predication compiler algorithm. Similar to the previous chapter, we assume a SPMD programming language that approximates NVIDIA's CUDA programming language and a SIMT architecture that closely resembles an NVIDIA GPU as the basis for the discussion. However, our compiler algorithm is applicable to other SPMD programming languages and data-parallel architectures that support vector predication and reduction operations on predicates. The compiler algorithm maps complex control flow found in SPMD programs down to predicated vector instructions with *consensual branches*, where the branch is only taken when all threads in a warp have the same predicate value. We describe our compiler foundation in Section 5.1, discuss issues related to implementing our compiler algorithm within the NVIDIA production CUDA compiler in Section 5.2, and present evaluation results in Section 5.3. We then discuss advantages of software divergence management in Section 5.4, outline future research directions on how to improve our software divergence management scheme in Section 5.5, before summarizing in Section 5.6.

5.1 Compiler Foundation

This section describes algorithms that allow a compiler to systematically map divergence present in SPMD programming models down to data-parallel hardware, unlike the heuristic algorithm used to opportunistically predicate control flow in the backend NVIDIA compiler. The proposed compiler algorithms remove all control flow by predicating and *linearizing* different execution paths. These algorithms are applicable to data-parallel architectures that can partially execute instructions and support consensual branches. For the discussion, without any loss of generality, we assume predication and consensual branches as the underlying mechanisms. Our general approach to predication is similar to if-conversion described by Mahlke et al. [84], which is a variant of the RK algorithm described in Park and Schlansker [101]. Because predicating compilers are well-studied and have been used effectively on a variety of platforms, this section focuses only on aspects of compilation most relevant to thread-aware predication. We also discuss optimizations to make predication more efficient.



(a) Control Flow Graph (CFG)

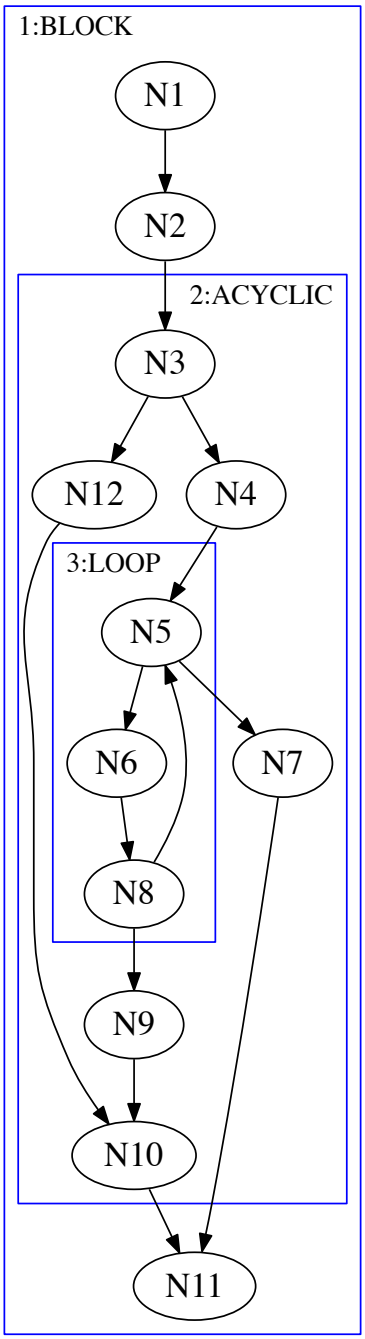


(b) Control Dependence Graph (CDG)

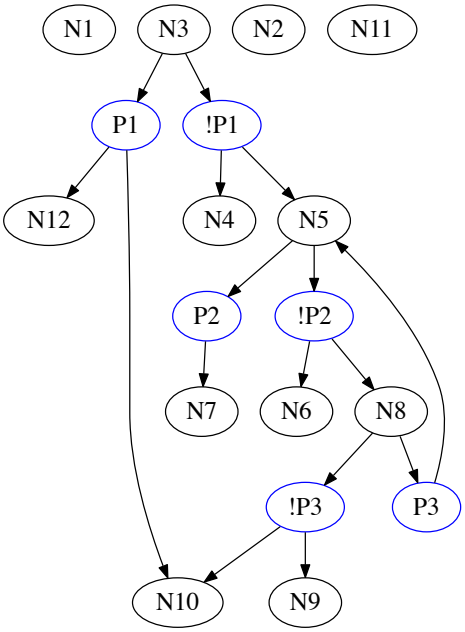
Figure 5.1: Thread-Aware Predication Code Example with Nested If-Then-Else Statements – (a) control flow graph, and (b) control dependence graph.

Thread-Aware Control Dependence Graphs

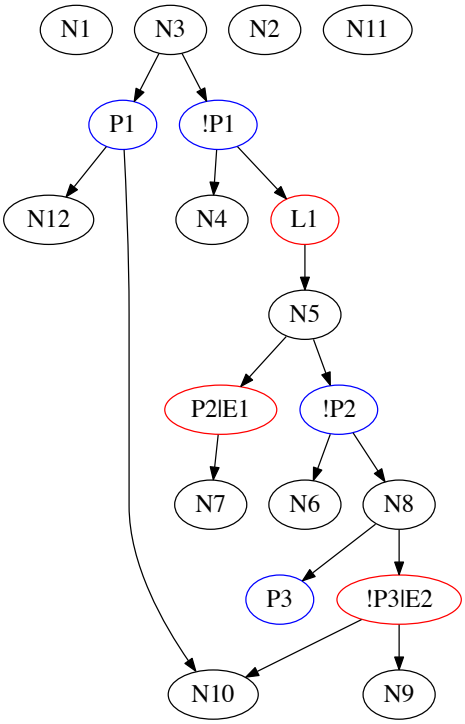
The control dependence graph (CDG) is the foundation upon which our predication algorithms rely [43, 133, 89]. A CDG relates nodes in a control flow graph (CFG) by their *control dependencies*. For example, Figure 5.1b shows a *labeled* CDG for the CFG in Figure 5.1a [133]. In our labeled CDG, we insert *predicate nodes* (sometimes called *region nodes*) between dependent basic blocks. The predicate nodes provide an explicit relationship between basic blocks and the predicates that guard their execution. For example, consider the if-then-else region $N4$, $N5$, and $N6$. The corresponding CDG has a predicate node $P2$ (and its negation $!P2$) controlling $N6$ and $N5$. As a practical matter, the compiler sets a predicate's value using the same test a branch would use.



(a) Control Flow Graph (CFG)



(b) Control Dependence Graph (CDG)



(c) Thread-aware Control Dependence Graph

Figure 5.2: Thread-Aware Predication Code Example with a Loop with Two Exits – (a) control flow graph, (b) control dependence graph, (c) modified control dependence graph with loop masks and exit masks.

Using the CDG, the compiler can simply trace a path from the entry node of a region to determine how to *guard* the execution of a basic block with predication. For instance, in 5.1b we see that $N5$'s instructions should only execute when $P1$ is true and $P2$ is false.

Figure 5.2b shows that a CDG's utility extends well beyond simple if-then-else regions, and can handle a variety of complex control flow such as the loop and unstructured control flow shown in the CFG in Figure 5.2a.

While the CDG as presented thus far applies to scalar processors, we can extend it to match the semantics of data-parallel architectures. With predication, when a group of threads execute the same loop simultaneously, the group must be sequenced until all the threads have exited the loop. We augment the CDG's basic structure with *loop masks* to track which threads are still actively executing the loop, *continue masks* to track which threads are in the current iteration of the loop, and *exit masks* to track which threads exited via specific loop exit blocks.

Figure 5.2c, which extends Figure 5.2b with these masks, serves as an example in the following description. In the example, $N4$ is the loop's *landing pad*, $N5$ is the *loop header*, and the exit paths are through predicates $P2$ and $!P3$. A *loop mask* represents the threads that are active in the loop body. The compiler initializes a loop mask in the loop's landing pad ($N4$) with a runtime value that represents the active threads at the loop's entrance. For our example, the compiler introduces a loop mask, $L1$, which guards the execution of the loop body. Exit masks, on the other hand, represent threads exiting the loop (through predicate nodes $P2$ and $!P3$ in our example). Exit masks are initialized to `false` in the loop's landing pad.

The compiler inserts instructions at the loop exits to keep the loop masks and exit masks up-to-date. For loop masks, the compiler inserts predicate `and` instructions to mask off threads that exit the loop. A consensual branch is added to all loop exits to check whether the loop mask is null (i.e., all threads are done executing the loop body). For exit masks, the compiler inserts predicate `or` instructions to aggregate the threads that have exited (*e.g.*, $P2|E1$ and $!P3|E2$).

While a *continue mask* is unnecessary for our example, we use them to optimize for the case where all threads in the current iteration execute a continue statement to move to the next iteration. The continue mask is added to the loop header block ($N5$), is initialized to the loop mask of the current iteration, and is iteratively `and`-ed at every continue and exit block with the negated mask of threads that leave the current loop iteration. A consensual branch is added to every continue block to jump to the loop header when the continue mask turns out to be null.

Because the resultant CDG is still valid, downstream predication algorithms can obliviously handle cyclic regions. Karrenberg and Hack [61] use similar loop masks and exit masks to vectorize loops. However, they do not use a CDG formulation to systematically generate predicates, and do not optimize for loop continues.

Static Branch-Uniformity Optimization

If the compiler can prove that a branch condition is *thread invariant*, meaning that all active threads simultaneously have the same value, the compiler can replace the branch with a consensual branch and forgo predicating the region. Consensual branches do not affect the hardware's divergence management as all active threads are guaranteed to branch in the same direction. This compile-

time *static branch-uniformity optimization* can avoid unnecessary work and also reduce register pressure. For example, in Figure 2.11a, if the compiler proves that `p0` is thread invariant, the compiler can safely remove the `p0` and `!p0` predicate guards for `op2` and `op3`. In more complex regions, removing the predicate guards can shorten the live range of the associated predicate, thereby reducing register pressure.

We use a modified version of *variance analysis* described in Chapter 4 to select thread-invariant predicates. A basic block is *convergent* if it is only control dependent on thread-invariant predicates. For convergent basic blocks, all threads that execute simultaneously will either enter with a full mask or not enter at all. Therefore, rather than predicating and linearizing convergent basic blocks, we omit the guard predicate and *preserve* the original control flow. We ignore the thread-invariant predicates in the CDG when generating guard predicates for other basic blocks. If the loop header is convergent, all threads will enter, execute, and exit the loop convergently; hence the CDG does not need to be transformed with loop, continue, and exit masks. Certain control edges must be preserved, otherwise the linearized basic block might execute incorrectly, as convergent basic blocks omit their guard predicates. The rule is to preserve outgoing edges of a convergent basic block if there is only one outgoing edge or when the branch condition of the convergent basic block is proven to be thread invariant.

We modified the variance analysis algorithm to work with the labeled CDG so that we could add an additional rule: if a basic block is controlled by a thread-variant predicate (i.e., proven to be non-convergent), mark all predicates controlling that basic block as thread variant. This constraint is added so that non-convergent basic blocks will have no preserved control edges coming in. For example, in Figure 5.2b, if predicate $P3$ is thread variant, then mark $P1$ as thread variant, as $N10$ is control dependent on both $P1$ and $!P3$. Otherwise, the compiler will insert a consensual branch at $N3$ since predicate $P1$ is invariant, and assuming the execution went down $N12$ and $N10$, there will be an uninitialized guard predicate representing threads from the $N9$–$N10$ control edge.

To see how this analysis works in practice, assume in Figures 5.1a/5.1b that the compiler can prove that $P1$ is thread invariant but cannot do the same for $P2$. As basic block $N2$ is convergent, the predicate generation algorithm can ignore predicate nodes $P1$ and $!P1$, and only consider predicate nodes $P2$ and $!P2$. As a result, basic blocks $N4$, $N7$, and $N3$ do not have guard predicates, while $N6$ and $N5$ are guarded by $P2$ and $!P2$ respectively. Control flow edges $N2$–$N4$ and $N2$–$N3$ are preserved as $N2$ is convergent and $P1$ is thread invariant. $N7$–$N8$ are also preserved as $N7$ is convergent and only has one outgoing edge. Similarly $N3$–$N8$ is preserved as $N3$ is convergent and only has one outgoing edge. As a result of this static branch-uniformity optimization, only $N4$–$N6$–$N5$–$N7$ are predicated. If all the predicates turn out to be thread invariant, all control flow will be preserved. On the other hand, if they are all thread variant, then all control flow will be predicated.

Runtime Branch-Uniformity Optimization

For branch conditions that the compiler cannot prove as thread invariant, the compiler can still optimize the control flow by inserting dynamic predicate uniformity tests that consensually branch

around whole regions when the active threads all agree. We refer to this as compiler-instigated *runtime branch-uniformity optimization*.

This optimization is guided by *structural analysis* to uncover control-flow structures of interest [111]. Structural analysis allows us to reconstruct control flow structure from a standard CFG. For example, Figure 5.1a overlays a structural analysis of a CFG with nested if-then-else structures. The structural analysis recursively discovers that blocks $N4$, $N5$, and $N6$ form an if-then-else region. This region is then compressed into an IFTHENELSE block and the algorithm repeats. Likewise, in Figure 5.2a structural analysis identifies a LOOP ($N5$, $N6$, and $N8$) and an ACYCLIC structure ($N3$, $N4$, $N7$, $N9$, $N10$, $N12$, and the loop).

We consider adding runtime checks to single-entry single-exit (SESE) substructures of IFTHEN-ELSE and IFTHEN flow structures. A simple heuristic algorithm decides to put a runtime uniformity check around the SESE region of the substructure when there are more than three instructions or more than two basic blocks in the SESE region. If the optimization is selected, the compiler adds a header block and a tail block around the SESE region. A consensual branch is added to the header block that branches to the tail block when the predicate guarding the region is uniform. One interesting ramification of this approach is that the inserted branches form scheduling barriers that constrain instruction scheduling.

As an example, assume in Figures 5.1a/5.1b that a runtime branch-uniformity check is added to substructures of 4:IFTHENELSE. A header block and a tail block are added around both $N6$ and $N5$. Consensual branches are added to both header blocks checking whether $P2$ and $!P2$ are false, respectively. If the respective predicates are false, instructions in $N6$ and $N5$ are neither fetched nor executed. As Figure 5.2a has no IFTHENELSE structures, no runtime uniformity checks are inserted.

Shin [112] describes a similar dynamic branch-uniformity optimization called BOSCC (branches-on-superword-condition-codes), but relies on predicate hierarchy graphs to nest regions that are covered by BOSCC to reduce runtime overhead of checking the uniformity of branch conditions. In contrast, we utilize the structural analysis to pick candidate SESE regions for runtime checks.

Linearizing the Control Flow

For predicated execution, basic block ordering is clearly important. Consider an if-then-else region for which the compiler has inserted runtime branch-uniformity checks. For cases in which the branches are not uniform, execution will fall through in a predicated fashion from the header, through *both* paths of control flow, and finally exit.

A simple approach to basic block ordering that topologically sorts the loop tree of the CFG as in [61] only works when *all* control flow will be predicated. As we have seen, this assumption is not valid for our solution, because our optimizations intentionally preserve some parts of the original control flow graph. To avoid inserting extraneous branches, which would defeat the purpose of our work, the basic blocks from a predicated region have to be placed contiguously. We refer to this placement problem as linearizing the control flow.

We again turn to structural analysis to achieve a linearized schedule. In addition to the IFTHEN and IFTHENELSE structures, our analysis also discovers BLOCK, LOOP, ACYCLIC, and OTHER

structures [111]. The OTHER structure is a catch-all category that represents regions we do not attempt to predicate and schedule. For these, we fall back to a low-performance escape hatch where we put sequencing code at the entry block and the exit block to sequence active threads through the structure one-by-one. Note that irreducible loops will be part of a single-entry single-exit OTHER structure. For the purposes of this thesis, the detailed form of the structures is not important. What is important is that the structures discovered in the analysis hierarchically group together basic blocks that need to be contiguous in a predicated execution model.

Once structural analysis reports the control-flow structures, we reverse the direction of all edges in the CFG, and then perform a depth-first post-order traversal from the exit node to generate a valid schedule. We reference the result of the structural analysis to pick which children to visit first to obtain a contiguous schedule of basic blocks from the same control-flow structure. The rule is to first pick children that are from the same innermost structure. To correctly schedule all structures in a loop, we remove backedges that connect loop tails to loop headers and make all edges from the outside point to the loop tail.

Karrenberg and Hack [60] describe a similar static branch-uniformity optimization to reduce register pressure while vectorizing OpenCL kernels for packed-SIMD units in x86 processors. Reducing register pressure is especially important on an x86 processor, as it has a limited number of scalar and vector registers. While their motivation is similar to ours, they use a different formulation. Uniform branches are identified with a dataflow lattice approach, while we use variance analysis that uses control dependence information to do so. To linearize basic blocks, they utilize a region analysis based on a depth-first search with post-dominator information to identify region exits, while we use structural analysis, which can identify irreducible regions more easily.

5.2 Implementation

As NVIDIA GPUs support both hardware-managed and software-managed divergence, we can compare these schemes by implementing our compiler algorithms in the production NVIDIA CUDA toolchain and running real workloads on existing hardware.

The CUDA compiler takes a CUDA program and translates it to native SASS instructions [93] through a two-step process. The CUDA LLVM compiler first takes a CUDA program and generates PTX instructions with virtual registers and simple branches to represent data and control dependencies. The `ptxas` backend compiler then generates native SASS instructions from the PTX code by allocating hardware registers, inserting instructions that sequence the divergence stack, and performing a very limited version of if-conversion with simple heuristics discussed in Section 2.3. When using our predication algorithms, our modified compiler disables passes that insert divergence stack instructions and perform if-conversion.

We implement our compiler algorithms described in Section 5.1 in the CUDA LLVM compiler. The LLVM compiler uses a static single assignment (SSA) based internal representation. SSA form is inherently incompatible with the conditional update semantics of predication. The production compiler toolchain in which we prototype these techniques is sufficiently rigid to preclude implementation of the predicate-aware techniques such as [38, 119]. Instead, to interop-

erate with LLVM's internal representation and built-in passes, during our predication pass we embed a throw-away instruction in each basic block to hold its guard predicate, ordering, and linearization information. The metadata held by the throw-away instruction survives various LLVM passes including the instruction DAG selection pass. We modify `ptxas` to accept the intermediate form with the throw-away instruction mapped to a pseudo-PTX instruction to deliver metadata needed for predication. The throw-away instruction withstands another set of optimization passes in `ptxas`, and is discarded in a late phase once the compiler predicates all instructions with the respective guard predicate and rewires the basic blocks with consensual branches to adhere to the ordering generated by our LLVM predication pass. The resulting binary can be executed on both Kepler and Fermi GPUs without modifications to the CUDA driver.

Limitations

While most SASS instructions accept a guard predicate, the shared memory atomic operations, which are implemented with a load-lock and store-unlock instruction sequence, do not. To support programs with conditional execution of shared atomic instructions, we modify `ptxas` to guard the lock/unlock sequence with a divergent branch and handle reconvergence through the divergence stack. Only 4 out of 28 benchmarks are programmed with shared memory atomic operations (SA column of Table 5.1 and 5.2).

Integer division and remainder instructions are expanded into a loop with conditional branches by `ptxas`, invalidating the ordering information generated by our LLVM predication pass. To avoid this problem, we call the "Expand Integer Division" pass to legalize integer division and remainder operations in the LLVM compiler before we call our predication pass. We also add this pass to the baseline compiler, which only uses the divergence stack to handle control flow, for a clearer comparison against thread-aware predication. Only 4 out of 28 benchmarks use an integer division or a remainder operation (DR column of Table 5.1 and 5.2).

5.3 Evaluation

Our study uses 11 benchmarks from Parboil [121] and 11 benchmarks from Rodinia [30], which cover compute-intensive domains including linear algebra, image processing, medical imaging, biomolecular simulation, physics simulation, fluid dynamics, data mining, and astronomy. We also added 6 benchmarks we wrote to characterize our thread-aware predication CUDA compiler, including a control-flow heavy N-queens benchmark and several FFT benchmarks with different radices. We characterize our thread-aware predication approach on an NVIDIA Tesla K20c GPU (Kepler GK110) and compare performance results with baseline runs that only use the divergence stack to handle control flow. To draw a clearer comparison between hardware and software divergence management schemes, we disable the limited if-conversion heuristic in the baseline compiler so that the baseline solely uses the divergence stack to handle the control flow, and therefore clearly delineate contributions of predication.

Table 5.1: Benchmark Statistics Compiled for Kepler and Run on Tesla K20c (GK110)

Application	Kernel	Compile-time Statistics																				Runtime Statistics						
		Structures			Inst.			Stack Inst.		CBranch Inst.			Registers				Pred Regs.				Br Uni.		Occup.		Runtime (ms)		Speedup
		BB	R	L	Br	SA	DR	Push	Pop	SBU	RBU	LC	S	P	PS	PSR	S	P	PS	PSR	U	D	S	PSR	S	PSR	
p-bfs	BFS	17	12	2	10	1	0	5	10	0	5	2	20	20	20	20	2	4	4	4	0.18	0.82	0.75	0.75	0.44	0.46	0.96×
	BFS_in_GPU	28	18	4	16	1	0	7	13	0	4	5	31	43	42	42	2	7	5	5	0.38	0.62	0.75	0.50	2.71	3.10	0.90×
	BFS_multi_blk	46	32	7	28	1	0	13	24	0	11	7	39	62	62	62	4	7	6	6	0.16	0.66	0.75	0.50	5.31	5.54	0.93×
p-cutcp	lattice6overlap	30	24	4	17	0	0	9	17	2	8	6	28	43	33	30	1	7	4	3	0.47	0.53	0.69	0.69	4.94	4.48	1.10×
p-histo	main	31	30	4	21	2	0	11	23	1	11	4	23	34	34	30	7	7	7	7	1.00	0.00	0.75	0.75	0.34	0.40	0.86×
	final	10	6	3	6	0	0	2	2	0	0	3	38	42	42	42	1	2	2	2	1.00	0.00	0.75	0.50	0.06	0.06	1.00×
	prescan	46	25	6	25	0	1	11	20	4	11	6	14	16	14	14	2	5	5	5	0.52	0.09	1.00	1.00	0.03	0.03	0.95×
	intermediates	353	208	64	208	0	0	143	174	0	80	64	22	24	24	24	3	5	5	5	0.46	0.54	1.00	1.00	0.21	0.23	0.94×
p-lbm	StreamCollide	3	1	0	1	0	0	1	2	0	1	0	34	41	41	42	1	1	1	1	0.00	1.00	0.75	0.63	2.13	2.12	1.01×
p-mri-gridding	uniformAdd	7	3	0	3	0	0	2	4	0	3	0	6	8	8	6	1	2	2	1	0.64	0.36	1.00	1.00	0.12	0.12	1.00×
	gridding	38	33	7	23	0	3	13	20	11	5	7	62	70	61	58	3	7	4	3	0.00	1.00	0.50	0.50	149.58	155.79	0.96×
	binning	6	3	0	3	0	0	1	3	0	2	0	8	11	11	8	1	4	4	3	0.00	1.00	1.00	1.00	1.94	1.94	1.00×
	reorder	3	1	0	1	0	0	0	0	0	1	0	14	14	14	14	1	1	1	1	0.33	0.67	1.00	1.00	2.55	2.55	1.00×
	scan_L1	20	11	2	11	0	0	6	12	2	3	2	17	16	16	16	2	6	4	4	0.55	0.45	1.00	1.00	0.81	0.81	0.99×
	splitRearrange	10	4	1	5	0	0	3	6	0	3	2	22	25	25	21	2	3	3	3	0.67	0.33	1.00	1.00	1.49	1.65	0.91×
	scan_inter1	5	3	1	3	0	0	1	2	1	1	1	16	15	15	15	1	3	1	1	0.67	0.33	0.61	0.61	0.01	0.01	1.05×
	scan_inter2	5	3	1	3	0	0	1	2	1	1	1	15	16	15	15	1	3	1	1	0.00	1.00	0.61	0.61	0.01	0.01	1.01×
	splitSort	21	12	3	12	4	0	10	16	2	5	3	43	48	41	41	3	7	4	4	0.31	0.69	0.63	0.63	4.35	4.28	1.02×
p-mri-q	ComputeQ	5	3	1	3	0	0	0	0	2	0	1	21	22	22	22	1	3	1	1	1.00	0.00	1.00	1.00	1.60	1.56	1.03×
	ComputePhiMag	3	1	0	1	0	0	0	0	0	1	0	10	10	10	10	1	1	1	1	1.00	0.00	1.00	1.00	0.00	0.00	1.04×
p-sad	mb_calc	27	18	5	17	0	0	9	12	0	8	5	49	62	62	62	3	6	6	6	0.67	0.33	0.50	0.50	8.90	8.31	1.05×
	larger_calc_8	6	3	1	3	0	0	2	3	0	1	1	26	24	24	24	1	2	2	2	0.50	0.50	1.00	1.00	2.92	2.85	1.03×
	larger_calc_16	4	2	1	2	0	0	1	1	0	0	1	26	24	24	24	1	2	2	2	0.63	0.38	0.25	0.25	0.57	0.61	0.96×
p-sgemm	mysgemmNT	6	3	2	3	0	0	0	0	1	0	2	45	38	49	49	1	4	1	1	1.00	0.00	0.63	0.56	2.01	1.87	1.04×
p-spmv	spmv_jds	7	4	1	4	0	0	3	4	0	2	1	19	22	22	20	1	5	5	6	0.75	0.25	0.48	0.48	0.11	0.11	0.98×
p-stencil	block2D_hybrid	34	14	1	14	0	0	12	24	1	12	1	31	42	40	37	7	7	7	7	0.71	0.29	1.00	0.75	0.66	0.68	0.98×
p-tpacf	gen_hists	56	37	5	30	1	0	22	40	4	21	5	29	36	31	31	4	7	4	4	0.59	0.41	0.38	0.38	2040.31	2143.03	0.95×

Note: p=Parboil for application names. Kernel names are abbreviated. BB=Basic Blocks, R=Regions, L=Loops, Br=Branches, SA=Shared Atomics, DR=Integer Division/Remainder, SBU=Static Branch-Uniformity optimization, RBU=Runtime Branch-Uniformity optimization, LC=Consensual Branches for Loops, S=Compiled with divergence stack, P=Compiled with thread-aware predication, PS=P+SBU, PSR=P+SBU+RBU, Br Uni.=Branch Uniformity, U=Uniform, D=Divergent, Occup=Occupancy.

Table 5.2: Benchmark Statistics Compiled for Kepler and Run on Tesla K20c (GK110) Cont'd

Application	Kernel	Compile-time Statistics																				Runtime Statistics						
		Structures			Inst.			Stack Inst.		CBranch Inst.			Registers				Pred Regs.				Br Uni.		Occup.		Runtime (ms)		Speedup
		BB	R	L	Br	SA	DR	Push	Pop	SBU	RBU	LC	S	P	PS	PSR	S	P	PS	PSR	U	D	S	PSR	S	PSR	
r-b+tree	findK	12	8	1	7	0	0	2	6	1	3	1	20	24	23	23	1	4	2	2	0.18	0.64	1.00	1.00	1.50	1.52	0.99×
	findRangeK	18	13	1	11	0	0	4	12	1	7	1	27	38	32	32	1	4	2	2	0.29	0.57	1.00	0.75	1.35	1.74	0.77×
r-backprop	layerforward	20	14	3	11	0	1	4	7	1	5	3	21	33	31	31	2	7	6	6	0.40	0.60	1.00	1.00	0.38	0.39	0.96×
	adjust_weights	3	1	0	1	0	0	0	0	0	1	0	19	19	19	19	1	2	2	2	0.00	1.00	1.00	1.00	0.34	0.34	1.00×
r-bfs	Kernel	8	6	1	5	0	0	1	2	0	3	1	16	18	18	18	1	2	2	2	0.40	0.60	1.00	1.00	0.43	0.44	0.95×
	Kernel2	4	2	0	2	0	0	0	0	0	2	0	11	11	11	11	1	2	2	2	0.50	0.50	1.00	1.00	0.05	0.05	0.94×
r-gaussian	Fan1	3	1	0	1	0	0	0	0	0	1	0	11	11	11	11	1	1	1	1	0.00	1.00	1.00	1.00	0.01	0.01	1.01×
	Fan2	5	3	0	3	0	0	0	0	0	3	0	12	15	15	12	1	2	2	2	0.00	1.00	0.25	0.25	0.25	0.26	0.90×
r-hotspot	calculate_temp	14	7	1	8	0	0	3	7	1	4	2	33	35	35	34	3	6	4	3	0.38	0.63	0.75	0.75	0.11	0.11	0.97×
r-lud	lud_diagonal	28	21	7	18	0	0	2	3	0	0	7	44	74	73	73	2	7	7	7	0.83	0.17	0.25	0.25	0.06	0.07	0.90×
	lud_internal	1	0	0	0	0	0	0	0	0	0	0	17	17	17	17	0	0	0	0	–	–	1.00	1.00	0.01	0.01	1.01×
	lud_perimeter	35	27	8	21	0	0	2	3	0	4	8	42	49	49	42	3	7	7	7	0.80	0.20	0.25	0.25	0.08	0.10	0.85×
r-nn	euclid	3	1	0	1	0	0	0	0	0	1	0	8	8	8	8	1	1	1	1	0.00	1.00	1.00	1.00	0.01	0.01	1.01×
r-pathfinder	dynproc	13	6	1	7	0	0	3	6	1	3	2	17	18	17	18	3	4	3	3	0.43	0.57	1.00	1.00	0.15	0.13	1.16×
r-srad-v1	srad	15	8	2	8	0	2	6	13	0	3	2	22	20	20	26	2	5	5	5	0.62	0.15	1.00	1.00	0.09	0.10	0.87×
	srad2	13	7	2	7	0	2	4	4	0	3	2	20	20	20	32	2	5	5	5	0.67	0.33	1.00	1.00	0.07	0.09	0.86×
	reduce	66	43	4	35	0	1	25	46	3	25	4	26	38	36	36	2	7	5	5	0.17	0.43	1.00	0.75	0.08	0.09	0.88×
	extract	3	1	0	1	0	0	0	0	0	1	0	4	4	4	4	1	1	1	1	0.00	1.00	1.00	1.00	0.01	0.01	1.00×
	prepare	3	1	0	1	0	0	0	0	0	1	0	10	10	10	10	1	1	1	1	0.00	1.00	1.00	1.00	0.02	0.02	1.01×
	compress	3	1	0	1	0	0	0	0	0	1	0	4	4	4	4	1	1	1	1	0.50	0.50	1.00	1.00	0.01	0.01	1.01×
r-srad-v2	srad_cuda_1	28	18	0	14	0	0	5	24	4	13	0	23	26	26	26	5	7	7	7	0.36	0.64	1.00	1.00	0.98	1.04	0.95×
	srad_cuda_2	11	5	0	5	0	0	3	8	2	4	0	20	22	22	20	3	5	5	4	0.40	0.60	1.00	1.00	1.01	1.02	1.00×
r-streamcluster	compute_cost	7	4	1	4	0	0	0	0	0	3	1	18	15	15	15	1	4	4	4	0.75	0.25	1.00	1.00	0.52	0.59	0.86×
nqueens	nqueens	47	44	13	32	0	0	11	16	4	7	14	27	54	54	57	2	7	7	7	0.58	0.42	1.00	0.50	55.06	56.05	0.98×
radix2fft	mp_radix2	6	3	1	3	0	0	0	0	2	0	1	35	37	35	35	1	3	1	1	1.00	0.00	0.50	0.50	0.01	0.01	1.01×
	sp_radix2	3	2	1	2	0	0	0	0	1	0	1	22	26	22	22	1	2	1	1	1.00	0.00	0.50	0.50	0.01	0.01	1.04×
radix3fft	radix3fftd	4	2	1	2	0	0	0	0	1	0	1	34	43	33	33	1	2	1	1	1.00	0.00	0.75	0.75	0.01	0.01	0.99×
radix4fft	radix4fftd	3	2	1	2	0	0	0	0	1	0	1	36	40	36	36	1	2	1	1	1.00	0.00	0.25	0.25	0.01	0.01	0.98×
radix5fft	radix5fftd	4	2	1	2	0	0	0	0	1	0	1	44	48	44	44	1	2	1	1	1.00	0.00	0.25	0.25	0.01	0.01	0.99×
radix6fft	radix6fftd	4	2	1	2	0	0	0	0	1	0	1	44	53	45	45	1	2	1	1	1.00	0.00	0.25	0.25	0.01	0.01	1.00×

Note: r=Rodinia for application names. FFT benchmark results are reported as one benchmark result. For abbreviations reference Table 5.1's notes.

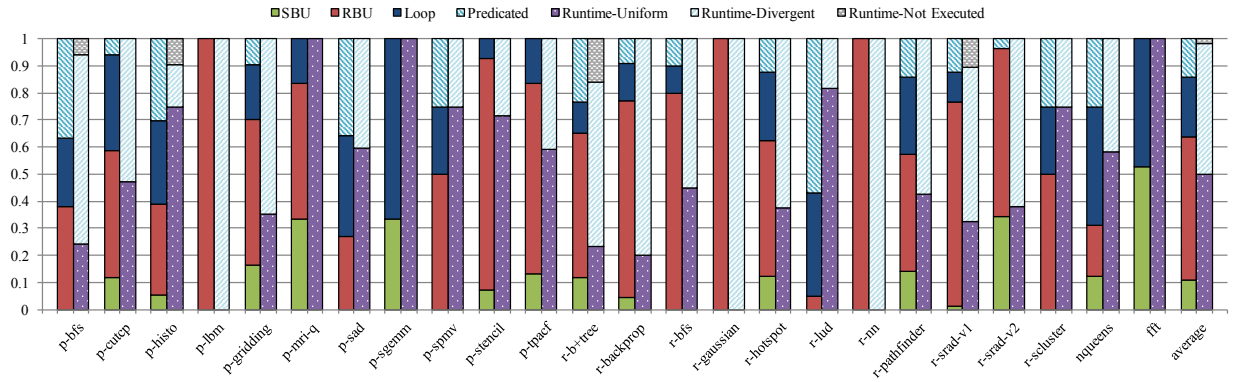
Benchmark Characterization

Table 5.1 and 5.2 report compile-time and runtime statistics of the different kernels of the 28 benchmarks. The BB, R, and L columns of the table, which count the number of basic blocks, regions, and loops of each kernel, show that the benchmarks are composed of a non-trivial number of control-flow structures. The Br column of the table counts the total number of conditional branches in each kernel. The push and pop of stack instructions columns count the number of baseline compiler generated instructions that sequence the divergence stack. The CBranch instruction columns count the number of consensual branches inserted by the thread-aware predication compiler. The SBU column counts the number of branches that are proved to be non-divergent by the static branch-uniformity optimization. The RBU column reports the number of consensual branches that were added by the runtime branch-uniformity optimization. Unlike the if-then-else statements, consensual branches are required to implement loop constructs correctly. The LC column counts these branches.

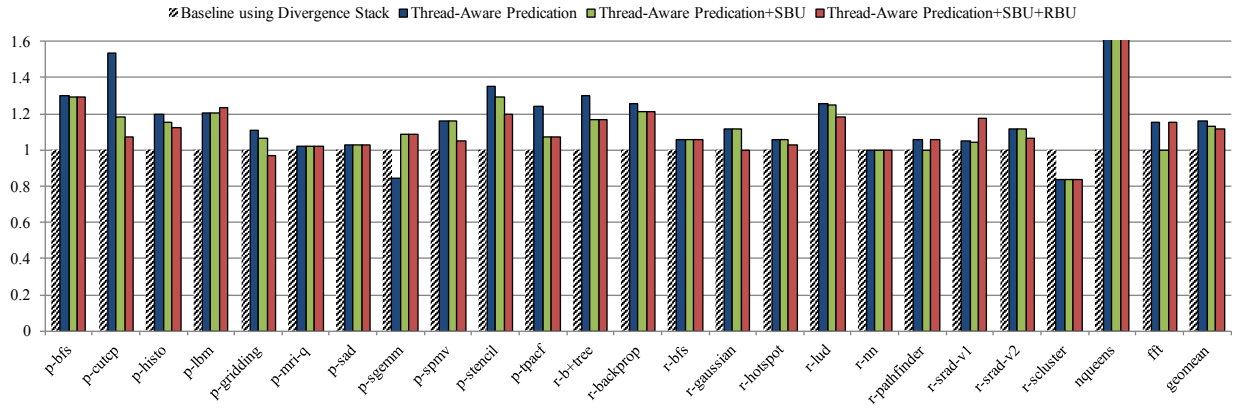
We have developed a SASS instrumentation tool, which injects instrumentation code before all conditional branches at the final pass of `ptxas` [118]; we use this tool to collect the runtime uniformity statistics and record them in the Br Uniformity columns. Figure 5.3a visualizes the classification of compile-time and runtime branches statistics. On average, the compiler proved 11% of branches to be non-divergent (SBU), added dynamic checks for 53% of the branches (RBU), turned 22% of the branches into consensual branches for loops (Loop), and removed the remaining with predication (Predicated). At runtime, 50% of the branches turned out to be uniform, and 48% to be divergent. The remaining 2% of the branches were not executed. In general, the predication compiler is doing a good job optimizing for branch uniformity. However, there are certain benchmarks such as `r-lud` where the compiler can improve, as the runtime-uniform bar is 17$\times$ taller the SBU+RBU bar.

The S column of both register sections counts the number of data registers and predicate registers used by the baseline compiler, which only uses the divergence stack to handle control flow. The P column captures the number of registers used by the thread-aware predication compiler. The PS and PSR columns count the number of registers used by the compiler when the static and runtime branch-uniformity optimizations are enabled respectively. Figure 5.3b shows the register usage normalized to the baseline register count. The general trend is that predication increases register pressure, since a conservative register allocator cannot reuse registers for both sides of a branch. Normally with branches, a register allocator can easily reuse the same register on different sides of the branch. With predication, the allocator would have to prove that certain predicate conditions are disjoint to do so. For some branches, the static branch-uniformity optimization can prove that a branch is non-divergent so that the compiler can safely remove the predicates from both sides of the branches, make register allocation easier, and alleviate register pressure. Runtime branch-uniformity tends to further reduce register pressure by preventing the compiler from blending instructions from both sides of the branches, hence reducing the live ranges of values.

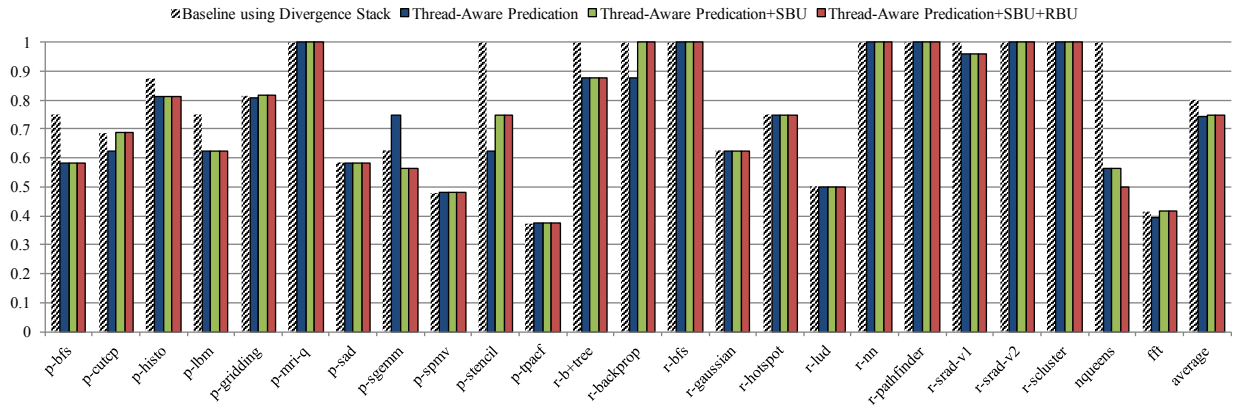
Register pressure affects occupancy (the number of threads that can execute simultaneously) as the threads share a common pool of physical registers. Figure 5.3c shows the average occupancy of kernels reported in the occupancy column of the benchmark statistics table, normalized to the



(a) Branch Classification



(b) Register Pressure



(c) Occupancy

Figure 5.3: Benchmark Characterization with Thread-Aware (TA) Predication – (a) branch classification, (b) register pressure, (c) occupancy. SBU=Static Branch-Uniformity optimization, RBU=Runtime Branch-Uniformity optimization. In (b), register usage for nqueens (outside figure) is 2× for TA-predication and +SBU data points and 2.1× for the TA+SBU+RBU.

maximum occupancy of the processor. Occupancy decreases for those applications that experience increased register pressure: `p-bfs`, `p-histo`, `p-lbm`, `p-sgemm`, `p-stencil`, `r-b+tree`, `r-srad-v1`, and `nqueens`. The static branch-uniformity optimization recoups occupancy lost by the baseline predication algorithm for `p-cutcp`, `p-stencil`, `r-backprop`, and `fft`. As discussed in the next section, occupancy has a strong influence on performance and is a critical metric for compiler optimizations in throughput processors.

Performance Analysis

Figure 5.4 shows the performance of all benchmarks normalized to the performance using the divergence stack. Table 5.1 and 5.2 have runtime and speedup numbers of all kernels that comprise these benchmarks for the baseline compiler and the thread-aware predication compiler with both branch-uniformity optimizations enabled.

The performance of the thread-aware predication compiler targeting the Kepler GPU only with predication and consensual branches is competitive with the baseline compiler using the divergence stack. The thread-aware predication compiler with both branch-uniformity optimizations generates code that is only 2.7% slower on average than the baseline compiler. Without any optimizations, the thread-aware predication compiler is 11.3% slower than the baseline. Static branch-uniformity optimization boosts performance by 4.7%, and the runtime branch-uniformity optimization adds an additional 3.9%. Runtime branch-uniformity optimizations harm performance in some cases, especially when the branch conditions are truly unbiased. In such cases, consensual branches added for runtime uniformity checks are pure overhead. Adaptive optimization could use online profiles and recompilation to elide unnecessary uniformity checks. By manually picking the best-performing cases, the thread-aware predication compiler is only 0.6% slower.

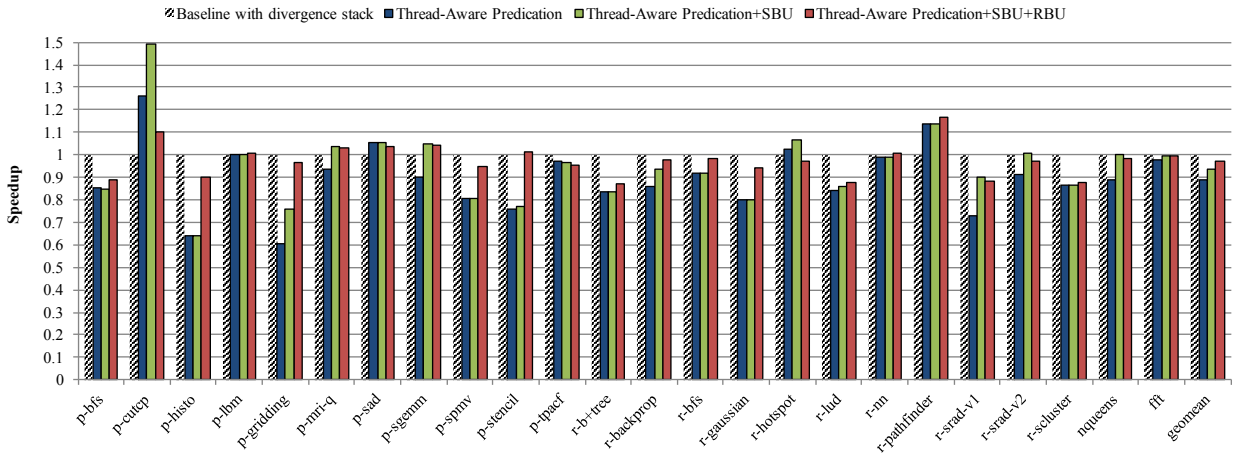


Figure 5.4: Speedup of Thread-Aware Predication Against Divergence Stack on NVIDIA Tesla K20c – SBU=Static Branch-Uniformity optimization, RBU=Runtime Branch-Uniformity optimization.

The static branch-uniformity optimization tends to increase performance, as this optimization reduces register pressure. The only exception is `p-tpacf`, where performance decreases monotonically with each additional branch-uniformity optimization. Upon inspection the compiler generates a better instruction schedule when both optimizations are disabled by intermixing more parallel execution paths. When both optimizations are disabled, the compiler uses 3 more predicate registers (Table 5.1 and 5.2), and has more freedom with instruction scheduling because more than 80% of the branches are predicated (Figure 5.3a).

The runtime branch-uniformity optimization increases performance for 11 out of 24 benchmarks by skipping regions with a null guard predicate. Among those 11 benchmarks, `p-histo`, `p-gridding`, `p-spmv`, and `p-stencil` benefit from these runtime checks by 26%, 21%, 14%, and 25% respectively. However, runtime checks can also reduce performance. The worst of the nine benchmarks that performed worse with this runtime optimization is `p-cutcp`, which dropped by 39%. For this benchmark, we found that the compiler is able to schedule multiple fused-multiply-add instructions from multiple execution paths simultaneously when the dynamic checks are not included.

Nine benchmarks performed better with the thread-aware predication compiler than the baseline compiler and 17(19) benchmarks performed within 0.95×(0.90×) of the performance of the baseline compiler respectively. Of the five benchmarks that see a performance loss of more than 10%, `p-bfs`, `r-b+tree`, and `r-srad-v1` exhibit reduced occupancy (Figure 5.3c). Table 5.1 shows that the two kernels with reduced occupancy in `p-bfs` use 11 and 23 additional registers respectively. Likewise, offending kernels `r-b+tree` and `r-srad-v1` use 5 and 10 more registers, respectively. For each of these cases, the additional registers per thread are enough to limit the parallelism that the processor can exploit. As mitigating register pressure is critical to obtaining performance in throughput processors, Section 5.5 discusses options for improving register allocation for predicated code. For the remaining two benchmarks `r-lud` and `r-scluster`, the compiler is not able to optimize for all branch uniformity exhibited during runtime (the runtime-uniform bar is much larger than the SBU+RBU bar in Figure 5.3a). We discuss options for improving branch-uniformity checks in Section 5.5.

Although not shown in the graph, the limited if-conversion heuristic implemented in the NVIDIA production compiler only makes 6 out of 24 benchmarks run faster when compared to the baseline compiler. Eleven benchmarks run slower with the if-conversion heuristic, while the remaining 7 benchmarks run at them same speed. The average performance does not change with the if-conversion heuristic. Out of those 6 benchmarks that run faster with the limited if-conversion heuristic, only 3 benchmarks are 1%, 2%, and 3% faster than code generated from our thread-aware predication compiler respectively.

Discussion on Area, Power, and Energy

Quantifying the impact of software divergence management on area, power, and energy is challenging, since we ran CUDA workloads on GPU silicon to obtain performance numbers and benchmark statistics. The primary motivation to remove the divergence stack is to reduce hardware design complexity and associated verification costs. We estimate that area and power savings from elim-

inating the divergence stack are not significant, so performance would serve as a good proxy for power and energy consumption. The performance of code generated by the thread-aware predication compiler is competitive to the one generated by the baseline compiler using the divergence stack. For that reason, we believe that the power and energy consumption of the software divergence management scheme is on par with the hardware scheme. With improvements proposed in Section 5.5, software divergence management has potential to outperform hardware divergence management schemes in terms of performance, and therefore power and energy efficiency as well.

5.4 Discussion

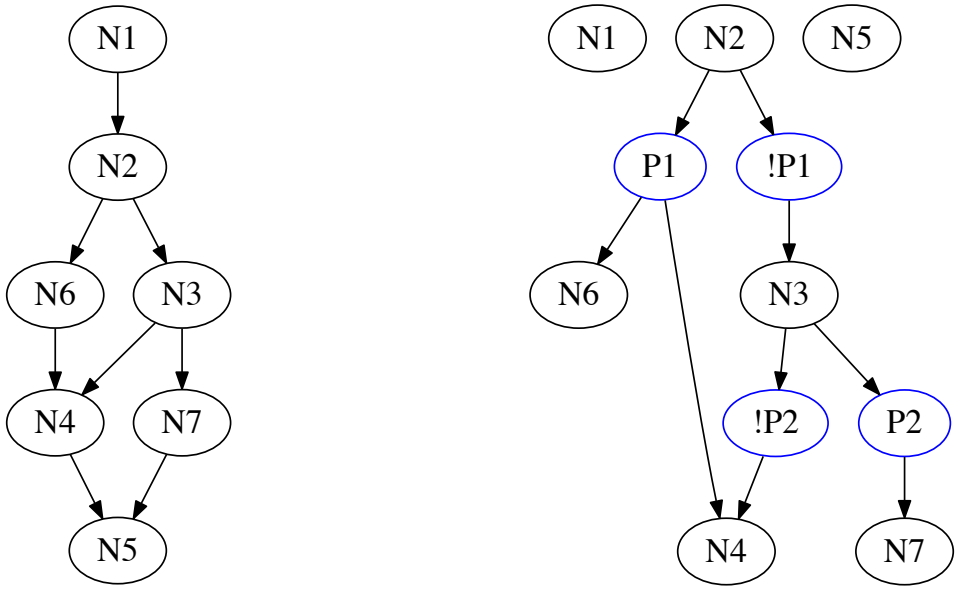
The risks and benefits of predication on latency-oriented architectures such as traditional CPUs are well understood. In that context, branch predictability and instruction path length play a major role [84, 83]. The tradeoffs associated with predication on throughput-oriented architectures with a divergence stack are less studied and less intuitive. This section discusses some of the reasons that extremely aggressive predication is effective on such architectures. We also discuss ways in which we can potentially co-design future throughput-oriented architectures to make predication even more effective.

Advantages of Software Divergence Management

In theory, the predicated code should perform as well as code that uses the divergence stack, since the underlying mechanism to handle divergent execution is fundamentally the same. With predication, the compiler is explicitly scheduling the operations in the same way a divergence stack would do implicitly. The reconvergence point when using a divergence stack is known to be the immediate post-dominator in a CFG for if-then-else statements [45]. For loops, the reconvergence point is the immediate post-dominator of all exit blocks, which is the post-tail block of a loop. The algorithms in Section 5.1 will find the same reconvergence points and put a predicate with reconverged threads rather than setting up a reconvergence point with a `push.stack` instruction. Whereas the hardware stack has to spill to DRAM if there is too much divergence, the predicating compiler correspondingly manages excessive divergence through explicit predicate register spills. Hence, the main benefit of software divergence management is reducing hardware complexity by eliminating hardware structures used for divergence management without altering the programmer's view of the machine.

There are additional advantages of managing divergence explicitly in software. Figure 5.5a shows a CFG for a simple short-circuit code segment. Following the reconvergence rule discussed above, when threads diverge at basic blocks $N2$ and $N3$, they will reconverge at $N5$ where parallel execution will resume. Basic block $N4$ can be a partial reconvergence point; however, with a divergence stack, threads will execute $N4$ serially. Using predication with the CDG shown in Figure 5.5b, diverged threads from $N2$ and $N3$ will join at $N4$ to execute in parallel.

Managing divergence explicitly by the compiler provides more control over irregularly structured code than the divergence stack mechanism can. The order of execution, under divergence



(a) Control Flow Graph (CFG) **(b)** Control Dependence Graph (CDG)

Figure 5.5: Short-Circuit Example Showing Limitations of the Divergence Stack – (a) control flow graph, (b) control dependence graph.

stack control, is nondeterministic; the hardware can choose to execute either side of the branch first. This nondeterminism puts a limit on what the compiler can guarantee when analyzing the control flow. For example, the variance analysis used for the static branch-uniformity optimization can make stronger guarantees when divergence is managed by the compiler as it can analyze hazards between different execution paths with a known execution order. Better variance analysis not only results in faster performance with fewer registers, but also opens up more opportunities for scalarizing SPMD code on data-parallel architectures [74].

Function Calls

Predicated function calls can be supported by a straightforward calling convention. The convention designates one predicate register (e.g., `p0` register) as the *entrance mask* to hold a mask of threads that are active at function entrance. The compiler then guards via predication all instructions in the function with the entrance mask. If the entrance mask is live across a function call, the compiler should move it to a callee-saved register or spill it to the stack before calling the function.

Predicated virtual function calls can be supported with a simple instruction added to the hardware. Figure 5.6b shows the predicated version of Figure 5.6a. The new `find_unique` instruction will return a unique value of a vector register (namely an address for the function) and a predicate mask of active threads that holds the unique value. Following the calling convention, we save the resulting predicate mask in `p0` and then jump to the unique program counter. When control from the function returns, we mask off the threads that executed the function, and check

```
1 # function pointer PC stored in r3
2 # predicate register p2 holds the active threads
3 @p2 jalr r3
```

(a) With a Divergence Stack.

```
1 loop:
2     p0, r4 = find_unique p2, r3
3 @p0 jalr r4 # known to be unique
4     p2 = p2 and !p0
5     cbranch.ifany p2, loop
```

(b) With Predication.

Figure 5.6: Supporting Virtual Function Calls with Predication – (a) divergence stack, (b) predication.

whether any active threads still remain. If so, we loop back and repeat with a new program counter until all active threads are sequenced.

5.5 Future Research Directions

Although our experimental thread-aware predication compiler is competitive in performance to a well-tuned production compiler that uses the divergence stack, we expect that we can make software divergence management even more effective with the following software and hardware improvements.

Tuning our compiler. Our heuristics for deciding when to enable the runtime branch-uniformity optimization have not been extensively tuned. More importantly, many of the downstream compiler passes have not been tuned with our optimizations in mind. In fact, some downstream compiler optimizations are not predicate-aware, rendering them ineffective. Some effort globally tuning the compiler and implementing common predicate-aware analyses and transformations could provide performance boosts.

Predication-aware register allocation. Register count affects occupancy, which has a strong influence on performance. Several studies [48, 39] look into techniques to reduce register pressure under predication for superscalar and VLIW architectures. We can apply similar techniques to reduce register count, and hence increase occupancy.

Better branch-uniformity optimizations. As shown in Figure 5.4, the compiler is only able to capture a small fraction of the runtime branch uniformity. The current variance analysis used in static branch-uniformity optimization only considers convergent basic blocks, where all threads

will enter with a full mask or not. The compiler does not analyze the case where a subset of the warp has the same branch condition. With the execution order of divergent regions understood by the compiler, we can extend the analysis to report scalar branch conditions across a subset of the warp. We spotted some cases where the structural analysis was not reporting all regions of interest to add runtime branch-uniformity checks. Other algorithms should also be considered to determine where and when to add these runtime checks.

Branch if any instruction. The current hardware only supports `cbranch.ifnone` or `cbranch.ifall` instructions. To emulate a `cbranch.ifany` instruction, we need two branches in a row, a `cbranch.ifnone` followed by an unconditional jump. A new instruction would improve performance as it would decrease instruction count and make unrolling easier by eliminating a branch instruction from the middle of an unrolled region.

Adaptive optimization. As shown in our performance results, static branch-uniformity optimization and runtime branch-uniformity optimization can sometimes reduce performance. Adaptive optimization in a just-in-time compilation scheme (such as implemented in the NVIDIA CUDA driver) could profile branch behaviors and generate code that selects the best optimizations on a per-branch basis. Feedback-directed optimization could similarly improve our results.

5.6 Predication Summary

Trading complexity back and forth between software and hardware is a classic debate in computer architecture. Divergence management is a prime target for these tradeoffs, with a wide range of software, hardware, and hybrid schemes implemented in the field. However, while hardware divergence management schemes have received a lot of attention from the academic research community, the benefits and drawbacks of software divergence management on data-parallel architectures have been less explored.

Hardware divergence management has its advantages. It enables a fairly conventional thread compilation model, makes register allocation easier, and simplifies the task of supporting complex irreducible control flow. However, in doing so the hardware takes on the burden of implementing fairly complex divergence management structures. By trading the complexity with the compiler to manage some or all divergence explicitly in software, we can potentially simplify the hardware without sacrificing programmability.

In this chapter, we present new compiler algorithms to systematically map arbitrarily nested control flow present in SPMD programs down to data-parallel architectures with predicates and consensual branches. We implement these compiler algorithms in a production CUDA compiler, and use it to run real workloads and gather runtime statistics on existing hardware. Our detailed performance analysis on an NVIDIA Tesla K20c show that software divergence management architectures can be competitive to hardware divergence management architectures. We anticipate that with our suggested software and hardware improvements, software divergence management schemes can be even more effective.

Chapter 6

The Hwacha Vector-Fetch Architecture

Our claim is that with a scalarizing compiler, traditional vector-like architectures can maintain the same level of programmability as other data-parallel architectures while being highly performant, efficient, yet a favorable compiler target. The Hwacha vector-fetch architecture is our attempt to construct a vector-like data-parallel machine that is even more efficient than a traditional vector machine. The *vector-fetch* instruction decouples the vector instruction stream into a separate thread in order to enable light-weight access-execute decoupling of the vector data stream. The Hwacha architecture also takes into account what we have learned from the previous chapters, and adds scalar execution resources and support for vector predication with consensual branches. All vector instructions, including vector gather and scatter operations, can be predicated. Virtual memory is supported with restartable exceptions. The architecture presents a clean assembly interface that enables dense packing of mixed-precision values in the vector register file as well as efficient subword parallelism for mixed-precision operations.

This chapter discusses the Hwacha architecture in detail. Section 6.1 describes the Hwacha assembly programming model, and Section 6.2 walks through the important Hwacha architectural features. We have developed the Hwacha architecture with multiple tapeouts on the ST 28 nm FD-SOI technology [136, 80, 135] and the IBM 45 nm SOI technology [78, 122]. Section 6.3 outlines the lineage and history of the Hwacha vector-fetch architecture. Chapter 7 details the Hwacha instruction set architecture, which has been developed as a non-standard extension to the RISC-V instruction set architecture [131, 132, 130, 129]. Chapter 8 describes the Hwacha decoupled vector microarchitecture. Chapter 9 discusses the implementation and evaluation of the Hwacha architecture with our custom LLVM-based scalarizing compiler using OpenCL microbenchmarks.

6.1 Hwacha Vector-Fetch Assembly Programming Model

The Hwacha vector-fetch assembly programming model builds on the traditional vector assembly programming model, with a key difference: the vector operations have been hoisted out of the *control thread* and placed in a separate *worker thread*. The control thread is in charge of the execution and is responsible for application setup, configuration, and stripmining the vectorized

loop. The control thread points to the worker thread with a *vector-fetch* instruction, and therefore the worker thread is often called the *vector-fetch block*. In the Hwacha assembly programming model, the control thread completes the stripmining loop faster and is able to continue doing useful work, while the worker thread is independently executing vector instructions.

User-Visible Register State

Figure 6.1 shows the Hwacha user-visible register state. Like the traditional vector machine, Hwacha has vector data registers (`vv`*0–255*) and vector predicate registers (`vp`*0–15*), but it also has two flavors of scalar registers. These are the vector *shared* registers (`vs`*0–63*, `vs`*0* is hardwired to constant 0), which can be read and written within a vector-fetch block, and vector *address* registers (`va`*0–31*), which are read-only within a vector-fetch block. This distinction supports non-speculative light-weight access-execute decoupling and is further described in Chapter 8, which details the Hwacha microarchitecture.

Vector data and shared registers may hold 8-, 16-, 32-, and 64-bit integer values and half-, single-, and double-precision floating-point values. Vector predicate registers are 1-bit wide, and hold boolean values that mask vector operations. Vector address registers hold 64-bit pointer values, and serve as the base and stride of unit-strided and constant-strided vector memory instructions.

In addition, the vector configuration register `vcfg`, which keeps the configuration state of the vector unit, and the vector length register `vlen`, which stores the maximum hardware vector length, are also visible to the user. The configuration state is described in Section 7.1. The maximum hardware vector length is configurable based on how many registers of each type a program uses. Regardless of how many registers are used, a hardware vector length of 8 is guaranteed. The vector length register (`vlen`) can be set to zero, in such case, all vector instructions will not be executed.

Assembly Programming Model

Figure 6.2 shows the CSAXPY kernel (the source is shown in Figure 2.2) mapped to the Hwacha assembly programming model. The structure of the stripmine loop in the control thread (line 1–16) is similar to the traditional vector code shown in Figure 2.5, however, all vector operations in the stripmine loop have been hoisted out into their own worker thread (line 18–25). The control thread first executes the `vsetcfg` instruction (line 2), which adjusts the maximum hardware vector length by taking the register usage (number of 64-, 32-, 16-bit vector data registers and vector predicate registers) into account. `vmcs` (line 3) moves the value of a scalar register from the control thread to a vector shared (`vs`) register. The stripmine loop sets the vector length with a `vsetvl` instruction (line 5), moves the array pointers to the vector address (`va`) registers with `vmca` instructions (line 6–8), then executes a vector-fetch (`vf`) instruction (line 9) causing the Hwacha unit to execute the vector-fetch block. The code in the vector-fetch block is equivalent to the vector code in Figure 2.5, with the addition of a `vstop` instruction, signifying the end of the

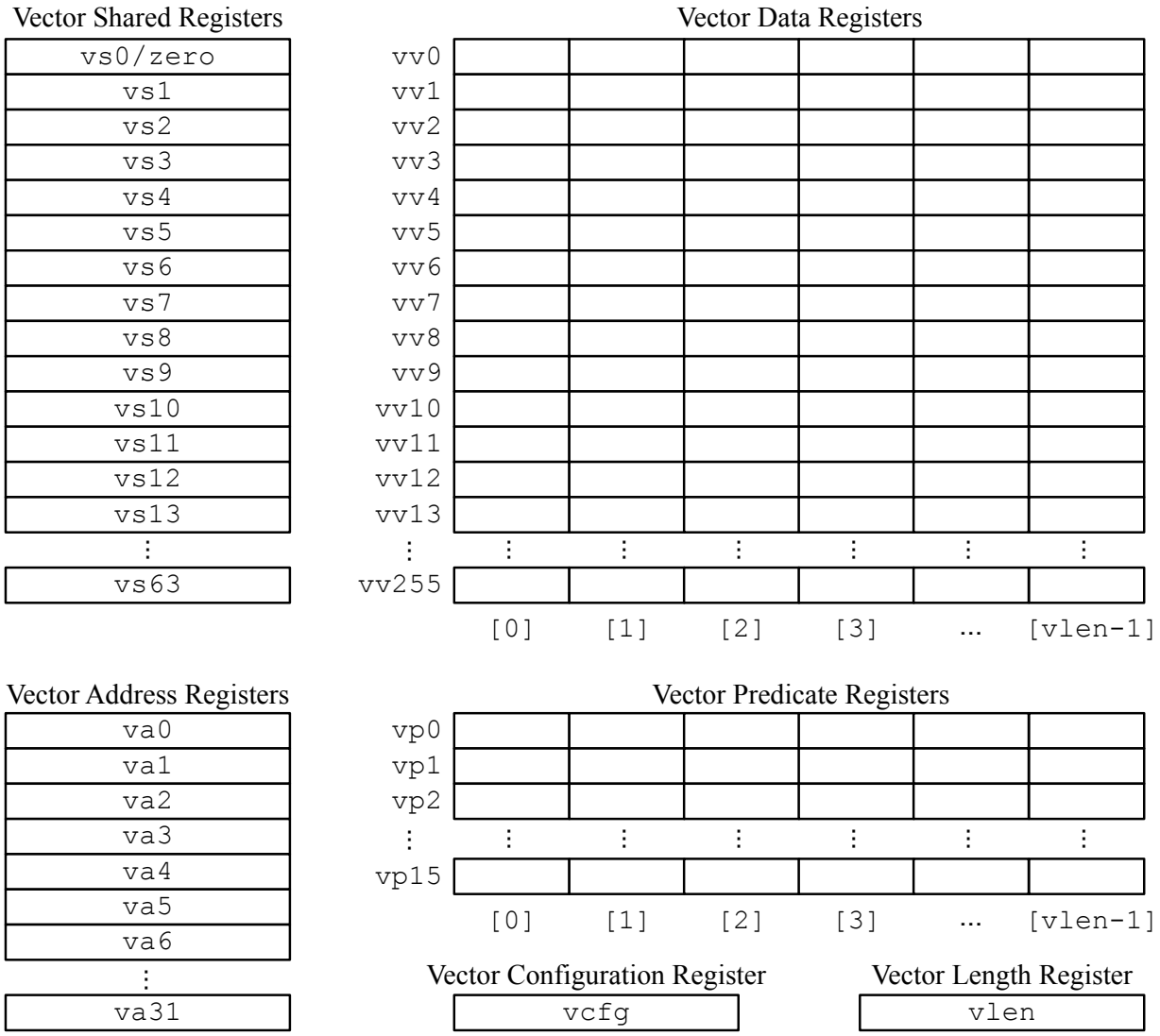


Figure 6.1: Hwacha User-Visible Register State – vector data registers (`vv`*0–255*), vector predicate registers (`vp`*0–15*), vector shared registers (`vs`*0–63*), vector address registers (`va`*0–31*), vector configuration register (`vcfg`), vector length register (`vlen`).

```
1  csaxpy_control_thread:
2       vsetcfg #v64, #v32, #v16, #vp
3       vmcs    vs1, a2
4  stripmine_loop:
5       vsetvl  t0, a0
6       vmca    va0, a1
7       vmca    va1, a3
8       vmca    va2, a4
9       vf      csaxpy_worker_thread
10      add     a1, a1, t0
11      slli    t1, t0, 2
12      add     a3, a3, t1
13      add     a4, a4, t1
14      sub     a0, a0, t0
15      bnez    a0, stripmine_loop
16      ret
17
18 csaxpy_worker_thread:
19      vlb     vv0, (va0)
20      vcmpez  vp0, vv0
21 !vp0 vlw     vv0, (va1)
22 !vp0 vlw     vv1, (va2)
23 !vp0 vfma    vv0, vv0, vs1, vv1
24 !vp0 vsw     vv0, (va2)
25      vstop
```

Figure 6.2: CSAXPY Kernel Mapped to the Hwacha Assembly Programming Model – Figure 2.2 shows the source code of the CSAXPY kernel. Register `a0` holds variable `n`, `a1` holds pointer `cond`, `a2` holds scalar `a`, `a3` holds pointer `x`, and `a4` holds pointer `y`. Contrast the Hwacha assembly programming model to the traditional vector assembly programming model shown in Figure 2.5.

block. The address bookkeeping is amortized across the entire vector and done once on the control thread (lines 10–13).

The combination of `vsetcfg` and `vsetvl` makes the Hwacha assembly code backward and forward compatible with different microarchitecture implementations. For example, the same code can run on wider vector machines with additional vector lanes. The end result of running the Hwacha assembly code on such a machine would be executing vector instructions with a longer hardware vector length.

For the CSAXPY example, factoring the vector code out of the stripmine loop reduces the scalar instruction count by 15%. However, for more complex loops with more vector instructions, the fraction of saved scalar instruction fetches per stripmine loop will increase (due to the Hwacha vector-fetch programming model). This allows the control thread to run ahead further, enabling a higher degree of access-execute decoupling.

6.2 Architectural Features

This section introduces the high-level design decisions of the Hwacha vector-fetch architecture: how the architecture hides memory latency, supports scalarization, predication, virtual memory, reconfigurable hardware vector length, and mixed-precision.

Hiding Memory Latency with Access-Execute Decoupling

With the processor-memory performance gap growing exponentially, high performance processors are architected to tolerate long memory latencies. There are generally four techniques to tolerate memory latency: static scheduling, dynamic scheduling, prefetching, and multithreading.

VLIW machines are built around static scheduling in which the compiler statically finds useful work to do while a load is in flight. Out-of-order processors exploit dynamic scheduling in which the microarchitecture dynamically exploits instruction-level parallelism by picking useful work to do from a wide range of instructions uncovered with branch speculation.

GPUs hide memory latency with multithreading [27]; while a thread is waiting for the load data to return, the machine switches threads to find useful work to do. This choice naturally falls out of the SPMD programming model, where the application kernel is expressed as highly threaded code. The GPU execution model reduces the program counter and instruction fetch overheads of multithreading, but significant overheads of the highly threaded code remains—the thread state. A thread has to park all of its state on the chip while waiting for the load data to return. For this reason, GPUs often have a very large register file built out of SRAM macros.

However, when the address stream is well known ahead of time, we can exploit prefetching rather than multithreading to minimize the on-chip buffer space needed to hide memory latency. The key insight is that prefetching only uses on-chip buffer space to hold the prefetched data, while multithreading requires the entire thread state to be present on the chip (every register except the destination register to hold the load data is overhead). Decoupled access-execute computer architectures [114] are a classic example that exploits prefetching. The access processor runs

ahead and prefetches the load data into a queue, while the execute processor catches up consuming the load data from the queue later on. Runahead processors [90, 29] also exploit prefetching with a scout thread. When a thread is stalled waiting for the load data to return, the processor checkpoints the architectural state, then switches to the runahead mode, where the scout thread continues to execute instructions from the stalled thread, prefetching data into caches long before it is needed. When the load data returns from memory, the processor resumes normal execution after restoring architectural state from the recent checkpoint.

The Hwacha vector-fetch architecture takes advantage of access-execute decoupling between the vector runahead unit (access processor) and the vector execution unit (execute processor), similar to schemes introduced by Batten et al. [22] and Espasa and Valero [41]. The key difference is that both units fetch vector instructions from the same vector-fetch block simplifying the assembly programming model; the access processor runs ahead and fetches vector memory instructions, which reference vector address registers (`va`*0–31*), and issues prefetch operations long before the data is needed by the execute processor. Unlike the original decoupled access-execute architecture [114], the Hwacha vector-fetch architecture has no risk of deadlocking, since the vector runahead unit does not rely on the vector execution unit to make any decisions. Note, this vector access-execute decoupling scheme is greatly simplified by disallowing the worker thread to write the vector address registers. Chapter 8 has more details on how the access-execute decoupling is implemented in the Hwacha microarchitecture.

Exploit Operand Uniformity with Scalarization

Chapter 4 shows that a simple compiler algorithm is able to statically scalarize one-third of the executed operations and register accesses. The Hwacha vector-fetch architecture takes advantage of these scalar values by putting them into vector shared registers (`vs`*0–63*) and using them across the entire vector. Scalar instructions (both scalar computational instructions and scalar memory access instructions) that read and write vector shared registers are allowed in the vector-fetch block. Vector instructions can source operands from vector shared registers. The control thread is also allowed to write a vector shared register via a `vmcs` instruction.

Early vector machines such as the Cray-1 [109], Fujitsu VP100/200 [123], NEC SX [128], Hitachi S820 [40] had scalar registers, scalar functional units, and datapath to forward scalar operands to vector functional units. Packed-SIMD extensions such as the Intel MMX [102], SSE [105], AVX [82], ARM NEON [10], SPARC VIS [126], MIPS MDMX [51], and PowerPC AltiVec [37] lacked support for vector-scalar operations, and therefore had to *splat* a scalar value to a vector register. Recent AMD GPUs [5, 7] added scalar registers to the architecture, however, the scalar registers are primarily used to manage divergence.

Handle Control Flow Efficiently with Predication

Chapter 5 shows that simple vector predication in conjunction with consensual branches is able to efficiently handle complex control flow (if-then-else statements, loops, function calls including virtual calls) found in parallel programming languages. The Hwacha vector-fetch architecture

stores predicates in vector predicate registers (`vp`*0–15*) and allows all vector instructions (including both vector computational instructions and vector memory access instructions) to be predicated. Consensual branches are also supported in a vector-fetch block with the following conditions: `ifnone`, `ifall`, and `ifany`. Logical operations on vector predicates are supported with an instruction that takes three input predicates and an 8-entry lookup table that encodes the logic function. Vector predicate memory access instructions can spill and refill contents of the vector predicate register to and from memory.

Early vector machines supported vector predication. The Cray-1 had one vector mask register on which vector instructions were implicitly predicated [109]. The Fujitsu VP100/200 [123], NEC SX [128], and Hitachi S820 [40] had a separate set of vector predicate registers, vector instructions that were explicitly predicated, and separate predicate logic operations that manipulated the vector predicates. The packed-SIMD extensions generally lack predication support, however, the recent Intel AVX-512 packed-SIMD extension adds predication support with vector predicate registers (`k`*0–7*) [56], but is only available on its Xeon line of server processors and Xeon Phi coprocessors [57] as of 2015. As discussed in Section 2.3, NVIDIA GPUs provide vector predicate registers and vector instructions that can be explicitly predicated, while AMD GPUs provide special predicate registers to hold vector comparison results and vector masks that implicitly predicate the vector instructions.

Virtual Memory Support with Restartable Exceptions

The Hwacha vector-fetch architecture is built on a framework for OS-friendly accelerators, suitable for use with multi-programmed applications running on a general-purpose operating system [127]. All addresses referenced in a vector-fetch block are virtual, and therefore need to be translated. Address translation on the vector unit can fail for various reasons, so we augment the Hwacha machine to handle restartable exceptions by providing a mechanism to save and restore the microarchitectural state of the vector unit. This mechanism is also used by the operating system to stop and reschedule a process that is in use of the Hwacha vector unit. We also define additional fence semantics that order the memory operations between the control thread and the worker thread.

Early vector machines such as the Cray-1 [109] required all pages it accessed to be pinned in physical memory, due to the difficulty of implementing precise exceptions or restartable exceptions. The IBM Vector Facility supported restartable exceptions by limiting the machine to only execute one vector instruction at a time [26]. Asanović [13] proposed a decoupled vector pipeline design that issues vector instructions to the vector datapath only when all addresses from previous vector memory instructions are known to not cause an exception. Espasa et al. [42] and Kozyrakis [65] renamed vector destination registers to implement precise exceptions. Once a vector instruction faults, the destination registers of all subsequent vector instructions are rolled back to the previous mapping to maintain preciseness. Since the Hwacha vector machine eschews vector register renaming, it must allow partial completion of more than one vector instruction, at the expense of larger architectural state.

Hampton [52] presented software restart markers as a foundation to handle exceptions in parallel architectures. The compiler is responsible for delimiting the program into idempotent regions.

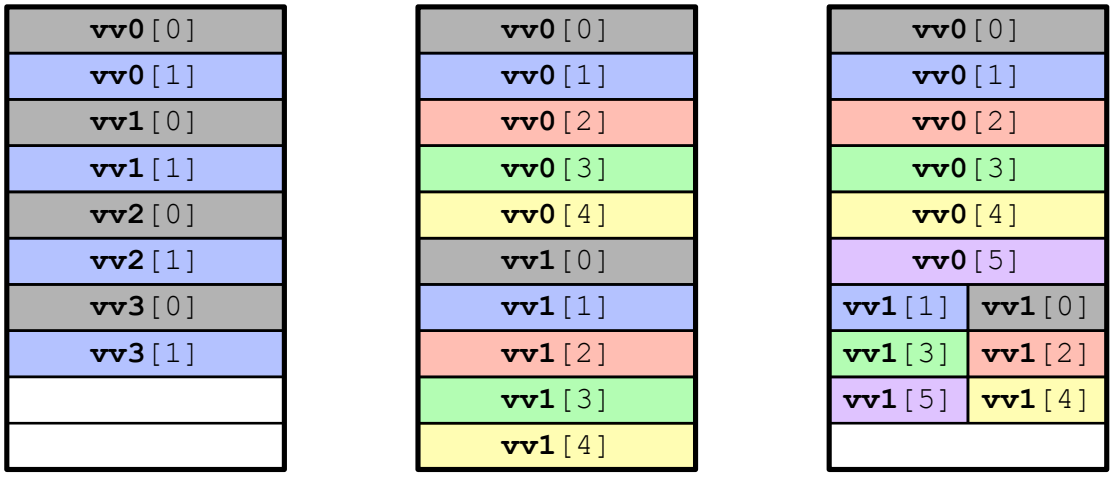
Once an exception occurs, the operating system will simply resume execution from the beginning of the faulting region. Although this approach has very low implementation overhead, it is constrained by the ability of the compiler to statically determine the idempotency of a region, and can hence have large execution overheads on some codes.

Our proposal is most similar to the DEC Vector VAX [24] design, which provided the OS with an opaque microarchitectural state save and restore mechanism, and also provided fence instructions to synchronize vector unit execution with the scalar processor.

Reconfigurable Hardware Vector Length

As the requirements for the vector length and number of vector registers vary from one kernel to another, the Hwacha vector-fetch architecture lets the vector register file and the hardware vector length to be dynamically reconfigured to maximize utilization. As a result, the Hwacha vector-fetch architecture is able to efficiently handle kernels on the both ends of vector register usage. For kernels that use a large number of vector registers, the Hwacha architecture can avoid spilling vector registers by allowing a large number of vector registers in the instruction encoding. For kernels that use a small number of vector registers, the Hwacha architecture can still maintain efficiency by trading less register usage with longer vector lengths.

Figure 6.3 demonstrates how we can dynamically reconfigure the vector register file and adjust the maximum hardware vector length. Assume we have a vector register file with 10 entries. For a kernel that uses 4 vector registers, the maximum hardware vector length is set to 2 (see



(a) `vsetcfg 4` HVL = 2

(b) `vsetcfg 2` HVL = 5

(c) `vsetcfg 1,1` HVL = 6

Figure 6.3: Reconfigurable Vector Register File – The `vsetcfg` instruction reconfigures the 64-bit vector register file and the hardware vector length to reflect the following register usage in a kernel: (a) four 64-bit vector registers, (b) two 64-bit vector registers, (c) one 64-bit vector register and one 32-bit vector register. HVL = hardware vector length.

Figure 6.3a). When the number of used vector registers halves to 2, the maximum hardware vector length is adjusted to 5 (see Figure 6.3b).

This register reconfiguration is similar to that in the early Fujitsu VP100/200 machines, but is more flexible, as neither the number of registers nor the vector length have to be a power of two. NVIDIA GPUs also adjust the number of warps that are scheduled to an individual streaming multiprocessor depending on the register usage of a kernel. This coarse-grain quantization at the warp level, however, is often known to result in pathological cases in which the occupancy drops dramatically with a slight change in register usage [73]. The Hwacha vector-fetch architecture is less sensitive to this problem as the configuration is done at a finer granularity.

Mixed-Precision Support

Many applications exhibit a broad variation in data value widths. Fixed-function accelerators attain significantly better energy efficiency and performance by exploiting minimal and heterogeneous word widths in their custom datapaths. By contrast, processors must support a range of conventional datatype widths to fulfill a general-purpose role. Thus, the datapath is typically fixed at the maximum precision potentially employed by any application. For certain applications (e.g., multimedia and signal processing) that do not require the highest available precision, unnecessary energy is expended obtaining an equivalent result.

Rarely is one global precision optimal throughout all stages of computation; for example, the widths of addresses and integer data often differ, and in widening arithmetic operations such as a fused multiply-add (FMA), the product of n-bit values is added to a $2n$-bit accumulator without intermediate rounding. For versatility, a processor should be able to simultaneously intermix several reduced-precision modes according to application-specific conditions.

The Hwacha vector-fetch architecture supports mixed-precision computation. Due to the intrinsic data independence, the wide datapath is naturally partitioned into multiple narrower elements. Compaction improves the utilization of vector register file accesses and interconnection fabric for operand communication, and additional vector functional units can be introduced to leverage subword parallelism with relatively small area cost. Most importantly, increased throughput enables faster race-to-halt into a low power state. Moreover, denser storage of elements lessens memory pressure and allows for longer vectors with the same register file capacity. The expanded buffering assists with decoupled execution in more effectively hiding memory latency.

The `vsetcfg` instruction is modified to also take into account the number of vector registers with narrower precision. Figure 6.3c depicts the case in which the programmer requested one 64-bit vector register and one 32-bit vector register. Assuming the vector register file has 10 entries, the maximum hardware vector length will be set to 6 instead of 5 (the case depicted in Figure 6.3b in which the programmer requested two 64-bit vector registers).

Asanović [13] and Kozyrakis [65] similarly describe how a vector machine can be treated as an array of *virtual processors* whose datapath widths are set collectively through a virtual processor width (VPW) register. However, the precision of an individual vector register cannot be configured independently, restricting fine-grain mixing of different data types.

Packed-SIMD extensions naturally support mixed-precision values within vector registers that are uniformly fixed in size. Consequently, the number of elements per vector register varies at different precisions. For example, a 512-bit AVX register can store 8 double-precision, 16 single-precision, and 32 half-precision floating-point numbers. However, this makes it difficult for the vector microarchitecture to chain mixed-precision operations. Equalizing vector lengths might involve splitting a vector across several architectural registers, depleting register encoding space as a result.

Some GPUs and microcontrollers implement limited support for variable precision by storing wider datatypes across several narrower architectural registers. For example, double-precision floating-point values might occupy pairs of 32-bit registers and be referenced solely through even register specifiers, effectively halving the available register set. Newer generation GPUs such as the NVIDIA Tegra X1 [95], AMD GCN3 [6], ARM Mali Midgard architecture [116, 11], and Imagination PowerVR Rogue architecture [117] have added native support for half-precision floating-point operations. However, these new GPU architectures expose half-precision floating-point operations as packed-SIMD instructions, making the assembly code less compatible across different generations.

Ou's master thesis [99] has more details on how the Hwacha vector-fetch architecture supports mixed-precision values and operations.

6.3 History

The Hwacha vector-fetch architecture builds on several earlier projects: T0, Scale, Maven, and earlier versions of Hwacha. This section outlines the lineage and history of vector machines that influenced the current Hwacha vector-fetch architecture design.

Lineage

The T0 (Torrent-0) vector microprocessor project at UC Berkeley and ICSI begun in 1992 with Krste Asanović as the lead architect and RTL designer, and Brian Kingsbury and Bertrand Irrisou as main VLSI designers [14, 16, 12, 13]. T0 was a vector processor based on the MIPS-II ISA, implemented in Hewlett-Packard's CMOS26G 1.0 um CMOS process with 2 metal layers. The resulting $16.75 \times 16.75\, mm^2$ chip operated at a maximum frequency of 45 MHz at 5 V and consumed less than 12 W. The T0 vector machine had a custom-designed 5-read-3-write register file that contained 16 vector registers, each holding 32×32-bit elements, split across 8 vector lanes each with two dynamically reconfigurable vector arithmetic pipelines, and interfaced with external SRAM as main memory.

The Scale (Software-Controlled Architecture for Low Energy) vector-thread architecture project at MIT begun in 2000 with Ronny Krashinsky and Christopher Batten as lead architects [68, 22, 66, 67]. The Scale vector processor had an SMIPS control processor (SMIPS stands for Scale MIPS, a subset of MIPS without the branch delay slot), and a vector-thread unit with 4 lanes each with 4 clusters that featured different types of execution resources. The instructions that ran on the

vector-thread unit were grouped into an *atomic instruction block* (AIB). All instructions in an AIB were executed before moving on to the next element in the vector. Scale included a do-across network for efficient cross-element communication, which were particularly useful when vectorizing loops with loop-carried dependencies. Scale implemented a cache refill-access decoupling scheme to hide memory latency. Scale did not support floating-point operations in hardware. The Scale chip was implemented in TSMC CL018G 180 nm CMOS process with 6 metal layers. The resulting 23.14 mm^2 chip operated at a maximum frequency of 260 MHz at 1.8 V consuming 0.4–1.1 W, depending on the workload.

The Maven (Malleable Array of Vector-thread ENgines) vector-thread architecture project at UC Berkeley and MIT begun in 2007 with Christopher Batten as the lead architect, and Yunsup Lee and Rimas Avizienis as main designers of the Maven vector-thread unit and scalar units, respectively [21, 72, 70, 71]. The goal of the Maven project was to explore the programmability and efficiency of a wide range of data-parallel accelerators including the MIMD, traditional vector, and the newly proposed Maven vector-thread architecture. The group designed a flexible RTL framework that instantiated all designs, which was then pushed through the VLSI flow to obtain accurate area, performance, and power/energy numbers to compare. The Maven design was never fabricated, however, many VLSI layouts were produced using the TSMC 65GPLUS 65 nm CMOS process with 9 metal layers. The sizes of the resulting accelerators were around 4–6.3 mm^2, and operated at a maximum frequency of 680–763 MHz at 1 V consuming 111–331 mW. The Maven vector-thread unit was configurable to have 1, 2, or 4 lanes, each with a vector register file that was optionally banked 1, 2, or 4 ways. Vector arithmetic instructions were packed into separate vector-fetch blocks to let the vector memory instructions run ahead and prefetch the needed vector data. Branch instructions were allowed in vector-fetch blocks to support kernels with irregular control flow, and were implicitly handled by the hardware (meaning that the hardware was responsible for the bookkeeping the divergence state). Explicit predication was not supported. Floating-point operations were supported in hardware.

Earlier versions of Hwacha

The Hwacha project started right after the Maven project in 2011. The first version of Hwacha had a similar assembly programming model as Maven, where the vector memory instructions were kept in the control processor's instruction stream, and the vector arithmetic instructions were hoisted out into a separate vector-fetch block. Branches were not allowed in vector-fetch blocks, and predication was not supported. Conditional move instructions were the only way to write data-dependent execution. The Hwacha project used the 64-bit RISC-V ISA [131, 129] as its base ISA, moving away from the Maven ISA. The vector microarchitecture changed significantly, where the vector lane was mainly redesigned to work with a banked vector register file that was split into 8 banks of area-efficient 1-read-1-write SRAM macros. The RTL was written from scratch in Verilog.

The second version of Hwacha was rewritten from scratch in an early version of Chisel [17], and mainly added support for virtual memory and restartable exceptions [127]. The vector runahead unit was rearchitected to prefetch vector data into the nearest cache as a result.

The third version of Hwacha was rewritten from scratch in Chisel. This was a result of designers learning better ways to express hardware designs in Chisel. The control logic and the vector memory unit were mostly rewritten in a cleaner way. The vector memory unit interfaced with the L2 cache directly rather than the L1 data cache.

Current version of Hwacha

This thesis documents the fourth version of the Hwacha vector-fetch architecture. The assembly programming model has changed to put all vector instructions into the vector-fetch block. This version supports full predication on all vector instructions and consensual branches. The RTL for the entire machine has been rewritten from scratch in Chisel. The vector lane has been rearchitected to use four 128-bit SRAM macros as opposed to eight 64-bit macros that were used in the previous versions to double the functional-unit throughput. Reduction operations, and variable latency operations such as floating-point divide and square root, integer divide and remainder operations are also supported in hardware.

Chapter 7

Hwacha Instruction Set Architecture

This chapter introduces the Hwacha instruction set architecture (ISA), developed as a non-standard extension to the free and open RISC-V ISA [131, 132, 130, 129]. The Hwacha ISA is a vector load-store architecture; to perform a compute operation, operands must be read and written to registers. The Hwacha ISA defines instructions for the control thread and the worker thread. The control thread is mapped to the control processor and is responsible for configuring the Hwacha vector unit, kicking off work to the Hwacha vector unit, and interacting with the operating system. The control thread instructions are 32 bits in length and are a greenfield extension in the CUSTOM major opcode space (i.e., overlaid on top of normal RISC-V instructions). The worker thread is mapped to the Hwacha vector unit and consists of vector, vector-scalar, and scalar instructions that carry out the actual data-parallel computation. The worker thread instructions are 64 bits in length and respect the variable-length RISC-V encoding. Section 7.1 and 7.2 lists the Hwacha instructions for the control thread and worker thread, respectively. Section 7.3 outlines future research directions.

7.1 Control Thread Instructions

There are three types of Hwacha control thread instructions—vector configuration, vector move, and vector-fetch instructions. Vector configuration instructions configure the Hwacha vector unit, vector move instructions transfer scalar values from the control thread to the worker thread, and the vector-fetch instruction kicks off a sequence of worker thread instructions. Control thread instructions follow the standard *RoCC* (Rocket Custom Coprocessor) instruction format shown in Figure 7.1.

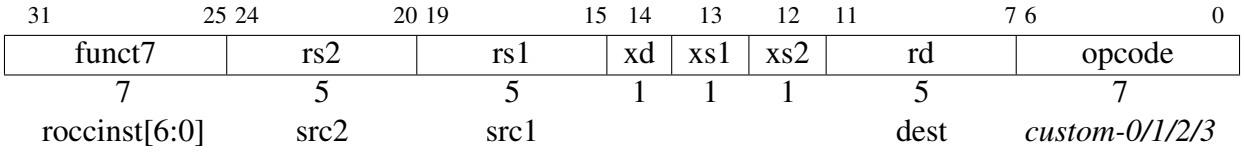


Figure 7.1: RoCC Instruction Format – The Hwacha control thread instruction format follows the standard RoCC instruction format.

Instruction	Opcode	Meaning
`vsetcfg rs1, imm[11:0]`	*custom-0*	Set maximum hardware vector length
`vsetvl rd, rs1`	*custom-0*	Set hardware vector length
`vgetcfg rd`	*custom-0*	Get vector unit configuration
`vgetvl rd`	*custom-0*	Get hardware vector length
`vuncfg`	*custom-0*	Unconfigure vector unit
`vmcs srd, rs1`	*custom-1*	Move value to vector shared register (`vs`*0–63*)
`vmca ard, rs1`	*custom-1*	Move value to vector address register (`va`*0–31*)
`vf rs1, imm[11:0]`	*custom-1*	Execute a block of worker thread instructions

Table 7.1: Listing of Hwacha Control Thread Instructions – The instructions are organized into 3 groups starting from the top: vector configuration instructions, vector move instructions, and the vector-fetch instruction.

Table 7.1 summarizes the instruction format, opcode, and the meaning of vector configuration instructions, vector move instructions, and the vector-fetch instruction. We omit the encoding details for the control thread instructions in this thesis for brevity; the actual control thread instruction encodings can be found in the Hwacha vector-fetch architecture manual [77].

Vector Configuration Instructions

VSETCFG configures the vector unit with a 64 bit constant built with the top 52 bits of the source register *rs1* combined with the 12 bit immediate value at the lowest 12 bits. The 64 bit configuration register `vcfg` layout is shown in Figure 7.2.

The assembler will take VSETCFG *#v64, #pred2:0*, or VSETCFG *#v64, #pred, #v32, #v16*, which will also generate sequence of instructions to build the corresponding 64 bit configuration object. *#v64* and *#pred* denotes the number of 64-bit vector and predicate registers used in the program, respectively. For further optimization, the number of 32-, 16-bit vector registers may be specified in the *#v32*, and *#v16* fields, respectively. These values can range from 0 to 256 for the number of vector data (`vv`) registers and 0 to 16 for the number of vector predicate (`vp`) registers. Once the vector unit is reconfigured, the maximum hardware vector length may be adjusted, and the vector length register is reset to 0. VSETVL sets the vector length register by taking the minimum

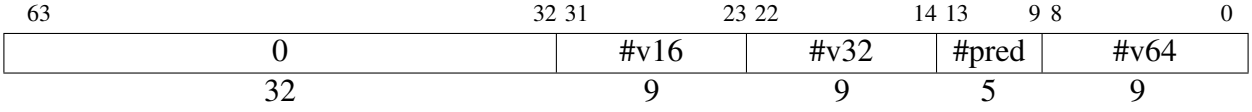


Figure 7.2: Layout of the `vcfg` Configuration Register – *#v64*, *#v32*, *#v16*, and *#pred* fields denote the number of 64-, 32-, and 16-bit vector data registers (`vv`*0–255*) and vector predicate registers (`vp`*0–15*), respectively.

of the requested vector length in register *rs1* and the maximum hardware vector length, and writes the set vector length to register *rd*.

VGETCFG writes the content of the vector configuration register (`vcfg`) to register *rd*. Similarly, VGETVL writes the content of the vector length register (`vlen`) to register *rd*.

VUNCFG clears the vector configuration register and sets the vector unit as unused. The vector unit must be configured before any subsequent vector instructions are issued. The vector unit begins operation in this unconfigured state. If the machine has not been configured since reset or the most recent VUNCFG, then any RoCC control thread instruction other than VSETCFG will trigger an accelerator disabled exception.

Vector Move Instructions

VMCS (Vector Move Control to Scalar) moves the value in register *rs1* to vector shared (`vs`) register *srd*. VMCA (Vector Move Control to Address) moves the value in register *rs1* to vector address (`va`) register *ard*.

Vector Fetch Instruction

The VF instruction executes a block of worker thread instructions in a vector-fetch block that resides at the target address (`vpc`), which is obtained by adding the 12-bit signed immediate to register *rs1*. The vector-fetch block contains vector instructions, vector-scalar instructions, and scalar instructions that operate on vector data (`vv`) registers, vector predicate (`vp`) registers, vector shared (`vs`) registers, and vector address (`va`) registers. The worker thread instructions are listed in the next section.

7.2 Worker Thread Instructions

The worker thread instructions are grouped into vector-fetch blocks. The first instruction of a vector-fetch block is pointed to by a vector-fetch instruction, which lives in the control thread's instruction stream.

Figure 7.3 depicts the four core instruction formats (VJ/VU/VI/VR) for the Hwacha worker thread instructions. The VR4 format extends the VR format with a third register specifier *rs3* for floating-point fused-multiply-add instructions. All worker thread instructions are 64 bits in length and must be aligned on a eight-byte boundary in memory. An instruction address misaligned exception is generated if the `vpc` is not eight-byte aligned on an instruction fetch. Following the RISC-V instruction format, the source register specifiers (*rs1* and *rs2*), destination register specifier (*rd*), and the predicate register specifier (*p*) are kept at the same position for all instruction formats to simplify decoding. Immediates are left aligned.

When the *d* flag (bit 63) is set, register *rd* is interpreted as a vector data register (*vd*). When it is cleared, register *rd* is interpreted as a vector shared register (*sd*). Similarly, the *1* flag (bit 62), the *2* flag (bit 61), and the *3* flag (bit 60) indicates whether *rs1*, *rs2*, and *rs3* refers to vector data register

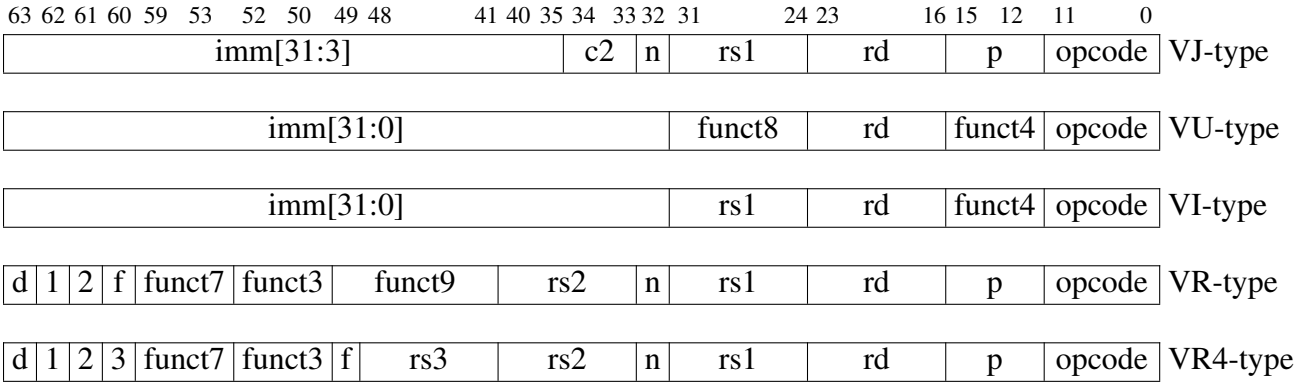


Figure 7.3: Hwacha Worker Thread Instruction Formats – There are four core instruction formats for the 64-bit Hwacha worker thread instructions, with the VR4 format extending the VR format with a third register specifier *rs3*.

inst[9:7] / inst[11:10]	000	001	010	011	100	101	110	111
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	*reserved*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	*reserved*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*custom-5*	*custom-2/rv128*	*reserved*
11	CTRL	JALR	*custom-4*	JAL	SYSTEM	*custom-6*	*custom-3/rv128*	*reserved*

Table 7.2: Hwacha Worker Thread Instruction Opcode Map – Hwacha worker thread instructions are 64 bits in length and respects the variable-length RISC-V encoding (inst[6:0]=0111111).

specifiers (*vs1*, *vs2*, and *vs3*) or vector shared register specifiers (*ss1*, *ss2*, and *ss3*) respectively. Certain instructions index vector address registers instead, in which case use *as1* and *as2* register specifiers. Note, the *ad* register specifier should never show up in worker thread instructions, since the vector address registers cannot be written by the worker thread.

An instruction with all source and destination operands marked as vector shared registers is considered a scalar instruction (with a `@s` prefix in the assembly), and therefore does not need to execute in vector fashion. An instruction with any operand marked as a vector data register is considered a vector instruction, and will execute for `vlen` elements. The `vlen` register is set by the VSETVL control thread instruction or cleared by the VSETCFG control thread instruction. The *p* field (bits 12–15) designates a predicate register that masks the vector instruction. The *n* flag (bit 32) negates the condition of the appointed predicate. Instructions that are not masked on a predicate are given the `@all` prefix in assembly.

Table 7.2 shows the major opcode allocation for the worker thread instructions. Major opcodes are given 12 bits, however, the lower 7 bits are fixed to `0111111` to respect the RISC-V variable-length encoding for 64 bit instructions. This leaves 5 bits or 32 major opcodes to use. The worker thread instructions consume 21 of these 32 major opcodes, and roughly follow the RISC-V major opcode allocation. We also omit the encoding details for the worker thread instructions in this

thesis for brevity; the actual worker thread instruction encodings can be found in the Hwacha vector-fetch architecture manual [77].

Vector Unit-Strided, Constant-Strided Memory Instructions

Vector unit-strided and constant-strided load and store instructions transfer values between vector data (vv) registers and memory. Unit-strided vector memory instructions transfer vectors whose elements are held in contiguous locations in memory. Constant-strided vector memory instructions transfer vectors whose elements are held in memory addresses that form an arithmetic progression. Table 7.3 lists all vector unit-strided and constant-strided memory instructions.

Instruction	Format	Meaning
`@[!]p vlb vd, as1`	VR	Vector unit-strided load byte, signed
`@[!]p vlbu vd, as1`	VR	Vector unit-strided load byte, unsigned
`@[!]p vlh vd, as1`	VR	Vector unit-strided load half-word, signed
`@[!]p vlhu vd, as1`	VR	Vector unit-strided load half-word, unsigned
`@[!]p vlw vd, as1`	VR	Vector unit-strided load word, signed
`@[!]p vlwu vd, as1`	VR	Vector unit-strided load word, unsigned
`@[!]p vld vd, as1`	VR	Vector unit-strided load double-word
`@[!]p vsb vd, as1`	VR	Vector unit-strided store byte
`@[!]p vsh vd, as1`	VR	Vector unit-strided store half-word
`@[!]p vsw vd, as1`	VR	Vector unit-strided store word
`@[!]p vsd vd, as1`	VR	Vector unit-strided store double-word
`@[!]p vlstb vd, as1, as2`	VR	Vector constant-strided load byte, signed
`@[!]p vlstbu vd, as1, as2`	VR	Vector constant-strided load byte, unsigned
`@[!]p vlsth vd, as1, as2`	VR	Vector constant-strided load half-word, signed
`@[!]p vlsthu vd, as1, as2`	VR	Vector constant-strided load half-word, unsigned
`@[!]p vlstw vd, as1, as2`	VR	Vector constant-strided load word, signed
`@[!]p vlstwu vd, as1, as2`	VR	Vector constant-strided load word, unsigned
`@[!]p vlstd vd, as1, as2`	VR	Vector constant-strided load double-word
`@[!]p vsstb vd, as1, as2`	VR	Vector constant-strided store byte
`@[!]p vssth vd, as1, as2`	VR	Vector constant-strided store half-word
`@[!]p vsstw vd, as1, as2`	VR	Vector constant-strided store word
`@[!]p vsstd vd, as1, as2`	VR	Vector constant-strided store double-word

Table 7.3: Listing of Vector Unit-Strided, Constant-Strided Memory Instructions – The base address is stored in vector address register *as1* and the constant stride is stored in vector address register *as2*. Vector load instructions copy values from memory into vector data register *vd*, while vector store instructions copy values in vector data registers *vd* to memory.

The vector predicate register *p* in conjunction with the *n* flag masks the vector memory instruction. Vector load instructions copy a vector of values from memory to vector register *vd*. Vector store instructions copy a vector of values in vector register *vd* to memory. The base address for the memory transfer is taken from vector address register *as1*. Vector address register *as2* holds the stride for the constant-stride vector memory operations. Following the RISC-V ISA, the vector memory instructions support four data widths (byte, half-word, word, and double-word) and two flavors of load instructions: one that sign-extends the load results and the other that zero-extends the load results.

Vector Indexed Memory Instructions

Vector indexed load and store instructions transfer vectors whose elements are located at offsets from a base address, with the offsets specified by the values of an index vector. The effective address of each element is the base address plus the offset register. These instructions are traditionally known as vector gathers and scatters. Table 7.4 lists all vector indexed memory instructions.

The base address for the indexed memory operation is taken from vector shared register *ss1*, while the offset from the base address is taken from vector data register *vs2*. The vector indexed memory operations are masked with vector predicate register *p* and the *n* flag, and support four data widths and two types of sign-extension for loads (same as strided vector memory operations).

Instruction	Format	Meaning
`@[!]p vlxb vd, ss1, vs2`	VR	Vector indexed load byte, signed
`@[!]p vlxbu vd, ss1, vs2`	VR	Vector indexed load byte, unsigned
`@[!]p vlxh vd, ss1, vs2`	VR	Vector indexed load half-word, signed
`@[!]p vlxhu vd, ss1, vs2`	VR	Vector indexed load half-word, unsigned
`@[!]p vlxw vd, ss1, vs2`	VR	Vector indexed load word, signed
`@[!]p vlxwu vd, ss1, vs2`	VR	Vector indexed load word, unsigned
`@[!]p vlxd vd, ss1, vs2`	VR	Vector indexed load double-word
`@[!]p vsxb vd, ss1, vs2`	VR	Vector indexed store byte
`@[!]p vsxh vd, ss1, vs2`	VR	Vector indexed store half-word
`@[!]p vsxw vd, ss1, vs2`	VR	Vector indexed store word
`@[!]p vsxd vd, ss1, vs2`	VR	Vector indexed store double-word

Table 7.4: Listing of Vector Indexed Memory Instructions – The base address is stored in vector shared register *ss1* and the offset from the base address is stored in vector data register *vs2*. Vector load instructions copy values from memory into vector data register *vd*, while vector store instructions copy values in vector data registers *vd* to memory.

Vector Atomic Memory Instructions

Vector atomic memory operation (AMO) instructions perform a vector of read-modify-write operations. Table 7.5 lists all vector AMO instructions.

Similar to AMO operations defined by the RISC-V ISA, the vector AMO instructions atomically load data values from addresses in register *rs1*, place the value into vector data register *vd*, apply the binary operator to loaded values and values in register *rs2*, then store the result back to addresses in register *rs1*. Both register specifiers *rs1* and *rs2* refer to vector data registers *vs1* and *vs2* or vector shared registers *ss1* and *ss2* depending on whether the *1* flag and the *2* flag are set, respectively. When *ss1* is used for the address, the AMO operation is preformed `vlen` times on the same address. Similarly, when *ss2* is used for the AMO operand, the AMO operation is performed `vlen` times with the same data. The result is always stored in vector data register *vd*. The vector AMO instruction is masked with vector predicate register *p* and the *n* flag.

Following the RISC-V ISA, instructions with the `w` suffix operate on words, while instructions with the `d` suffix operate on double-words. The supported AMO operations are swap, integer add, logical AND, logical OR, logical XOR, and signed and unsigned integer minimum and maximum. The vector AMO instructions optionally provide release consistency semantics at the vector instruction granularity with the *aq* bit and the *rl* bit. The "A" standard extension chapter of the RISC-V user-level ISA specification [131] has more details on the release consistency semantics.

Instruction	Format	Meaning
`@[!]p vamoswap.{w|d} vd, (rs1), rs2`	VR	Vector atomic swap
`@[!]p vamoadd.{w|d} vd, (rs1), rs2`	VR	Vector atomic addition
`@[!]p vamoand.{w|d} vd, (rs1), rs2`	VR	Vector atomic bitwise AND
`@[!]p vamoor.{w|d} vd, (rs1), rs2`	VR	Vector atomic bitwise OR
`@[!]p vamoxor.{w|d} vd, (rs1), rs2`	VR	Vector atomic bitwise XOR
`@[!]p vamomin.{w|d} vd, (rs1), rs2`	VR	Vector atomic 2's comp. minimum
`@[!]p vamomax.{w|d} vd, (rs1), rs2`	VR	Vector atomic 2's comp. maximum
`@[!]p vamominu.{w|d} vd, (rs1), rs2`	VR	Vector atomic unsigned minimum
`@[!]p vamomaxu.{w|d} vd, (rs1), rs2`	VR	Vector atomic unsigned maximum

Table 7.5: Listing of Vector Atomic Memory Instructions – The address is stored in register *rs1* (vector data register *vs1* or vector shared register *ss1*) and the value sent to the memory is stored in register *rs2* (*vs2* or *ss2*). The result of the atomic memory operation (AMO) is stored in vector data register *vd*.

Vector Integer Compute Instructions

Vector integer compute instructions take two input operands and produce one output operand. Table 7.6 lists all vector integer compute instructions. The vector integer compute instructions are vector versions of RV64IM instructions, therefore follow the same RISC-V instruction semantics.

Instruction		Format	Meaning
`@[!]p veidx`	`vd, rs1`	VR	Vector return element index
`@[!]p vadd`	`rd, rs1, rs2`	VR	Vector add registers
`@[!]p vsub`	`rd, rs1, rs2`	VR	Vector subtract registers
`@[!]p vsll`	`rd, rs1, rs2`	VR	Vector shift left logical by register
`@[!]p vsrl`	`rd, rs1, rs2`	VR	Vector shift right logical by register
`@[!]p vsra`	`rd, rs1, rs2`	VR	Vector shift right arithmetic by register
`@[!]p vand`	`rd, rs1, rs2`	VR	Vector bitwise AND by register
`@[!]p vor`	`rd, rs1, rs2`	VR	Vector bitwise OR by register
`@[!]p vxor`	`rd, rs1, rs2`	VR	Vector bitwise XOR by register
`@[!]p vslt`	`rd, rs1, rs2`	VR	Vector set if less than, 2's comp
`@[!]p vsltu`	`rd, rs1, rs2`	VR	Vector set if less than, unsigned
`@[!]p vmul`	`rd, rs1, rs2`	VR	Vector multiply, return lower bits
`@[!]p vmulh`	`rd, rs1, rs2`	VR	Vector multiply signed, return upper bits
`@[!]p vmulhu`	`rd, rs1, rs2`	VR	Vector multiply unsigned, return upper bits
`@[!]p vmulhsu`	`rd, rs1, rs2`	VR	Vector multiply signed-unsigned, return upper bits
`@[!]p vdiv`	`rd, rs1, rs2`	VR	Vector signed division
`@[!]p vdivu`	`rd, rs1, rs2`	VR	Vector unsigned division
`@[!]p vrem`	`rd, rs1, rs2`	VR	Vector signed remainder
`@[!]p vremu`	`rd, rs1, rs2`	VR	Vector unsigned remainder
`@[!]p vaddw`	`rd, rs1, rs2`	VR	Vector add registers, 32-bit
`@[!]p vsubw`	`rd, rs1, rs2`	VR	Vector subtract registers, 32-bit
`@[!]p vsllw`	`rd, rs1, rs2`	VR	Vector shift left logical by registers, 32-bit
`@[!]p vsrlw`	`rd, rs1, rs2`	VR	Vector shift right logical by registers, 32-bit
`@[!]p vsraw`	`rd, rs1, rs2`	VR	Vector shift right arithmetic by registers, 32-bit
`@[!]p vmulw`	`rd, rs1, rs2`	VR	Vector multiply, 32-bit
`@[!]p vdivw`	`rd, rs1, rs2`	VR	Vector signed division, 32-bit
`@[!]p vdivuw`	`rd, rs1, rs2`	VR	Vector unsigned division, 32-bit
`@[!]p vremw`	`rd, rs1, rs2`	VR	Vector signed remainder, 32-bit
`@[!]p vremuw`	`rd, rs1, rs2`	VR	Vector unsigned remainder, 32-bit

Table 7.6: Listing of Vector Integer Compute Instructions – These instructions carry out the integer computation by taking one or two input operands from register *rs1* (vector data register *vs1* or vector shared register *ss1*) and register *rs2* (*vs2* or *ss2*), and storing the result in register *rd* (*vd* or *sd*).

The only exception is the vector element index instruction (`veidx`), which takes one input operand and returns the element index left shifted by the value from the second input operand.

The vector predicate register *p* in conjunction with the *n* flag masks the vector instruction. Flags *d*, *1*, and *2* indicate whether registers *rd*, *rs1*, and *rs2* should be taken from vector data (`vv`) registers (if set) or vector shared (`vs`) registers (if cleared). When all source and destination operands are marked to use vector shared registers, the instruction is decoded as a scalar instruction. For this case, the *p* register should be set to zero and the *n* field should be cleared, otherwise, an illegal instruction exception will be raised.

Vector Reduction Instructions

Vector reduction instructions perform a reduction operation on the vector input operand and produce one scalar output operand. Table 7.7 lists the only vector reduction operation that is defined.

The vector first instruction (`vfirst`) finds the first value of vector data register *vs1* that is not masked off under vector predicate *p* (negated if flag *n* is set) and writes the result in vector shared register *sd*. If the predicate register is entirely empty, the result of the reduction is zero. This instruction can be used in conjunction with a vector compare instruction to iterate through all distinct values in a vector data register.

Instruction	Format	Meaning
`@[!]p vfirst sd, vs1`	VR	Vector return first active element

Table 7.7: Listing of Vector Reduction Instructions – The VFIRST instruction returns the value of the first active element of vector predicate mask *p*. Returns zero when no elements are active.

Vector Floating-Point Compute Instructions

Vector floating-point compute instructions take one, two, or three input operands and produce one output operand. Table 7.8 and 7.9 list all vector floating-point compute instructions. These instructions use either the VR-type format or the VR4-type format depending on the number of input operands. The vector floating-point compute instructions are vector versions of RV64FD instructions, therefore follow the same RISC-V instruction semantics. On top of the single- and double-precision vector floating-point operations, half-precision versions of the vector floating-point compute instructions are also available. Since the integer and floating-point values are stored in the same vector data (`vv`) registers, vector versions of the `fmv` instructions, which move values between RISC-V `x` registers and RISC-V `f` registers, are omitted.

Similar to vector integer compute instructions, flags *d*, *1*, *2*, and *3* indicate whether operands should be taken from vector data (`vv`) registers or vector shared (`vs`) registers. The instruction is decoded as a scalar instruction when all input and output operands are marked to use vector shared registers. Otherwise, the instruction is decoded as a vector instruction, and is masked under vector predicate register *p* and the *n* flag.

Instruction		Format	Meaning
`@[!]p vfadd.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP add registers
`@[!]p vfsub.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP subtract registers
`@[!]p vfmul.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP multiply registers
`@[!]p vfdiv.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP divide registers
`@[!]p vfsqrt.{d|s|h}`	`rd, rs1`	VR	Vector FP square root
`@[!]p vfmadd.{d|s|h}`	`rd, rs1, rs2, rs3`	VR4	Vector FP `rs1`×`rs2`+`rs3`
`@[!]p vfmadd.{d|s|h}`	`rd, rs1, rs2, rs3`	VR4	Vector FP `rs1`×`rs2`+`rs3`
`@[!]p vfnmsub.{d|s|h}`	`rd, rs1, rs2, rs3`	VR4	Vector FP −(`rs1`×`rs2`−`rs3`)
`@[!]p vfnmsub.{d|s|h}`	`rd, rs1, rs2, rs3`	VR4	Vector FP −(`rs1`×`rs2`−`rs3`)
`@[!]p vfsgnj.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP inject sign
`@[!]p vfsgnjn.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP inject comp of sign
`@[!]p vfsgnjx.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP multiply signs
`@[!]p vfmin.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP select minimum
`@[!]p vfmax.{d|s|h}`	`rd, rs1, rs2`	VR	Vector FP select maximum
`@[!]p vfclass.{d|s|h}`	`rd, rs1`	VR	Vector FP classify value

Table 7.8: Listing of Vector Floating-Point Compute Instructions – These instructions carry out the floating-point computation by taking one, two, or three input operands from register *rs1* (vector data register *vs1* or vector shared register *ss1*), register *rs2* (*vs2* or *ss2*), register *rs3* (*vs3* or *ss3*), and storing the result in register *rd* (*vd* or *sd*). Optionally, a static rounding mode can be specified as the last operand of the instruction.

Instruction		Format	Meaning
`@[!]p vfcvt.d.{s|h}`	`rd, rs1`	VR	Vector FP convert to double
`@[!]p vfcvt.s.{d|h}`	`rd, rs1`	VR	Vector FP convert to single
`@[!]p vfcvt.h.{d|s}`	`rd, rs1`	VR	Vector FP convert to half
`@[!]p vfcvt.w[u].{d|s|h}`	`rd, rs1`	VR	Vector FP convert to [un]signed 32-bit int
`@[!]p vfcvt.{d|s|h}.w[u]`	`rd, rs1`	VR	Vector FP convert from [un]signed 32-bit int
`@[!]p vfcvt.l[u].{d|s|h}`	`rd, rs1`	VR	Vector FP convert to [un]signed 64-bit int
`@[!]p vfcvt.{d|s|h}.l[u]`	`rd, rs1`	VR	Vector FP convert from [un]signed 64-bit int

Table 7.9: Listing of Vector Floating-Point Convert Instructions – These instructions carry out floating-point conversion by taking the input operand from register *rs1* (vector data register *vs1* or vector shared register *ss1*), and storing the result in register *rd* (*vd* or *sd*). Optionally, a static rounding mode can be specified as the last operand of the instruction.

The rounding mode is either defined statically as part of the worker thread instruction, or controlled by the dynamic rounding mode held in the `frm` register. The dynamic rounding mode of the worker thread is inherited from the dynamic rounding mode of the control thread at vector-fetch issuance. Floating point exceptions generated by the worker thread are accrued in the control thread's `fflags` register. Once the control thread reads this register it sees the accrued exceptions in program order. The worker thread is unable to read and write the dynamic rounding mode and the exception flags.

Vector Compare Instructions

Vector compare instructions compare two registers and store the resulting flag into a predicate register. Table 7.10 lists all vector compare instructions.

The vector compare instruction is masked under vector predicate register *p* and the *n* flag, takes two input operands from either vector data registers (*vs1* and *vs2*) or vector shared registers (*ss1*, and stores the result in vector predicate register *pd*.

The supported integer comparisons are set if equal, less than, less than unsigned. Instructions that produce opposite results are omitted, as all vector instructions can take the negated predicate condition as an input by setting the *n* flag. In addition to integer comparisons, a set of floating-point comparisons (if equal, less than, and less than or equal) are supported in different precisions. The vector floating-point comparison instructions are vector versions of RISC-V floating-point comparison instructions, therefore follow the same RISC-V instruction semantics.

Instruction	Format	Meaning
`@[!]p vcmpeq  pd, rs1, rs2`	VR	Vector set if equal
`@[!]p vcmplt  pd, rs1, rs2`	VR	Vector set if less than, 2's comp
`@[!]p vcmpltu pd, rs1, rs2`	VR	Vector set if less than, unsigned
`@[!]p vcmpfeq.{d|s|h} pd, rs1, rs2`	VR	Vector FP set if equal
`@[!]p vcmpflt.{d|s|h} pd, rs1, rs2`	VR	Vector FP set if less than
`@[!]p vcmpfle.{d|s|h} pd, rs1, rs2`	VR	Vector FP set if less than or equal

Table 7.10: Listing of Vector Compare Instructions – These instructions compare two values in register *rs1* (vector data register *vs1* or vector shared register *ss1*) and register *rs2* (*vs2* or *ss2*), and store the result in vector predicate register *pd*.

Vector Predicate Memory Instructions

Vector predicate memory instructions spill and refill contents of the predicate register to and from memory. Table 7.11 lists all vector predicate memory instructions.

These instructions are not predicated, meaning that they always execute under a full mask. The vector predicate load instruction restores the predicate mask from memory, while the store instruction spills the content of the predicate mask to memory. VPS stores out each predicate as a

sign-extended byte, while VPL loads each predicate from the least significant bit of every byte. In both cases, the base address for the memory transfers are taken from vector address register *as1*.

Instruction	Format	Meaning
`@all vpl pd, as1`	VR	Vector load predicate
`@all vps pd, as1`	VR	Vector store predicate

Table 7.11: Listing of Vector Predicate Memory Instructions – Each predicate interfaces with a byte of memory.

Vector Predicate Compute Instructions

Vector predicate compute instructions perform a logic operation on three input predicates and produce an output predicate. Table 7.12 lists all vector predicate compute instructions.

Following the vector predicate memory instructions, vector predicate compute instructions always execute under a full mask (note the `@all` prefix). The VPOP instruction takes an 8 entry truth table that serves as a lookup table for a three-input logic function to build arbitrary logic functions. A handful of well-known logic operations are defined as pseudo instructions in the assembler.

Instruction	Format	Meaning
`@all vpop pd, ps1, ps2, ps3, tt`	VR	Vector predicate operation
`@all vpclear pd`	VR	`pd = false`
`@all vpset pd`	VR	`pd = true`
`@all vpxorxor pd, ps1, ps2, ps3`	VR	`pd = (ps1 ^ ps2) ^ ps3`
`@all vpxoror pd, ps1, ps2, ps3`	VR	`pd = (ps1 ^ ps2) | ps3`
`@all vpxorand pd, ps1, ps2, ps3`	VR	`pd = (ps1 ^ ps2) & ps3`
`@all vporxor pd, ps1, ps2, ps3`	VR	`pd = (ps1 | ps2) ^ ps3`
`@all vporor pd, ps1, ps2, ps3`	VR	`pd = (ps1 | ps2) | ps3`
`@all vporand pd, ps1, ps2, ps3`	VR	`pd = (ps1 | ps2) & ps3`
`@all vpandxor pd, ps1, ps2, ps3`	VR	`pd = (ps1 & ps2) ^ ps3`
`@all vpandor pd, ps1, ps2, ps3`	VR	`pd = (ps1 & ps2) | ps3`
`@all vpandand pd, ps1, ps2, ps3`	VR	`pd = (ps1 & ps2) & ps3`

Table 7.12: Listing of Vector Predicate Compute Instructions – The generic `vpop` instruction takes an 8 entry truth table (`tt`) that serves as a lookup table for a three-input logic function. The instructions that are listed below are pseudo instructions with the truth table set up to implement the given logic function.

Scalar Memory Instructions

Scalar memory instructions transfer values between vector shared (`vs`) registers and memory. Table 7.13 lists all scalar memory instructions.

Scalar load instructions copy the value from memory to vector shared register *sd*. Scalar store instructions copy the value in vector shared register *ss2* to memory. The effective address is obtained from either address register *as1* or shared register *ss1* depending on the instruction type. Scalar memory instructions support four data widths and two types of sign-extensions for load operations (same as vector memory operations).

Instruction	Format	Meaning
`@s vlab sd, as1`	VR	Scalar load byte from, signed, addr
`@s vlabu sd, as1`	VR	Scalar load byte from, unsigned, addr
`@s vlah sd, as1`	VR	Scalar load half-word from, signed, addr
`@s vlahu sd, as1`	VR	Scalar load half-word from, unsigned, addr
`@s vlaw sd, as1`	VR	Scalar load word from, signed, addr
`@s vlawu sd, as1`	VR	Scalar load word from, unsigned, addr
`@s vlad sd, as1`	VR	Scalar load double-word from, signed, addr
`@s vsab as1, ss2`	VR	Scalar store byte from, addr
`@s vsah as1, ss2`	VR	Scalar store half-word from, addr
`@s vsaw as1, ss2`	VR	Scalar store word from, addr
`@s vsad as1, ss2`	VR	Scalar store double-word from, addr
`@s vlsb sd, ss1`	VR	Scalar load byte from, signed, shared
`@s vlsbu sd, ss1`	VR	Scalar load byte from, unsigned, shared
`@s vlsh sd, ss1`	VR	Scalar load half-word from, signed, shared
`@s vlshu sd, ss1`	VR	Scalar load half-word from, unsigned, shared
`@s vlsw sd, ss1`	VR	Scalar load word from, signed, shared
`@s vlswu sd, ss1`	VR	Scalar load word from, unsigned, shared
`@s vlsd sd, ss1`	VR	Scalar load double-word from, signed, shared
`@s vssb ss1, ss2`	VR	Scalar store byte from, shared
`@s vssh ss1, ss2`	VR	Scalar store half-word from, shared
`@s vssw ss1, ss2`	VR	Scalar store word from, shared
`@s vssd ss1, ss2`	VR	Scalar store double-word from, shared

Table 7.13: Listing of Scalar Memory Instructions – There are two types of scalar memory instructions: one that gets the address from vector address register *as1*, the other that gets the address from vector shared register *ss1*.

Scalar Compute Instructions

In addition to scalar compute instructions with all register operands marked to use vector shared (vs) registers, we also define scalar compute instructions that take immediate operands using the VI-type format and the VU-type format. Table 7.14 lists all scalar compute instructions that take a 32 bit immediate operand.

Following the RISC-V ISA, VADDI, VSLTI, VANDI, VORI, VXORI, VSLTI, and VSLTIU instructions operate on the sign-extended 32-bit immediate and the value stored in vector shared register *ss1*, and writes the result vector shared register *sd*. We follow the same RISC-V instruction semantics, and ignore the arithmetic overflow and simply take the low 64 bits of the result. The same holds for VSLLI, VSLRI, and VSRAI operations. The 6-bit *shamt* value is held in the lower bits of the 32-bit immediate. Instructions that produce a 32 bit result (denoted with the w suffix) sign-extends the 32 bit result to 64 bits.

The VLUI instruction loads the 32-bit immediate into the upper 32 bits of vector shared register *sd* and fills the lower 32 bits with all zeros. A sequence of VLUI and VADDI can be used to generate a 64 bit constant, for both integer and floating-point values. The VAUIPC instruction forms a 64 bit constant by shifting the 32 bit immediate to the upper 32 bits, filling in the lower 32 bits with zeros, and then adding the current vpc to this value. The resulting 64 bit constant is written to vector shared register *sd*.

Instruction	Format	Meaning
`@s vaddi sd, ss1, imm[31:0]`	VI	Add imm
`@s vslli sd, ss1, shamt[5:0]`	VI	Shift left logical by imm
`@s vsrli sd, ss1, shamt[5:0]`	VI	Shift right logical by imm
`@s vsrai sd, ss1, shamt[5:0]`	VI	Shift right arithmetic by imm
`@s vandi sd, ss1, imm[31:0]`	VI	Bitwise AND by imm
`@s vori sd, ss1, imm[31:0]`	VI	Bitwise OR by imm
`@s vxori sd, ss1, imm[31:0]`	VI	Bitwise XOR imm
`@s vslti sd, ss1, imm[31:0]`	VI	Set if less than imm, 2's comp
`@s vsltiu sd, ss1, imm[31:0]`	VI	Set if less than imm, unsigned
`@s vaddiw sd, ss1, imm[31:0]`	VI	Add imm, 32-bit
`@s vslliw sd, ss1, shamt[4:0]`	VI	Shift left logical by imm, 32-bit
`@s vsrliw sd, ss1, shamt[4:0]`	VI	Shift right logical by imm, 32-bit
`@s vsraiw sd, ss1, shamt[4:0]`	VI	Shift right arithmetic by imm, 32-bit
`@s vlui sd, imm[31:0]`	VU	Load upper imm
`@s vauipc sd, imm[31:0]`	VU	Add upper imm to vpc

Table 7.14: Listing of Scalar Compute Instructions – These scalar compute instructions with immediate operands complement the scalar compute instructions with all register operands marked to use vector shared (vs) registers.

Control Flow Instructions

There are three types of control flow instructions—stop, fence, and consensual jumps. Table 7.15 lists all control flow instructions.

The VSTOP instruction halts the execution of the current vector-fetch block. The VFENCE instruction orders all memory accesses before and after the fence instruction so that all threads observe any memory operation preceding the fence before they observe any memory operation following the fence.

The vector consensual jump and link (VCJAL) instruction and the vector consensual jump and link register (VCJALR) instruction both use the VJ-type format with a 29-bit immediate, which encodes a signed offset in multiple of 8 bytes. When the consensual condition is met (ALL bits in vector predicate register *p* are set, or ANY bit in vector predicate register *p* is set), the VCJAL instruction stores the address of the instruction following the jump (`vpc`+8) in vector shared register *sd*, and jumps to the target address, which is computed by adding the sign-extended immediate to `vpc`. The indirect consensual jump instruction VCJALR calculates the target address by adding the sign-extended immediate to vector shared register *ss1*, then setting the lowest three bits of the result to zero. Similar to VCJAL, when the consensual condition is met, the address of the instruction following the jump is stored into vector shared register *sd*, and the jump is taken.

Instruction	Format	Meaning
`@all vstop`	VJ	Halt vector-fetch block
`@all vfence`	VJ	Memory ordering fence
`@[!]p vcjal.{all|any} sd, imm[31:3]`	VJ	Jump and link
`@[!]p vcjalr.{all|any} sd, ss1, imm[31:3]`	VJ	Jump and link register

Table 7.15: Listing of Control Flow Instructions – Consensual branch instructions jump to the target address computed by adding the sign-extended immediate value to either the current `vpc` or address in vector shared register *ss1*, and writes `vpc`+8 into vector shared register *sd*.

Exceptions

The worker thread can generate two types of exceptions—misaligned exceptions and illegal instruction exceptions. If the address of a load or store is not aligned with its data width, a misaligned load or store exception will be generated, respectively. If the worker thread instructions are not aligned to an 8 byte boundary, an instruction misaligned exception will be generated.

Illegal instruction exceptions are generated for several reasons. If an instruction is marked to use a vector shared register for its destination register while any of its source registers refer to vector data registers, an illegal instruction exception will be generated. When a vector data register is configured with a precision that is smaller than the specified precision of the instruction, an illegal instruction will be generated. If an instruction attempts to reference a vector register (data

or predicate) that is out of range with respect to the vector register set currently configured by VSETCFG, an illegal instruction exception will be generated.

7.3 Future Research Directions

This section briefly describes some possible directions for future improvements with respect to the Hwacha instruction set architecture.

Polymorphic Instruction Set. The current practice is to assign a distinct opcode for each supported data type, as is the convention in general-purpose ISAs. For example, there are separate instructions VFMADD.D, VFMADD.S, and VFMADD.H to denote double-, single-, and half-precision fused multiply-adds, respectively. For a polymorphic instruction set, the input and output precisions of an operation would instead be determined by the source and destination register specifiers in conjunction with the register width configuration from `vcfg`. Orthogonality in the instruction set could therefore be achieved without excessive consumption of opcode space or encoding complexity. However, these advantages must be weighed against the vector registers losing the ability to hold values narrower than their configured width, which may constrain register reuse by a compiler.

The RISC-V "V" Standard Extension for Vectors. In a traditional vector instruction set architecture, vector instructions are intermingled within the control thread's instruction stream. Shared registers are simply overlaid on top of the control thread's state, and scalar instructions are executed as control thread instructions. Contrast this to the Hwacha instruction set design, in which a separate set of vector shared registers are defined alongside the control thread's state, and a separate instruction stream for the worker thread is present. There are benefits to this Hwacha approach (see Chapter 8), however, we plan to define a traditional vector like instruction set to serve as the standard "V" extension for RISC-V. The vector instructions will be 32 bits in length to ease decoding and to enable simple implementations that will be competitive to packed-SIMD vector machines such as Intel AVX [56] and ARM NEON [10]. RISC-V's `x` and `f` registers will serve as scalar registers that are shared across the entire vector.

Chapter 8

Hwacha Decoupled Vector Microarchitecture

This chapter details the Hwacha decoupled vector microarchitecture that executes the Hwacha instruction set architecture described in Chapter 7. In Section 8.1, we first discuss how we modified the open-source Rocket Chip SoC generator to provide a system architecture that is comparable to the commercially available data-parallel accelerators. We take advantage of the RTL libraries that come with the generator, including the Rocket in-order core that executes the RISC-V instruction set, multiple levels of coherent caches, and the standardized RoCC accelerator interface that is used to attach the Hwacha vector accelerator. We present the overall machine organization in Section 8.2, and describe the details of the vector frontend in Section 8.3, vector runahead unit (VRU) in Section 8.4, vector execution unit (VXU) in Section 8.5, and the vector memory unit (VMU) in Section 8.6. The design space for the Hwacha vector accelerator is described in Section 8.7, including issues that arise with the multi-lane configuration.

8.1 System Architecture

Figure 8.1 illustrates the overall system architecture of the Hwacha vector microprocessor. We use the open-source Rocket Chip SoC generator to elaborate our design [79]. The generator consists of highly parameterized RTL libraries written in Chisel [17]. In this section, we discuss the salient capabilities of the generator that allows us to integrate the Hwacha vector accelerator productively within a modern SoC environment while being efficient and providing a simple assembly programming model.

A tile consists of a Rocket control processor and a RoCC (Rocket Custom Coprocessor) socket. Rocket is a five-stage in-order RISC-V scalar core that interface with its private blocking L1 instruction cache and non-blocking L1 data cache [78]. The RoCC socket provides standardized interfaces for issuing commands to a custom accelerator, and interacting with the memory system. The Hwacha decoupled vector accelerator and its blocking vector instruction cache are designed to fit within the RoCC socket. The control thread and the worker thread of the Hwacha assembly

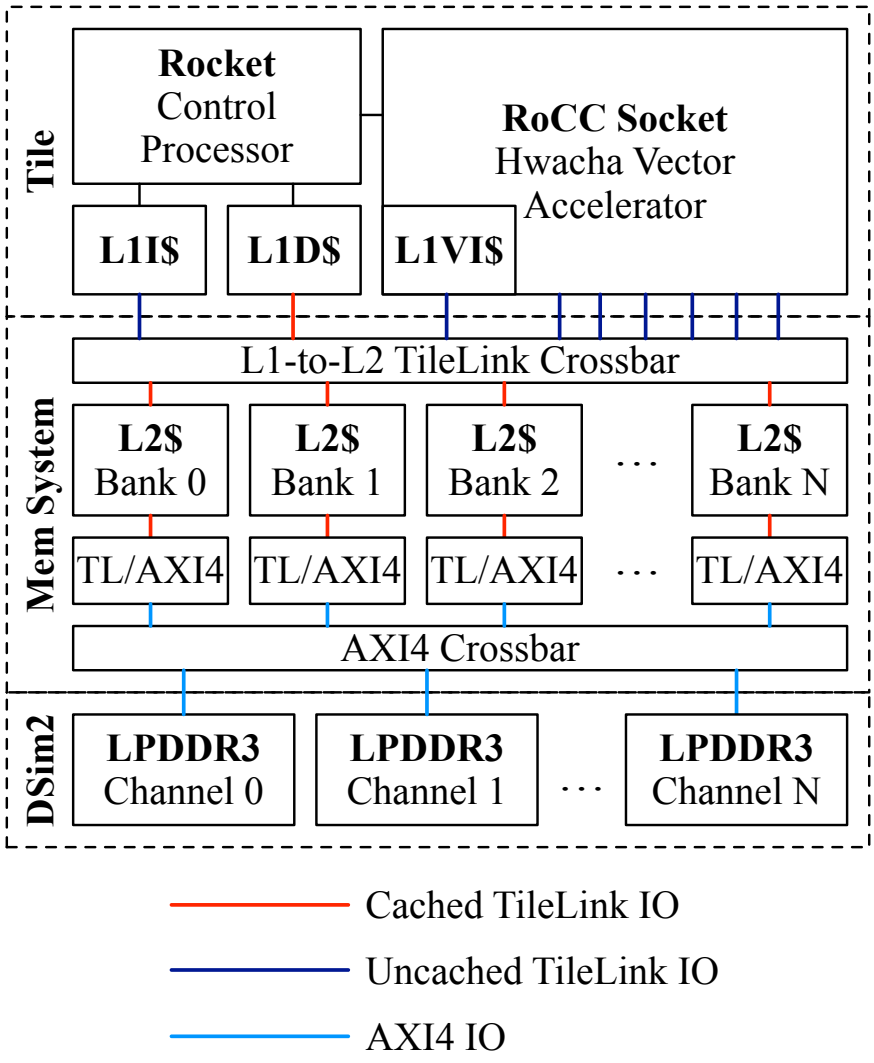


Figure 8.1: System Architecture Provided by the Rocket Chip SoC Generator – The tile consists of the Rocket control processor and the Hwacha vector accelerator that fits within the RoCC (Rocket Custom Coprocessor) socket. The tile is connected to a shared banked L2 cache, which talks to a bank of simulated multi-channel LPDDR3 memory interfaces.

programming model (see Chapter 6.1) are mapped to the Rocket control processor and Hwacha vector accelerator, respectively.

The shared L2 cache is banked, set-associative, and fully inclusive of the L1 caches. Addresses are interleaved at cache line granularity across banks. The tile and L2 cache banks are connected through an on-chip network that implements the TileLink cache coherence protocol [32]. There are two flavors of TileLink IO: cached and uncached. The cached TileLink interface is used by clients that create private copies of cache blocks such as the L1 data cache and L2 cache banks. These cache blocks are kept coherent throughout the memory system. The uncached TileLink interface is

used for clients that do not keep private copies, such as the Hwacha vector unit. Note, instruction caches use uncached TileLink IO, as the cache-coherence protocol does not keep the content of instruction caches coherent with respect to the data stream. The L2 cache banks are coherence master endpoints that implement a cache-coherence protocol, which is selected during elaboration. There is also an option to accelerate the protocol using directory bits that live in the L2 cache tag array.

TileLink offers several capabilities in support of the Hwacha vector accelerator. *Cache coherence* between the L1 data cache and the vector accelerator preserves the shared memory abstraction between the control processor and vector accelerator. This keeps the assembly programming model simple. There is no need to keep two separate address spaces—one for host memory and the other for accelerator's target memory. When the vector accelerator makes a read request to a cache line, the L2 cache bank looks up the directory bit to quickly determine whether the cache line resides in the L1 data cache. If so, the L2 cache bank will then take appropriate steps to guarantee that the L1 data cache does not hold any dirty data. The cache-coherence protocol dictates how the L2 cache bank enforces that invariant. If the cache line has an exclusive state, the L2 cache bank will send a message to the L1 data cache requesting the line to be downgraded to a shared state. If the cache line is already in a shared state, no action is taken. When the vector accelerator makes a write request, the L2 cache bank checks whether the cache line is in a shared or exclusive state, and sends a message to the L1 data cache asking it to drop the line, if necessary. *Sub-cache-block accesses* to words within a cache line reduce memory bandwidth for vector gathers and scatters. *Data prefetch requests* help the system overlap data transfer and computation to achieve better bandwidth utilization. These prefetches are efficiently merged with subsequent sub-block accesses. *Atomic memory operations*, which are performed by ALUs inside each L2 cache bank, offload the work of reduction computations.

The refill ports of the L2 cache banks are connected to a bank of cached- TileLink-IO-to-AXI4 converters. The AXI4 interfaces are then routed to the appropriate LPDDR3 channels through the AXI4 crossbars. The LPDDR3 memory channels are implemented in the testbench, where the DRAM timing is simulated using DRAMSim2 [107].

The memory system parameters, such as the cache size, associativity, cache-coherence protocol, and the number of L2 cache banks and memory channels are set from a configuration object during elaboration. The configuration object also holds design parameters for the Rocket control processor and the Hwacha vector accelerator.

The Hwacha vector accelerator is influenced by several system-level decisions inherent to the Rocket Chip SoC generator. In mapping the control thread to the Rocket scalar core, we exploit vector-fetch decoupling to push the limits of in-order processors. The unified and coherent virtual address space enables restartable exceptions for the Hwacha vector accelerator. By connecting the accelerators to the L2 cache instead of the L1 data cache, we have traded off longer average access latency for substantially higher bandwidth to the cache. However, this decision makes memory access coalescing a more important design feature for the accelerator. Exploiting these built-in SoC generator features allowed us to substantially improve the capabilities of the Hwacha vector accelerator while simultaneously making it simple to apply as much parameter tuning as possible to balance the memory system and the vector accelerator design.

Figure 8.2: Block Diagram of the Hwacha Decoupled Vector Accelerator – The Hwacha decoupled vector accelerator consists of a RoCC unit, scalar unit, vector runahead unit (VRU), master sequencer, multiple vector lanes, each with a vector execution unit (VXU) and a vector memory unit (VMU). VCMDQ = vector command queue, VRCMDQ = vector runahead command queue, FPREQQ = floating-point request queue, FPRESPQ = floating-point response queue.

8.2 Machine Organization

The Hwacha decoupled vector microarchitecture combines ideas from access-execute decoupling [114], decoupled vector architectures [41], and cache refill-access decoupling [22] in order to decouple the data access stream from the execute stream to efficiently hide memory latency, but applies them to work within a cache-coherent memory system with no risk of deadlocking. Extensive decoupling enables the microarchitecture to effectively tolerate long and variable memory latencies with an in-order design.

Figure 8.2 presents the high-level anatomy of the Hwacha vector accelerator. Hwacha is situated as a discrete accelerator with its own independent frontend, which consists of the RoCC unit, scalar unit, vector runahead unit (VRU), and an L1 vector instruction cache. The RoCC unit distributes the control thread instructions coming from the Rocket control processor to the scalar unit and the VRU. The scalar unit receives all instructions including the vector configuration instructions (`vsetcfg` and `vsetvl`), vector move instructions for both vector shared registers and vector address registers (`vmcs` and `vmca`), and vector-fetch instructions (`vf`) through the vector command queue (VCMDQ). Since the VRU only needs to decode the constant-strided vector memory instructions from the vector-fetch instruction stream to issue prefetches into the memory system, the RoCC unit only sends certain instructions through the vector runahead command queue (VRCMDQ) such as the vector configuration, `vmca`, and `vf` instructions. The scalar unit and the VRU both have ports into the L1 vector instruction cache. This property allows the VRU to also prefetch instructions for the scalar unit. This vector-fetch decoupling relieves the Rocket control processor to resolve address calculations for upcoming vector-fetch blocks, among other bookkeeping actions, well in advance of the vector accelerator.

The scalar unit goes ahead and fetches instructions from a given vector-fetch block, executing scalar compute and scalar memory instructions on the unit itself, while pushing vector compute and vector memory operations to the master sequencer. The scalar unit interfaces with the floating-point request and response queues (FPREQQ and FPRESPQ) to execute scalar floating-point compute instructions on the floating-point functional units shared with the Rocket control processor. A scalar memory unit (SMU) talks to the L2 cache directly to carry out scalar memory instructions.

Hwacha consists of one or more replicated vector lanes assisted by the master sequencer. Internally, the vector lane is bifurcated into two major components: the vector execution unit (VXU) and the vector memory unit (VMU). The VXU encompasses the vector register file, predicate register file, and various functional units. The VMU coordinates data movement between the VXU and the memory system. The master sequencer interfaces with lane sequencers and VMUs of all vector lanes to coordinate the execution of in-flight vector instructions. The master sequencer distributes work to individual vector lanes, keeps track of progress, and retires fully-executed vector instructions. Unlike traditional vector machines, in which the control logic keeps all vector lanes in lock-step, Hwacha keeps all vector lanes executing in a decoupled fashion. This means that each lane will need its own sequencing logic to step through vector operations, but can better tolerate irregular memory access patterns, since the vector lanes will naturally slip its execution and adapt to the memory system's behavior. However, the master sequencer will sync up all vector lanes when

executing a vector reduction operation (such as consensual branch instructions and the `vfirst` instruction) before sending the result back to the scalar unit.

Hwacha also features a VRU that exploits the inherent regularity of constant-strided vector memory accesses for aggressive yet extremely accurate prefetching. Unlike out-of-order cores with SIMD extensions that rely on reorder buffers and GPUs that rely on multithreading to hide memory latency, the Hwacha decoupled vector microarchitecture is particularly amenable to prefetching without requiring a large amount of state.

8.3 Vector Frontend: RoCC Unit and Scalar Unit

The vector frontend includes the RoCC unit, scalar unit, an L1 vector instruction cache, and two sets of vector command queues that connect the Rocket control processor with the Hwacha vector accelerator.

The RoCC unit, shown in Figure 8.3, is responsible for interfacing with the Hwacha control thread instructions that are coming from the Rocket control processor. This unit routes the control thread instructions to respective vector command queues, calculates the maximum hardware vector length given the register usage (`vsetcfg` instruction), adjusts the vector length given the application vector length (`vsetvl` instruction), responds to other vector configuration instructions (`vgetcfg`, `vgetvl`, and `vuncfg` instructions). Table 8.1 summarizes the actions taken by the RoCC unit in response to the Hwacha control thread instructions described in Section 7.1. This table also shows which control thread instructions get routed to which vector command queue. Note, the `vmcs` instruction does not get enqueued into the vector runahead command queue, since the VRU only decodes constant-strided vector memory instructions that access vector address registers.

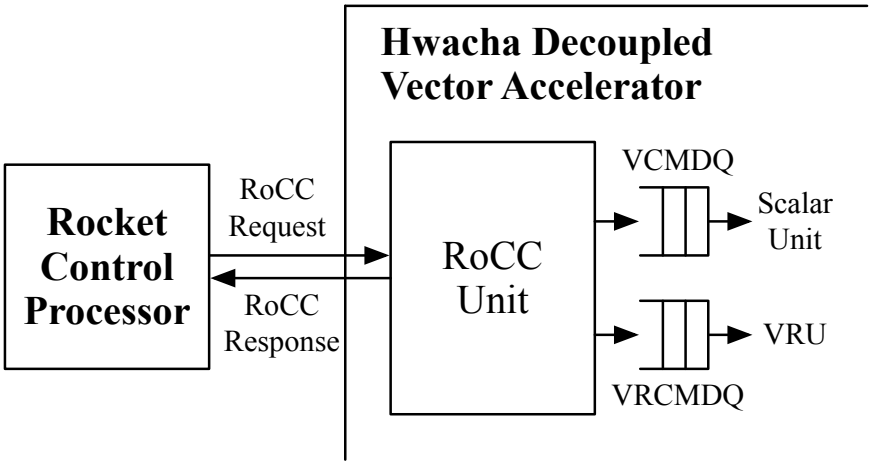


Figure 8.3: Block Diagram of the RoCC Unit – The RoCC unit interfaces with the Rocket control processor's RoCC port and vector command queues connected to the scalar unit and the vector runahead unit (VRU).

Control Thread Instruction	VCMDQ Enqueue?	VRCMDQ Enqueue?	RoCC Response?
`vsetcfg`	Y	Y	N
`vsetvl`	Y	Y	Y
`vgetcfg`	N	N	Y
`vgetvl`	N	N	Y
`vuncfg`	N	N	N
`vmcs`	Y	N	N
`vmca`	Y	Y	N
`vf`	Y	Y	N

Table 8.1: Actions Taken by the RoCC Unit for Each Hwacha Control Thread Instruction – VCDMQ = vector command queue, VRCMDQ = vector runahead command queue.

The scalar unit includes the vector shared register file and the vector address register file, and possesses a fairly conventional single-issue, in-order, four-stage pipeline (see Figure 8.4). The scalar unit receives Hwacha control thread instructions through the VCMDQ. The `vsetcfg` instruction zeros out the internal vector length register, while the `vsetvl` instruction sets the internal vector length register. The `vmcs` and `vcma` instructions directly write the accompanied value into the vector shared register file and the vector address register file, respectively. Upon encountering a `vf` instruction, the scalar unit begins fetching Hwacha worker thread instructions at the accompanying PC from the 4 KB two-way set-associative L1 vector instruction cache, continuing until it reaches a `vstop` instruction in the vector-fetch block.

Table 8.2 summarizes the actions taken by the scalar unit for each type of worker thread instructions described in Section 7.2. Among the worker thread instructions that are fetched, the

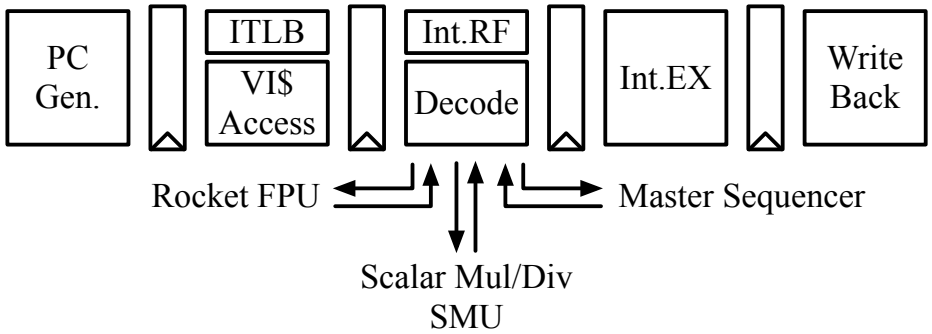


Figure 8.4: Pipeline Diagram of the Scalar Unit – The scalar unit is a fairly conventional single-issue, in-order, four-stage pipeline fetches that executes Hwacha worker thread instructions. SMU = scalar memory unit.

Worker Thread Instruction Group	Scalar Pipe	Scalar Mul/Div	Scalar Memory Unit	Rocket FPU	Master Sequencer	Set SB?
Scalar INT Compute	Y	N	N	N	N	N
Scalar INT Mul/Div	N	Y	N	N	N	Y
Scalar Memory	N	N	Y	N	N	Y
Scalar FP Compute	N	N	N	Y	N	Y
Vector	N	N	N	N	Y	N
Vector Reduction	N	N	N	N	Y	Y

Table 8.2: Actions Taken by the Scalar Unit For Each Hwacha Worker Thread Instruction Group – Scalar integer compute instructions exclude multiply and divide instructions, and vector instructions exclude vector reduction instructions. INT = integer, FP = floating-point, FPU = floating-point unit, SB = scoreboard.

scalar unit handles the scalar integer computation instructions locally on the pipeline (excluding the integer multiply and divide instructions). For other instructions, the scalar unit steers the operation to other units on the Hwacha vector accelerator in the decode stage. Integer multiply and divide instructions are steered to a decoupled functional unit on the scalar unit; scalar memory instructions are sent to the SMU; the scalar floating-point instructions interface with the FPREQQ and FPRESPQ to use the shared floating-point functional units on the Rocket FPU. For these instructions, the control logic sets the scoreboard bit to interlock on the destination register until the result is sent back to the scalar unit. Vector instructions are sent to the master sequencer. The scoreboard bit is set for a vector reduction operation to prevent the scalar unit from accessing the result before it is sent back from the vector lanes.

8.4 Vector Runahead Unit

The vector runahead unit (VRU), shown in Figure 8.5, takes advantage of the decoupled nature of the Hwacha vector-fetch architecture to hide memory latency and maximize functional unit utilization.

The VRU has a separate VRCMDQ between the Hwacha accelerator and the Rocket control processor. It receives `vsetvl`, `vmca`, and `vf` instructions through the VRCMDQ. The `vsetvl` instruction sets the current vector length. The `vmca` instruction sends over the addressing information, which is written into a separate vector address register file that resides on the VRU. Upon receiving a `vf` instruction, the VRU fetches instructions from the L1 vector instruction cache and decodes unit-strided load and store instructions. Using the previously collected address information along with the vector length, the VRU issues prefetching commands directly to the L2, in anticipation of loads and stores issued by the vector lanes. Unlike other machines, these prefetches are in most cases non-speculative. Since the address registers and vector length cannot be changed

by the worker thread, the VRU is certain what data will be touched for each vector load and store instruction.

Efficiently using L2 tracking resources and managing the runahead distance are critical to balancing latency-hiding with allowing the rest of the machine to make forward-progress at a reasonable pace. We limit the VRU to using at most one-third of the outstanding access trackers in the L2 cache, since in the unit strided case, the VRU's prefetch blocks are twice as large as the execution unit's loads and stores.

In managing the runahead-distance of the VRU, the controller must avoid two extremes. A VRU that runs too close to real-time execution risks invoking a performance penalty. This penalty arises not only from the obvious inability to hide latency, but also because the VRU wastes L2 tracking resources and creates a hotspot around one bank of the L2 cache. A VRU that runs too far ahead of real-time execution has the potential to remove cache lines from the L2 that are in-use or that have been prefetched but not yet used.

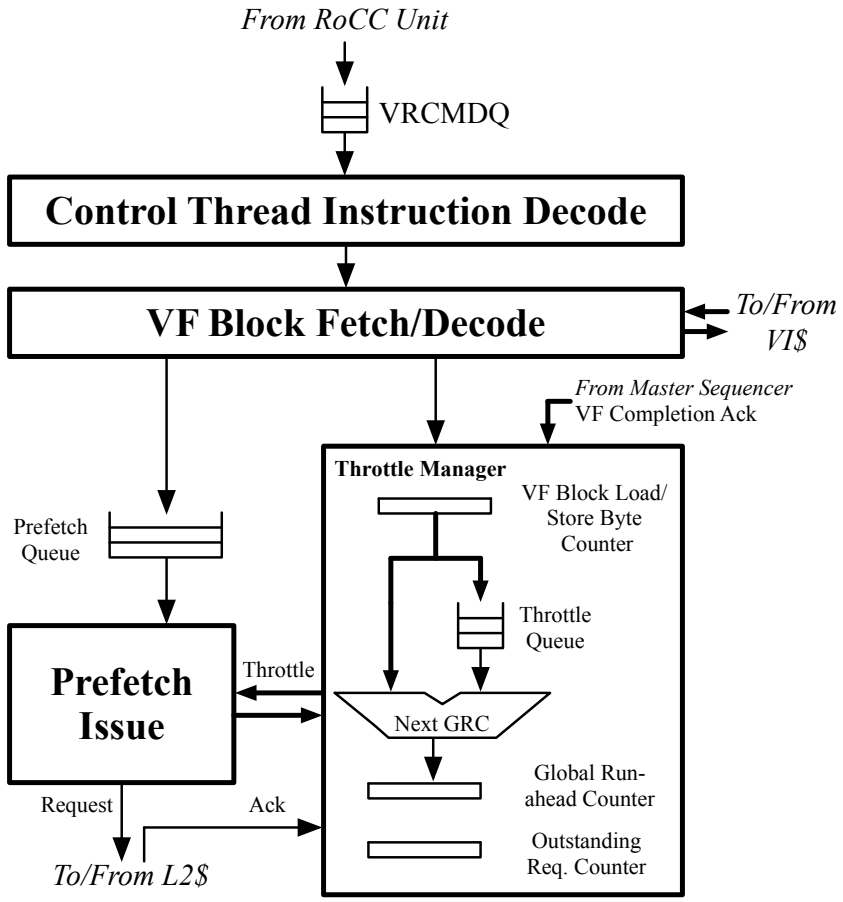


Figure 8.5: Block Diagram of the Vector Runahead Unit (VRU) – VRCMDQ = vector runahead command queue, VF = vector fetch, VI$ = vector instruction cache, GRC = global runahead counter.

To prevent the VRU from running too close to the execution units, we ignore a small number of vector-fetch blocks at startup. We observe that sacrificing the prefetch of the vector loads and stores from one or two initial vector-fetch blocks greatly increases the ability of the VRU to runahead in a steady state. To prevent the VRU from running too far ahead of the execution units, we implement a throttling scheme that keeps track of the outstanding memory loads and stores in terms of bytes that have been decoded by the VRU but have not yet been consumed by the execution units. However, for the Hwacha vector accelerator, this scheme is hindered by predicated vector memory operations and consensual branches. Our scheme ensures that the bookkeeping in the VRU's throttle mechanism is synchronized at the end of each vector-fetch block, regardless of the presence of unexecuted vector memory operations due to predication and consensual branches. In our scheme, the VRU maintains a queue containing individual load/store byte counters for each vector-fetch block that the VRU has seen, but that has not been acknowledged by the vector lanes. A global counter is also incremented by this per-block byte count whenever the VRU finishes decoding a vector-fetch block. When the vector lanes complete the execution of a vector-fetch block, an acknowledgement is sent to the VRU, which pops an entry off of the byte count queue, and decrements the global byte counter by the appropriate amount. This global counter is then used to throttle the runahead distance of the VRU.

8.5 Vector Execution Unit

The vector execution unit (VXU), depicted in Figure 8.6, is broadly organized around four banks. Each bank contains a 256×128-bit 1-read-1-write (1R1W) 8T SRAM that forms a portion of the vector register file (VRF), and a 256×2-bit 5-read-1-write (5R1W) predicate register file (PRF). Operand latches buffer up operands to emulate a multi-ported register file using an SRAM-based 1R1W register file. A crossbar connects the banks to the long-latency functional units, grouped into clusters whose members share the same operand, predicate, and result lines. Also private to each bank are a local integer arithmetic logic unit (ALU) and predicate logic unit (PLU).

Figure 8.7 depicts the systolic bank execution scheme of the VXU. The entire Hwacha vector machine is built around this stall-free systolic schedule that sustains n operands per cycle to the shared functional units after an initial n-cycle latency. The figure shows the simplified VXU structure with four banks, each with an 1R1W register file (big square) and two operand latches (two small rectangles next to the big square). Each operand latches can present their values to a global operand line, which feeds into the shared functional unit at the bottom. The result is fanned out to all banks. Several micro-ops (μops) are defined that carry out a specific task within the systolic array, such as the *read-rf* μop (read register file), *write-rf* μop (write register file), *opl-top* μop (write register file read result to top operand latch), *opl-bottom* μop (write register file read result to bottom operand latch), *xbar* μop (present operand latches to crossbar), and the *fop* μop (perform operation on shared functional unit). These μops are presented to the first bank on the top, and sequentially traverse to the next bank every cycle. At cycle 0, *read-rf* and *opl-top* μops are executed on the first bank. At cycle 1, these two μops are executed on the second bank, while a new set of *read-rf* and *opl-bottom* μops are presented to the first bank. At cycle 2, the *xbar* μop is executed

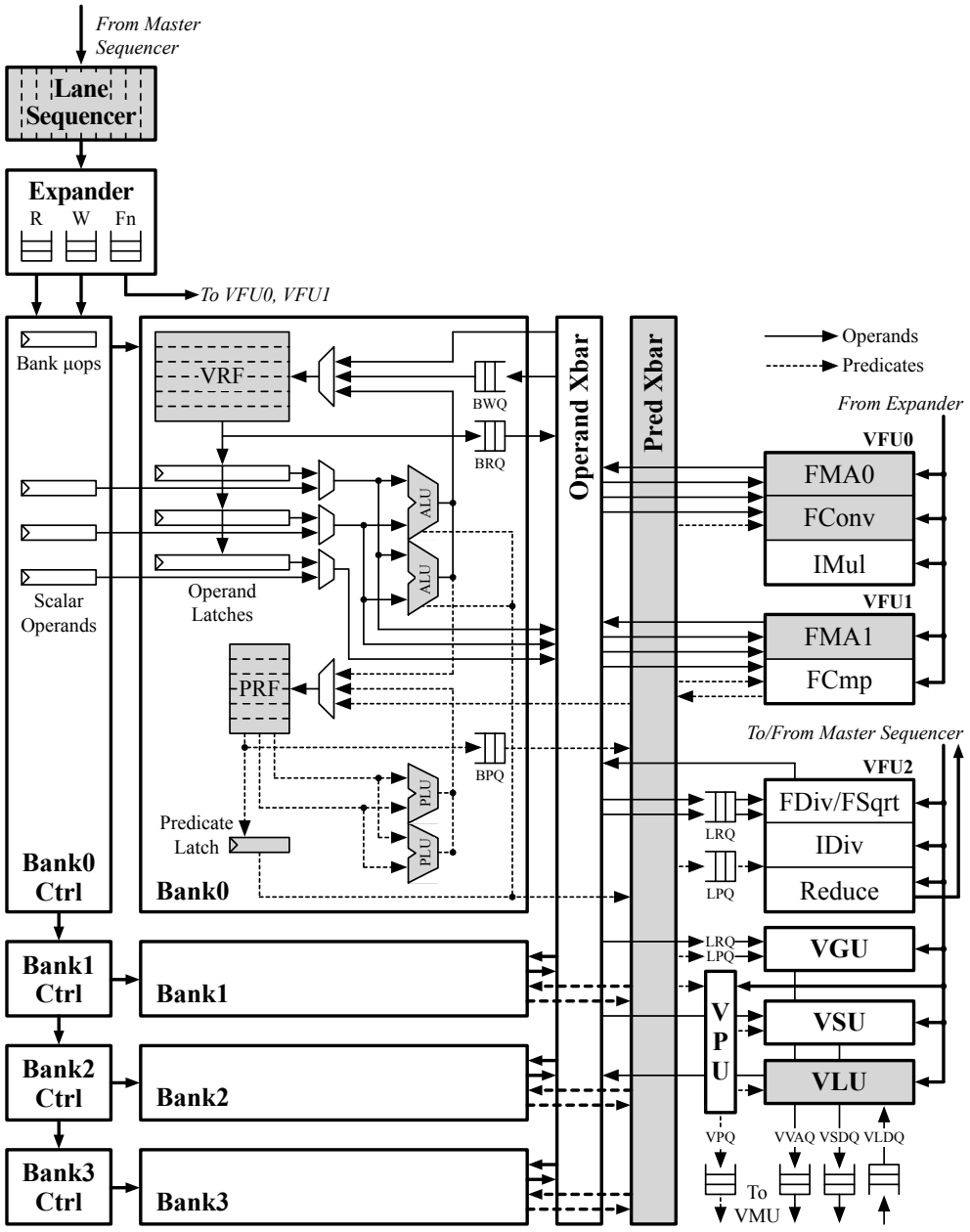


Figure 8.6: Block Diagram of the Vector Execution Unit (VXU) – VRF = vector register file, PRF = predicate register file, ALU = arithmetic logic unit, PLU = predicate logic unit, BRQ = bank operand read queue, BWQ = bank operand write queue, BPQ = bank predicate read queue, LRQ = lane operand read queue, LPQ = lane predicate read queue, VFU = vector functional unit, FP = floating-point, FMA = FP fused multiply add unit, FConv = FP conversion unit, FCmp = FP compare unit, FDiv/FSqrt = FP divide/square-root unit, IMul = integer multiply unit, IDiv = integer divide unit, VPU = vector predicate unit, VGU = vector address generation unit, VSU = vector store-data unit, VLU = vector load-data unit, VPQ = vector predicate queue, VVAQ = vector virtual address queue, VSDQ = vector store-data queue, VLDQ = vector load-data queue.

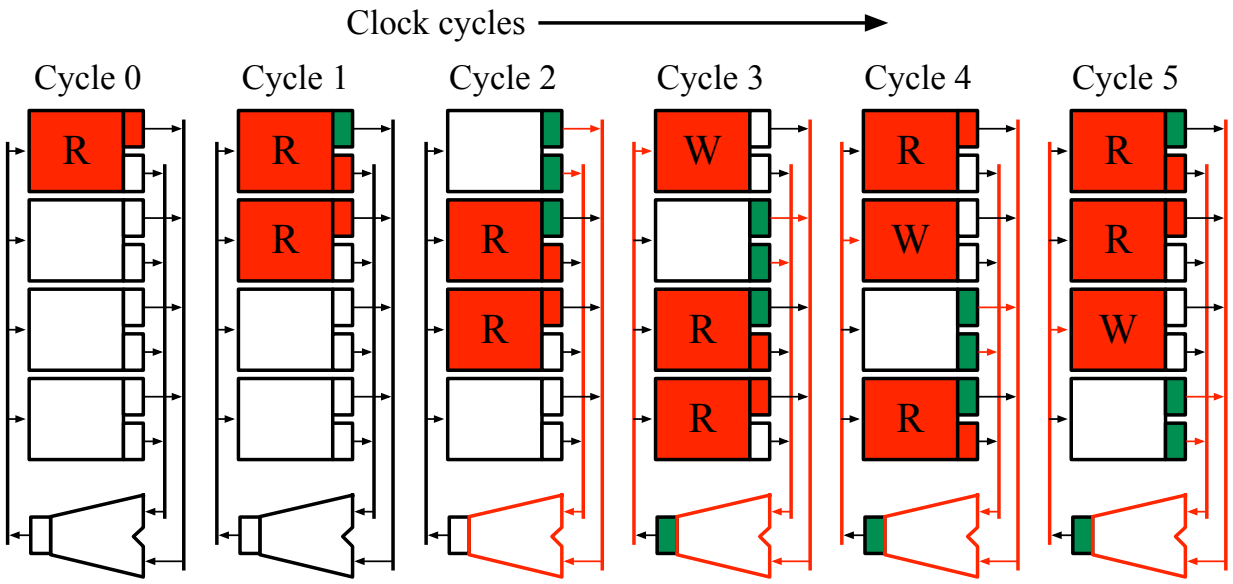


Figure 8.7: Systolic Bank Execution Diagram – In this example, after a 2-cycle initial startup latency, the banked register file is effectively able to read out 2 operands per cycle.

on the first bank, while the *fop* µop fires up the shared functional unit. At cycle 3, the *write-rf* µop writes back the result into the first bank. At cycle 4 and 5, the result is written back to the second bank and the third bank, respectively, while new *read-rf* and *opl* µops enter the system to read out the next batch of elements to the shared functional unit.

The VXU in Figure 8.6 executes the vector instructions in a similar fashion, however, has to cope with variable-latency functional units such as the memory system, integer divide unit, and the floating-point divide and square root unit. Additionally, the VXU deals with data hazards, structural hazards, and bank hazards to correctly carry out the vector computation. The sequencer (see Figure 8.8) picks a hazard-free vector operation every cycle that will execute on the systolic bank structure, and sends it to the expander (see Figure 8.9) to break the vector operation into bank µops that the systolic bank datapath understands.

Sequencer

Figure 8.8 details the sequencer logic for n vector lanes and m sequencer slots, which monitors the progress of every active vector operation within the Hwacha vector accelerator. The sequencer is split into two portions: the master sequencer and the per-lane sequencers. The master sequencer is shared among all vector lanes, and holds the common dependency information and other static state. Each lane has its own lane sequencer, which keeps track of progress within that particular lane. Due to these per-lane sequencers, the vector lanes can naturally slip in execution with respect to other vector lanes and adapt to the system behavior.

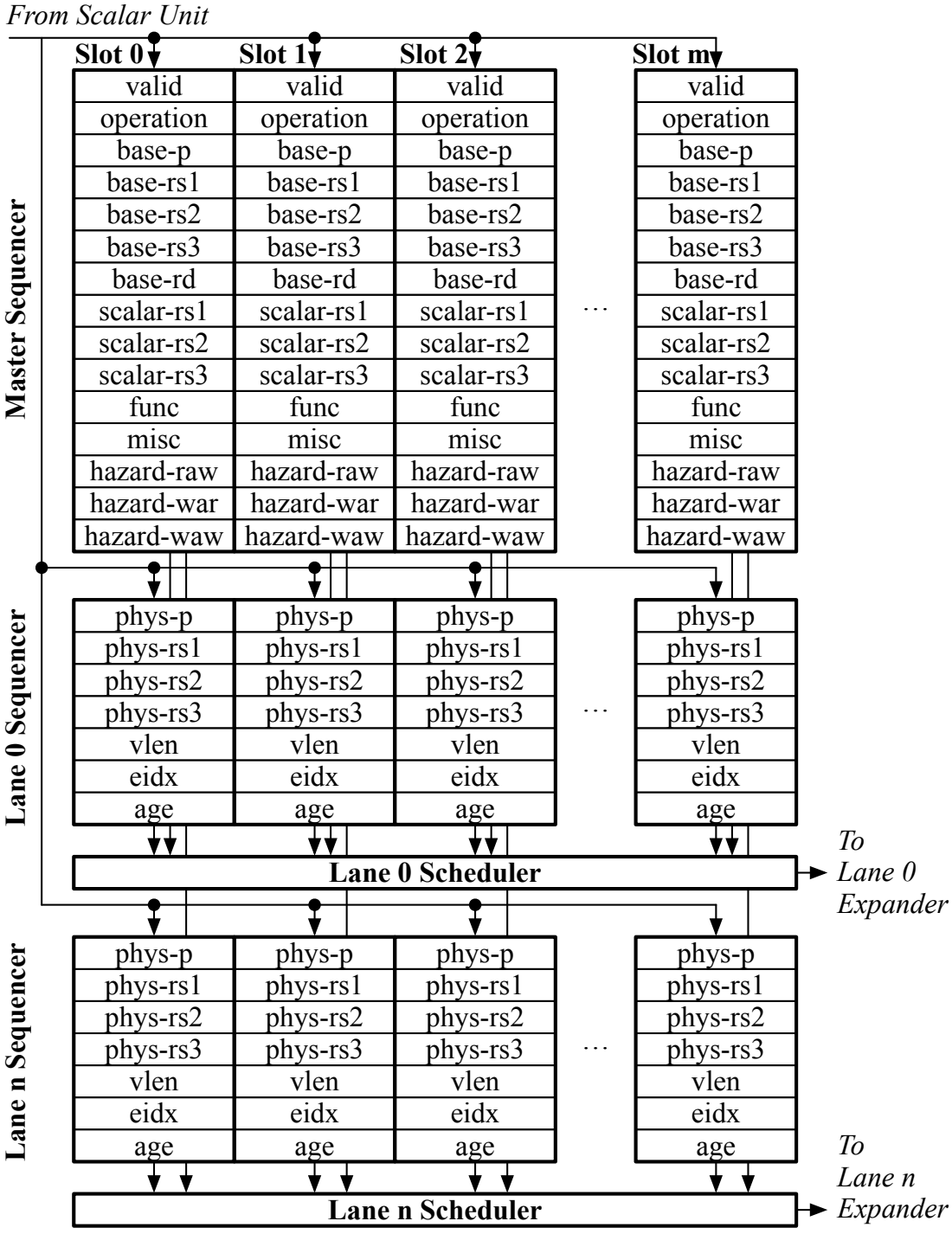


Figure 8.8: Block Diagram of the Sequencer – The sequencer is split into the master sequencer, which is shared among all vector lanes, and per-lane sequencers, which are replicated per vector lane.

The master sequencer slot has a *valid* bit, and keeps track of the sequencer operation type (*operation* field), base register specifier information (*base-p*, *base-rs1*, *base-rs2*, *base-rs3*, *base-rd* fields) to detect data hazards against future vector operations that have not been issued yet, scalar operands (*scalar-rs1*, *scalar-rs2*, *scalar-rs3* fields), miscellaneous information (*func* and *misc* fields), and data hazard information (*hazard-raw*, *hazard-war*, *hazard-waw* fields) with respect to previous sequencer operations that have already been issued. The lane sequencer slot keeps track of physical register specifier information (*phys-p*, *phys-rs1*, *phys-rs2*, *phys-rs3* fields), number of pending vector elements (*vlen* field), element index (*eidx* field), and age information (*age* field).

The scalar unit fetches, decodes, and issues a vector instruction to the master sequencer and all lane sequencers simultaneously. When a vector instruction is issued, the *valid* bit is set, the *vlen* field is loaded with the current vector length stored in the `vlen` register, and the data hazard fields (*hazard-** fields) are dynamically calculated by comparing the vector instruction's register usage with the previously issued sequencer operations' base register specifier information stored in the master sequencer. The scalar unit will stall issuing a vector instruction if there is not enough open sequencer slots to take the vector instruction. A sequencer slot is freed when all vector lanes have fully executed the sequencer operation for all vector elements.

Table 8.3 lists all possible sequencer operations that occupy one sequencer slot and their associated actions. Table 8.4 summarizes the sequencer operations that are issued for each type of worker thread instructions described in Section 7.2. Note, some vector instructions require mul-

Sequencer Operation	Sequencer Action
VIU	If no hazards, sequence integer ALU operation.
VIMU	If no hazards, sequence integer multiply operation.
VIPU	If no hazards, sequence predicate logic operation.
VFMU	If no hazards, sequence floating-point FMA operation.
VFCU	If no hazards, sequence floating-point comparison operation.
VFVU	If no hazards, sequence floating-point conversion operation.
VPU	If space in all BPQs, sequence predicate readout to BPQs.
VSU	If space in all BRQs, sequence register readout to BRQs.
VGU	If space in LPQ/LRQ for VGU, sequence pred/reg readout to LPQ/LRQ.
VQU	If space in LPQ/LRQ for VFU2, sequence pred/reg readout to LPQ/LRQ.
VIDU	Bookkeeping operation for integer divide operation.
VFDU	Bookkeeping operation for floating-point divide/square-root operation.
VCU	Bookkeeping operation for VMU address translation.
VLU	Bookkeeping operation for vector load writeback operation.

Table 8.3: List of Sequencer Operations – This table lists all sequencer operations and their associated actions. All operations are sequenced in terms of *strips* (i.e., eight 64 bit elements).

Worker Thread Instructions	VIU	VIMU	VIPU	VFMU	VFCU	VFVU	VQU	VIDU	VFDU	VPU	VGU	VCU	VSU	VLU
Vector Integer Compute	✓													
Vector Integer Multiply		✓												
Vector Integer Divide							✓	✓						
Vector Integer Reduction							✓							
Vector Predicate Compute			✓											
Vector Predicate Reduction							✓							
Vector Floating-Point FMA				✓										
Vector Floating-Point Div/Sqrt							✓		✓					
Vector Floating-Point Compare					✓									
Vector Floating-Point Convert						✓								
Vector Atomic Memory											✓	✓	✓	✓
Vector Indexed Load											✓	✓		✓
Vector Indexed Store											✓	✓	✓	
Vector Constant-Strided Load										✓		✓		✓
Vector Constant-Strided Store										✓		✓	✓	

Table 8.4: Sequencer Operations Issued for Each Hwacha Worker Thread Instruction Group – The sequencer operations are described in Table 8.3. Note, some vector instructions require multiple sequencer slots. For those instructions, the sequencer slots must be allocated in the order presented in this table.

tiple sequencer slots. The vector atomic memory operations require four sequencer operations (VGU, VCU, VSU, and VLU operations) to carry out the specific vector memory operation. The VGU operation reads out the addresses from the vector register file and pushes them to the vector virtual address queue (VVAQ). The VCU operation is a bookkeeping operation that interfaces with the VMU and only lets the following VSU and VLU operation to proceed when the address translation has succeeded. The VSU operation reads out the vector of AMO data and pushes it out to the bank read queue (BRQ). The data in the BRQ is swizzled and aligned by the VSU functional unit, and pushed to the vector store data queue (VSDQ). The VLU operation is a bookkeeping operation that keeps track of the load data that has been written back from the bank write queue (BWQ) to the vector register file. The VLU functional unit takes the load data from the vector load data queue (VLDQ), swizzles, aligns, and writes the load data to the respective BWQs. The subsequent sequencer operations that use the AMO result will be blocked waiting on the *vlen* entry of the VLU sequencer slot to decrement.

Execution is managed in *strips* that complete eight 64-bit elements worth of work, corresponding to one pass through the banks. The lane sequencer acts as an out-of-order, albeit non-

speculative, issue window: hazards are continuously examined for each operation; when clear for the next strip, an age-based arbitration scheme determines which ready operation to send to the expander. Lane schedulers pick out one sequencer operation every cycle by taking into account the data hazard information and resource requirements from the master sequencer, current progress from the *vlen* fields, age information from the *age* fields, and current resource usage by peeking into the expander. The picked sequencer operation is sent to the expander.

Expander

The expander converts a sequencer operation into its constituent µops, low-level control signals that directly drive the systolic bank datapath. Table 8.5 lists all bank µops that are used by the expander. These µops are inserted into shift registers with the displacement of read and write µops coinciding exactly with the functional unit latency. The shift registers shift down every cycle, and the µop at the end of the shift registers will be sent to the first bank of the systolic datapath. The µops then sequentially traverse all the banks cycle by cycle.

Bank µop	**Bank µop Action**
sram-read	Read from VRF.
sram-write	Select VRF writeback mux and write to VRF.
pred-read	Read from PRF.
pred-write	Select PRF writeback mux and write to PRF.
opl	Write to operand latch.
pdl	Write to predicate latch.
sreg	Use given scalar operand instead.
xbar	Drive crossbar with operands.
fop-alu	Use ALU functional unit local to the bank.
fop-plu	Use PLU functional unit local to the bank.
fop-brq	Write to local BRQ.
fop-bpq	Write to local BPQ.
fop-vfu0-fma0	Use FMA0 functional unit on VFU0.
fop-vfu0-imul	Use IMul functional unit on VFU0.
fop-vfu0-fconv	Use FConv functional unit on VFU0.
fop-vfu1-fma1	Use FMA1 functional unit on VFU1.
fop-vfu1-fcmp	Use FCmp functional unit on VFU1.
fop-vfu2	Write to LPQ/LRQ for VFU2.
fop-vgu	Write to LPQ/LRQ for VGU.

Table 8.5: List of Bank Micro-Operations (µops) – This table lists all bank µops used by the expander. The expander consists of shift registers, one for each corresponding bank µop.

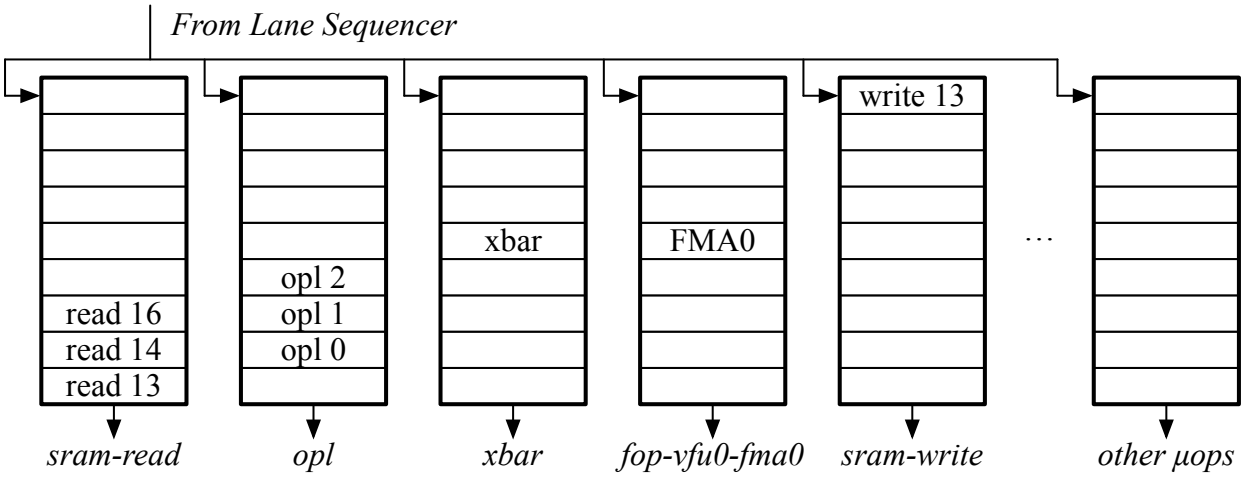


Figure 8.9: Block Diagram of the Expander – The expander consists of an array of shift registers, one for each corresponding bank µop listed in Table 8.5. These shift registers shift down every cycle. The bank µops at the end of the shift registers are sent to bank 0.

Figure 8.9 depicts the expander logic. The figure shows an example where the lane sequencer sends a VFMU operation to the expander. Assuming that the VFMU sequencer operation encodes a three-input FMA operation, the expander writes three *sram-read* µops, three *opl* µops, an *xbar* µop, an *fop-vfu0-fma0* µop, and an *sram-write* µop into the shift register slots shown in the figure. The three *sram-read* µops tell the banks to read three operands from physical register rows 13, 14, and 16. The *opl* µops write the value read out of the vector register file into operand latches 0, 1, and 2. The *xbar* µop presents the operand latches on the operand crossbar. The *fop-vfu0-fma0* µop is scheduled at the same cycle as the *xbar* µop to use the operands present on the crossbar. The *sram-write* µop is precisely scheduled to use the 4-cycle latency FMA functional unit. The shift registers for other bank µops are omitted for brevity.

Table 8.6 summarizes the bank µops that are scheduled in the expander for each sequencer operation described in Table 8.3. For example, for the VFMU sequencer operation, the expander schedules the *sram-read*, *sram-write*, *pred-read*, *opl*, *pdl*, *sreg*, *xbar*, *fop-vfu-fma0*, and *fop-vfu-fma1* bank µops accordingly. If the VFMU operation encodes a floating-point multiply then only two *sram-read* and *opl* bank µops are scheduled. Three of each bank µops are scheduled for a floating-point fused-multiply-add operation. If one of the operands came from the vector shared register file, the *sreg* bank µop will be scheduled instead of an *sram-read* bank µop. Depending on which FMA functional unit is available, either one of the *fop-vfu-fma0* or *fop-vfu-fma1* bank µops will be marked in the respective shift register.

The sequencer operations that use the functional units that are local to the bank (e.g., VIU, VIPU, VPU, and VSU) do not use the *xbar* bank µop. The sequencer operations that only access the predicate register file (e.g., VIPU and VPU) do not use the *sreg* bank µop. Among all sequencer operations, the VIPU operation is the only one that writes to the predicate register file, and hence schedule the *pred-write* bank µop. The sequencer operations that use the variable-latency

Seq Op	*sram-read*	*sram-write*	*pred-read*	*pred-write*	*opl*	*pdl*	*sreg*	*xbar*	*fop-alu*	*fop-plu*	*fop-brq*	*fop-bpq*	*fop-vfu0-fma0*	*fop-vfu0-imul*	*fop-vfu0-fconv*	*fop-vfu1-fma1*	*fop-vfu1-fcmp*	*fop-vfu2*	*fop-vgu*
VIU	✓	✓	✓		✓	✓	✓		✓										
VIMU	✓	✓	✓		✓	✓	✓	✓						✓					
VIPU			✓	✓		✓				✓									
VFMU	✓	✓	✓		✓	✓	✓	✓					✓			✓			
VFCU	✓	✓	✓		✓	✓	✓	✓									✓		
VFVU	✓	✓	✓		✓	✓	✓	✓							✓				
VPU			✓									✓							
VSU	✓		✓		✓	✓	✓				✓								
VGU	✓		✓		✓	✓	✓	✓											✓
VQU	✓		✓		✓	✓	✓	✓										✓	
VIDU																			
VFDU																			
VCU																			
VLU																			

Table 8.6: Bank μops Scheduled for Each Sequencer Operation – The sequencer operations are listed in Table 8.3, and the bank μops are listed in Table 8.5.

functional units do not explicitly schedule a *sram-write* bank μop. Variable-latency functional units instead deposit results into per-bank BWQs for decoupled writes; the sequencer monitors retirement asynchronously with the bookkeeping operations. Note, these bookkeeping sequencer operations such as the VIDU, VFDU, VCU, and VLU operations do require any bank μops to be sent down the systolic bank datapath, therefore are not sent to the expander.

Vector chaining arises naturally from interleaving bank μops belonging to different sequencer operations. The lane scheduler clears all hazards before sending back-to-back sequencer operations that are chained together; the expander simply converts them into the corresponding bank μops that are legal to execute in an interleaved fashion.

8.6 Vector Memory Unit

The per-lane vector memory units (VMUs) are each equipped with a 128-bit interface to the shared L2 cache. This arrangement delivers high memory bandwidth, albeit with a trade-off of increased latency that is overcome by decoupling the VMU from the rest of the vector unit. Figure 8.10 outlines the organization of the VMU.

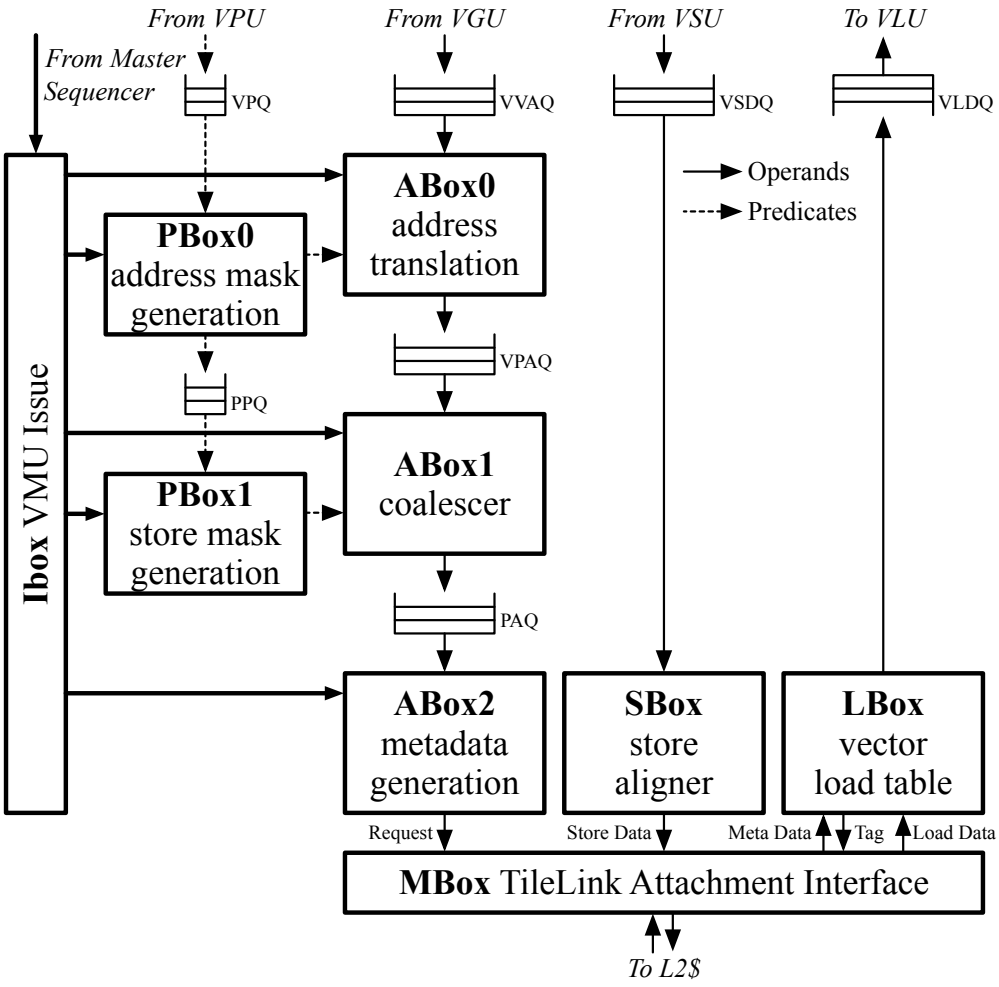


Figure 8.10: Block Diagram of the Vector Memory Unit (VMU) – VPU = vector predicate unit, VGU = vector address generation unit, VSU = vector store-data unit, VLU = vector load-data unit, VPQ = vector predicate queue, VVAQ = vector virtual address queue, VSDQ = vector store-data queue, VLDQ = vector load-data queue, VPAQ = vector physical address queue, PPQ = pipe predicate queue, PAQ = pipe address queue.

As a memory operation is issued to the lane, the VMU command queue is populated with the operation type, vector length, base address, and stride. Address generation for constant-stride accesses proceeds without VXU involvement. For indexed operations such as gathers, scatters, and AMOs, the vector address generation unit (VGU) reads offsets from the VRF into the vector virtual address queue (VVAQ). Virtual addresses are then translated and deposited into the vector physical address queue (VPAQ), and the progress is reported to the VXU. The departure of requests is regulated by the lane sequencer via the VCU sequencer operation to facilitate restartable exceptions.

The address pipeline is assisted by a separate predicate pipeline. Predicates must be examined to determine whether a page fault is genuine, and are used to derive the store masks. The VMU supports limited density-time skipping given power-of-2 runs of false predicates.

Unit strides represent a very common case for which the VMU is specifically optimized. The initial address generation and translation occur at a page granularity to circumvent predicate latency and accelerate the lane sequencer check. To more fully utilize the available memory bandwidth, adjacent elements are coalesced into a single request prior to dispatch. The VMU correctly handles edge cases with base addresses that are not 128-bit-aligned and irregular vector lengths, or not a multiple of the packing density [115].

The vector store-data unit (VSU) multiplexes elements read from the VRF banks into the vector store-data queue (VSDQ) using the bank read queues (BRQs). An aligner module following the VSDQ shifts the data entries appropriately for constant-stride stores, vector scatters, and AMOs with non-ideal alignment.

In reverse, the vector load-data unit (VLU) routes data from the vector load-data queue (VLDQ) to their respective banks via the bank write queues (BWQs). As the memory system may arbitrarily order responses, two VLU optimizations become crucial. The first is an opportunistic writeback mechanism that permits the VRF to accept elements out of sequence; this reduces latency and area compared to a reorder buffer. The VLU maintains a bit vector of retired elements, and interacts with the VLU sequencer operation for bookkeeping. The VLU is also able to simultaneously manage multiple operations to avoid artificial throttling of successive loads by the VMU.

8.7 Design Space

The Hwacha microarchitecture has been developed as a flexible generator to enable massive design space exploration. Table 8.7 lists a relevant subset of Hwacha design parameters that can be adjusted to tune the Hwacha design at Chisel elaboration time. The number of various structures are exposed as a tunable parameter, such as the number of vector lanes and sequencer entries. The number of pipeline stages for all functional units are tunable. Various queue structures also expose the number of entries as parameters. Options to instantiate a private FPU for the Hwacha scalar unit exist. Finally, the VRU and the mixed-precision extensions can be optionally turned off. Chapter 9 evaluates a baseline Hwacha design against a Hwacha design with mixed-precision support in order to see how the mixed-precision feature affects area, performance, and power/energy consumption.

Parameter	Description	Default Value
HwachaNLanes	Number of vector lanes	1
HwachaNSeqEntries	Number of sequencer entries	8
HwachaStagesALU	Number of ALU pipeline stages	1
HwachaStagesPLU	Number of PLU pipeline stages	0
HwachaStagesIMul	Number of IMul pipeline stages	3
HwachaStagesDFMA	Number of double-precision FMA pipeline stages	4
HwachaStagesSFMA	Number of single-precision FMA pipeline stages	3
HwachaStagesHFMa	Number of half-precision FMA pipeline stages	3
HwachaStagesFConv	Number of FConv pipeline stages	2
HwachaStagesFCmp	Number of FCmp pipeline stages	1
HwachaNVVAQEntries	Number of VVAQ entries	4
HwachaNVPAQEntries	Number of VPAQ entries	24
HwachaNVSDQEntries	Number of VSDQ entries	4
HwachaNVLDQEntries	Number of VLDQ entries	4
HwachaNVLTEntries	Number of Vector Load Table entries	64
HwachaNDTLB	Number of data TLB entries	8
HwachaNPTLB	Number of prefetch TLB entries	2
HwachaLocalScalarFPU	Instantiate separate FPU for scalar unit	False
HwachaBuildVRU	Instantiate VRU	True
HwachaConfMixedPrec	Enable Mixed Precision	False

Table 8.7: Tunable Hwacha Design Parameters and Default Values – This table lists the various Hwacha design parameters, their description, and default values.

Mixed-Precision Support

Our microarchitectural changes for mixed-precision extension focus on the modules shaded in Figure 8.6, with modifications falling into two broad categories: datapath (parallel functional units, and subword compaction/extraction logic), and control (data hazard checking when chaining vector operations of unequal throughput). For more details on the mixed-precision extension for Hwacha, see Ou's master thesis [99].

Register Mapping. To preserve control logic regularity, all elements with the same index must reside within the same SRAM bank. This avoids structural hazards that would otherwise require μops to be scheduled differently for distinct banks, counter to the systolic execution schedule. The VRF banks are segmented into doubleword, word, and halfword regions. Thus, the starting physical address of a vector register can be straightfowardly calculated by the sum of the architectural register identifier and the associated region offset, and elements in the vector are traversable by

a constant stride equal to the total number of architectural registers of that type. The size of the SRAM arrays remain unchanged. However, the predicate register file is widened to 8 bits, such that all predicates corresponding to a maximally packed SRAM entry of eight halfwords are accessible through a single port in one cycle. The predicate mapping is also striped in a similar manner.

Spatial Parallelism. The overall rate of an operation is constrained by the precisions of the source and destination registers, as well as the availability of parallel functional units for that particular operation. Both the register precisions and operation rate are determined during decode and recorded in the master sequencer entry. These two pieces of information, along with a subword index in the μop, control a set of muxes interposed between the SRAM port and operand latches that unpack and sign-extend the subwords as desired. The sign extension is necessary for producing consistent results between an implementation that incorporates mixed-precision support and one that omits it. Another set of muxes in front of the write port performs the inverse repacking. For each vector lane, we instantiate two more single-precision and six more half-precision FMA units, as well as an additional half-to-single and single-to-half conversion unit, to enable full throughput for these operations.

Multi-Rate Vector Chaining. Chaining allows vector operations to execute in an interleaved manner. Interim results may be consumed by subsequent operations without waiting for the entire vector to complete. Fundamentally, newer and faster operations must be prevented from overtaking older and slower operations when a data hazard exists. Comparing the remaining vector lengths between active sequencer entries is a necessary but not sufficient method: As these values are updated at time of sequencing rather than commit, the strip may still be partially in-flight. Although a conservative scheme is possible by maintaining a separation greater than a strip between dependent operations, this degrades the performance gains of chaining. Instead, the sequencer also examines the shift registers in the expander for pending write μops that conflict. For a uniform rate, it previously sufficed to search for collisions in the physical register addresses. This logic must now be expanded into an interval check to determine any overlaps in the strip(s) used by both operations. The primary cost is in extra comparators.

Multilane Configuration

Hwacha is parameterized to support any power-of-2 number of identical lanes. Although the master sequencer issues operations to all lanes synchronously, each lane executes entirely decoupled from one another.

To achieve more uniform load-balancing, elements of a vector are striped across the lanes by a runtime-configurable multiple of the sequencer strip size (the *lane stride*), as shown in Figure 8.11. This also simplifies the base calculation for memory operations of arbitrary constant stride, enabling the VMU to reuse the existing address generation datapath as a short iterative multiplier. The striping does introduce gaps in the unit-stride operations performed by an individual VMU, but the VMU issue unit can readily compensate by decomposing the vector into its con-

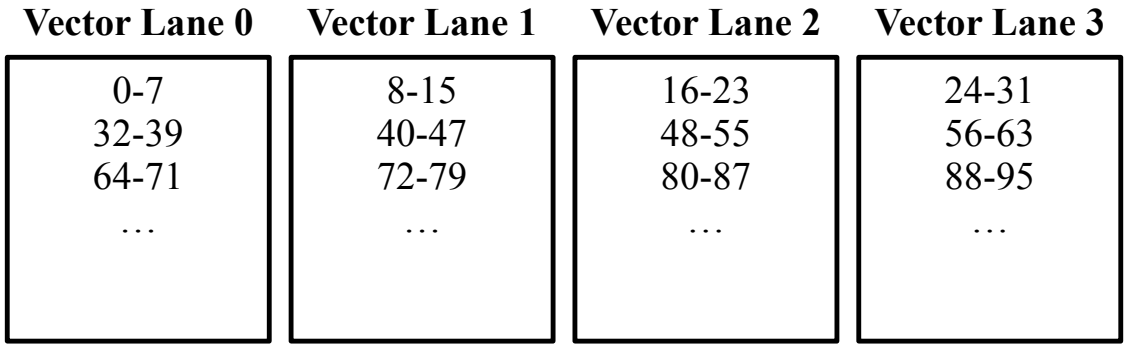


Figure 8.11: Mapping of Elements Across a Four-Lane Hwacha Vector Machine – Shows an example element mapping for a four-lane Hwacha vector accelerator. In this example, the lane stride is set to one times the strip size or 8 elements.

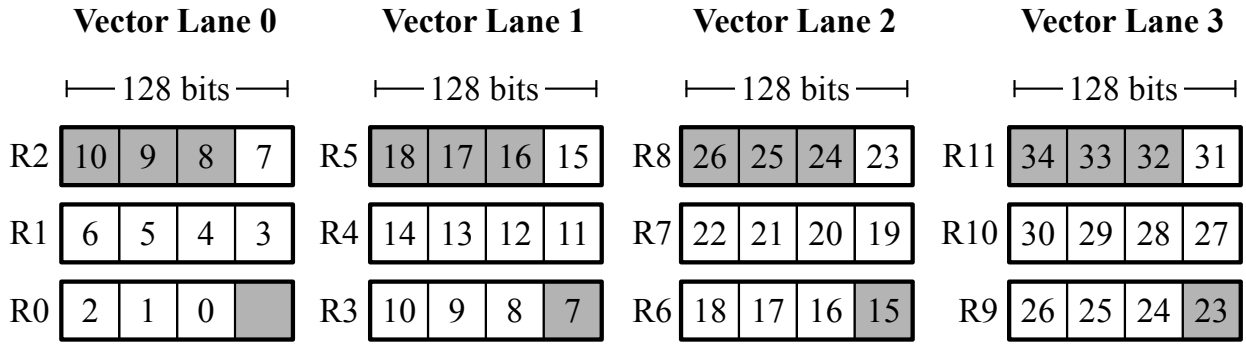


Figure 8.12: Example of Redundant Memory Requests by Adjacent Lanes – These occur when the base address of a unit-strided vector in memory is not aligned at the memory interface width (128 bits)—in this case, `0x????????4`. Each block represents a 128 bit TileLink beat containing four 32 bit elements. Shaded cells indicate portions of a request ignored by each lane. Note that R2 overlaps with R3, R5 with R6, etc.

tiguous segments, while the rest of the VMU remains oblivious. Unfavorable alignment, however, incurs a modest waste of bandwidth as adjacent lanes request the same cache line at these segment boundaries. Figure 8.12 provides an example of such a situation.

Chapter 9

Hwacha Evaluation

This chapter presents Hwacha evaluation results. We first discuss the details of our evaluation framework. Using this framework, we compare a baseline Hwacha design to a Hwacha design with mixed-precision support to see how the mixed-precision feature affects area, performance, and power/energy consumption. Chapter 8 details the Hwacha design space with other parameters that can be modified. We also validate the Hwacha design against the ARM Mali-T628 MP6 GPU by running a suite of microbenchmarks compiled from the same OpenCL source code using our custom LLVM-based scalarizing compiler and ARM's stock OpenCL compiler on respective hardware.

9.1 Evaluation Framework

This section outlines how we evaluate the Hwacha design, and validate our decoupled vector-fetch architecture against a commercial GPU using compiled OpenCL kernels. Our evaluation framework is described in Figure 9.1. The high-level objective of our framework is to compare realistic area, performance, and power/energy numbers using detailed VLSI layouts and compiled OpenCL kernels (as opposed to using hand-tuned assembly code).

As a first step towards that goal, we write a set of OpenCL microbenchmarks for the study (see Section 9.2), and develop our own LLVM-based scalarizing compiler that targets the Hwacha vector-fetch assembly programming model (see Section 9.3). These microbenchmarks are compiled twice with our custom compiler and ARM's stock OpenCL compiler. Section 9.4 details our implementation strategy. We select parameters for the Rocket Chip SoC generator to match the Samsung Exynos 5422 SoC, which has an ARM Mali-T628 MP6 GPU. We chose that specific SoC because it ships with the ODROID-XU3 development board that has instrumentation capabilities to separately measure power consumption of the Mali GPU. Our simulated memory system is validated against the memory system of the Exynos 5422 SoC in Section 9.5. We synthesize and place-and-route both Hwacha designs (with and without mixed-precision support) in a commercial 28 nm process resembling the 28 nm high-κ metal gate (HKMG) process that is used to fabricate the Exynos 5422 SoC. We run compiled microbenchmarks on both gate-level simulators

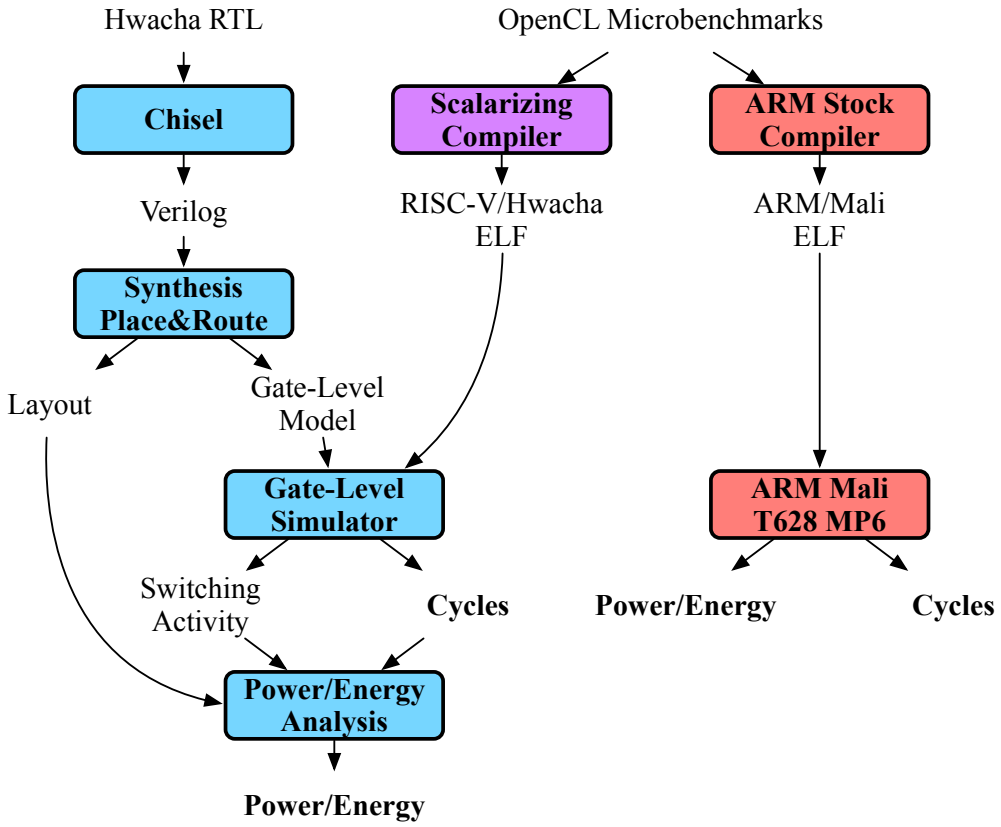


Figure 9.1: Evaluation Framework – The Hwacha RTL is pushed through synthesis and place-and-route to generate area numbers and a gate-level simulator that runs OpenCL microbenchmarks and produces cycle counts and switching activities, which are later combined with the layout information for power/energy consumption numbers. We also run the same set of OpenCL microbenchmarks on an ARM Mali T628 GPU (part of a Samsung Exynos 5422 SoC) to capture performance and power/energy consumption numbers. The OpenCL microbenchmarks are compiled with our custom LLVM-based scalarizing compiler as well as ARM's stock OpenCL compiler.

to obtain accurate switching activities and performance numbers. The switching activities are then combined with the VLSI layout to generate accurate power/energy consumption numbers. Area, performance, and energy numbers are subsequently compared against each other, but are also validated against ARM Mali numbers obtained from the ODROID-XU3 board in Sections 9.6–9.8.

9.2 Microbenchmarks

For the study, we have developed four types of OpenCL kernels in four different precisions. Table 9.1 lists all microbenchmarks, and Figure 9.2 shows the OpenCL kernels of selected microbenchmarks. The microbenchmarks are named with a prefix and a suffix. The prefix denotes the precision of the operands: `h`, `s` and `d` for half-, single-, and double-precision, respectively. `sd` signifies that the benchmark's inputs and outputs are in single-precision, but the intermediate computation is performed in double-precision. Similarly, `hs` signifies that the inputs and outputs are in half-precision, but the computation is performed in single-precision.

The suffixes denote the type of the kernel: `axpy` for $\mathbf{y} \leftarrow a\mathbf{x}+\mathbf{y}$, a scaled vector accumulation (see Figure 9.2a); `gemm` for dense matrix-matrix multiplication (see Figure 9.2b); and `filter` for a Gaussian blur filter, which computes a stencil over an image. The `mask` versions of `filter` accept an additional input array determining whether to compute that point, thus exercising predication (see Figure 9.2c). For reference, we also wrote hand-optimized versions of `gemm-opt` in order to gauge the code generation quality of our custom OpenCL compiler. For $C \leftarrow A \times B$,

Kernel	Mixed-Precision	Predication
{s,d}axpy		
{hs,sd}axpy	✓	
{s,d}gemm		
{hs,sd}gemm	✓	
{s,d}gemm-opt		
{hs,sd}gemm-opt	✓	
{s,d}filter		
{hs,sd}filter	✓	
mask-{s,d}filter		✓
mask-{hs,sd}filter	✓	✓

Table 9.1: Listing of Evaluated Microbenchmarks – The prefix denotes the precision: `d` for double-precision, `s` for single-precision, `hs` for half-to-single upconvert, and `sd` for single-to-double upconvert. The suffix denotes the type of microbenchmark: `axpy` for $\mathbf{y} \leftarrow a\mathbf{x}+\mathbf{y}$, `gemm` for dense matrix-matrix multiplication, `gemm-opt` for hand-optimized versions of `gemm`, `filter` for a Gaussian blur filter, `mask-filter` for masked versions of `gemm`.

```
1 __kernel void daxpy(__global double *src, __global double *dst, double factor)
2 {
3   long i = get_global_id(0);
4   dst[i] += src[i] * factor;
5 }
```

(a) daxpy

```
1 __kernel void dgemm(__global double *A, __global double *B, __global double *C,
2                     int ldc)
3 {
4   long i = get_global_id(0);
5   long k = 4*get_global_id(1);
6   long j = 4*get_global_id(2);
7
8   #pragma unroll
9   for (long jj=j; jj<j+4; jj++) {
10    double c = C[i+jj*ldc];
11    for (long kk=k; kk<k+4; kk++) {
12      c += A[kk+jj*ldc] * B[i+kk*ldc];
13    }
14    C[i+jj*ldc] = c;
15  }
16 }
```

(b) dgemm

```
1 __kernel void mask_dfilter(__global double *src, __global double *dst, long ldc,
2                            __global short *mask, // omitted in non-masked version
3                            double m0, double m1, double m2,
4                            double m3, double m4, double m5,
5                            double m6, double m7, double m8)
6 {
7   long x = get_global_id(0);
8   long y = 2*get_global_id(1);
9
10  #pragma unroll
11  for (long yy=y; yy<y+2; yy++) {
12    if (mask[x+yy*ldc]) { // omitted in non-masked version
13      double i00 = src[(x-1)+(yy-1)*ldc]*m0;
14      double i01 = src[(x)  +(yy-1)*ldc]*m1;
15      double i02 = src[(x+1)+(yy-1)*ldc]*m2;
16      double i03 = src[(x-1)+(yy)  *ldc]*m3;
17      double i04 = src[(x)  + yy  * ldc]*m4;
18      double i05 = src[(x+1)+(yy)  *ldc]*m5;
19      double i06 = src[(x-1)+(yy+1)*ldc]*m6;
20      double i07 = src[(x)  +(yy+1)*ldc]*m7;
21      double i08 = src[(x+1)+(yy+1)*ldc]*m8;
22
23      dst[x+yy*ldc] = i00 + i01 + i02 + i03 + i04 + i05 + i06 + i07 + i08;
24    }
25  }
26 }
```

(c) mask-dfilter

Figure 9.2: OpenCL Kernels of Evaluated Microbenchmarks – (a) daxpy, (b) dgemm, and (c) mask-dfilter. The non-masked dfilter omits the mask condition check in (c).

`gemm-opt` loads unit-strided vectors of C into the vector register file, keeping them in place while striding through the A and B matrices. The values from B are unit-stride vectors; the values from A reside in shared registers.

9.3 Scalarizing OpenCL Compiler

We have developed an OpenCL compiler as a custom LLVM [69] backend based on the PoCL OpenCL runtime [58]. The main challenges in generating Hwacha vector code from OpenCL kernels are moving thread-invariant values into scalar registers [120, 31, 74], identifying stylized memory access patterns, and using predication effectively [61, 33, 73]. Thread-invariance is determined using the variance analysis presented in Chapter 4 and is performed at both the LLVM IR level and machine instruction level. This promotion to scalar registers avoids redundant values being stored in vector registers, improving register file utilization. In addition to scalarization, thread invariance is used to drive the promotion of loads and stores to unit-strided or constant-strided accesses. Performing this promotion is essential for the decoupled architecture because it enables prefetching of the vector loads and stores.

To fully support OpenCL kernel functions, the compiler must also generate predicated code for conditionals and loops. Generating efficient predication without hardware divergence management requires additional compiler analyses. We implement predication compiler algorithms described in Chapter 5.

Collecting energy results on a per-kernel basis requires very detailed, hence time-consuming, simulations. This presents a challenge for evaluating OpenCL kernels, which typically make heavy use of the OpenCL runtime and online compilation. Fortunately, OpenCL supports offline compilation, which we rely upon to avoid simulating the execution of the compiler. To obviate the remaining runtime code, we augment our OpenCL runtime with the ability to record the inputs and outputs of the kernel. Our runtime also generates glue code that pushes these inputs into the kernel code, and verifies that the outputs match after the execution. The effect is that only the kernel code of interest is simulated with great detail, substantially reducing simulation runtime.

9.4 Implementation

We implement two Hwacha designs with and without mixed-precision support and compare them to observe the impact of our mixed-precision extensions, and to validate against the ARM Mali-T628 MP6 GPU. We first analyze the Samsung Exynos 5422 SoC and the ARM Mali-T628 GPU not only to understand their architecture and microarchitecture, but also to pick out parameters for the Rocket Chip SoC generator and the Hwacha vector unit to make them as comparable as possible. We then discuss our RTL development and VLSI flow.

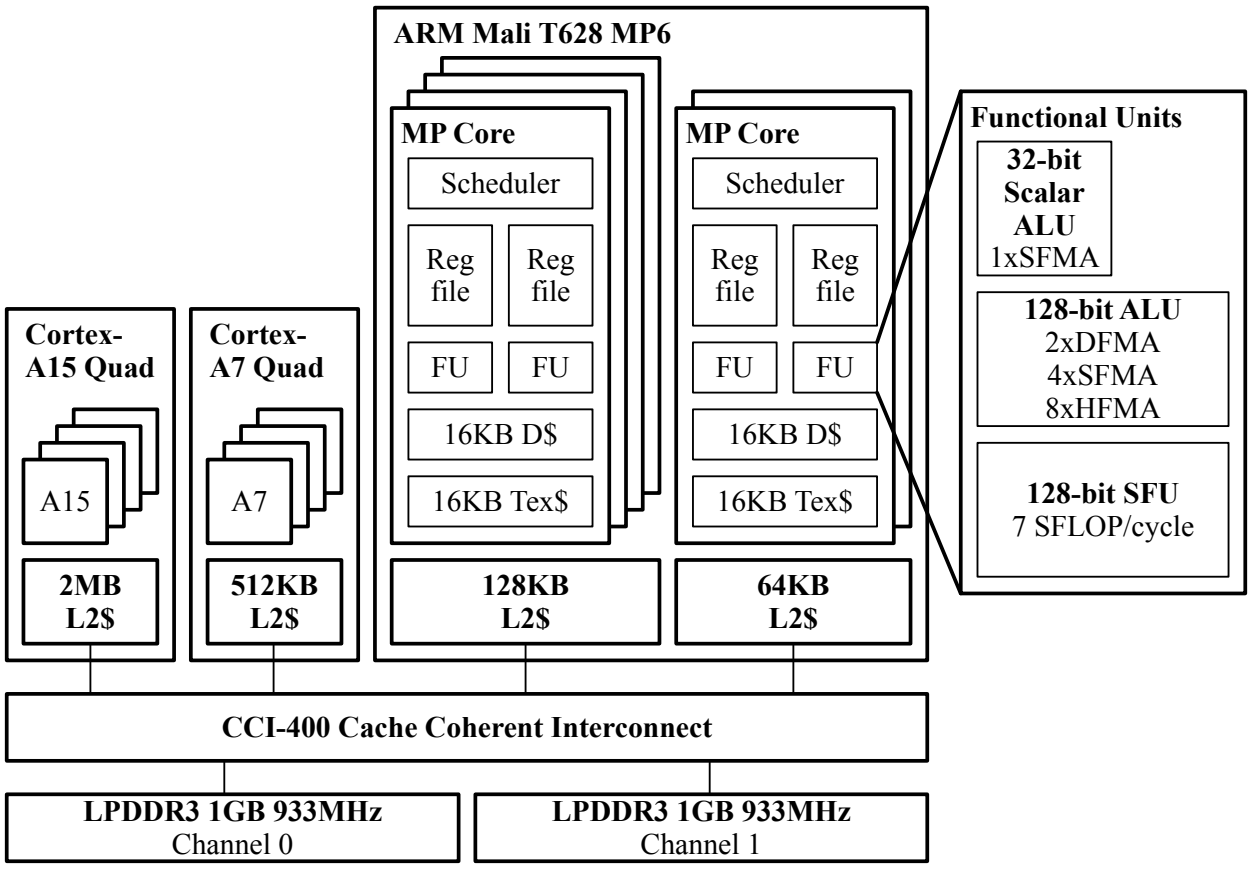


Figure 9.3: Block Diagram of the Samsung Exynos 5422 SoC Block Diagram – The SoC contains a quad Cortex-A15 out-of-order processor complex, a quad Cortex-A7 in-order processor complex, an ARM Mali-T628 MP6 GPU complex. All modules have private L2 caches, which talk to the dual LPDDR3 memory channels through the CCI-400 cache coherent interconnect.

Samsung Exynos 5422 and the ARM Mali-T628 MP6 GPU

Figure 9.3 shows the block diagram of the Samsung Exynos 5422 SoC. The quad Cortex-A15 complex, quad Cortex-A7 complex, and the ARM Mali-T628 MP6 are connected through the CCI-400 cache coherent interconnect that talks to two LPDDR3 channels of 1 GB running at 933 MHz [9, 54]. Table 9.2 presents the specific Rocket Chip SoC generator parameters we chose to match the Samsung Exynos 5422 SoC.

The Mali-T628 MP6 GPU has six shader cores (termed MPs, or multiprocessors) that run at 600 MHz, exposed as two sets of OpenCL devices. Without explicitly load balancing the work on these two devices by software, the OpenCL kernel can either only run on the two shader core device or on the four shader core device. We first run the microbenchmarks on the device with two shader cores, named *Mali2*, and again on the device with four shader cores, named *Mali4*. Each shader core has four main pipes: two arithmetic pipes (128-bit wide VLIW SIMD execution pipelines), a load/store pipe with a 16 KB data cache, and a texture pipe with a 16 KB texture cache.

Component	Settings
Hwacha vector unit	baseline/mixed-precision, 1/2/4 lanes
Hwacha L1 vector inst cache	4 KB, 2 ways
Rocket L1 data cache	16 KB, 4 ways
Rocket L1 inst cache	16 KB, 4 ways
L2 Cache	64 KB/bank, 8 ways, 4 banks
Cache coherence	MESI protocol, directory bits in L2$ tags
DRAMSim2	LPDDR3, 933 MHz, 1 GB/channel, 2 channels

Table 9.2: Used Rocket Chip SoC Generator Parameters – The parameters are configured to make the generated Rocket Chip SoC as comparable as the Samsung Exynos 5422 SoC.

Threads are mapped to one of these four main pipes. The compiler needs to pack three instructions per very long instruction word for the arithmetic SIMD pipelines. The three instruction slots are a 32-bit scalar FMA (fused-multiply-add) unit, a 128-bit SIMD unit (which supports two 64-bit FMAs, four 32-bit FMAs, or eight 16-bit FMAs), and a 128-bit SFU (special functional unit) for dot products and transcendentals. Each shader core has an associated 32 KB of L2 cache, making the total capacity 192 KB. Further details on the Mali architecture and how the L2 cache is split among these two devices are sparse; an AnandTech article [116] and an ARM presentation [11] on the Midgard GPU architecture provide some insight into the organization of Mali.

To measure power consumption of the various units, we sample the current through three separate power rails, distinguishing the power consumption of the CPU complex, the GPU, and the memory system. We average these samples over the kernel execution, and use the average power and kernel runtime to compute the energy consumed by the Mali GPU during kernel execution. We examine this comparison in the next section.

One MP possesses approximately the same arithmetic throughput as one Hwacha vector lane with mixed-precision support. The Hwacha vector lane is 128 bits wide, and has two vector functional units that each support two 64-bit FMAs, four 32-bit FMAs, or eight 16-bit FMAs.

Due to time constraints, only the single-lane Hwacha configuration has been fully evaluated. Consequently, it must be noted that the comparisons against the Mali2 and Mali4 devices are not perfectly fair from Hwacha's perspective, given that Mali2 and Mali4 have the advantage of twice and quadruple the number of functional units, respectively. Nevertheless, the results are encouraging in light of this fact, although they should be considered still preliminary, as there are substantial opportunities that remain to tune the benchmark code for either platform.

RTL Development and VLSI Flow

The Hwacha RTL is written in Chisel [17], a domain-specific hardware description language embedded in the Scala programming language. Because Chisel is embedded in Scala, hardware devel-

opers can leverage features of the modern Scala programming language for increased productivity, such as parameterized types, object-oriented programming, and functional programming. Chisel generates both a cycle-accurate software model as well as synthesizable Verilog that can be mapped to standard FPGA and ASIC flows. Our article on agile hardware development methodology [79] has more details on how Chisel is used in conjunction with the VLSI tools to increase designer productivity. We also use a custom random instruction generator tool to facilitate verification of the Hwacha vector unit.

We use the Synopsys physical design flow (Design Compiler, IC Compiler) to map the Chisel-generated Verilog to a standard cell library and memory-compiler-generated SRAMs in a widely-used commercial 28 nm process technology, that resembles the 28 nm HKMG process in which the Exynos 5422 is fabricated. We use eight layers out of ten for routing, leaving two for the top-level power grid. The flow is highly automated to enable quick iterations through physical design variations. When coupled with the flexibility provided by Chisel, this flow allows a tight feedback loop between physical design and RTL implementation. The rapid feedback is vital for converging on a decent floorplan to obtain acceptable quality of results: a week of physical design iteration on the single-lane Hwacha design resulted in approximately 100 layouts and around a 50% faster clock frequency.

We combine the layout information with activity factors generated via gate-level simulation of the place-and-routed netlist to calculate power consumption. Parasitic RC constants for every wire in the gate-level netlist are computed using the TLU+ model. Each microbenchmark is executed in the gate-level simulator to produce an activity factor for every net in the design. The combination of activity factors and parasitics are fed into PrimeTime PX to produce an average power number for each benchmark. We derive energy dissipation for each benchmark by multiplying the average power with the runtime (i.e., cycle count divided by implementation clock rate).

9.5 Memory System Validation

In order to provide reasonable DRAM model for our performance and power/energy comparisons, we utilize DRAMsim2 [107] in our simulations. To ensure a competitive baseline for our benchmarks, we supplied DRAMsim2 with timing parameters of a Micron LPDDR3 part matching those of the dual-channel 933 MHz LPDDR3 on the Samsung Exynos 5422 SoC. We use `ccbench` to empirically confirm that our simulated memory hierarchy is similar to that of the Exynos 5422 SoC.

The `ccbench` benchmarking suite [28] contains a variety of benchmarks to characterize multi-core systems. We use `ccbench`'s `caches` benchmark, which performs a pointer chase to measure latencies of each level of the memory hierarchy. In unit-stride mode, each pointer in the array of pointers points to the next contiguously placed pointer in memory. In cache-line stride mode, each pointer points to another pointer cache-line size away in memory, to avoid spatial prefetching. In random-stride mode, each pointer points to a random pointer in the array, to avoid both stride and stream prefetching. The size of the array of pointers can be varied to exercise differing pieces of the memory hierarchy.

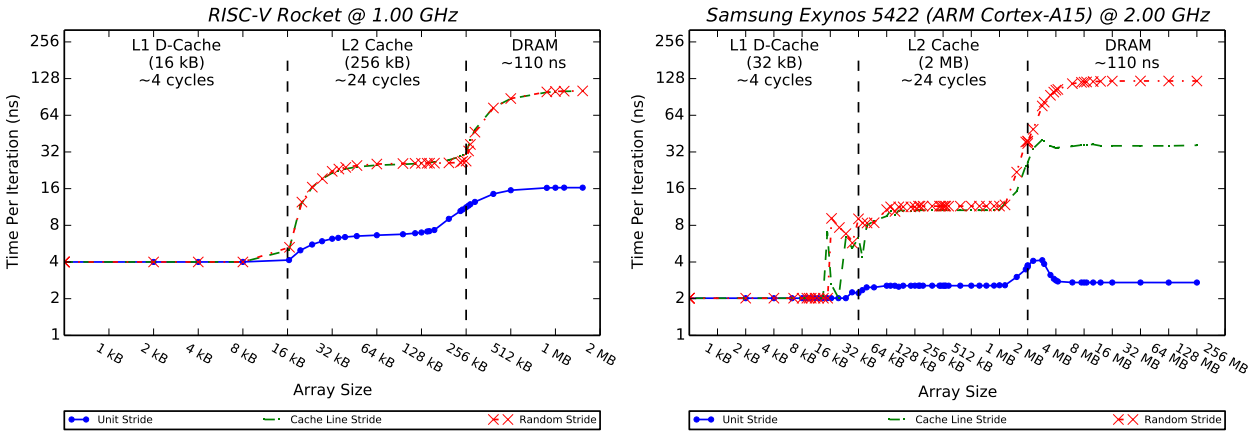


Figure 9.4: Memory System Validation – We run the `ccbench` "`caches`" memory system benchmark on the RISC-V Rocket processor and the ARM Cortex-A15 processor to validate the memory system of the Rocket Chip SoC generator.

Figure 9.4 compares the performance of our cycle-accurate simulated memory hierarchy against the Exynos 5422 SoC using the `caches` benchmark in `ccbench`. On the simulated RISC-V Rocket core, `ccbench` measures cycles, which we normalize to nanoseconds by assuming the 1 GHz clock of previous silicon implementations of Rocket cores [78, 136]. On the Exynos 5422 SoC, `ccbench` measures wall-clock time to produce our results.

This baseline comparison highlights two important features that validate our experiments. Firstly, while the L1 and L2 cache sizes differ between the Rocket core and the Exynos 5422 SoC, the L1 and L2 caches have similar latencies in terms of processor clock cycles. With both a 1 GHz Rocket core and a 2 GHz ARM Cortex-A15, the L1 hit latency is approximately 4 cycles and the L2 hit latency is approximately 22 cycles. Secondly, both the simulated LPDDR3 used in our experiments and the LPDDR3 in the Exynos 5422 SoC achieve similar latencies of approximately 110 ns.

Nevertheless, one significant difference remains in the inclusion of a streaming prefetcher within the ARM Cortex-A15, which reduces the latency of unit-stride and non-unit-stride loads and stores [8].

9.6 Area and Cycle-Time Comparison

Table 9.3 lists the total area numbers, cycle-times, and clock frequencies obtained for a variety of Hwacha configurations. Recall that Mali2 is clocked at 600 MHz but contains approximately twice the number of functional units. To attempt to match functional unit bandwidth with a single-lane Hwacha design, we target Hwacha for a nominal frequency of 1.2 GHz. While actual frequencies fall slightly short, they are still generally above 1 GHz. However, the aggressive physical design does involve a trade-off in area.

	Hwacha Baseline				Hwacha MXP			
Lanes	1	1	2	4	1	1	2	4
Area (mm^2)	2.23	2.11	2.60	3.59	2.32	2.21	2.82	4.04
Cycle-Time (ns)	0.95	0.90	0.93	0.93	0.98	0.94	1.02	1.08
Frequency (GHz)	1.05	1.11	1.08	1.08	1.02	1.06	0.98	0.93
PNR?	✓				✓			

Table 9.3: VLSI Quality of Results – Columns not marked as PNR are results from synthesis. MXP = Mixed-Precision.

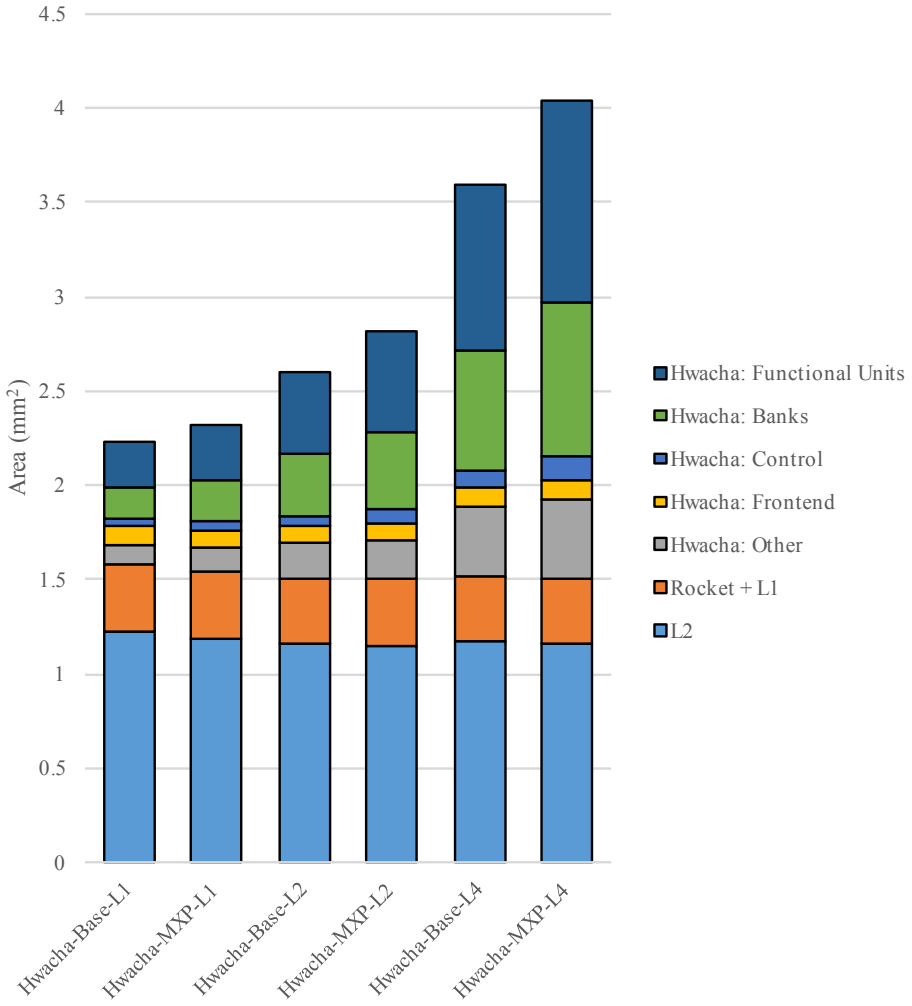
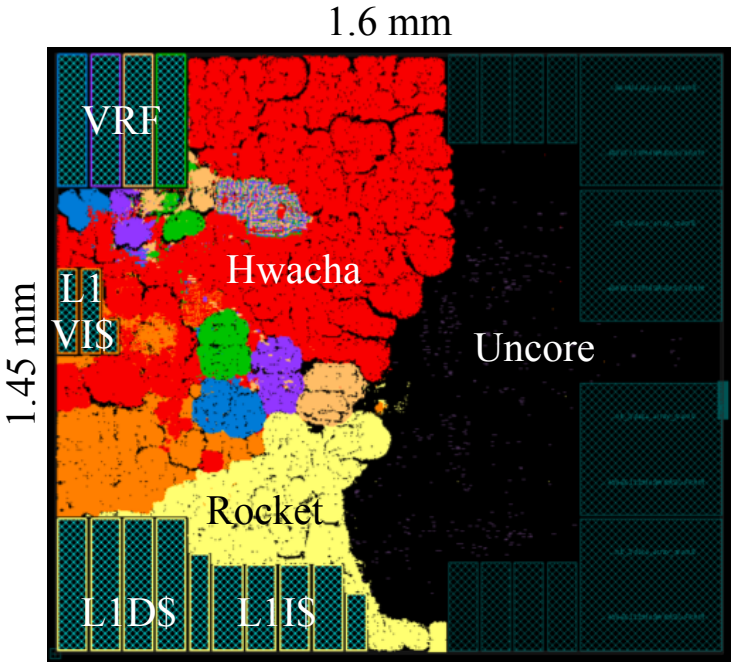
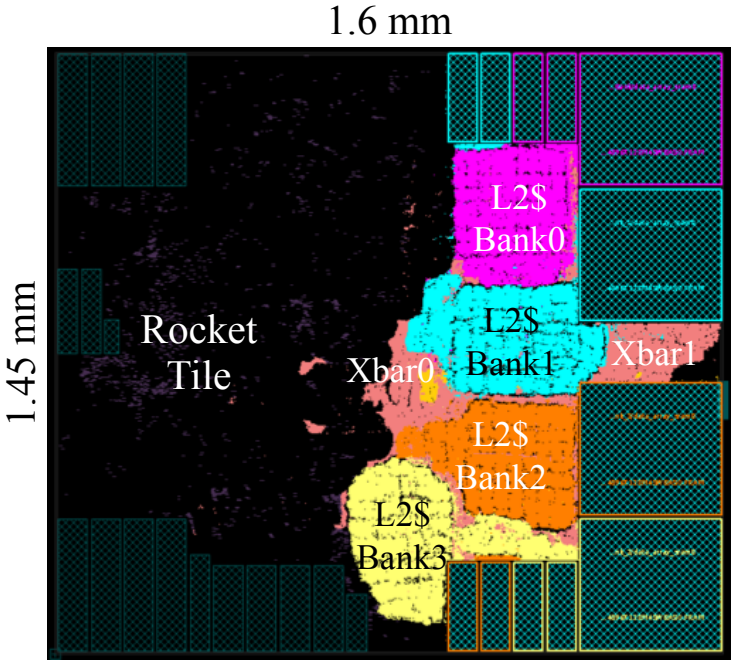


Figure 9.5: Area Distribution for the 1/2/4-Lane Hwacha Baseline and Hwacha MXP (Mixed-Precision) Designs – Data for the 2-lane and 4-lane designs are from synthesis only.



(a) Tile: Rocket and Hwacha



(b) Uncore: L2 Cache and Interconnect

Figure 9.6: Layout of the Single-lane Hwacha Design with Mixed-Precision Support – (a) Tile, and (b) Uncore. VRF = vector register file, VI$ = vector instruction cache, Xbar0 = L1-to-L2 TileLink crossbar, Xbar1 = AXI4 crossbar.

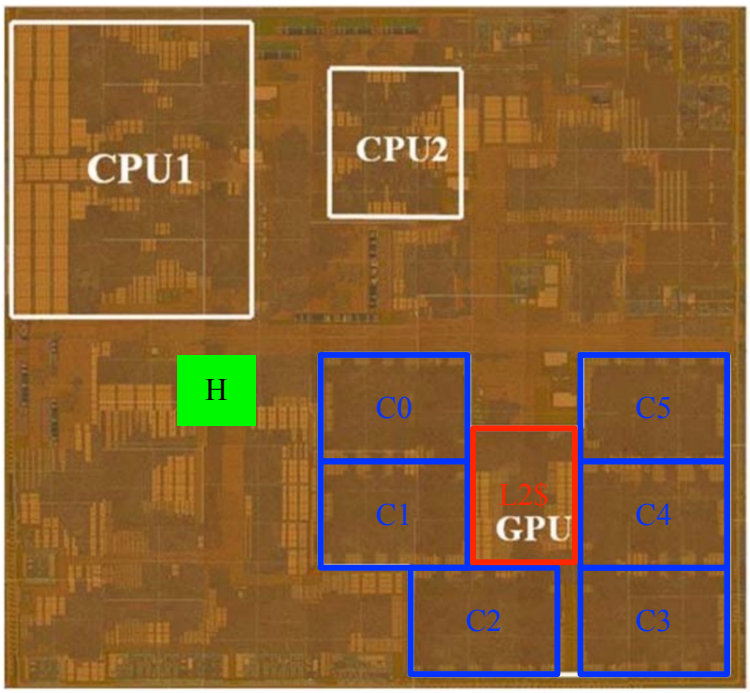


Figure 9.7: Annotated Die Photo of the 20 nm Samsung Exynos 5430 SoC – The total die area is 110.18 mm^2, and the ARM Mali T628 MP6 GPU takes up 25 mm^2 of the total SoC area [44]. The ARM Mali GPU has 6 shader cores and a 512 KB L2 cache [104]. Each shader core (C0–C5) and the L2 cache area (L2$) are highlighted in the die photo. Contrast the ARM Mali GPU with the green H box, which is the Hwacha design in Figure 9.6 (single-lane Hwacha design with mixed-precision support and a 256 KB L2 cache) shrunk down to 20 nm and drawn at scale.

Figure 9.5 shows the area distribution of the baseline Hwacha design and the Hwacha design with mixed-precision support for 1-, 2-, and 4-lane configurations. As seen in the figure, the area overhead for the mixed-precision extensions spans from 4.0% in the single-lane design to 12.5% in the four-lane design. The additional functional units account for a large portion of the increase. The banks also become somewhat larger from the widening of the predicate register file, as does the control due to the non-trivial amount of comparators needed to implement the expander hazard check in the sequencers. Area for functional units, banks, control, and the vector memory units (part of the other bar) grows proportional to the number of lanes for the multi-lane designs.

Figure 9.6 shows the layout of the single-lane Hwacha design with mixed-precision support and a 256 KB L2 cache. Various module hierarchies and SRAM macros are highlighted in the layout figure.

The Samsung Exynos die photos and area breakdowns are rarely published. The 28 nm Exynos SoC was published at the ISSCC conference in 2013 [113], however, the GPU was excluded from the die photo. The 2015 ISSCC paper [104] had a die photo of the entire 20 nm Exynos SoC (see Figure 9.7), however, it did not give out any area numbers. A follow-on Anandtech article [44] provides detailed area breakdown of various Exynos SoCs including the sizes of the ARM Mali GPU.

According to the article, the Exynos 5420 SoC (comparable to the Exynos 5422 SoC) is about 137 mm^2 in total area, and the ARM Mali T628 MP6 with a 192 KB L2 cache is about 30 mm^2. Using the numbers in Figure 9.5, a rough estimate of a comparable 6-lane Hwacha design with a 256 KB L2 cache comes out to be 6.2 mm^2 in the same 28 nm technology. Note that the Hwacha design does not have dedicated hardware units for graphics such as SFUs (special functional units) present in ARM Mali GPUs. Conservatively assuming that the overhead of graphics is 50%, the Hwacha design with mixed-precision support is still 3× more area-efficient than the ARM Mali T628 MP6 GPU.

9.7 Performance Comparison

For the set of hand-optimized assembly and OpenCL benchmarks, Figure 9.8 graphs the speedup normalized to baseline Hwacha running the OpenCL version. Compared to Mali2, Hwacha suffers from a slight disadvantage in functional unit bandwidth from being clocked closer to 1 GHz rather than the ideal 1.2 GHz. Compared to Mali4, Hwacha has less than half the functional unit bandwidth.

For `*axpy`, MXP (mixed-precision support) has a marginal effect on performance. As a streaming kernel, it is primarily memory-constrained and therefore benefits little from the higher arithmetic throughput offered by MXP. `*axpy` is also the only set of benchmarks in which Mali2 and Mali4 consistently outperforms Hwacha by a factor of 1.5 to 2. This disparity most likely indicates some low-level mismatches in the outer memory systems of Mali and our simulated setup—for example, in the memory access scheduler. We used the default parameters for the memory access scheduler that were shipped with the DRAMSim2 project.

The benefits of MXP become clearer with `*gemm` as it is more compute-bound, and more opportunities for in-register data reuse arise. As expected for `dgemm*` and `sdgemm*`, no difference in performance is observed between the baseline and MXP, since the two designs have the same number of double-precision FMA units. A modest speedup is seen with `sgemm-unroll`, although still far from the ideal factor of 2 given the single-precision FMA throughput. Curiously, MXP achieves almost no speedup on `sgemm-unroll-opt`. It is possible that the matrices are simply too undersized for the effects to be major. `hgemm*` and `hsgemm*` demonstrate the most dramatic improvements; however, the speedup is sublinear since, with the quadrupled arithmetic throughput, memory latency becomes more problematic per Amdahl's law.

A significant gap is apparent between the OpenCL and hand-optimized versions of the same benchmarks. The primary reason is that the latter liberally exploits *inter-vector-fetch optimizations* whereby data is retained in the vector register file and reused across vector-fetch blocks. It is perhaps a fundamental limitation of the programming model that prevents this behavior from being expressed in OpenCL, resulting in redundant loads and stores at the beginning and end of each vector-fetch block.

For all precisions of `*gemm`, Mali2 performs surprisingly poorly, by a factor of 3 or 4 slowdown relative to the Hwacha baseline. Mali4 performs about 2× better than Mali2, however, is still

slower than the Hwacha baseline. We speculate that the working set is simply unable to fit in cache. Thus, this particular run should not be considered to be entirely fair.

Finally, the baseline and MXP perform about equivalently on `*filter`. A slight improvement is discernible for `sfilter` and `mask-sfilter`, and a more appreciable speedup is evident with `hsfilter` and `mask-hsfilter`. The performance of Mali2 is generally about half that of Hwacha, and Mali4 is on par with Hwacha.

9.8 Energy Comparison

Figure 9.9 graphs the energy consumption for each benchmark, normalized once again to the baseline Hwacha results for the OpenCL versions. Note that the Mali GPU runs on a supply voltage of 0.9 V, whereas we overdrive Hwacha with a 1 V supply to meet the 1 GHz frequency target.

For `*axpy`, Hwacha MXP dissipates slightly more energy than the Hwacha baseline, with dynamic energy from the functional units comprising most of the difference. In other benchmarks as well, the functional units account for a higher proportion of losses in Hwacha MXP, which may indicate sub-par effectiveness of clock gating with the extra functional units. Consistent with its performance advantage, Mali2 and Mali4 are twice as energy-efficient on `*axpy` than Hwacha.

The results for `*gemm` are much more varied. Although Hwacha MXP is less energy-efficient than the baseline on benchmarks for which it can provide no performance advantage, such as `dgemm`, it is more so on `sgemm`, `hsgemm`, and `hgemm`. These collectively demonstrate a consistent downward trend of increasingly significant reductions in energy consumption as the precision is lowered. Mali data points are again an outlier here, and no conclusion should be drawn.

Energy dissipation on `*filter` generally mirrors performance. Overall, Hwacha MXP is slightly worse than the Hwacha baseline except on `hsfilter`. Mali similarly retains an advantage as it does with performance, with some exceptions involving reduced-precision computation, i.e., both masked and non-masked versions `sfilter` and `hsfilter`. On these, the energy efficiency of Hwacha MXP is on par with Mali2 and worse when compared to Mali4.

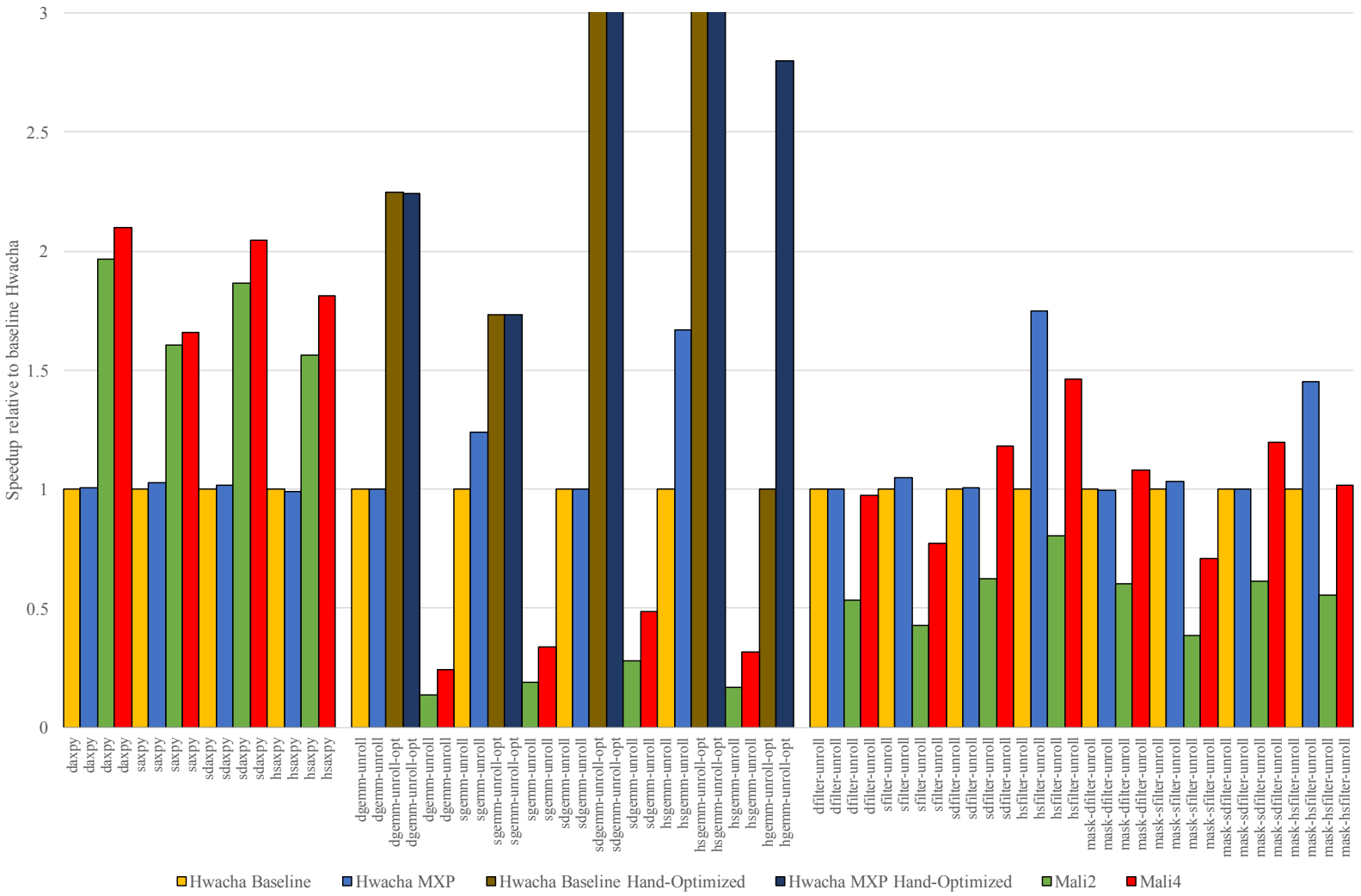


Figure 9.8: Hwacha Performance Results – Higher is better. Due to scale, bars for certain benchmarks have been truncated. *sdgemm-unroll-opt* has speedups 14.0× on the Hwacha baseline and 13.8× on Hwacha MXP. *hsgemm-unroll-opt* has speedups 12.0× on the Hwacha baseline and 19.0× on Hwacha MXP.

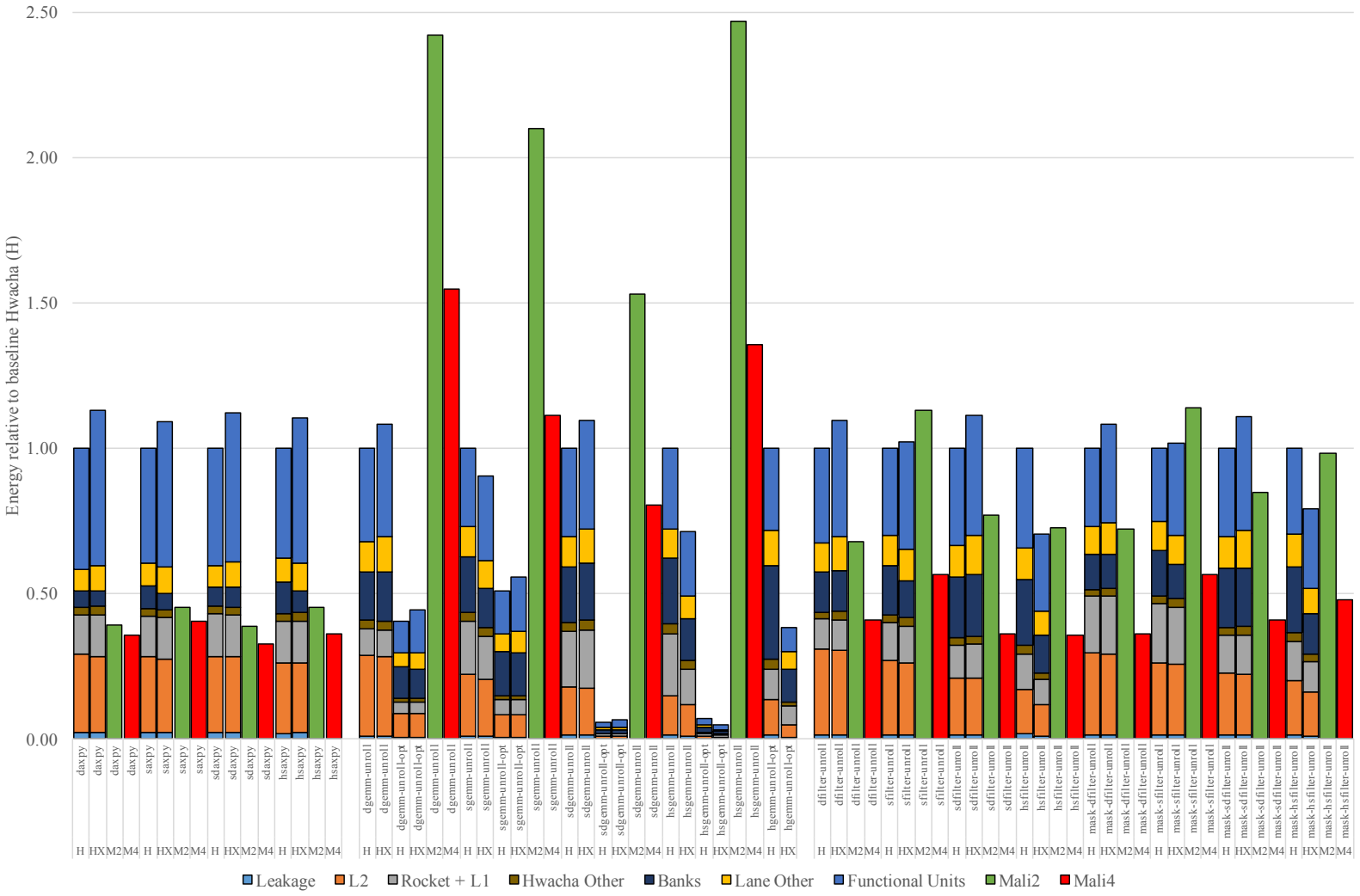


Figure 9.9: Hwacha Energy Results – Lower is better. H = Hwacha baseline, HX = Hwacha MXP, M2 = Mali2, M4 = Mali4.

Chapter 10

Conclusion

In the post-Moore's law era, computer architects are forced to incorporate custom accelerators alongside the general-purpose processor to achieve higher performance and better energy efficiency. It is up to the computer architect either to build a sea of fixed-function accelerators or to embrace general-purpose specialization by deploying a handful of accelerators that are more flexible and programmable. This thesis has explored a new approach to building a highly performant and efficient data-parallel accelerator that maintains the same level of programmability as other data-parallel accelerators, by pushing the burden of factoring out redundant values and predicating the control flow found in data-parallel programs into a scalarizing compiler, therefore simplifying the underlying vector hardware.

10.1 Thesis Summary and Contributions

This thesis began by introducing data-parallel programming languages, assembly programming models, architectural features, and divergence management schemes of various data-parallel architectures out in the field. We discussed the overheads and inefficiencies of these data-parallel architectures, and proposed that a simple traditional vector-like architecture coupled with a scalarizing compiler is able to improve performance and energy efficiency while maintaining the same level of programmability as other architectures. We outlined the Hwacha vector-fetch architecture, its instruction set architecture, microarchitecture, and implementation in a modern 28 nm process node. We validated our Hwacha design against an ARM Mali-T628 MP6 GPU also implemented in a similar 28 nm process node, and showed that our new approach is competitive to industry-built data-parallel architectures. These ideas and results form the key contributions of this thesis, which are outlined below.

- **Survey of Data-Parallel Architectures on their Assembly Programming Models, Architectural Features, and Compiler Support –** This thesis introduced implicit autovectorization and the explicit Single-Program Multiple-Data (SPMD) model as the widely used parallel programming languages for data-parallel architectures. We outlined a wide range of data-parallel architectures and their assembly programming models as well as architectural

features to give programmers and compiler writers an idea on how code gets executed on these machines, and how parallel programming models get mapped down to these architectures. We presented divergence management schemes of these data-parallel architectures in more detail, as they often dictate the underlying machine organization. See Chapter 2 for more details.

- **Scalarizing Compilers –** This thesis established the foundation for scalarizing compilers. They are tasked to find the most efficient mapping of explicitly parallel programs down to the data-parallel architecture alongside scalar execution resources by automatically picking out redundant values and operations, and breaking complex control flow down to simple vector predication. Chapter 3 presented the overheads of the explicit SPMD programming model, crystallized the concept of scalarizing compilers, and compared them to vectorizing compilers that are tasked to automatically pick out parallel parts from single-threaded code and map them onto the data-parallel architecture. Chapter 4 and 5 described the compiler foundation, implementation in a production CUDA compiler, and evaluation results of the scalarization and predication compiler algorithms, respectively.

- **The Hwacha Vector-Fetch Architecture –** This thesis proposed the Hwacha vector-fetch architecture to show that traditional vector-like architectures coupled with a scalarizing compiler can maintain the same level of programmability as other data-parallel architectures while being highly performant, efficient, yet a favorable compiler target. The Hwacha vector-fetch architecture decouples the vector instruction stream into a separate thread with a vector-fetch instruction in order to enable light-weight access-execute decoupling of the vector data stream. We proposed and implemented the new Hwacha decoupled vector microarchitecture that exploits the vector-fetch decoupling within a cache-coherent memory system to efficiently tolerate long and variable memory latencies. We validated the Hwacha design against the ARM Mali-T628 MP6 GPU by running a suite of microbenchmarks compiled from the same OpenCL source code with our custom LLVM-based scalarizing compiler and ARM's stock OpenCL compiler. Chapter 6 detailed the assembly programming model and architectural features of the Hwacha vector-fetch architecture. Chapter 7, Chapter 8, and Chapter 9 described the Hwacha instruction set architecture, decoupled vector microarchitecture, and evaluation results, respectively.

This thesis documents the long journey on the Hwacha vector-fetch architecture project. Figure 10.1 shows the timeline history for the Hwacha project and the other projects to which I have contributed. The Maven vector-thread architecture project explored the programmability and efficiency of a wide range of data-parallel accelerators including MIMD, traditional vector, and the newly proposed Maven vector-thread architecture. The Hwacha vector-fetch architecture was developed in four phases (see Section 6.3 for detailed history), including the two major phases in which scalarization and predication ideas were developed. The Hwacha architecture was co-developed with multiple tapeouts for the Raven project and the EOS project. The 28 nm Raven chips combined a 64 bit RISC-V processor and the Hwacha vector accelerator with on-chip switched-capacitor DC-DC converters and adaptive clocking [136, 80, 135]. The 45 nm

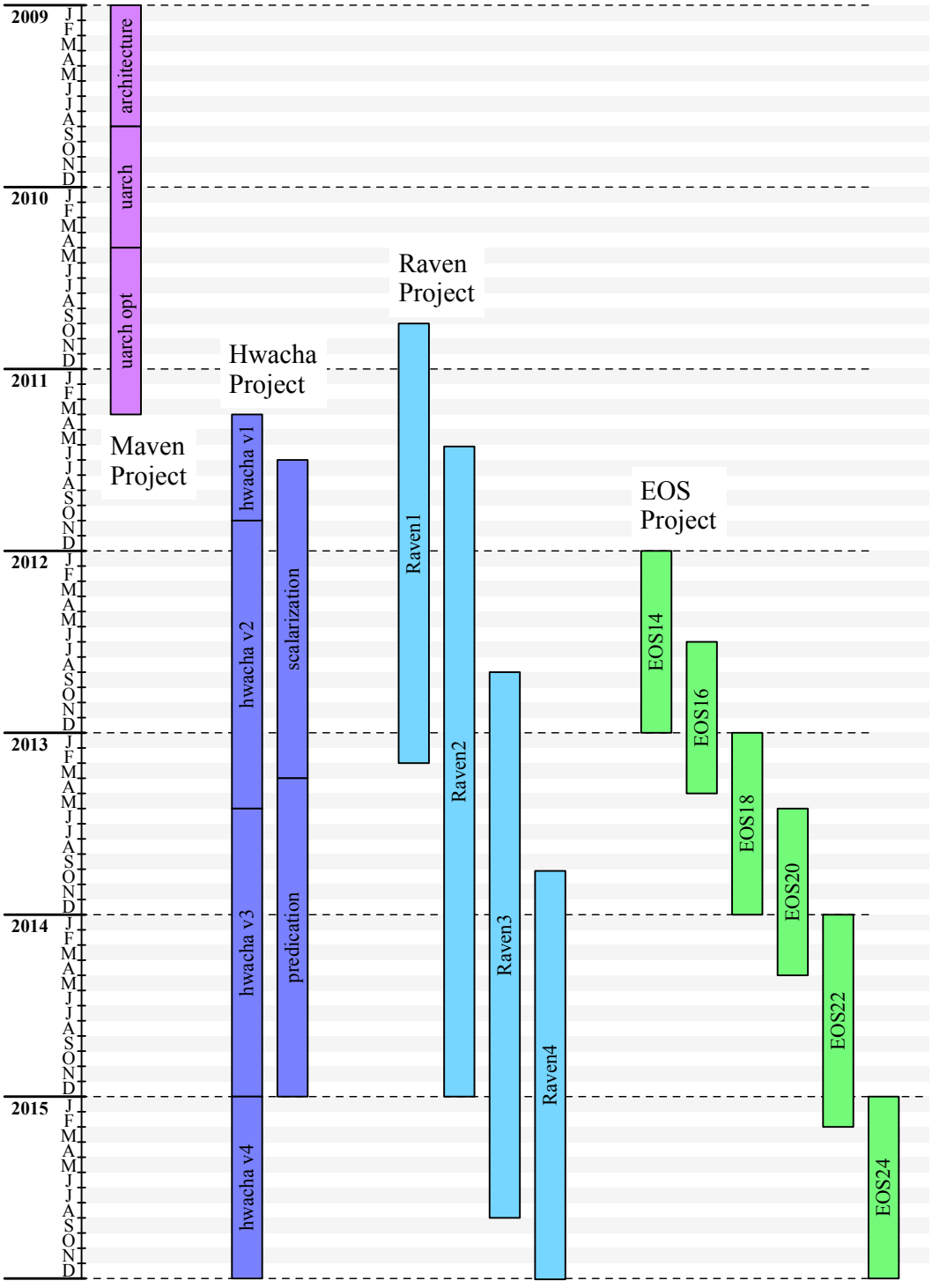


Figure 10.1: Thesis Timeline – We have co-developed the Hwacha vector-fetch architecture with multiple tapeouts for the Raven project and EOS project. [79] has more details on the four 28 nm Raven chips and the six 45 nm EOS chips we have built.

EOS chips integrated a 64 bit dual-core RISC-V processor and the Hwacha vector accelerator with monolithically-integrated silicon photonics links [78, 122]. Our IEEE Micro magazine article [79] details our agile hardware development methodology on how we co-designed our architecture and chips together, and how it enabled us to build industry-competitive 1 GHz–1.65 GHz RISC-V vector microprocessors with a small team. Overall, it took about 5 years to go from the initial concept to an actual implementation that can execute compiled OpenCL benchmarks and generate performance and energy numbers for a comparison against an industry-built GPU.

10.2 Future Work

Sections 4.5, 5.5, and 7.3 discussed specific directions for future work with respect to scalarization, predication, and the Hwacha vector instruction set architecture. The last section of this thesis briefly outlines more general thoughts on future work.

Open-Source the Hwacha Implementation. We plan to open-source the Hwacha golden functional model, Hwacha RTL, Hwacha verification suite, and our custom LLVM-based scalarizing compiler, which takes OpenCL programs and generates Hwacha assembly code, in the near future. This is the actual design we have taped-out multiple times on the 28 nm and 45 nm process nodes that achieved 1 GHz–1.65 GHz. We have been finding good design patterns to express hardware in Chisel across four generations of Hwacha RTL that have been written from scratch every generation. We also hope that the open-source Hwacha RTL can serve as a good example for other Chisel designers to see how modern programming language features such as object-oriented programming, functional programming, parameterized types, abstract data types, operator overloading, and type inference can improve hardware designer productivity and increase code reuse.

Execute the Vector-Fetch Block Temporally for Better Energy Efficiency. The current Hwacha microarchitecture maintains an instruction window that keeps track of a couple of vector instructions that are in flight, and only retires the oldest vector instruction from the instruction window once all elements in the vector are fully sequenced. This means that all vector operands are read from and written to the SRAM-based vector register file, which is expensive to access in terms of energy. The next step would be to explore microarchitectural ideas to alter the execution order of all operations within the vector-fetch block to better expose temporal locality. Rather than executing the vector-fetch block for all elements at once, an alternate scheduling scheme would execute the vector-fetch block for a strip of elements (i.e., a subset of the vector) before moving on to the next strip. With this new scheduling scheme, the compiler will be able to exploit short definition-use chains [89], and put these operands in a separate vector temporary register file, which would typically be mapped to a small flip-flop-based register file to reduce operand access energy [46, 62]. The vector temporary registers will let the compiler to shrink the element state that needs to be preserved in the SRAM-based vector register file across vector-fetch blocks, therefore increasing the maximum hardware vector length, which in turn can better hide long memory latencies for vector gather operations. The alternate scheduling scheme will also increase the likelihood of a

strip having the same consensual branch condition, therefore improving performance and energy efficiency of executing code with highly irregular control flow.

Further Evaluate the Benefits of Mixed-Precision Support. GPUs have been gradually adding better hardware support for mixed-precision operations. The recently added half-precision operations in various GPUs are a good example of this. However, the benchmarks and applications that we have used in this thesis do not fully take advantage of this feature yet. The next step would be to further evaluate the benefits of the mixed-precision extension with benchmarks that are written with first-class mixed-precision support in mind. We may also benefit from floating-point precision-optimizing tools such as Precimonius [108] to assist programmers in writing mixed-precision floating-point benchmarks.

Further Evaluate the Multi-Lane and Multi-Core Hwacha Decoupled Vector Accelerator. Chapter 9 has focused on evaluating the single-lane Hwacha design point. However, the Hwacha microarchitecture and the RTL is designed to support multi-lane and multi-core design points. With these multi-lane and multi-core design points in mind, we should consider dialing down the clock frequency that is used to synthesize and place-and-route the design. The next step would be to push these additional design points through the VLSI flow to not only see the impact on area, performance, and energy efficiency numbers, but also compare these numbers with high-end data-parallel accelerators that have more execution resources.

Further Generalize Accelerator Support in RISC-V. The Hwacha vector-fetch architecture was co-developed with the Rocket Chip SoC generator as the driving example on how to attach accelerators within the generator framework. We have distilled the important interactions between a RISC-V core and an accelerator down to the RoCC (Rocket Custom Coprocessor) interface. The next step would be to further generalize this RoCC interface to support a wider range of accelerators, and to develop software tools that generate custom compiler toolchains for new accelerators.

Bibliography

[1] John R. Allen and Ken Kennedy. “Automatic Loop Interchange”. In: *SIGPLAN Symposium on Compiler Construction (SIGPLAN)* 19.6 (June 1984), pp. 233–246.

[2] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. “Conversion of Control Dependence to Data Dependence”. In: *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Jan. 1983), pp. 177–189.

[3] Randy Allen and Ken Kennedy. “Automatic Translation of FORTRAN Programs to Vector Form”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.4 (Oct. 1987), pp. 491–542.

[4] Randy Allen and Ken Kennedy. “Vector Register Allocation”. In: *IEEE Transactions on Computers* 41.10 (Oct. 1992), pp. 1290–1317.

[5] AMD. *AMD Graphic Core Next Architecture*. AMD Fusion Developer Summit 11. 2011.

[6] AMD. *Graphics Core Next Architecture, Generation 3*. AMD White Paper. 2015.

[7] AMD. *Southern Islands Series Instruction Set Architecture*. AMD White Paper. 2012.

[8] ARM. *ARM Cortex-A15 MPCore Processor*. Technical Reference Manual. 2013.

[9] ARM. *big.LITTLE Technology: The Future of Mobile*. ARM White Paper. 2013.

[10] ARM. *Introducing NEON Development Article*. ARM White Paper. 2009.

[11] ARM. *Midgard GPU Architecture*. Oct. 2014.

[12] Krste Asanović. *Torrent Architecture Manual*. Tech. rep. EECS Department, University of California, Berkeley, Dec. 1996.

[13] Krste Asanović. “Vector Microprocessors”. PhD thesis. EECS Department, University of California, Berkeley, 1998.

[14] Krste Asanović, James Beck, Bertrand Irissou, Brian Kingsbury, Nelson Morgan, and John Wawrzynek. “The T0 Vector Microprocessor”. In: *Symposium on High Performance Chips (Hot Chips)* (Aug. 1995).

[15] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006.

[16] Krste Asanović, Brian Kingsbury, Bertrand Irissou, James Beck, and John Wawrzynek. "T0: A Single-Chip Vector Microprocessor with Reconfigurable Pipelines". In: *European Solid-State Circuits Conference (ESSCIRC)* (Sept. 1996).

[17] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Design Automation Conference (DAC)* (June 2012), pp. 1212–1221.

[18] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. "Analyzing CUDA Workloads Using a Detailed GPU Simulator". In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (Apr. 2009), pp. 163–174.

[19] Thomas Ball. "What's in a Region?: or Computing Control Dependence Regions in Near-linear Time for Reducible Control Flow". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 2.1-4 (Mar. 1993), pp. 1–16.

[20] Utpal Banerjee. "Data Dependence in Ordinary Programs". MA thesis. Department of Computer Science, University of Illinois at Urbana-Champaign, 1976.

[21] Christopher Batten. "Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators". PhD thesis. Massachusetts Institute of Technology, 2010.

[22] Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanović. "Cache Refill/Access Decoupling for Vector Machines". In: *International Symposium on Microarchitecture (MICRO)* (Dec. 2004).

[23] A.J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (Oct. 1966), pp. 757–763.

[24] Dileep Bhandarkar and Richard Brunner. "VAX Vector Architecture". In: *International Symposium on Computer Architecture (ISCA)* (May 1990), pp. 204–215.

[25] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2008), pp. 101–113.

[26] W. Buchholz. "The IBM System/370 Vector Architecture". In: *IBM Systems Journal* 25.1 (1986), pp. 51–62.

[27] Gregory T. Byrd and Mark A. Holliday. "Multithreaded Processor Architectures". In: *IEEE Spectrum* 32.8 (Aug. 1995), pp. 38–46.

[28] Christopher Celio. *Characterizing Multi-Core Processors Using Micro-benchmarks*. UC Berkeley Parlab Winter 2012 Retreat. 2012.

[29] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. "Rock: A High-Performance Sparc CMT Processor". In: *IEEE Micro Magazine* (Mar. 2009), pp. 6–16.

[30] 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: *International Symposium on Workload Characterization (IISWC)* (Oct. 2009), pp. 44–54.

[31] Sylvain Collange. *Identifying Scalar Behavior in CUDA Kernels*. Tech. rep. hal-00555134. Université de Lyon, Jan. 2011.

[32] Henry M. Cook, Andrew S. Waterman, and Yunsup Lee. *TileLink Cache Coherence Protocol Implementation*. White Paper. 2015.

[33] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintão Pereira, and Wagner Meira Jr. "Divergence Analysis and Optimizations". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Oct. 2011), pp. 320–329.

[34] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (4 Oct. 1991), pp. 451–490.

[35] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits (JSSC)* 9.5 (Oct. 1974), pp. 256–268.

[36] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. "Ocelot: a Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Sept. 2010), pp. 353–364.

[37] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. "AltiVec Extension to PowerPC Accelerates Media Processing". In: *IEEE Micro Magazine* 20.2 (Mar. 2000), pp. 85–95.

[38] Benoît Dupont de Dinechin. "Using the SSA-Form in a Code Generator". In: *International Conference on Compiler Construction (CC)* (Apr. 2014), pp. 1–17.

[39] Alexandre E. Eichenberger and Edward S. Davidson. "Register Allocation for Predicated Code". In: *International Symposium on Microarchitecture (MICRO)* (Nov. 1995), pp. 180–191.

[40] Christopher Eoyang, Raul H. Mendez, and Olaf M. Lubeck. "The Birth of the Second Generation: The Hitachi S-820/80". In: *International Conference on High Performance Networking and Computing (Supercomputing)* (Nov. 1988), pp. 296–303.

[41] Roger Espasa and Mateo Valero. "Decoupled Vector Architectures". In: *International Symposium on High-Performance Computer Architecture (HPCA)* (Feb. 1996), pp. 281–290.

[42] Roger Espasa, Mateo Valero, and James E. Smith. "Out-of-Order Vector Architectures". In: *International Symposium on Microarchitecture (MICRO)* (Dec. 1997), pp. 160–170.

[43] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and its Use in Optimization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9 (3 July 1987), pp. 319–349.

[44] Andrei Frumusanu. *The Samsung Exynos 7420 Deep Dive - Inside A Modern 14nm SoC*. AnandTech Article. 2015.

[45] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. “Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 6.2 (June 2009), pp. 1–35.

[46] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. “A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors”. In: *ACM Transactions on Computer Systems (TOCS)* 30.2 (Apr. 2012), 8:1–8:38.

[47] Syed Zohaib Gilani, Nam Sung Kim, and Michael J. Schulte. “Power-efficient Computing for Compute-intensive GPGPU Applications”. In: *International Symposium on High-Performance Computer Architecture (HPCA)* (Feb. 2013), pp. 330–341.

[48] David M. Gillies, Dz-ching Roy Ju, Richard Johnson, and Michael Schlansker. “Global Predicate Analysis and Its Application to Register Allocation”. In: *International Symposium on Microarchitecture (MICRO)* (Dec. 1996), pp. 114–125.

[49] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. “Practical Dependence Testing”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (May 1991), pp. 15–29.

[50] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Taylor. “The GreenDroid Mobile Application Processor: An Architecture for Silicon’s Dark Future”. In: *IEEE Micro Magazine* 31.2 (Mar. 2011), pp. 86–95.

[51] Linley Gwennap. “Digital, MIPS Add Multimedia Extensions”. In: *Microprocessor Report* 10.15 (Nov. 1996), pp. 1–5.

[52] Mark Hampton. “Reducing Exception Management Overhead with Software Restart Markers”. PhD thesis. Massachusetts Institute of Technology, 2008.

[53] Mark Hampton and Krste Asanović. “Compiling for Vector-Thread Architectures”. In: *International Symposium on Code Generation and Optimization (CGO)* (Apr. 2008).

[54] HardKernel. *ODROID-XU3 Block Diagram*. HardKernel Products Wiki Page.

[55] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach 5th Edition*. Morgan Kaufmann, 2011.

[56] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. Intel White Paper. 2015.

[57] Intel. *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. Intel White Paper. 2012.

[58] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. "pocl: A Performance-Portable OpenCL Implementation". In: *International Journal of Parallel Programming* 43.5 (Oct. 2015), pp. 752–785.

[59] Handel Jones. *Why Migration to 20nm Bulk CMOS and 16/14nm FinFETs is not Best Approach for Semiconductor Industry*. Tech. rep. IBS, Inc., Jan. 2014.

[60] Ralf Karrenberg and Sebastian Hack. "Improving Performance of OpenCL on CPUs". In: *International Conference on Compiler Construction (CC)* (Mar. 2012), pp. 1–20.

[61] Ralf Karrenberg and Sebastian Hack. "Whole-Function Vectorization". In: *International Symposium on Code Generation and Optimization (CGO)* (Apr. 2011), pp. 141–150.

[62] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. "GPUs and the Future of Parallel Computing". In: *IEEE Micro Magazine* 31.5 (Sept. 2011), pp. 7–17.

[63] Andrew Kerr, Gregory Diamos, and S. Yalamanchili. "Dynamic Compilation of Data-parallel Kernels for Vector Processors". In: *International Symposium on Code Generation and Optimization (CGO)* (Apr. 2012), pp. 23–32.

[64] Ji Kim, Christopher Torng, Shreesha Srinath, Derek Lockhart, and Christopher Batten. "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures". In: *International Symposium on Computer Architecture (ISCA)* (June 2013), pp. 130–141.

[65] Christoforos Kozyrakis. "Scalable Vector Media-processors for Embedded Systems". PhD thesis. EECS Department, University of California, Berkeley, 2002.

[66] Ronny Krashinsky. "Vector-Thread Architecture and Implementation". PhD thesis. Massachusetts Institute of Technology, 2007.

[67] Ronny Krashinsky, Christopher Batten, and Krste Asanović. "Implementing the Scale Vector-Thread Processor". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 13.3 (July 2008).

[68] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. "The Vector-Thread Architecture". In: *International Symposium on Computer Architecture (ISCA)* (June 2004).

[69] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *International Symposium on Code Generation and Optimization (CGO)* (Mar. 2004), pp. 75–88.

[70] Yunsup Lee. "Efficient VLSI Implementations of Vector-Thread Architectures". MA thesis. EECS Department, University of California, Berkeley, 2011.

[71] Yunsup Lee, Rimas Avižienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. "Exploring the Tradeoffs Between Programmability and Efficiency in Data-Parallel Accelerators". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (Aug. 2013), 6:1–6:38.

[72] Yunsup Lee, Rimas Avižienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators". In: *International Symposium on Computer Architecture (ISCA)* (June 2011), pp. 129–140.

[73] Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W. Keckler, and Krste Asanović. "Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures". In: *International Symposium on Microarchitecture (MICRO)* (Dec. 2014), pp. 101–113.

[74] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanović. "Convergence and Scalarization for Data-Parallel Architectures". In: *International Symposium on Code Generation and Optimization (CGO)* (Feb. 2013), pp. 1–11.

[75] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and Krste Asanović. *The Hwacha Microarchitecture Manual, Version 3.8.1*. Tech. rep. UCB/EECS-2015-263. EECS Department, University of California, Berkeley, Dec. 2015.

[76] Yunsup Lee, Colin Schmidt, Sagar Karandikar, Daniel Dabbelt, Albert Ou, and Krste Asanović. *Hwacha Preliminary Evaluation Results, Version 3.8.1*. Tech. rep. UCB/EECS-2015-264. EECS Department, University of California, Berkeley, Dec. 2015.

[77] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1*. Tech. rep. UCB/EECS-2015-262. EECS Department, University of California, Berkeley, Dec. 2015.

[78] Yunsup Lee, Andrew Waterman, Rimas Avižienis, Henry Cook, Chen Sun, Vladimir Stojanović, and Krste Asanović. "A 45nm 1.3GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector Accelerators". In: *European Solid-State Circuits Conference (ESSCIRC)* (Sept. 2014), pp. 199–202.

[79] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtić, Stevo Bailey, Milovan Blagojević, Pi-Feng Chiu, Rimas Avižienis, Brian Richards, Jonathan Bachrach, David Patterson, Elad Alon, Borivoje Nikolić, and Krste Asanović. "An Agile Approach to Building RISC-V Microprocessors". In: *IEEE Micro Magazine* (Mar. 2016).

[80] Yunsup Lee, Brian Zimmer, Andrew Waterman, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Ben Keller, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Henry Cook, Rimas Avižienis, Brian Richards, Elad Alon, Borivoje Nikolic, and Krste Asanović. "Raven: A 28nm RISC-V Vector Processor with Integrated Switched-Capacitor DC-DC Converters and Adaptive Clocking". In: *Symposium on High Performance Chips (Hot Chips)* (Aug. 2015).

[81] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro Magazine* 28.2 (Mar. 2008), pp. 39–55.

[82] Chris Lomont. *Introduction to Intel Advanced Vector Extensions*. Intel White Paper. 2011.

[83] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. "A Comparison of Full and Partial Predicated Execution Support for ILP Processors". In: *International Symposium on Computer Architecture (ISCA)* (June 1995), pp. 138–149.

[84] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. "Effective Compiler Support for Predicated Execution Using the Hyperblock". In: *International Symposium on Microarchitecture (MICRO)* (Dec. 1992), pp. 45–54.

[85] David Maier. "The Complexity of Some Problems on Subsequences and Supersequences". In: *Journal of the ACM (JACM)* 25.2 (Apr. 1978), pp. 322–336.

[86] Rick Merritt. "ARM CTO: power surge could create 'dark silicon'". In: *EE Times* (Oct. 2009).

[87] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* (Apr. 1965), pp. 114–117.

[88] Gordon E. Moore. "No Exponential is Forever: But "Forever" Can Be Delayed!" In: *International Solid-State Circuits Conference (ISSCC)* (Feb. 2003), pp. 20–23.

[89] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[90] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors". In: *International Symposium on High-Performance Computer Architecture (HPCA)* (Feb. 2003), pp. 129–140.

[91] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable Parallel Programming with CUDA". In: *ACM Queue Magazine (ACMQ)* 6.2 (Mar. 2008), pp. 40–53.

[92] John Nickolls, Richard Craig Johnson, Robert Steven Glanville, and Guillermo Juan Rozas. *Unanimous Branch Instructions in a Parallel Thread Processor*. US Patent 8,677,106. Mar. 2014.

[93] NVIDIA. *CUDA Binary Utilities*. NVIDIA Application Note. 2014.

[94] NVIDIA. *NVIDIA CUDA C Programming Guide 4.2*. Apr. 2012.

[95] NVIDIA. *NVIDIA Tegra X1: NVIDIA's New Mobile Superchip*. Jan. 2015.

[96] NVIDIA. *NVIDIA's Next Gen CUDA Compute Architecture: Kepler GK110*. NVIDIA White Paper. 2012.

[97] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. NVIDIA White Paper. 2009.

[98] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. "The Program Dependence Web: a Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 1990), pp. 257–271.

[99] Albert Ou. "Mixed-Precision Vector Processors". MA thesis. EECS Department, University of California, Berkeley, 2015.

[100] Albert Ou, Quan Nguyen, Yunsup Lee, and Krste Asanović. "A Case for MVPs: Mixed-Precision Vector Processors". In: *International Workshop on Parallelism in Mobile Platforms (PRISM)* (June 2014).

[101] Joseph C. H. Park and Mike Schlansker. *On Predicated Execution*. Tech. rep. HPL-91-58. Hewlett Packard Laboratories, May 1991.

[102] Alex Peleg and Uri Weiser. "MMX Technology Extension to the Intel Architecture". In: *IEEE Micro Magazine* 16.4 (July 1996), pp. 42–50.

[103] Matt Pharr and William R. Mark. "ispc: A SPMD Compiler for High-Performance CPU Programming". In: *Innovative Parallel Computing (InPar)* (May 2012).

[104] Jungyul Pyo, Youngmin Shin, Hoi-Jin Lee, Sung-il Bae, Min-Su Kim, Kwangil Kim, Ken Shin, Yohan Kwon, Heungchul Oh, Jaeyoung Lim, Dong-Wook Lee, Jongho Lee, Inpyo Hong, Kyungkuk Chae, Heon-Hee Lee, Sung-Wook Lee, Seongho Song, Chung-Hee Kim, Jin-Soo Park, Heesoo Kim, Sunghee Yun, Uk-Rae Cho, Jae Cheol Son, and Sungho Park. "20nm High-K Metal-Gate Heterogeneous 64b Quad-Core CPUs and Hexa-Core GPU for High-Performance and Energy-Efficient Mobile Application Processor". In: *International Solid-State Circuits Conference (ISSCC)* (Feb. 2015), pp. 420–421.

[105] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. "Implementing Streaming SIMD Extensions on the Pentium-III Processor". In: *IEEE Micro Magazine* 20.4 (July 2000), pp. 47–57.

[106] John H. Reif and Harry R. Lewis. "Efficient Symbolic Analysis of Programs". In: *Journal of Computer and System Sciences* 32.3 (June 1986), pp. 280–314.

[107] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A Cycle Accurate Memory System Simulator". In: *IEEE Computer Architecture Letters (CAL)* 10.1 (Jan. 2011), pp. 16–19.

[108] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. "Precimonious: Tuning Assistant for Floating-Point Precision". In: *International Conference on High Performance Networking and Computing (Supercomputing)* (Nov. 2013), 27:1–27:12.

[109] Richard M. Russell. "The CRAY-1 Computer System". In: *Communications of the ACM (CACM)* 21.1 (Jan. 1978), pp. 63–72.

[110] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Sugerman, and Pat Hanrahan. "Larrabee: A Many-Core x86 Architecture for Visual Computing". In: *IEEE Micro Magazine* 29.1 (Jan. 2009), pp. 10–21.

[111] M. Sharir. "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers". In: *Computer Languages* 5.3-4 (Jan. 1980), pp. 141–153.

[112] Jaewook Shin. "Introducing Control Flow into Vectorized Code". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Sept. 2007), pp. 280–291.

[113] Youngmin Shin, Ken Shin, Prashant Kenkare, Rajesh Kashyap, Hoi-Jin Lee, Dongjoo Seo, Brian Millar, Yohan Kwon, Ravi Iyengar, Min-Su Kim, Ahsan Chowdhury, Sung-il Bae, Inpyo Hong, Wookyeong Jeong, Aaron Lindner, Ukrae Cho, Keith Hawkins, Jae Cheol Son, and Seung Ho Hwang. "28nm High-K Metal-Gate Heterogeneous Quad-Core CPUs for High-Performance and Energy-Efficient Mobile Application Processor". In: *International Solid-State Circuits Conference (ISSCC)* (Feb. 2013), pp. 154–155.

[114] James E. Smith. "Decoupled Access/Execute Computer Architectures". In: *ACM Transactions on Computer Systems (TOCS)* (Nov. 1984), pp. 289–308.

[115] James E. Smith, Greg Faanes, and Rabin Sugumar. "Vector Instruction Set Support for Conditional Operations". In: *International Symposium on Computer Architecture (ISCA)* (June 2000), pp. 260–269.

[116] Ryan Smith. *ARM's Mali Midgard Architecture Explored*. AnandTech Article. 2014.

[117] Ryan Smith. *Imagination's PowerVR Rogue Architecture Explored*. AnandTech Article. 2014.

[118] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. "Flexible Software Profiling of GPU Architectures". In: *International Symposium on Computer Architecture (ISCA)* (June 2015), pp. 185–197.

[119] Arthur Stoutchinin and Francois De Ferriere. "Efficient Static Single Assignment Form for Predication". In: *International Symposium on Microarchitecture (MICRO)* (Dec. 2001), pp. 172–181.

[120] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. "Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs". In: *International Symposium on Code Generation and Optimization (CGO)* (Apr. 2010), pp. 111–119.

[121] 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*. Tech. rep. IMPACT-12-01. University of Illinois, Urbana-Champaign, Mar. 2012.

[122] Chen Sun*, Mark T. Wade*, Yunsup Lee*, Jason S. Orcutt*, Luca Alloatti, Michael S. Georgas, Andrew S. Waterman, Jeffrey M. Shainline, Rimas R. Avižienis, Sen Lin, Benjamin R. Moss, Rajesh Kumar, Fabio Pavanello, Amir H. Atabaki, Henry M. Cook, Albert J. Ou, Jonathan C. Leu, Yu-Hsin Chen, Krste Asanović, Rajeev J. Ram, Milos A. Popović, and Vladimir M. Stojanović. "Single-Chip Microprocessor that Communicates Directly Using Light". In: *Nature* 528 (Dec. 2015), pp. 534–538.

[123] Hiroshi Tamura, Sachio Kamiya, and Takahiro Ishigai. "FACOM VP-100/200: Supercomputers with ease of use". In: *Parallel Computing* 2.2 (June 1985), pp. 87–107.

[124] Michael B. Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *IEEE Micro Magazine* 33.5 (Sept. 2013), pp. 8–19.

[125] *The OpenCL Specification Version 1.2*. Khronos OpenCL Working Group. 2011.

[126] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. "VIS Speeds New Media Processing". In: *IEEE Micro Magazine* 16.4 (July 1996), pp. 10–20.

[127] Huy Vo, Yunsup Lee, Andrew Waterman, and Krste Asanović. "A Case for OS-Friendly Hardware Accelerators". In: *Workshop on the Interaction between Operating System and Computer Architecture (WIVOSCA)* (June 2013).

[128] Tadashi Watanabe. "Architecture and performance of NEC supercomputer SX system". In: *Parallel Computing* 5.1–2 (July 1987), pp. 247–255.

[129] Andrew Waterman. "Design of the RISC-V Instruction Set Architecture". PhD thesis. EECS Department, University of California, Berkeley, 2016.

[130] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Compressed Instruction Set Manual, Version 1.9*. Tech. rep. UCB/EECS-2015-209. EECS Department, University of California, Berkeley, Nov. 2015.

[131] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014.

[132] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.7*. Tech. rep. UCB/EECS-2015-49. EECS Department, University of California, Berkeley, May 2015.

[133] Michael Weiss. "The Transitive Closure of Control Dependence: the Iterated Join". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1 (2 June 1992), pp. 178–190.

[134] Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili. "Characterization and Transformation of Unstructured Control Flow in Bulk Synchronous GPU Applications". In: *International Journal of High Performance Computing Applications (IJHPCA)* 26.2 (May 2012), pp. 170–185.

[135] Brian Zimmer, Yunsup Lee, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtić, Ben Keller, Stevo Bailey, Milovan Blagojević, Pi-Feng Chiu, Hanh-Phuc Le, Po-Hung Chen, Nicholas Sutardja, Rimas Avižienis, Andrew Waterman, Brian Richards, Phillippe Flatresse, Elad Alon, Krste Asanović, and Borivoje Nikolić. "A RISC-V Vector Processor with Simultaneous-Switching Switched-Capacitor DC-DC Converters in 28nm FDSOI". In: *IEEE Journal of Solid-State Circuits (JSSC)* (Apr. 2016).

[136] Brian Zimmer, Yunsup Lee, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtić, Ben Keller, Stevo Bailey, Milovan Blagojević, Pi-Feng Chiu, Hanh-Phuc Le, Po-Hung Chen, Nicholas Sutardja, Rimas Avižienis, Andrew Waterman, Brian Richards, Phillippe Flatresse, Elad Alon, Krste Asanović, and Borivoje Nikolić. "A RISC-V Vector Processor with Tightly-Integrated Switched-Capacitor DC-DC Converters in 28nm FDSOI". In: *Symposium on VLSI Circuits* (June 2015), pp. C316–C317.