UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title

Design Techniques for Specialized Short-Vector Machines

Permalink

https://escholarship.org/uc/item/1tv3m0vb

ISBN

9798293893614

Author

Grubb, Daniel

Publication Date

2025-08-01

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Design Techniques for Specialized Short-Vector Machines

by

Daniel Grubb

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolić, Chair
Professor Krste Asanović
Professor Christopher Fletcher
Dr. Ali Sazegari

Summer 2025

Design Techniques for Specialized Short-Vector Machines

Copyright 2025
by
Daniel Grubb

Abstract

Design Techniques for Specialized Short-Vector Machines

by

Daniel Grubb

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

Systems-on-chip (SoCs) for mobile devices need flexible yet energy efficient processing capabilities in order to target evolving multimedia and embedded machine-learning applications while coping with constrained device battery life. Digital signal processors (DSPs) are frequently integrated into these SoCs under a wide variety of physical constraints to fill this role. DSPs have evolved significantly over the last four decades, but have always had the goal of providing programmable, domain-specialized computation in constrained deployments, guided by trends in silicon design and the important applications of the day. The emergence of modern scalable vector ISAs, such as the RISC-V "V" vector extension (RVV), presents the opportunity to design DSPs by building on a scalable, efficient vector machine template.

This thesis presents techniques for designing and implementing domain-specialized short-vector machines, suitable for deployment as DSPs in mobile SoCs. First, this thesis discusses a generator for baseline short-vector machines implementing the RVV 1.0 specification, including quantitative performance, power, and area (PPA) results and a qualitative comparison to other DSP core paradigms. This thesis presents techniques for specializing short-vector machines for mobile multimedia workloads to enhance their efficiency and enable new PPA trade-offs, including the design of novel vector instructions that compose neatly with the baseline vector machine. This thesis presents contributions to Hammer, a reusable physical design framework, as a means for rapidly evaluating and implementing these designs. Finally, this thesis presents Cygnus, a 1GHz heterogeneous octa-core RISC-V vector processor for digital signal processing. Cygnus demonstrates the silicon implementation of instances of the short-vector machine generator described in this thesis organized in a big/little architecture. Cygnus achieves over 90% datapath utilization on matrix multiplication and convolution kernels, and 414 GOPS/W and 109 GFLOPS/W on INT8 and FP32 matrix multiplication kernels, respectively.

To my family.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **vii**

1 Introduction **1**

2 Background **4**
2.1 Mobile SoCs 4
2.2 Mobile Multimedia Applications 7
2.3 Mobile Multimedia Processing 9
2.4 DSP Cores 12
2.5 Modern Scalable Vector Architectures 16

3 A Short-Vector Machine Platform for DSP Cores **20**
3.1 Short-Vector Machines 21
3.2 Short-Vector Machines as DSP Cores 25
3.3 The Saturn Vector Unit 28
3.4 Saturn Evaluation 38
3.5 Related Work, Summary, and Conclusions 43

4 Techniques for Specializing Short-Vector Machines **45**
4.1 Baseline Short-Vector Machines 46
4.2 Short-Vector Machines as Templates for Domain Specialization 47
4.3 Vector Backend Sequencer Configurations 51
4.4 Vector Operation Fusion 55
4.5 Adding Specialized Vector Instructions 57
4.6 Accelerating Multidimensional Workloads 59
4.7 Accelerating Convolutions 63
4.8 Specialized Vector Instruction Results 70
4.9 Summary 75

5 Silicon Implementation and Design Methodologies **77**
5.1 Hammer: a Reusable VLSI Flow Generator 78
5.2 Hammer Evaluation 81
5.3 Saturn Implementation and Evaluation 84
5.4 Cygnus DSP Chiplet 87

6 Conclusion **102**

Bibliography **105**

List of Figures

2.1 A high-level view of the frontend processing in the SIFT pipeline [74]. 8
2.2 A summary of mobile multimedia processing categories in mobile SoCs. 11
2.3 The grouping of a subset of the 32 architectural vector registers according to when the `LMUL` setting in RVV is ≥ 1. Increasing `LMUL` groups more vector registers together. 18
2.4 A vector-length-agnostic FP32 vector-scalar multiply loop in RVV assembly. . . 19
2.5 RVV processing example as in Figure 2.4, for a vector machine with `VLEN=128b`. The input vector contains 19 single-precision elements. `LMUL` is set to 4, so `VLMAX=16`. On the first iteration, four vector registers are grouped such that 16 elements will be processed. On the second iteration, the vector length is trimmed to process the remaining 3 elements. 19

3.1 High-level view of a short-vector machine. 22
3.2 Overview of the unified VRF, shown with 4 banks and `VLEN/DLEN=2`. 23
3.3 Vector-length-agnostic FP32 FMA kernel leveraging vector register grouping. . . 27
3.4 Block diagram overview of the Saturn vector unit. VDQ = Vector Dispatch Queue, VRF = Vector Register File, VLSU = Vector Load-Store Unit. 30
3.5 Saturn vector backend configuration block diagrams [132]. 34
3.6 Execute sequencer block diagram for a Saturn instance with `VLEN/DLEN=2`. . . . 35
3.7 Saturn performance on SGEMM. 40
3.8 Saturn datapath utilization on SGEMM. 41
3.9 Saturn area results for several configurations [132]. 42
3.10 Saturn power results for several configurations [132]. 42
3.11 Annotated Saturn PnR results for a V512D256 configuration [132]. 43

4.1 Baseline and multi-FMA sequencer configurations of Saturn. 52
4.2 Saturn multi-FMA sequencer performance. 53
4.3 Saturn multi-FMA sequencer normalized area cost. 54
4.4 Example of fusable vector move and FMA instructions. 55
4.5 Convolution vectorization example using offset vector loads to align the data. . . 56
4.6 Offset load-based convolution code snippet. 56
4.7 Convolution with sliding input code snippet. 57
4.8 Convolution with sliding accumulator code snippet. 57

4.9 `vmvmacc` instruction operation. 61
4.10 Mapping of source vector registers for the `vmvmacc` instruction for several `LMUL` settings. 61
4.11 Snippet of GEMM inner loop with and without the `vmvmacc` instruction. 62
4.12 `vconv2` instruction functionality. 65
4.13 Tiled convolution using the `vconv4` instruction. 65
4.14 Convolution with `vconv2` and `vconv4` instructions. 66
4.15 Decoupled functional unit integration in Saturn. The decoupled functional unit can be implemented in a more flexible manner than the pipelined functional units and advertises its outstanding element group writes to the sequencers for the purposes of hazard checking. 68
4.16 Comparison of matrix multiplication performance of Saturn instances, including the implementation of the `vmvmacc` instruction. 71
4.17 Normalized area results for the `vmvmacc` instruction implementation in the Intel16 process technology. 72
4.18 Normalized energy efficiency results for executing a 64x64 matrix multiplication with the `vmvmacc` instruction. 72
4.19 Specialized convolution instruction normalized performance comparison for a 5x5 2D convolution. 73
4.20 Specialized convolution instruction normalized energy efficiency results for a 5x5 2D convolution. 74
4.21 Performance and energy efficiency of the unsharp mask pipeline for Saturn variants with and without the specialized convolution instructions implemented. . . 75

5.1 Hammer framework block diagram [72]. 79
5.2 Hammer evaluation of design reuse across various chips [72]. 83
5.3 Hammer simulation input keys. 86
5.4 Hammer power input keys. 88
5.5 Block diagram of the Cygnus DSP chiplet architecture [59]. 91
5.6 Block diagram of the Cygnus memory hierarchy [59]. 92
5.7 Cygnus GDS rendering with block overlays [59]. The Big RVV tiles have `VLEN=512b` and `DLEN=256b`. The Little RVV tiles have `VLEN=128b` and `DLEN=64b`. The LLC is 512KB and comprises four banks. 95
5.8 Cygnus package [59]. 96
5.9 Cygnus test setup [59]. 96
5.10 Cygnus shmoo plot [59]. 97
5.11 Vector core 64x64 SGEMM results [59]. 98
5.12 Vector core 128x128 IGEMM results [59]. 99
5.13 Cygnus multi-core 192x192 SGEMM results [59]. 100
5.14 Cygnus multi-core 384x384 IGEMM results [59]. 100
5.15 Performance and energy efficiency results for the big and little vector core tiles executing kernels with varying arithmetic intensities [59]. 101

5.16 Utilization of the big and little vector core tiles executing an SGEMM across a sweep of square matrix sizes. The little vector cores achieve a higher utilization at smaller problem sizes, demonstrating how utilizing the little vector cores may be advantageous [59]. 101

List of Tables

2.1 A high-level comparison of some of the features of SVE and RVV. 17

3.1 Saturn functional unit summary. 39

5.1 Comparison of chips using Hammer [72]. 82
5.2 Comparison of Big and Little Vector Units 94

Acknowledgments

Writing these acknowledgements, I'm struck by feelings of joy about how many amazing and talented people I've met during this journey. I hope these acknowledgements can convey even a fraction of how thankful I am. After spending nearly a decade at Berkeley for both my undergraduate and graduate studies, I feel incredibly grateful to have lived these years in such a wonderful and vibrant place, both academically and socially.

I would like to start by thanking Bora Nikolić, my advisor, for all his advice and continual support throughout my PhD. Bora, I appreciate the trust you have in your students to take on big systems projects and your ability to help guide them through it, exemplified by starting me off with managing a big tapeout in my first year of grad school. I have also always relished chatting in group meetings and one-on-ones about the goings-on of the day, whether it be semiconductor news or basketball. I would next like to thank Ali Sazegari both as a mentor and a member of my dissertation committee. Ali, I deeply appreciate the perspective you brought as a PhD and an industry expert; it has shaped my development as a grad student and an engineer. I have learned a great deal from your ability to identify the most important details when building and evaluating systems, and to communicate about them effectively. I would like to thank Krste Asanović for all his advising on the Saturn project and for serving on my dissertation committee. Saturn project meetings were always enlightening and were critical for the work in this dissertation. I would also like to thank Christopher Fletcher for serving on my dissertation committee and Yakun Sophia Shao for serving on my quals committee. There are many other faculty members that I learned from while at Berkeley who contributed to the foundation of my EECS experience. I would also like to thank the amazing staff at BWRC, Slice, and EECS, including Candy, Mikaela, Brian, James, Caroline, Jeff, Ria, Tami, Kosta, Judy, Jean, and Shirley.

I feel very fortunate to have built a connection with some of those who were senior students at BWRC when I was just starting out. While there are always senior students when you start out in grad school, less than two years after starting we would be in the midst of a pandemic, and that kind of connection was much more difficult to form. Bora connected me to the people at Zendar to interview for an internship the summer before I started grad school and I went to work for Antonio Puglielli, who had also been my EE140 GSI the previous Fall. Antonio, you are a great mentor and you provided me with great advice (and retrospectively some eerily prescient warnings) about grad school. Antonio also introduced me to another critical piece of the BWRC puzzle, Ajith Amerasekera. Ajith, I always deeply appreciate talking with you, with your incredible depth of experience and insight. With regard to the GPS SoC project for my first part of grad school, I'd like to thank Sijun Du for being a great collaborator. The whole tapeout sprint lasting until early Summer 2019 was a very formative experience. I would like to thank John Wright and Colin Schmidt for apprenticing Harrison Liew and I on taping out. Regularly working 20+ hour days in the lead up to the tapeout deadline wasn't always fun, but we powered through with copious amounts of caffeine from BWRC Diet Coke and coffee. The experience was made bearable by the camaraderie. I learned so much from both of you and it helped define

the remainder of my PhD, between the knowledge of chip design I gained and leading to developing and maintaining Hammer with Harrison and others. On that note, I'd also like to thank Edward Wang for starting the Hammer project.

Harrison, thank you for being such a talented and cool person. You were my partner for lots of projects, and it was only fitting that we became roommates by the end. I have learned a lot from you, not just technically, but also about life; I admire your spirit for just going out and doing things. After the doldrums of COVID, I was so happy to be part of a cohort of students in a revitalized BWRC and Slice Lab. Ken Ho and Felicia Guo, I feel very fortunate to have met both of you and I'm incredibly glad we became friends. We all learned a lot, including the importance of keeping the BWRC coffee machine clean. What more can I say than you do good work?

The Saturn project was another defining portion of my PhD. Working with Jerry Zhao, Miles Rusch, Tianrui Wei, Kevin Anderson, and Albert Ou was fantastic. Jerry, thank you for being a great collaborator, leader, and friend. I learned a lot from you, and collaborating on Saturn and related projects was incredibly rewarding, as was stopping by Triple Rock and Jupiter. Albert, I really enjoyed collaborating with you and I appreciate all the knowledge that you have shared on vector architectures and beyond. I'm very happy we ended up as officemates. Kevin, thank you for being a bastion of sanity. I'd also like to thank Vikram Jain, who was a wonderful injection of new perspective and experience into the lab. Vikram, collaborating with you, and everyone else on the Cygnus team, was great and helped bring my PhD full circle. I'll always remember when we got the first vector code running on Cygnus. I'm excited to see the research that will come out of your lab.

There are so many others who made my experience at Berkeley special. Avi and Lars, I'm so glad we became friends, and I'm looking forward to attending more sporting events together in the future. Jean-Luc, it was quite fortuitous that we became roommates based on a mid-COVID eecs-grads-housing cold email; it was a blast cooking chicken parmesan that was way too big. Vighnesh and Bob, thank you for being my TAs in EECS151 and being part of the cohort that toughed it out at Berkeley through all these years. Doing things like wandering around the DAC showrooms and cooking pizzas in a wood-fired oven were lots of fun. Nayiri, thank you for joining Harrison and me in working on Hammer and completing the quadrangle of Bora group desks at BWRC. Thank you to Dan, Seah, Abe, Hasan, Paul, Joonho, Yufeng, Ethan, Zhenghan, Yue, Raghav, Junsun, Zhaokai, Dima, Ella, Kunmo, Alon, Sean, and everyone else at BWRC and Adept/Slice for supporting me on this journey and making my experience amazing.

I owe so much to my family. My parents, Mary and John, have given so much to me and I am so deeply thankful for them. Without their guidance and support, I would not be where I am today. It also didn't hurt that they instilled a love for Cal in me from an early age. Thank you, Mom and Dad. To my siblings Scott and Sophia, thank you for being amazing role models and dear friends. I greatly benefited from being the youngest and having you help show me the way.

Finally, I would like to thank my absolutely amazing girlfriend, Gitika. I cannot articulate how thankful I am for your support, cheering, and patience. You were right there with me

through all the ups and the downs of this whole process. Your constant belief in me kept me going through the tough times. You always showed me how to keep a good perspective on things (despite my sometimes resistance) and you always knew how to make me laugh. I am delighted by your all-encompassing creativity, whether it manifests in your writing, in decorating a space, or when crafting a new hot take. Your determination to achieve at the highest level on your own journey (with the talent to match) has inspired me ever since our first date at Cornerstone. You beat me to finishing grad school and I could not be more proud of you. I could not have done this without you.

Chapter 1

Introduction

Mobile devices, such as smartphones, laptops, and wearables, owe their tremendous functionality and battery life in no small part to the performance and efficiency of their underlying systems-on-chip (SoCs). Mobile SoCs are naturally heterogeneous given the breadth of their use, integrating blocks across the programmability continuum, from high-performance CPUs to largely fixed-function datapaths for video and audio codecs. Some of the most important functionality of mobile devices is how they interface with the user and the outside world. Cameras, speakers, microphones, radio transceivers, displays, and more all generate signals, which must be processed. This processing encompasses application domains such as computer vision, computational photography, digital signal processing, and embedded machine learning. These workloads tend to be highly data-parallel, making them good targets for acceleration. However, these workloads are also highly varied and fast evolving, demanding processing flexibility to ensure that devices can target new applications efficiently. Digital signal processors (DSPs) often fill this role, providing a balance between programmability and efficiency. Mobile SoCs often integrate DSPs across a variety of design points, from large, highly performant cores down to small, highly constrained cores. Domain- or application-specific accelerators have been hailed as the solution to the challenges in SoC design stemming from troubling scaling trends in deeply scaled CMOS processes. Surely, they are a critical part of the solution, but they often lack the processing flexibility required in the mobile SoC context to obviate the need for DSPs.

Originally built largely as real-time filtering machines, DSPs have evolved tremendously over the last four decades to target applications in computer vision, audio, communications, machine learning, and more. In the mobile SoC context, energy efficiency is often of the highest concern given the need for ever-increasing functionality and the constraints of battery life. To improve their energy efficiency, DSP cores are extended with domain-specialized features like specialized vector instructions. In fact, the design of DSP cores has always been guided by the computational characteristics of their target application domains. Since a given mobile SoC may deploy a multitude of DSP cores across different design points, designing a scalable machine that can be parameterized to meet various PPA constraints while retaining a common programming model is advantageous. This is made even more

important by the high costs of SoC design and software. Furthermore, these DSP cores are often targeted by low-level performance programmers to squeeze out the best possible performance and energy efficiency for a given application. Designing DSP cores that reduce the burden of meticulous static scheduling and promote kernel performance portability can improve programmer productivity and provide other design savings.

To design energy efficient, domain-specialized DSP cores, optimizations must be applied across the stack, from co-designing important algorithms with the instruction set down to the gate-level implementation. This thesis focuses on the design of these specialized DSP cores, beginning with the design of a *short-vector machine* generator that implements a modern scalable vector ISA. This thesis then outlines design techniques for specializing these cores to improve their energy efficiency for mobile multimedia applications. Finally, this thesis presents contributions to a reusable physical design framework suitable for rapid design evaluation and implementation, building to the tapeout of a chip that leveraged this framework and integrated a number of the aforementioned DSP cores.

Some of the content in this thesis is adapted from the following publications, with permission from their co-authors: "Instruction Scheduling in the Saturn Vector Unit" [131], "The Saturn Microarchitecture Manual" [132], "Hammer: A Modular and Reusable Physical Design Flow Tool" [72], "Cygnus: A 1 GHz Heterogeneous Octa-Core RISC-V Vector Processor for DSP" [59].

Chapter 2 provides salient background for this thesis, beginning with a discussion of mobile SoCs, mobile multimedia applications, and common multimedia processing blocks. This contextualizes the role that DSP cores play in mobile SoCs and points out the important common characteristics of mobile multimedia workloads that can be leveraged for designing efficient hardware. This is followed by a brief history of DSP cores and a discussion of their important components and common design paradigms. Finally, modern scalable vector ISAs are introduced, including the RISC-V "V" vector extension (RVV), which will be the baseline ISA leveraged by other parts of this thesis.

Chapter 3 discusses the design of a baseline short-vector machine generator. It begins by qualitatively comparing short-vector machines against the dominant DSP core paradigms of today, namely VLIW and narrow RISC machines, each typically integrating P-SIMD extensions of some form. This is followed by a discussion on Saturn, a parameterized Chisel generator for RVV 1.0 vector units, suitable for integration into mobile SoCs. Saturn demonstrates the advantages of short-vector machines in the DSP core context. Importantly, it shows the use of efficient hardware sequencing mechanisms and a decoupled access-execute-type architecture to achieve high datapath and memory resource utilizations on both small and large problem sizes, rather than a reliance on extensive static scheduling and high instruction throughput. Saturn-based designs serve as our baseline template for designing specialized short-vector machines.

Chapter 4 focuses on design techniques for improving the PPA characteristics of short-vector machines for mobile multimedia workloads. Examining the PPA results of differently parameterized Saturn instances in Chapter 3, we find that the PPA design space is quite coarse: large changes in performance require large changes in power and area. By design-

ing cost-effective and flexible domain-specialized features, we can significantly improve the efficiency of short-vector machines and enable a finer-grained design space. Chapter 4 then presents a set of specialized vector instructions that improve our baseline's performance and energy efficiency by more naturally capturing the dataflow of the target computation; enabling the integration of additional functional units; amortizing the overall costs of instruction fetch, decode, and sequencing; and reducing the movement of data in and out of the vector register file. A particular focus is placed on multidimensional kernels important in mobile multimedia workloads, such as convolutions and matrix multiplications. PPA results for select instructions integrated into Saturn are presented.

Chapter 5 centers on how short-vector machines can be rapidly evaluated and implemented. Physical design methodologies that enable design reuse can improve the design space exploration process, potentially resulting in more efficient silicon implementations and reduced design times. The first part of Chapter 5 presents contributions to Hammer, a reusable physical design framework. An empirical study shows that Hammer enabled significant design flow reuse across a diverse series of chips taped out in various process technologies. The second part of Chapter 5 presents the Cygnus chiplet, a 16mm^2 octa-core RISC-V vector processor for DSP applications, taped out in the Intel16 process technology using Hammer. Cygnus is the first silicon demonstration of the Saturn vector unit and achieves over 90% datapath utilization on matrix multiplication and convolution kernels, as well as best-in-class energy efficiencies of 414 GOPS/W and 109 GFLOPS/W on INT8 and FP32 matrix multiplication kernels, respectively.

Chapter 6 concludes this thesis, providing a summary of the important points and contributions, as well as a discussion of potential directions for future work.

Chapter 2

Background

Mobile systems-on-chip (SoCs) provide the computation necessary for the key functionality of mobile phones, laptops, tablets, wearables, and more. Mobile SoCs have been able to achieve much success by taking advantage of cutting-edge silicon technology advancements, integrating billions of transistors that make up numerous subsystems. Despite the difficulty and expense of SoC design, many companies now design their own mobile SoCs to differentiate products with new functionality while maintaining sufficient device battery life.

Multimedia applications in the domains of image processing, audio processing, and more are among the most important for mobile devices, as they are such a prominent part of the user experience. These applications change rapidly and grow in complexity, making them difficult targets for SoC architects looking to design ever more efficient and performant devices. While multimedia processing in mobile SoCs comes in many forms, digital signal processors, or DSP cores, fill an important niche by providing high-performance and energy-efficient execution for multimedia applications in constrained deployments.

This chapter motivates and provides context for the remainder of this thesis. Section 2.1 explores mobile SoC architectures and discusses why DSP cores are important components. Section 2.2 discusses mobile multimedia applications and their salient characteristics. A particular focus is placed on image processing due to its ubiquity and importance. Section 2.3 reviews some important categories of multimedia processing in mobile SoCs, including common design patterns for building specialized image processing hardware. Section 2.4 discusses DSP cores and how they leverage the nature of multimedia algorithms and economical hardware features to be efficient, yet flexible, compute platforms. Finally, section 2.5 explores the notable characteristics of modern scalable vector ISAs.

2.1 Mobile SoCs

Mobile devices must deliver a massive variety of functionality for users. This is particularly true in the multimedia application domain, as reflected by some of the most prominent benchmark suites today [39][60]. Mobile SoCs have become product differentiators because

they can enable new and improved functionality while maintaining device battery life. This has driven companies to design their own complex mobile SoCs despite the immense costs of design, verification, and manufacturing required [2][78][96][108][44][62].

Mobile SoCs must be energy efficient because of the constraints of limited device battery life. Given the slowing of transistor density scaling and the lack of Dennard-style scaling of power density [33] in modern CMOS processes, SoCs must leverage domain specialization to be efficient [49]. Thus, mobile SoCs are naturally *heterogeneous*. In other words, mobile SoCs comprise many subsystems, each tailored to a particular domain or functionality, with a wide range of programmability. This is reflected in the typical subsystem makeup of many mobile SoCs:

- The application CPUs are responsible for running general-purpose code and orchestrating other parts of the SoC. This subsystem is mostly made up of high-performance, speculative, out-of-order (OoO) CPUs. These machines are tailored towards the best performance on general-purpose code, where single-thread performance is often most critical. Instructions per cycle (IPC) is a critical metric and typically higher performance implies higher overall energy efficiency. Even within this processing complex, there is heterogeneity. Big/little architectures integrate some application cores that are higher performance and others that are less performant but lower power [65][133]. This paradigm has been dominant in mobile SoCs, though some are now moving away from it, opting for systems made from all big cores [79].

- The GPU is used for graphics rendering for the device and may also be used for data-parallel compute for domains like machine learning. GPUs are very programmable but far less general-purpose than the CPUs.

- The "neural processing unit," or NPU, is typically some form of large array of multiply-accumulate units (MACs) targeted at machine-learning workloads [3][103]. NPUs are a relatively new addition to mobile SoCs but are now ubiquitous. The computational primitive is often matrix multiplication, although other functionality may be supported. NPUs are more domain-specialized than the CPUs and GPU, delivering high throughput and energy efficiency on a narrower domain of workloads at the cost of flexibility.

- The camera subsystem and image signal processor (ISP) processes pixels from the camera and images stored on the device. The ISP is used for correcting and enhancing images and providing computation for applications like video playback.

- A mobile device integrates many sensors and audio functionality, which all must be managed and their data processed.

- The display subsystem manages the display output that may be rendered by the GPU or ISP, and processes user input.

- A mobile device like a smartphone supports a myriad of radio connectivity, including cellular, WiFi, Bluetooth, and locationing (e.g. GNSS), all of which require processing for transmission or reception.

The CPU, GPU, and NPU subsystems are largely made up of their eponymous compute blocks. The makeup of the other subsystems, largely responsible for multimedia processing, among other functionality, is somewhat less clear. Not captured directly in the above list is the processing that underlies these blocks. In many cases, these blocks contain one or more DSP-style cores. These DSP cores often make up the backbone of efficient yet flexible specialized processing in mobile SoCs. While DSP cores traditionally targeted DSP kernels like FIR filters and FFTs [73], they have evolved to encompass a much wider range of target domains, including computer vision, audio, machine learning, and more [21][118][27]. Section 2.4 will discuss the specifics of DSP cores in more detail.

Domain-specific accelerators have commonly been cited as a solution to the limitations stemming from the slowdown in silicon technology scaling [49]. Fixed-function solutions for a multimedia algorithm can be orders of magnitude more energy efficient than programmable ones [45][77]. So why do we care about DSP cores at all in this age of accelerators? While domain-specific accelerators are a critical part of the solution to this problem, they do not obviate the need for DSP cores. There are several important reasons for this.

Processing flexibility is critical in mobile SoCs, particularly for multimedia applications given that they change rapidly over time. For instance, machine learning became pervasive in mobile devices for computer vision, audio processing, and natural language processing over a relatively short span of time in recent years. A mobile SoC must be capable of efficiently processing future applications not yet imagined at the time of tapeout. Domain-specific accelerators deliver higher efficiencies than more programmable blocks for a particular domain of workloads, but are often too inflexible to target a sufficiently broad range of applications efficiently. DSP cores provide this flexibility and can be specialized to enhance their efficiency.

As CMOS process technologies have advanced, they have become more expensive. NRE costs have increased in terms of design, verification, and manufacturing. The slowing of Moore-style transistor density scaling [83] also means that CMOS technology advancements cannot be solely relied upon for uplifts in performance. This is exacerbated by issues with interconnect scaling and signal integrity [47]. Thus, integrating bespoke accelerators for every kernel or application in a domain is not necessarily practical in terms of design costs and the allocation of silicon area. Some sacrifice in energy efficiency may be made in favor of a flexible, compact DSP core capable of delivering the required performance for a whole class of applications.

Given these reasons, it is important to understand how to design DSP cores and how to specialize them in a flexible, scalable, and reusable manner. To do this, we will study multimedia algorithms and design patterns for specialized processing in mobile SoCs.

2.2 Mobile Multimedia Applications

Image processing, along with the related domains of computer vision and computational photography, is one of the most important and computationally intensive classes of multimedia algorithms in mobile devices. Pixel processing and computational photography process camera sensor data and enable photo editing and image enhancements [119]. Pixel data from the camera must be processed through a series of stages including demosaicing, noise reduction, color space conversion, and compression, among other steps [19]. Object detection and image stereo matching may be underlain by feature-detection pipelines like the Scale-Invariant Feature Transform (SIFT) [74]. Object recognition, mobile visual search, and face recognition are other important computer vision applications underlain by both learning-based and classical techniques [123]. Beyond image processing, audio-processing applications, such as text-to-speech and active noise cancellation, may use a mix of classical and learning-based algorithms [102]. This is just a small sampling of the types of multimedia applications that mobile SoCs may support, but from it we can discuss some of their characteristics which are also more broadly applicable. We will mostly focus on image processing as a driving application domain given its importance and general computational complexity.

One of the defining characteristics of image-processing workloads, and many other multimedia workloads, is that they tend to be highly data-parallel. To be efficient, processing hardware must leverage this inherent parallelism in a cost-effective manner. The input data is frequently multidimensional, which may also be leveraged by processing hardware. Mobile multimedia applications are also highly varied in terms of their operations, their problem sizes, and the datatypes they operate on [64]. Image-processing pipelines commonly take in narrow integer data and then operate on wider integers or floating-point datatypes for increased precision. These characteristics have implications for the varied types of operations and datatypes the processing in mobile SoCs must support. Beyond the varied nature of a particular workload, multimedia applications tend to change rapidly over time, more rapidly than the refresh of a mobile SoC. This is embodied by the evolution of machine-learning applications for mobile devices, but even the use of more classical algorithms changes quickly as new applications develop. Many computer vision and audio applications may use a combination of learning-based and classical techniques. These characteristics of mobile multimedia workloads explain why processing flexibility is a **requirement** in mobile SoCs.

Figure 2.1 shows an overview of the first few steps of the SIFT feature-detection pipeline [74]. A scale-space representation is generated from the input image by blurring it with a series of Gaussian filters and downsampling it to create a Gaussian pyramid. Difference-of-Gaussian (DoG) images are then generated by subtracting adjacent images in this space. The result of these frontend steps are passed to a backend that detects potential image keypoints by finding local extrema, refines them, and finally generates descriptors that can be used in keypoint matching algorithms. Other feature-detection algorithms, such as Oriented FAST and Rotated BRIEF (ORB) [106], Speeded Up Robust Features (SURF) [16], Harris corner detection [46], and Canny edge detection [22], while differing in their details and complexity, all exhibit similar computational characteristics. Beyond feature detection, many other

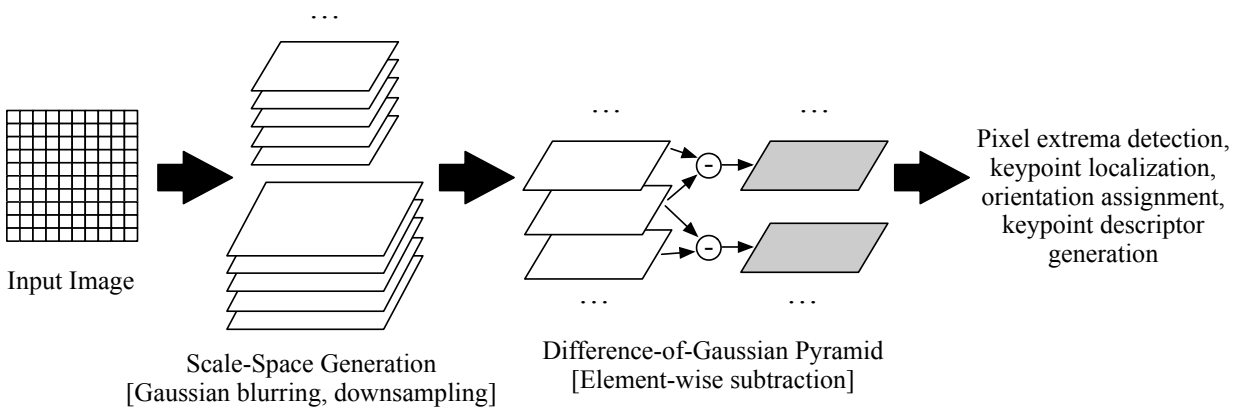


Figure 2.1: A high-level view of the frontend processing in the SIFT pipeline [74].

algorithms in computer vision and in computational photography also follow similar patterns.

Most common image-processing kernels can be sorted into several categories [119]. Point-wise kernels operate on pixels independently of one another, which might include operations like scaling and compositing. Neighborhood kernels generate output pixels by considering a group of pixels, or neighborhood, around each corresponding input pixel. This is perhaps best embodied by the convolution operation. Global operators consider pixels from across an image or across a large window of pixels, such as in image histogramming. The Gaussian filtering operations and the pixel extrema detection in Figure 2.1 are examples of neighborhood operators, while the final DoG computation is a point-wise operation.

These types of kernels each have different data reuse characteristics. Individual point-wise operators have very low reuse since they execute just a single operation on each pixel. Neighborhood operators tend to have reuse that grows with the size of the neighborhood used. For a 2D convolution with a kernel size, `KxK`, over an image of size `NxN`, the operational intensity is proportional to $\texttt{K}^2$. For 2D separable filters, such as Gaussian and Sobel, the operational intensity is proportional to `K`.

Figure 2.1 shows how image-processing applications are often made up of a series of many operations that are organized into a pipeline. Pipeline stages can be fused together to increase locality among the inputs and outputs of various pipeline stages. This is a critical performance optimization for image-processing pipelines [98]. Doing so increases the effective arithmetic intensity of a set of operations, even if each individual operation has a lower arithmetic intensity. For example, some of the successive operations in the SIFT pipeline across some of the steps shown in Figure 2.1 might be fused together to improve performance, rather than finishing an entire step for all data at once before moving on.

Beyond performance implications, maximizing locality in image-processing pipelines is critical for energy efficiency. Image data in mobile devices must typically be read from the sensor or DRAM, and then written to DRAM. The energy and latency cost of reads

from DRAM is extremely high, far higher than any single arithmetic operation that will be executed [53][20]. Thus, this cost must be amortized by executing a number of operations all together, ideally with each pixel only being read from DRAM a single time.

Beyond classical image processing, machine learning continues to grow in importance in mobile devices and is used heavily in computer vision applications [55][80]. For deep neural networks like convolutional neural networks (CNNs), inference workloads are organized into pipelines of layers that compute various operations, such as different flavors of convolutions, fully connected layers, pooling, activation functions, and more. These layers may be underlain by kernels such as matrix multiplications, 2D multilayer convolutions, and matrix-vector multiplications. Large matrix multiplications and 2D multi-layer convolutions tend to have very high levels of reuse, making them ideal candidates for compute accelerators.

2.3 Mobile Multimedia Processing

Mobile SoCs employ a number of processing design patterns for efficiently accelerating multimedia applications, from fixed-function to programmable. Where a particular block is designed to land on this continuum is determined by the performance and energy efficiency requirements of a particular application domain and silicon area constraints.

These design decisions are underlain by the trade-offs in CMOS implementations between flexibility and efficiency. Fundamentally, the most efficient way to process a particular computation is with a fixed-function datapath and specialized local memories that manage data movement [77]. The precise implementation can be guided by the energy-delay characteristics of fixed-function designs, which differ based on architectural, microarchitectural, and circuit-level parameters (e.g. supply voltage) [97].

The higher energy efficiency of these blocks compared to programmable solutions comes from the fact that no power or silicon area is spent on general-purpose processing capabilities like instruction fetch, decode, and sequencing. Furthermore, these blocks do not waste energy through implicit data movement through a cache hierarchy. The cost of data movement is often very high, even in comparison to the computation itself [53]. Thus, for fixed-function designs, computation and local memory movement make up a much larger proportion of the total energy cost [45].

Fixed-function datapaths for multimedia applications in mobile SoCs are most often implemented for video codecs such as H.264 [127], VP9 [85], and AV1 [29]. The NVIDIA Jetson Orin Series SoC implements hardware accelerators for H.264, VP9, and AV1 [62], with many other mobile SoCs implementing similar functionality. For these applications, fixed-function datapaths are both practical and justified. Video and audio codecs are standardized, meaning that fixed hardware blocks can be designed without fear of the target algorithm changing. Furthermore, these codecs are very computationally intensive, making them good targets for hardware accelerators to achieve large energy savings in a mobile SoC.

Mobile SoCs also often have specialized datapaths for camera ISP pipelines [19] for reasons similar to video codecs. The throughput requirements for image data coming from the

camera is very high and it must be processed before sending it to DRAM to save energy. As described in the previous section, image-processing hardware must maximally leverage locality to be efficient. These pipelines tend to have some limited programmability to allow image statistics to improve image quality, but the overall structure of the pipeline may be largely fixed, with specialized line buffer memories connecting each computational stage to maximize efficiency [124][50][20].

While these application-specific accelerators are terrific for matching the needs of their target applications efficiently, they are inflexible. Thus, they are practical only for the types of applications that need not change much and might otherwise be prohibitively expensive to deploy onto a programmable processor.

The next class of multimedia processing blocks we will discuss is domain-specific accelerators [49]. Mobile systems have deployed a plethora of programmable, domain-specific accelerators. These are blocks that can achieve impressive throughputs and energy efficiencies in an application domain, while retaining a greater level of flexibility than application-specific accelerators.

The Pixel Visual Core (PVC) is a programmable image processor for mobile phones [101]. The PVC comprises a 2D array of processing elements (PEs) that are interconnected in a north-south-east-west (NSEW) pattern for local communication of data. To handle the padding of image edges required for many convolutions, the PVC adds a set of halo PEs. The PVC also takes a page out the book of traditional image processors by using a more generalized and dynamic form of line buffers to manage image data efficiently. The PVC leverages the 2-dimensional nature of the domain to spatially map image-processing kernels and reduce the cost of data movement, just as much older supercomputers did in a similar fashion [13][14].

Other spatial arrays of many forms have seen tremendous interest in recent years due to the rise in importance of machine-learning applications [40][61]. Spatial arrays again leverage the 2D nature of operations like matrix multiplications and convolutions, enabling the scaling of the number of useful processing elements, while retaining a modicum of programmability. Having dedicated in-place accumulators improves data locality and energy efficiency.

The Qualcomm Hexagon is a family of VLIW DSPs integrated into Snapdragon SoCs that may target high performance and energy efficient computation for digital signal processing, image processing, machine learning, and communications [96]. In recent years, the Hexagon has been extended to accelerate matrix computation [75]. The Hexagon was extended with a matrix computation capability, to augment its vector processing, in order to increase throughput and efficiency in the machine-learning domain.

These domain-specific accelerators in mobile SoCs are tremendously successful in domains with a relatively narrow set of key kernels. Matrix accelerators targeting the matrix-multiply primitive in machine-learning applications are perhaps the best example of this. However, domain-specific accelerators are ill-suited to computation outside their target domain and are often inefficient when operating on small problem sizes or problem sizes with dimensions that do not match those of the spatial array. Because they are particularly useful when scaled up in size, they often consume significant silicon area.

Multimedia Processing Category	Description	Examples	Target Domain Examples
Fixed-function or limited programmability	Hardware blocks with very high efficiency and throughput on a very narrow application range; may have some limited programmability to tune parameters or adjust target output	Camera ISPs, video codecs	Image processing, computational photography, video processing
Domain-specific with some programmability	Accelerators that can efficiently compute a set of primitives with enough programmability to target a set of applications in a particular domain	Systolic arrays, outer product units, NPUs, other PE arrays, mobile GPUs	Machine learning, linear algebra, graphics processing
Constrained, flexible, domain-specialized	Processors for power- and area-constrained deployments requiring high flexibility to target a host of multimedia applications	DSP cores, VLIW, narrow-RISC, P-SIMD, vector extensions	Signal processing, image processing, machine learning
Highly flexible, general-purpose	High-throughput processing capable of achieving good performance on non-optimized vector applications	Superscalar OoO CPUs with SIMD extensions	General-purpose vectorized applications

Figure 2.2: A summary of mobile multimedia processing categories in mobile SoCs.

The final class of multimedia processing we will discuss in this section is tightly integrated SIMD processing [57]. Packed-SIMD-based (P-SIMD) multimedia extensions to scalar ISAs proliferated in the 1990s as an inexpensive way to increase multimedia processing throughput using sub-word data-parallel processing. Since then, P-SIMD ISAs have been implemented in most cores found in mobile SoCs, from high-performance out-of-order cores to small embedded cores. These P-SIMD ISA extensions are a cost-effective mechanism for exploiting data-level parallelism in multimedia applications. As a result, P-SIMD extensions have evolved to define wide vector registers and datapaths in order to further improve performance and energy efficiency.

P-SIMD ISA extensions represent a highly flexible form of acceleration for multimedia workloads. Being tightly coupled to a scalar processor means that the use of SIMD functionality is highly programmable. This flexibility comes at an efficiency cost compared to domain-specific and fixed-function accelerators. As a result, P-SIMD ISAs contain many specialized instructions tuned to particular multimedia computations in a bid to improve their energy efficiency and performance. For example, P-SIMD ISAs may include specialized instructions for operating on complex data or for computing the sum-of-absolute differences (SAD) kernel [4].

Figure 2.2 summarizes some of the processing paradigms for mobile multimedia applications discussed in this section. The categories of processing blocks generally change from more application- or domain-specific to more flexible moving down the chart. As noted in section 2.2, DSP cores are ubiquitous in mobile SoCs as a baseline of programmable yet efficient computation. Almost all DSP cores integrate some form of SIMD-based data-parallel processing due to its cost-efficacy. The following section discusses DSP cores, particularly their key features that make them staples of mobile SoCs.

2.4 DSP Cores

DSP Core History

Over their long history, digital signal processors have undergone many evolutions in their architectures, target application domains, and deployments. DSPs are, however, all united by the goal of delivering performant and efficient numerical computation for DSP applications, and related domains, while remaining programmable.

The TMS320 line of DSPs from Texas Instruments, first introduced in the early 1980s, is one of the most well-known and successful entries in this space [73]. These DSP chips took advantage of advances in semiconductor technology to implement new hardware features that enabled real-time processing for DSP applications. These chips integrated a dedicated multiplier datapath, and then a multiply-accumulate (MAC) unit. This was critical since MAC operations underlie almost all DSP kernels, and were previously typically implemented much more slowly with shifts and adds. High instruction throughput and clock speeds were achieved by using a Harvard architecture, separating instruction memories from data memories, and pipelining. Furthermore, specialized instructions were designed to orchestrate data movement and to combine common operations together to allow for more efficient loops. While the design focus and scale of silicon implementation capabilities of these early DSPs were vastly different from those today, the challenges their designers identified and the spirit of their solutions closely mirror modern DSPs.

A major sea change in DSP design was the adoption of the very long instruction word (VLIW) paradigm. The TI C6x line of VLIW fixed-point and floating-point DSPs was introduced in 1997 in the same TMS320 line of products as the originals [113]. At the time, there was a desire to scale the number of concurrent arithmetic and memory operations to meet the growing computational needs of DSP applications. Advances in VLSI technology made the integration of more functional units in a single chip possible. However, the architects were concerned with the cost of the control logic required to utilize them efficiently. A superscalar processor with sufficient instruction throughput to effectively utilize all of the functional units would be costly from the silicon area and design complexity perspectives. The architects adopted the VLIW paradigm, aiming to drive the functional units concurrently with a high-IPC frontend that requires only minimal control logic.

VLIW machines shift the complexity of instruction scheduling from the hardware to the

compiler or kernel writer [50]. While relying on the compiler to solve this problem for general-purpose systems is known to be problematic [49], the proposal for DSPs is more enticing. Many DSP workloads consist of regular loops and a high density of arithmetic operations that can be more easily scheduled using software pipelining [68] and loop unrolling techniques. These DSPs also adopt lightweight techniques to compress incomplete instruction packets that would otherwise need to explicitly encode no-ops [105].

Around the same time, in the late 1990s, interest grew in customizable DSP IP cores. The Tensilica Xtensa platform was introduced for building customized processors from a simple five-stage, in-order core [42]. The Xtensa design emphasized flexible processor IP that could be modified rapidly and brought to market quickly. The Tensilica Instruction Extension (TIE) framework allows designers to define custom instructions and additional hardware structures that are directly integrated with the base processor and software toolchain. The outcome of having such a platform is the ability to rapidly design domain-specific instructions to enhance performance and efficiency for a target application domain with a simple processor as the base.

Lx was a platform for designing VLIW-based DSP cores [37]. It studied the efficacy of scaling the DSP core to more concurrent operations and adding domain-specialized functional units, with the outcome being that doing so can yield massive enhancements for target DSP applications. However, this comes at the expense of growing hardware costs, which may be applicable only to a very narrow range of applications and yields diminishing returns. Countless more works in this space arose, laying the basis for the specialized processing in DSP cores used today [43].

Moving later into the 21st century, battery-powered mobile devices began to require their underlying SoCs to deliver more functionality, necessitating more specialized processing. Furthermore, the slowing of silicon technology scaling meant that energy efficiency and area efficiency became ever more important. A 2013 BDTI article entitled "Are DSPs Dying?" [18] pointed out that while DSP chips had seemingly left the public eye, DSP cores were more popular than ever. DSPs had simply mostly gone "underground," having continued to grow as components of SoCs as IP rather than as separate chips.

Today, DSP cores are an integral part of many mobile and embedded SoCs, spanning a wide design space. Vendors like Tensilica, Synopsys, and Ceva provide DSP core IP specialized for audio, computer vision, radio applications, and more [21][118][27]. Qualcomm integrates multiple instances of the Hexagon DSP as cornerstones of their Snapdragon SoCs [31].

Metrics

When evaluating DSP cores, there are several critical metrics. The three most fundamental metrics are performance (e.g. `operations/s`), power, and silicon area (PPA). From power, performance, and area we can find energy efficiency (i.e. energy per operation), which is critical in battery-powered devices. Any of these metrics may have an absolute specification or budget to meet, with the others then typically optimized given that constraint. We can

also find area efficiency (i.e. `operations/s/mm`^{2}), which is a measure of performance per silicon area. Area efficiency is important given that DSP cores are often used in highly physically constrained deployments and silicon area is costly.

Hardware resource utilization is a measure of what proportion of the maximum throughput is achieved for a particular resource, such as a particular arithmetic unit or memory unit. Maximizing utilization of the implemented resources has positive implications for energy efficiency and area efficiency. Underutilized resources cost silicon area and consume leakage power, which degrades efficiency. Recall that a given kernel can be characterized as being compute-bound or memory-bound. This informs which hardware resource should be considered when evaluating utilization. When comparing different designs against each other, utilization is most meaningful when put in the context of other metrics. For example, if two designs have the same maximum throughput for a particular resource, then the design with the higher utilization will have the higher performance. If their maximum throughputs are mismatched, this should be taken into account. Given that DSP cores are highly physically constrained, they strive to maximize resource utilization using techniques in order to maximize their energy efficiency and area efficiency.

Static code size for DSP kernels is also critically important given that embedded deployments may have hard constraints on available memory space. Microarchitectural features that allow programmers to write concise code that performs well are thus important to consider.

Beyond these metrics for a particular DSP core instance, flexibility and scalability are also important. Being highly flexible means that a DSP core can efficiently target many different types of applications, including those that may not be designed at the time of tapeout. Flexibility future-proofs the design and allows more applications to run on a core taking up minimal area. Scalability means that a style of DSP core may be scaled up and down to meet the PPA specification of a particular deployment. While features will doubtlessly change between design points, designing DSP cores that can be scaled up and down while retaining the same architecture and similar microarchitectures can save design costs. It also allows programmers to target the cores in similar fashions.

DSP Cores Today

Broadly, DSP cores today fall into two classes: typically higher performance VLIW cores and typically lower area narrow, in-order cores [51]. DSP cores derive their efficiency from leveraging the parallelism available in their target application domain with economical features. They may leverage all three classic forms of parallelism: data-level parallelism (DLP), instruction-level parallelism (ILP), and task-level parallelism (TLP) [50].

Since DLP is so abundant in DSP applications, integrating data-parallel processing is one of the most economical ways to improve efficiency in DSP cores. As a result, almost all DSP cores integrate SIMD datapaths. These SIMD units tend to be tightly integrated with the core. The class of core will determine the SIMD datapath width. Some may be as narrow as the width of a scalar register, while others may be very wide, upwards of 1Kbit

[75]. SIMD datapaths can significantly increase the energy efficiency of a core by increasing compute throughput and amortizing the energy costs of instruction fetch and sequencing over multiple operations encoded in a single instruction.

DSP cores leverage ILP through pipelining and also potentially VLIW frontends. High-performance DSP cores tend to use VLIW frontends because they can allow an increase in the number of concurrently issued instructions to drive functional units with little increase in the complexity of the instruction fetch and sequencing hardware. However, as previously noted, this shifts the complexity of instruction scheduling to the compiler or kernel writer.

Given the inflexibility of their execution and tightly coupled SIMD datapaths, both VLIW and narrow, in-order DSP cores rely on extensive static scheduling to achieve good resource utilization and performance. Techniques like loop unrolling and software pipelining can be highly effective for hiding functional unit latencies and amortizing loop overhead costs [68][100].

However, extensive static scheduling can lead to significantly increased static code size. This has been mitigated in many DSP cores by using mixed-instruction widths and techniques like rotating register files [52]. Furthermore, VLIW cores must handle no-ops stemming from incomplete instruction packets. While DSP cores have long dealt with this by delineating instruction packet boundaries without explicitly encoding no-ops [37], it is not free given that variable instruction packet boundaries add complexity to the VLIW instruction fetch unit.

The inflexibility of execution in these machines means that dynamic behaviors like variable memory latencies and loop control may not be handled well. These issues are mitigated by the use of explicitly managed, predictable latency memories and predication, but it does limit their efficacy when implemented in more dynamic systems.

TLP is also used in DSP core systems in several different forms. The first is with multithreading. Some DSP cores, such as the Qualcomm Hexagon, support several hardware threads that can be finely or coarsely interleaved [31]. This can help machine utilization by switching to a different thread when one is blocked, as in the case of waiting for a memory response. DSP multicore systems can leverage TLP and DLP by running different tasks on different DSP cores instances or by dividing up a single problem across multiple cores.

As previously alluded to, *specialization* is a key feature of DSP cores. While programmable systems can be orders of magnitude less efficient than their fixed-function counterparts [77][45], processing flexibility is a requirement in mobile SoCs. Domain specialization is a critical technique for narrowing this efficiency gap.

DSP cores have long included specialized features to enhance performance and efficiency [36]. Zero-overhead loops have been used to reduce the overhead of bookkeeping code in DSP loops by implementing loop counting in hardware. DSP cores have also integrated specialized addressing modes, like circular-buffer addressing and bit-reversed addressing to concisely encode memory access patterns found in filtering, FFTs, and other kernels.

DSP cores are also often integrated with predictable latency tightly coupled memories (TCMs). TCMs may not be cached and so can be managed by software, rather than having data movement managed implicitly through a cache hierarchy. This can allow for increased

memory bandwidth and can reduce the energy required for memory accesses at the cost of having to explicitly control it. Furthermore, some DSPs also integrate TCMs that are highly banked to enable high throughput scatter/gather memory accesses, which may be useful for sparse computations, lookup tables, and more. Examples of these include the capabilities in the Qualcomm Hexagon [31] and Tensilica Supergather [17].

One of the most powerful ways in which DSP cores can be specialized is through the use of specialized instructions. By encoding more complex computation patterns, significant energy savings can be made by driving more functional units at once or combining data movement or permutation with computation. These energy savings are achieved by further amortizing the costs of instruction fetch and sequencing and by reducing the number of accesses to register files and the memory system.

Specialized instructions may fuse multiple operations together that can be done in parallel or contribute to the same output. In more extreme cases, a single instruction may encode a complex kernel, effectively turning the DSP core into an accelerator for the life of the instruction. For example, Hameed, et al., [45] designed many such "magic" instructions to push the limits of the efficiency of a customizable DSP core for video coding applications. As another example, the Qualcomm Hexagon's vector extension, HVX, includes an instruction for computing an image histogram [31]. This plays into the continuum of DSP core flexibility and efficiency: more complex, specialized instructions enable large gains in efficiency at the cost of general utility.

To summarize, mobile SoCs require flexible, yet efficient, processing in many subsystems for multimedia processing. DSP cores often occupy this role. In essence, DSP cores integrate economical hardware features to efficiently leverage the parallelism available in multimedia applications. DSP cores often use a "cheap" frontend, in the form of VLIW or a narrow, in-order scalar machine, to reduce the area and power costs of instruction sequencing. This leads to a deep reliance on the extensive static scheduling of kernels. This frontend then drives a SIMD backend. These cores are then augmented with specialized instructions and hardware structures to enhance their energy efficiency. DSP core deployments come with many different PPA requirements and so it is beneficial to have solutions that are scalable and flexible in implementation. Ideally, this would mean that hardware design work can be reused and software might be portable in terms of compatibility and performance. The use of the *short-vector machine* design pattern in the DSP core context is the focus of Chapter 3 of this thesis. The following section gives background on modern scalable vector ISAs as the foundation for these machines.

2.5 Modern Scalable Vector Architectures

In recent years, scalable vector extensions to the Arm and RISC-V ISAs have emerged. This includes the Arm Scalable Vector Extension (SVE) [116], the ARM M-Profile Vector Extension (MVE) or Helium [5], and RVV [104]. These modern scalable vector architectures can trace their roots back to the original Cray-style vector machines [107]. While vector

architectures were long dominant in the supercomputer market and remain in use today [87], P-SIMD ISAs have been the data-parallel ISA extension paradigm of choice in mobile devices. Scalable vector ISAs have a number of well-known advantages over P-SIMD ISAs that can provide improved performance, efficiency, software portability, programmer ergonomics, and more [50][6].

SVE and RVV, both being scalable vector ISAs that define a rich set of vector memory and arithmetic operations, have some important similarities and differences, some of which are summarized in Table 2.1. These ISAs support a wide variety of classes and types of implementations, from the HPC to embedded domains, and from tightly coupled to decoupled vector implementations [35][6][48], respectively [120][132]. This freedom of implementation allows architects to customize their designs and devote hardware resources to portions of the machine as they see fit according to the constraints of their design.

Table 2.1: A high-level comparison of some of the features of SVE and RVV.

Feature	SVE	RVV
Vector Length Control	Predicate registers	Vector length register
Conditional Execution	Predicate registers	Mask in vector register
Overlay VRF on Scalar FRF?	Yes, FRF overlay	No, separate VRF and FRF
Vector Register Grouping	Limited multi-vector instructions	Dynamic vector register grouping with `LMUL` CSR field
Vector-Scalar Operations	Splat scalar value or index into vector	Vector-scalar instruction variants
Instruction Element Width	Encoded in instruction	`SEW` field in `vtype` CSR for non-vector memory instructions

Implementations of SVE and RVV define a vector length, `VLEN`, which is the width in bits of an architectural vector register. SVE chose to overlay the vector registers on the scalar floating-point registers, similar to ARM Neon, which may allow for a reduction in total area in some implementations. The Streaming SVE (SSVE) variant of SVE does not overlay the vector registers on the scalar floating-point register file, which may help implementations physically decouple the vector unit from the associated scalar unit. This is particularly relevant for implementations of Arm's Scalable Matrix Extension (SME) [4]. RVV does not overlay the vector registers on the scalar floating-point register file and defines scalar-vector variants of vector arithmetic instructions.

SVE and RVV both support a vector-length-agnostic programming model. Ideally, some performance may be portable across different implementations with vector-length-agnostic code. SVE and RVV differ in how they support the vector-length-agnostic programming

model. SVE relies on predication, defining a vector predicate register file and instructions to set the predicates based on loop iteration variables. RVV defines a vector length register, `vl`, which contains the number of elements that will be updated by a given vector instruction. This register is set by an instruction that tests a requested application vector length against the current maximum vector length of the machine, returning the minimum of the two. Some implementations may use these features to reduce execution time when the application vector length is shorter than the maximum vector length by ending execution when all active elements have been processed. SVE also uses predication to implement conditional vector execution. In contrast, RVV uses a mask held in the `v0` vector register to support conditional vector execution. Both ISAs provide instructions to manipulate these predicates or masks.

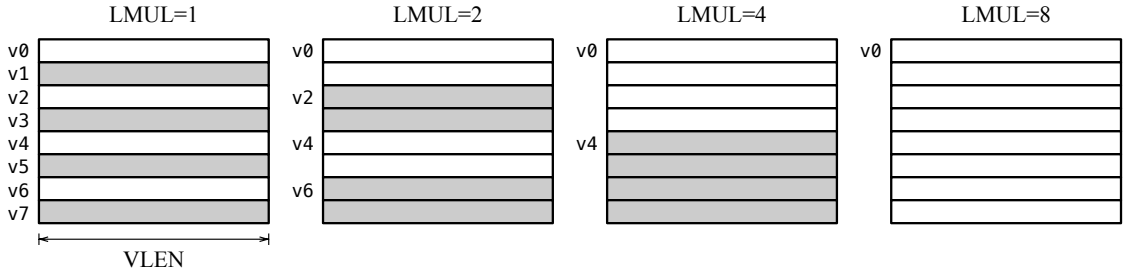


Figure 2.3: The grouping of a subset of the 32 architectural vector registers according to when the `LMUL` setting in RVV is ≥ 1. Increasing `LMUL` groups more vector registers together.

RVV also defines a vector register grouping feature, defining `LMUL`, the vector length multiplier. `LMUL` simplifies how mixed precision vector instructions are handled. Importantly, vector register grouping can also dynamically extend the vector length, allowing machines that efficiently support this feature to be more efficient when operating on long application vectors, as we shall discuss in the following chapter. Figure 2.3 depicts how some of the `LMUL` settings group architectural vector registers together.

SVE and RVV define a rich set of vector memory operations, including unit-stride, constant-stride, and scatter/gather instructions. RVV and SVE also define segment memory operations that unpack and pack "arrays of structs" [15]. They also define a rich set of vector element permutation operations, useful for many DSP kernels.

Figure 2.4 shows an example of a vector-length-agnostic vector-scalar floating-point multiply kernel in RVV, including the use of vector register grouping. The `vsetvli` instruction configures the vector length register, `SEW`, and `LMUL`. Figure 2.5 shows how this kernel would map to a particular RVV implementation, including the various RVV parameters. In the first iteration, all elements in a group of four vector registers are used to process the computation, while on the second iteration, the vector length is trimmed to process the remaining elements.

```
# FP32 vector-scalar multiply loop (C[i] = A[i] * b)
# a0 <- N: number of elements, a1 <- A pointer, a2 <- C pointer
rvv_vsfmul_loop:
  # t0 <- updated vector length, a0: requested vector length
  # Set SEW->32b, LMUL->m4 (group four vector registers)
  vsetvli t0, a0, e32, m4, ta, ma
  vle32.v v4, (a1)       # Unit-stride load from A
  vfmul.vf v8, v4, fa0   # Multiply A by b scalar
  vse32.v v8, (a2)       # Store result
  sh2add a1, t0, a1      # Update pointers
  sh2add a2, t0, a2
  sub a0, a0, t0         # Update remaining elements
  bnez a0, rvv_vsfmul_loop    # Loop if not done yet
```

Figure 2.4: A vector-length-agnostic FP32 vector-scalar multiply loop in RVV assembly.

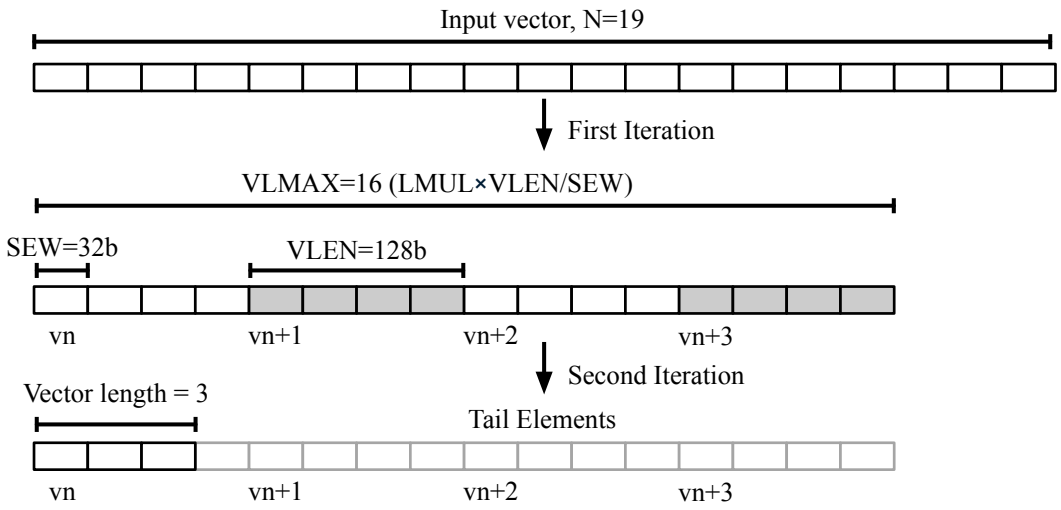


Figure 2.5: RVV processing example as in Figure 2.4, for a vector machine with `VLEN=128b`. The input vector contains 19 single-precision elements. `LMUL` is set to 4, so `VLMAX=16`. On the first iteration, four vector registers are grouped such that 16 elements will be processed. On the second iteration, the vector length is trimmed to process the remaining 3 elements.

Chapter 3

A Short-Vector Machine Platform for DSP Cores

All modern mobile SoCs require flexible yet efficient processing for multimedia applications, a role often filled by DSP cores. Performance, energy efficiency, silicon area, and code size are all critical metrics given that these cores are highly constrained and contribute to the overall quality of the mobile device. Modern scalable vector ISAs, like RVV, have potentially beneficial features when implemented in the DSP core design space, which raises the question of how to design such a machine effectively. This chapter discusses the design of *short-vector machines* in the context of building efficient and compact DSP cores. Importantly, these short-vector machines may serve as a baseline platform for flexible, domain-specialized computing. Short-vector machines strive to be energy efficient and area efficient by maximizing the utilization of their execution and memory resources for a wide variety of application types and problem sizes. Key to their design is harnessing economical hardware mechanisms. In addition, they support an ergonomic and portable vector-length-agnostic programming model.

This chapter aims to describe how short-vector machines may be designed. It begins by defining the key characteristics of short-vector machines before discussing their deployment as DSP cores. Then, the key details and motivation of the Saturn vector unit, or Saturn, are presented. Saturn is a Chisel generator for short-vector machines implementing the RVV 1.0 specification[1]. Saturn will serve as the generator for our baseline short-vector machine design, and is a platform for exploring scalable, specialized vector DSP cores.

Section 3.1 covers the definition of short-vector machines in the context of this thesis. Section 3.2 discusses the deployment of short-vector machines as DSP cores. Section 3.3 discusses the design and implementation of the Saturn vector unit. Section 3.4 presents an evaluation of Saturn. Section 3.5 concludes this chapter with a discussion on the Saturn work, including related work and potential future work.

[1]Saturn was built as a collaboration among a group of students in the Slice Lab at UC Berkeley [132], including Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, and Kevin Anderson.

3.1 Short-Vector Machines

A short-vector machine, in the context of this thesis, is a vector processor that implements a modern scalable vector ISA with a short architectural vector length. We will refer to the number of bits in an architectural vector register as `VLEN`, matching the terminology of the RVV specification. Short-vector machines integrate spatial SIMD functional units for high compute throughput on data-parallel applications. The width of these functional units can be decoupled from the width of the vector units. We will refer to the width, in bits, of the SIMD functional units as `DLEN`. Fully pipelined functional units of width `DLEN` can accept a new `DLEN` bits-wide group of vector elements for execution from the vector register file every cycle, which can be referred to as a "beat" of execution.

Given that the architectural vector length is short and the integration of SIMD functional units, short-vector machines are characterized by having a small ratio of `VLEN` to `DLEN`, the base chime length of the machine. For example, a short-vector machine may have a `VLEN=256b` and a `DLEN=128b`, giving a base chime length of 2. This would mean that a given vector instruction would be executed over 2 beats. Furthermore, recall the vector register grouping feature using `LMUL` in RVV, which allows up to 8 vector registers to be grouped together to act as a single vector register. Implementing dynamic vector register grouping means that chimes can vary in length. With vector register grouping and a base chime length of 2, a vector instruction may execute for up to 16 beats. Thus, short-vector machines operate with a balance of spatial and temporal execution. This is in contrast to typical P-SIMD implementations that execute vector instructions in a purely spatial manner and typical long-vector machines that execute vector instructions in a deeply temporal manner.

Given we are targeting the DSP core design space, short-vector machines must be capable of efficiently processing both short and long application vector lengths, which may in turn leverage these variable chimes. Short-vector machines strive to be energy efficient and area efficient by leveraging economical microarchitectural features to achieve high utilization of their datapath and memory system resources, despite having a short architectural vector length. This is in contrast to long-vector machines which use long architectural vector registers and execute long chimes temporally over many cycles, amortizing the costs of instruction sequencing and achieving high functional unit utilizations.

Figure 3.1 shows a high-level overview of a short-vector machine, including its main constituent blocks. The scalar unit is an in-order core that fetches and decodes all instructions. The vector unit frontend executes in lockstep with the latter pipeline stages of the scalar pipeline, decoding vector instructions and performing checks for memory faults. The vector load-store unit (VLSU) handles the communication of the vector unit with the SoC's memory system. The vector backend comprises the logic that sequences vector instructions, the vector register file (VRF), and the actual SIMD functional units of width `DLEN`.

With short chimes, achieving high utilization of the datapath and memory resources requires high sequencing throughput of vector operations, given that each beat may execute a significant portion of a single vector instruction. Vector operations should be capable of being executed back-to-back, assuming there are no hazards. Furthermore, the allowance of

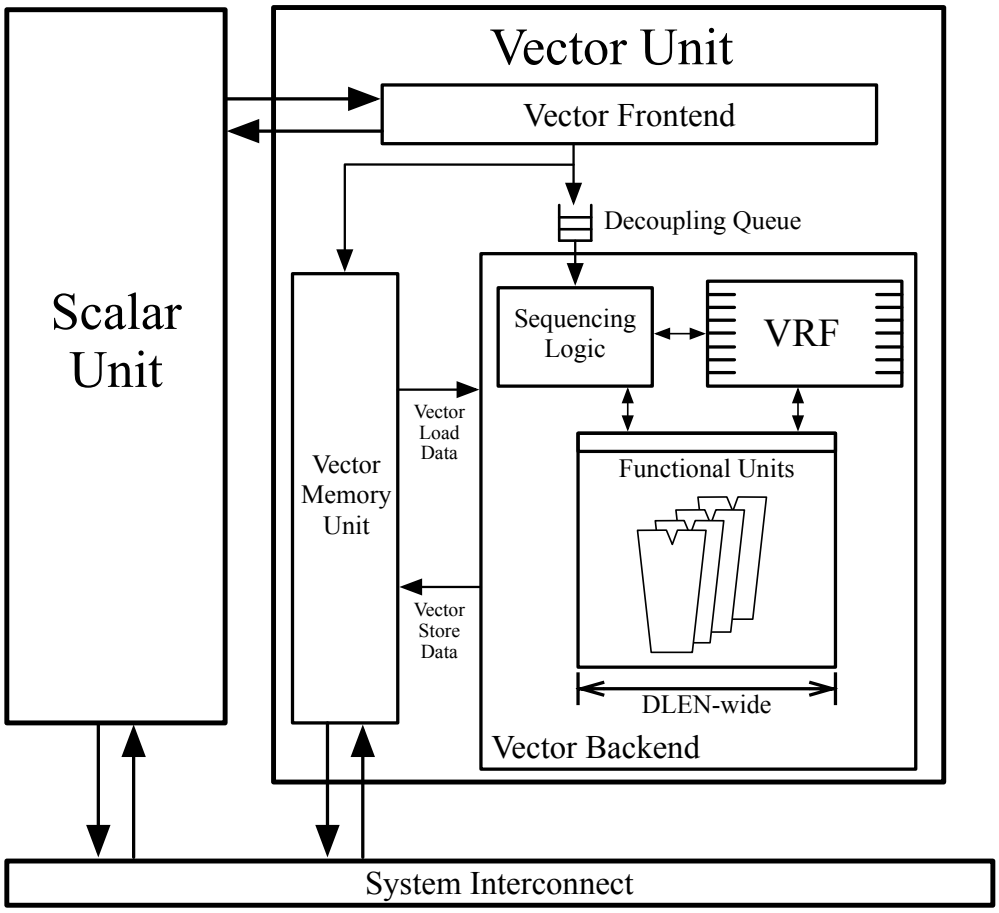


Figure 3.1: High-level view of a short-vector machine.

any dead time [6] is out of the question. A single cycle of dead time on a short-vector machine would severely limit the maximum possible datapath utilization. In order to sequence vector operations back-to-back, the sequencing logic must be able to do the following in a single clock cycle:

- Resolve all data and structural hazards.
- Orchestrate the movement of data from the VRF to the vector functional units.
- Accept a new instruction from its issue queue immediately as it finishes sequencing a vector instruction.

Mechanisms for general-purpose high instruction throughput, such as wide, out-of-order instruction issue and register renaming, are unlikely to be cost-effective in the DSP core design space. Thus, the sequencing logic in the vector backend must be able to resolve hazards

and schedule vector operations efficiently in the presence of multiple independent sequencing agents, without limiting the physical implementation of the design. This includes enabling vector chaining in a granular fashion and possibly executing some vector operations in an out-of-order fashion through the direct resolution of RAW, WAR, and WAW hazards. By integrating multiple sequencing agents in the vector unit, execution across various functional units and the vector memory unit can be overlapped.

The design of the VRF is a key part of every vector unit. The short-vector machine designs we will discuss employ a unified, banked VRF, with each bank being multi-ported such that it can read all operands for a vector operation from a bank in a single cycle. The VRF banking allows the various sequencing agents to simultaneously access different banks, helping to enable concurrent execution when properly provisioned. When the VRF is banked at a width finer than the architectural vector length, `VLEN`, as is often the case, chaining can occur at a sub-vector length granularity. With each bank having sufficient ports, all vector operands for most vector instructions can be read in a single beat, which helps when processing short application vectors.

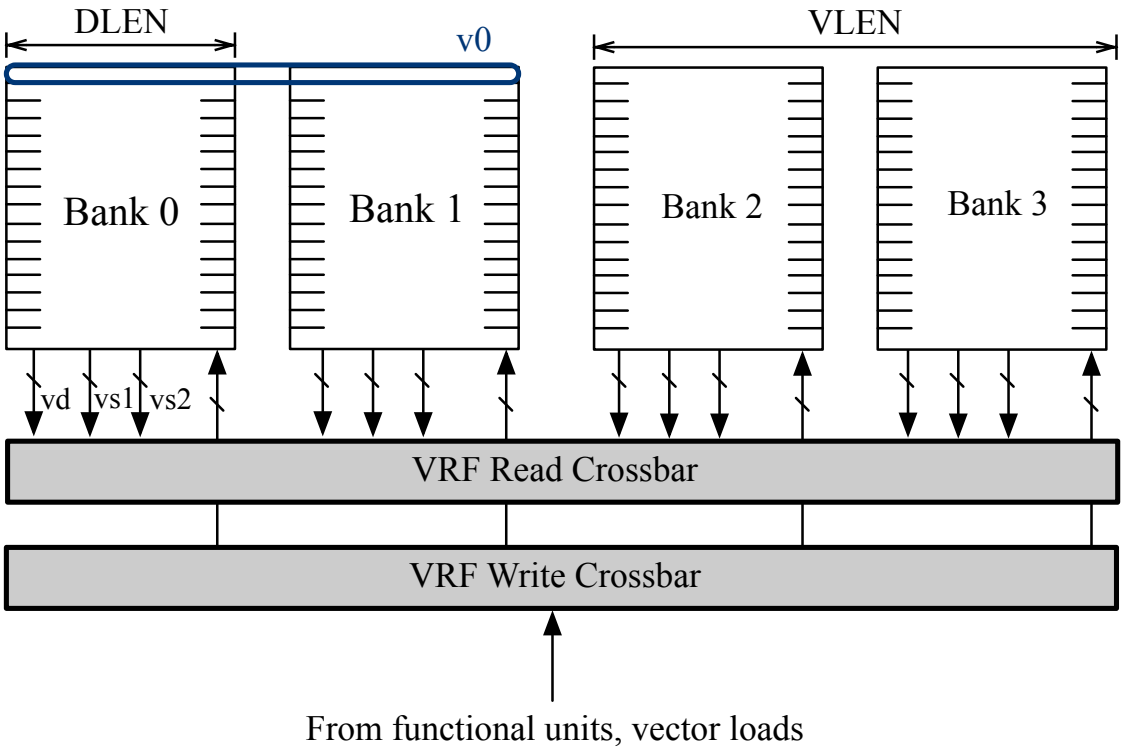


Figure 3.2: Overview of the unified VRF, shown with 4 banks and `VLEN/DLEN=2`.

Figure 3.2 shows a high-level overview of one possible configuration of the VRF in a short-vector machine, with four banks and a `VLEN` to `DLEN` ratio of two. Each bank is of

width `DLEN` and has three read ports such that all arguments for a ternary vector operation can be read at once. Each architectural vector register is striped across two banks, as shown with `v0`. The VRF writes come from the various vector functional units as well as from vector loads fetched by the vector memory unit.

Short-vector machines can effectively leverage the decoupled vector machine design pattern [35][6]. The decoupled access-execute (DAE) architecture paradigm partitions a machine into an "access" processor, responsible for memory accesses, and an "execute" processor, responsible for computation, decoupled from each other by queues for communication [115]. This decoupling can significantly improve performance by enabling the tolerance of memory latencies. The DAE paradigm applied in the short-vector machine context separates the vector memory unit from the vector execution unit, with decoupled interfaces between them. This allows the vector memory unit and the vector execution unit to run ahead or behind one another. Decoupling is an economical technique that helps improve utilization by queuing up work for the various units and helping to tolerate memory latencies. Furthermore, the vector unit is decoupled from the scalar core, which allows the scalar unit to queue up work for the vector unit without blocking issue, unless the queues fill. This allows the scalar unit to run ahead of the vector unit, overlapping scalar bookkeeping and control code with the vector execution and fetching scalar operands that will be needed by the vector unit in the future.

All together, these features in short-vector machines lead to zero dead time, granular vector chaining, and limited out-of-order execution, enabling high resource utilization on short chimes with minimal control hardware costs. We will explore these concepts more concretely in our discussion of the Saturn vector unit in the following sections, including quantitative results. In the physical evaluation presented in Section 3.4, we will show the control costs are minimal in comparison to the costs of the VRF and functional units.

Why short vectors?

Now that we have a high-level picture of what a short-vector machine looks like, we might want to step back and ask why should we focus on "short vectors" at all? As noted in Chapter 2, multimedia workloads in DSP and image processing feature widely varying application vector lengths. Modern vector benchmark suites reflect this [99][64]. Particularly in the mobile and embedded domain, application vector lengths are frequently short, much shorter than those found in the HPC domain in which long-vector machines are typically deployed. For example, BLASFEO is a framework for embedded optimization applications, featuring very small matrices [38]. Given that we want to maximize energy efficiency and area efficiency, the machine must be capable of achieving good resource utilization even on applications with short application vector lengths. Long-vector implementations derive their efficiency from sequencing long chimes that amortize many costs in the system and achieve high datapath utilizations [125]. However, this means they cannot always efficiently target applications with short application vector lengths. In contrast, short-vector machines are built with mechanisms that are specifically geared towards sequencing short vectors.

Beyond just the nature of mobile multimedia workloads, physical silicon constraints bring another key motivation for designing short-vector machines. In mobile SoCs, DSP cores are highly constrained in terms of area and power. Using shorter architectural vector lengths means that the total storage in the VRF is smaller. Reducing the total area of the machine may allow it to be deployed in more cases. Furthermore, this reduction in area also brings a reduction in leakage power, an issue that gets worse as silicon nodes continue to scale [56][54]. In other words, being able to achieve high utilization of SIMD datapaths with short architectural vector lengths without a high instructions-per-cycle (IPC) scalar core bodes well for overall processor energy efficiency and area efficiency.

While short-vector machines can efficiently target short application vectors, this does not mean that they are inefficient when processing long application vectors. The vector register grouping feature using the `LMUL` parameter in RVV can be used to dynamically increase the vector length by grouping a number of vector registers together. Short-vector machines should strive to support this feature efficiently. By doing so, instruction fetch costs can be amortized by encoding more operations per instruction when the application vectors are sufficiently long. More beats of execution are queued up at once, which can help the vector backend achieve improved resource utilization. The caveat is this is only beneficial when the application vector lengths are suitably long and vector register pressure is not too high. Thus, short-vector machines are capable of being efficient on both short and long application vector lengths. We will see how this feature can be efficiently implemented in the following sections.

3.2 Short-Vector Machines as DSP Cores

We will now discuss how short-vector machines can match the characteristics of DSP cores and how they may have positive characteristics when compared to other paradigms. Data-parallel processing with SIMD functional units is a near-universal hallmark of DSP cores going back over a quarter century, particularly as P-SIMD extensions to general-purpose ISAs proliferated in the 1990s [90][86][36]. Short-vector machines can effectively harness the data parallelism inherent in most multimedia workloads, targeting a wide variety of applications and problem sizes efficiently. Short-vector machines implement the rich features of a modern scalable vector ISA, making them highly flexible. In fact, scalable vector machines have long been used to target DSP workloads [117][11], though they were traditionally deployed in the vector supercomputer space.

In Chapter 2, we discussed the predominant paradigms in DSP core design, namely VLIW and narrow, in-order RISC pipelines with SIMD datapaths. VLIW-based DSP cores derive their efficiency by minimizing the area and energy costs of instruction fetch and sequencing in hardware. This shifts the scheduling complexity almost entirely to software, with the compiler or programmer being responsible for scheduling the code. This allows VLIW DSP cores to scale to higher instruction throughputs and drive additional functional units. Narrow RISC pipelines with tightly integrated SIMD datapaths tend to derive their efficiency by

simply using a minimal scalar core. However, this can limit datapath utilization due to insufficient instruction throughput. Both of these approaches rely on extensive static scheduling, typically including loop unrolling and software pipelining. This bloats code size, which is a key metric for DSP cores in constrained deployments. Beyond code size concerns, this makes them difficult to program and undermines code portability since microarchitecture-specific scheduling is essential for achieving good performance.

In contrast, short-vector machines leverage a mix of spatial and temporal execution, dynamic vector operation scheduling, and decoupling. The vector unit hardware effectively unrolls the beats of execution in a vector stripmine loop when the chime length is greater than one. This reduces the reliance of short-vector machines on static scheduling. Each instruction can encode more beats of work and the pipeline can continue to issue and execute other instructions concurrently. Because of these hardware mechanisms, stripmine loops that leverage vector register grouping may not need to unroll as much or be software pipelined, while still achieving high vector resource utilizations. This reduced reliance on static scheduling means that code size can be reduced while remaining efficient.

Short-vector machines support the vector-length-agnostic (VLA) programming model of modern scalable vector ISAs, so kernels can be written to omit tail code in the vectorized dimension, which can also reduce code size. P-SIMD ISAs that do not support VLA programming must include tail code when the application vector length is not an integer multiple of the hardware vector length. By encoding more work per instruction, the scalar unit IPC required to keep the vector functional units utilized is reduced and the cost of instruction fetch is further amortized. This means that the scalar unit needs not implement costly features that would otherwise be needed to improve IPC, without compromising on performance. Again, the caveat to these points is that when the application vectors are so short, these advantages are reduced. Since the execution width is quantized to `DLEN` bits, vector lengths that are not an integer multiple of the datapath width will leave some functional units unused on the final beat of a vector operation. This has a greater impact when the vector length is short. When the vector length drops to `DLEN/SEW` elements, each instruction will only execute for a single beat.

The vector FMA loop in Figure 3.3 is written in RVV assembly. This loop uses vector register grouping by setting `LMUL=8`. In a concise loop, this kernel leverages the mechanisms in the short-vector machine backend to efficiently sequence the vector operations, with the operations effectively unrolled by the hardware. This particular kernel is memory-bound given that it must load three vectors and store one vector for every vector FMA instruction. This loop executed in a short-vector machine should be able to attain close to 100% utilization of the vector load pipe. If a similarly structured kernel were to have a higher arithmetic intensity, the utilization of the vector FMA units would correspondingly increase. As vector register pressure increases in a particular kernel, the `LMUL` may need to be decreased, but the vector register grouping will typically remain useful.

The equivalent loop in a P-SIMD ISA would require more extensive static scheduling, possibly including loop unrolling and software pipelining, to achieve the same utilization, resulting in a larger static code size. Furthermore, the P-SIMD case would also require tail

code to handle vectors that are not an integer multiple of the hardware vector length, while the RVV version handles it automatically given that it is vector-length-agnostic. Porting such a kernel to a different DSP core with a slightly different pipeline may require rescheduling the code, while the RVV version using vector register grouping might be largely portable between different short-vector machine instances. While this is a simple example, many DSP functions take a very similar form. More complex functions can also typically leverage this feature to achieve high resource utilizations with concise vector kernels.

```
// FP32 VFMA loop (D[i] = C[i] + A[i] * B[i])
// a0 <- N: number of elements
// a1 <- A, a2 <- B, a3 <- C, a4 <- D
vfma_loop:
  vsetvli t0, a0, e32, m8, ta, ma
  vle32.v v0, (a1)
  sh2add a1, t0, a1
  vle32.v v8, (a2)
  sh2add a2, t0, a2
  vle32.v v16, (a3)
  sh2add a3, t0, a3
  vfmacc.vv v16, v0, v8
  sub a0, a0, t0
  vse32.v v16, (a4)
  sh2add a4, t0, a4
  bnez a0, vfma_loop
```

Figure 3.3: Vector-length-agnostic FP32 FMA kernel leveraging vector register grouping.

These characteristics of short-vector machines imply further advantages beyond just PPA, efficiency, and code size for a particular DSP core instance. Recall that a given mobile SoC may deploy multiple DSP cores across different PPA design points. Ideally, we would want a single type of machine that can be scaled up and down, ideally with kernel portability between them. Short-vector machines, as we will see, can be scaled to meet different design points while retaining the same structure and programming model. When designing a short-vector machine instance, the `VLEN` and `DLEN` parameters can be scaled together to meet performance and area constraints. The nature of short-vector machines incentivizes performance programmers to write concise stripmine loops that maximize vector register grouping, without inducing excessive vector register spills, to achieve good performance and efficiency. Even though the target DSP cores may be of vastly different sizes, they sequence variable chime operations in a similar way. This, together with the VLA programming model, means that the same kernel will likely have significant performance portability between different short-vector machine design points. Furthermore, decoupling and dynamic vector operation

scheduling can help the performance of sub-optimally scheduled vector code. By relying less on microarchitecture-specific static scheduling and scalar instruction throughput, short-vector machine platforms can have an advantage in programmer ergonomics and portability over other DSP core paradigms.

DSP cores have also traditionally integrated a number of specialized features to increase performance and improve energy efficiency for DSP workloads. Short-vector machines implementing a modern scalable vector ISA cover the functionality of many of these features already. The role that mechanisms to reduce loop control overheads like zero-overhead loops can be filled by a decoupled implementation that allows the scalar unit to run ahead and overlap scalar control code with vector execution. Widening vector arithmetic instructions help fill the niche that accumulator guard bits in traditional DSPs occupy. Short-vector machines can leverage specialized memories for more efficient accesses, just as other DSPs do, in addition to connecting to a more conventional cached memory hierarchy.

We have now motivated the usage of short-vector machines in the DSP core context. While a quantitative comparison against other DSP core design paradigms is outside the scope of this thesis, we have provided a qualitative comparison here, including some of the advantages that short-vector machines have. We will now move on to the design and implementation of a short-vector machine generator.

3.3 The Saturn Vector Unit

The Saturn vector unit [132][131], or Saturn, is a parameterized Chisel [10] generator for short-vector machines implementing the RVV 1.0 specification[2]. Saturn-generated designs are fully synthesizable. The `VLEN`, `DLEN`, and many other design parameters are configurable such that it can generate a wide variety of design points.

The goals of the Saturn project are severalfold. Saturn was designed to be a high-quality, open-source implementation of the RVV 1.0 specification, demonstrating how to implement the complete ISA extension, including its more complex features, in a compact manner. In particular, Saturn targets the DSP core design space as a short-vector machine generator. Saturn was also intended to enable research into programmable, domain-specialized systems, including serving as a platform for exploring the design of custom vector instructions and accelerator integration. The implementation was explicitly designed for ASIC design flows,

[2]A more complete discussion on the history of the Saturn project and contributions can be found in The Saturn Microarchitecture Manual technical report [132]. Jerry Zhao was the project lead, and implemented the instruction scheduling mechanism, load-store unit, and scalar core integration. I implemented the SIMD floating-point units, helped bring up the scalar core interface, and helped in debugging the vector load-store unit. I also contributed to the design and implementation of the core subsystems of the vector backend, including the instruction sequencing mechanism, the vector register file, and various other functional units. I led the Saturn physical implementation and evaluation for PPA feedback and optimization, including writing optimized vector kernels. Miles Rusch implemented several variants of SIMD integer multipliers. Tianrui Wei assisted with earlier versions of Saturn and performed key work that would evolve into the 1.0 version. Many others contributed to the project, in particular, Albert Ou.

rather than supporting efficient synthesis and mapping for FPGAs. This choice allows for a meaningful quantitative PPA evaluation of short-vector microarchitectures and techniques for domain specialization using real silicon PDKs.

The Saturn vector unit project originally began in the Fall of 2021, as it was identified that we needed a new platform for exploring programmable and efficient domain-specialized systems. Saturn went through a number of design iterations until it reached its current trajectory, which began to take shape in 2023.

The following sections will present the design details of Saturn to demystify the design of short-vector machines and a quantitative analysis of its performance, power, and area.

Saturn System Overview

Figure 3.4 shows an overview of the Saturn vector unit integrated with an in-order scalar core. The three main subsystems are the scalar unit and Saturn frontend, the vector load-store unit (VLSU), and the vector backend.

The Saturn generator is built around the `VLEN` and `DLEN` parameters, as described in the previous sections. The SIMD datapaths, the memory interfaces, and the VRF banks are all of width `DLEN` bits, with typical values being 128 or 256 for a DSP core design point. The total length of each vector register is `VLEN` bits, which is broken up into `DLEN` bits-wide chunks. Each chunk comprises a number of elements, varying based on the current `SEW` setting, which will execute together as an *element group*. The number of element groups will determine other physical parameters of the system such as the width of the hazard tracking masks. In later versions of Saturn, another parameter to decouple the width of the memory interface from the execution width was added, but that is beyond the scope of this thesis.

All instructions are fetched and decoded by the in-order scalar unit. The Saturn frontend serves as the interface between the scalar unit and the rest of the Saturn backend and VLSU. Vector instructions are passed on to the backend, and also the VLSU in the case of vector memory instructions, where they are buffered by a series of queues.

The sequencer blocks are responsible for orchestrating the execution of the vector operations and the movement of data between the VRF, the functional units, and the VLSU. The thicker lines in Figure 3.4 denote paths along which groups of `DLEN` bits-wide vector data move. The thinner lines stemming from the sequencer blocks denote paths of control signals that communicate the readiness of various blocks and arbitration for access to resources like VRF read and write ports.

The VLSU is responsible for actually executing vector memory operations, while the load and store sequencers in the Saturn backend coordinate the movement of data to and from the VLSU. The VLSU is decoupled from the Saturn backend and includes circuitry for address generation and for organizing load response data and vector data to be stored. It also provides interfaces to ensure proper memory ordering among vector and scalar memory accesses.

The following sections will explore these subsystems in more detail and discuss how Saturn's design features embody a short-vector machine.

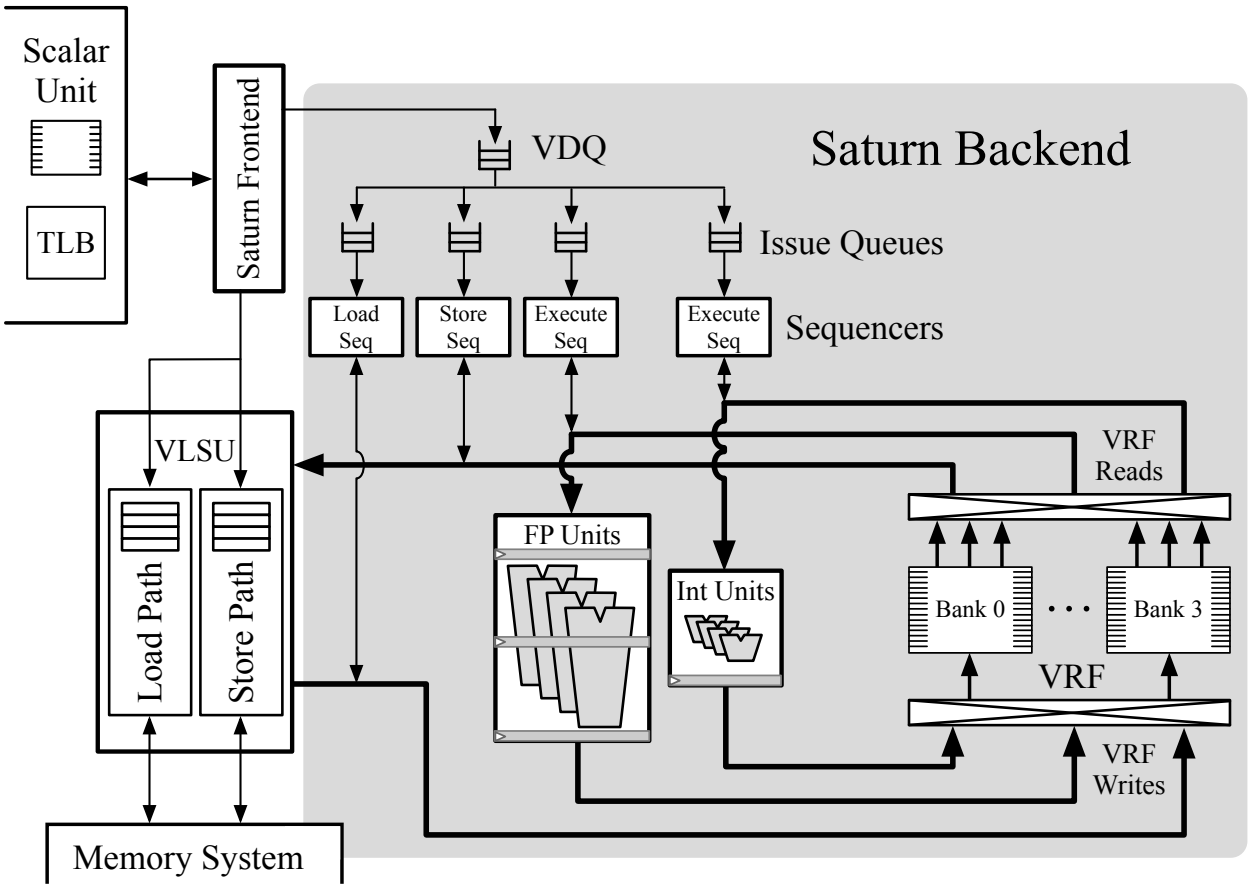


Figure 3.4: Block diagram overview of the Saturn vector unit. VDQ = Vector Dispatch Queue, VRF = Vector Register File, VLSU = Vector Load-Store Unit.

Scalar Unit and Saturn Frontend

While vector instruction execution occurs in the Saturn backend and VLSU, the scalar unit maintains some critical functionality for the Saturn vector unit, aside from simply fetching and decoding vector instructions before passing them off to the vector unit. As we are focusing on the short-vector machine design point for DSP cores, we restrict our discussion to in-order scalar units. The scalar unit maintains the specified control and status registers (CSRs) for RVV [104]. These CSRs include the vector length register, `vl`, and the `vtype` register that maintains the selected element width, `SEW`, and the vector register grouping factor, `LMUL`.

The Saturn frontend interfaces with the scalar unit. The scalar unit must implement a Saturn generator-compatible interface. The current Saturn generator can interface with a Rocket core instance [7] or a, potentially multi-issue, Shuttle core instance[3]. While the Saturn

[3]https://github.com/ucb-bar/shuttle

backend strives to minimize its dependence on the performance of the scalar core, there are some implementation details with important implications for the overall performance of the machine. If application vectors are particularly short, a higher issue rate into the vector backend is required to maintain good utilization of the datapaths and other resources. Thus, a modest-width, multi-issue scalar core capable of maintaining a reasonable IPC can increase the maximum performance of the system. Up to one vector instruction can be sent to the vector backend per cycle. Since these instructions will typically be encoding multiple beats of work, this is sufficient to saturate the vector memory and arithmetic resources, as the application allows.

As they proceed through the scalar pipeline, each vector instruction carries with it a copy of the proper vector configuration information, including the `vl` and `LMUL`. Not only is this information needed in the vector backend, but also for address checks to establish the reach of a vector memory instruction. Changes to these CSRs using the RVV configuration-setting instructions (e.g. `vsetvli`) instructions should be forwarded to younger vector instructions so as not to induce excessive bubbles in the scalar pipeline. Shuttle-based implementations do not co-issue a vector instruction with an older configuration-setting instruction, but do implement this bypassing.

In Saturn, all vector instructions execute post-commit. To support virtual memory and precise exceptions, the vector memory instructions must be verified to be free-of-fault ahead of the commit point. In a DSP core design point, virtual memory may not be supported, but the decision to support it in Saturn was made to demonstrate how it can be done and to simplify software development. The Saturn frontend unit interfaces with the scalar core starting at the execute stage. The frontend checks for vector memory faults by accessing the scalar unit's TLB. For unit-stride vector memory instructions, the extent of the access can be determined by using the instruction's base address and corresponding `vtype` and `vl`. Valid instructions accessing a single page can be checked in a single beat and sent to the VLSU and backend. Valid instructions that cross a page will be cracked into two accesses such that all accesses in the VLSU will always be to a single page. These cracked accesses will be merged into the proper vector register group later on for execution. Vector memory accesses that may access potentially many pages can be checked iteratively such that each element is checked in turn and sent to the VLSU. The Saturn VLSU operates solely on physical addresses. A vector instruction that does not commit at the write-back stage due to these processes will simply replay, killing younger instructions and executing again with the state of progress maintained by Saturn.

A committing vector instruction is sent to the vector backend, as well as the VLSU in the case of vector memory instructions. This instruction bundle includes the instruction itself, vector metadata like the `vtype` and `vl`, and any immediate or scalar register read operands from the scalar integer or floating-point register file.

By designing a decoupled vector unit as opposed to one that is tightly coupled, the scalar unit issue will not be blocked by committed vector instructions except in the case of the vector decoupling queues completely filling, memory access conflicts, or vector instructions that move data from the vector unit to the scalar unit. This allows the scalar unit to run ahead

to overlap control and bookkeeping code with vector execution, load scalar operands for future vector instructions, and fetch more vector instructions to be queued up for execution.

Vector Load-Store Unit

The Saturn VLSU serves as the "access" processor following the decoupled short-vector machine design pattern. This decoupling enables the Saturn VLSU to run ahead of the vector backend, fetching data for future vector arithmetic instructions, and run behind it, storing data as the backend continues executing older instructions. This is a cost-effective technique for improving the utilization of Saturn's hardware resources.

Committed vector memory instructions, and their corresponding metadata, are sent to the VLSU and the vector backend. The VLSU is responsible for load and store address generation, aligning and re-ordering memory requests and responses, and arranging the vector data for and from the vector backend. Saturn generates a VLSU with separate paths for vector loads and stores. A typical Saturn system has the vector memory port connected directly to the outer memory system, typically backed by a lower-level cache, bypassing the scalar unit's L1 caches.

Queues in the VLSU track in-flight vector memory operations. These queue entries contain the base of the vector memory accesses and their bounds, along with other salient information, such as the stride and `vl`. To maintain scalar-vector and vector-vector memory ordering, other memory accesses can look up potentially conflicting accesses in these queues using this information. Scalar memory accesses that conflict with an older in-flight vector memory access can simply be replayed at the commit point until the vector access has completed. Ordering across the vector load and stores pipes is maintained so that conflicting vector memory accesses can be stalled as needed.

The vector address sequencers generate memory requests based on the type of vector memory operation and their alignment. The VLSU does not assume that load requests will respond in-order and implements a simple reorder buffer to handle this. Outstanding store requests are tracked until they are acknowledged. Rotation buffer circuits shift and align vector load and store data to arrange it appropriately for the vector backend and the `DLEN`-wide memory interface, respectively. These circuits also arrange the non-contiguous data from strided and indexed accesses. The load and store paths also each contain a segment buffer circuit to properly arrange vector data for vector segment loads and stores. The VLSU will clear an entry from its queues tracking in-flight vector memory operations once its final memory transaction completes.

Saturn vector unit instances can also integrate a tightly coupled memory (TCM). A TCM is a software-managed scratchpad memory. TCMs are commonly integrated with DSP cores to provide efficient, high-throughput memory accesses. The TCM is not cached, so accesses have more predictable latencies at the cost of having to manage them explicitly. TCMs are commonly used to tile kernels for the vector unit outside of the outer memory system and can be optimized for irregular memory accesses [17]. The usage of TCMs in the context of a multi-vector core SoC is explored further in Chapter 5.

Vector Backend

Committed vector instructions are sent to the Saturn backend from the Saturn frontend. They are first buffered by the VDQ, a queue that decouples the vector backend from the scalar unit and the frontend, as shown in Figure 3.4. From the VDQ, vector operations are sent to one of a number of issue queues. These issue queues are each private to one or more sequencers, the blocks that handle the actual vector operation scheduling. A sequencer will dequeue a vector operation from its issue queue when it is ready and sequence all the beats of execution. The destination issue queue is determined by the type of vector instruction, since issue queues can be parameterized to accept only specified vector instructions.

The VDQ and issue queues together buffer instructions for the vector backend. Since all vector execution is done post-commit, the scalar unit can run ahead and overlap execution of scalar instructions with the processing in the vector backend. This buffering helps the sequencers remain busy and hides the overheads associated with scalar control code. When an issue queue fills, it may block instructions leaving the VDQ. Empirically, we find that generally only a relatively small number of queue slots, around 3 to 5, are required to avoid stalling issue from the scalar unit to the vector backend in most cases.

Each entry in the VDQ and issue queues encompasses an entire vector instruction and its metadata, including `vl` and `LMUL`. Even when the number of beats of execution for the instruction is greater than one, as is typically the case, each queue slot encodes an entire instruction. Cracking the instruction into its individual beats of execution in these queues would require many more slots. For example, for machines with a base chime length of two, an instruction with an `LMUL` of eight would require sixteen queue slots on its own, one for each beat of execution. Keeping the instruction "compressed" and using the instruction metadata for hazard checking can save considerable hardware resources.

Each vector instruction is assigned a unique tag that can be used to track its relative ordering. This tag is carried with the instruction for the remainder of its execution. The tag is used by the sequencing logic to disambiguate the ordering of in-flight vector instructions for the purposes of hazard detection. An instruction's tag is freed when it is completely sequenced. Once freed, it can be assigned to another vector instruction entering the Saturn backend. Thus, the bitwidth of the tag is sufficiently large to encompass the maximum possible number of in-flight instructions in the vector backend.

Vector instructions are kept in-order through the VDQ, and sequencer dequeue and process instructions from its issue queue in-order. However, the sequencers schedule vector operations independently such that they may execute out-of-order with respect to vector instructions held in other sequencers and issue queues. This dynamic scheduling of vector instructions can help the performance of sub-optimally scheduled code and allows operations within different units to proceed when they are safe to do so, regardless of issue order.

Each vector functional unit is part of a grouping of functional units called an execution unit. Each execution unit is driven by a single sequencer. Functional units within the same execution unit can operate concurrently, with the sequencer issuing up to one beat of work per cycle to a single functional unit. Instantiating multiple sequencers allows for more

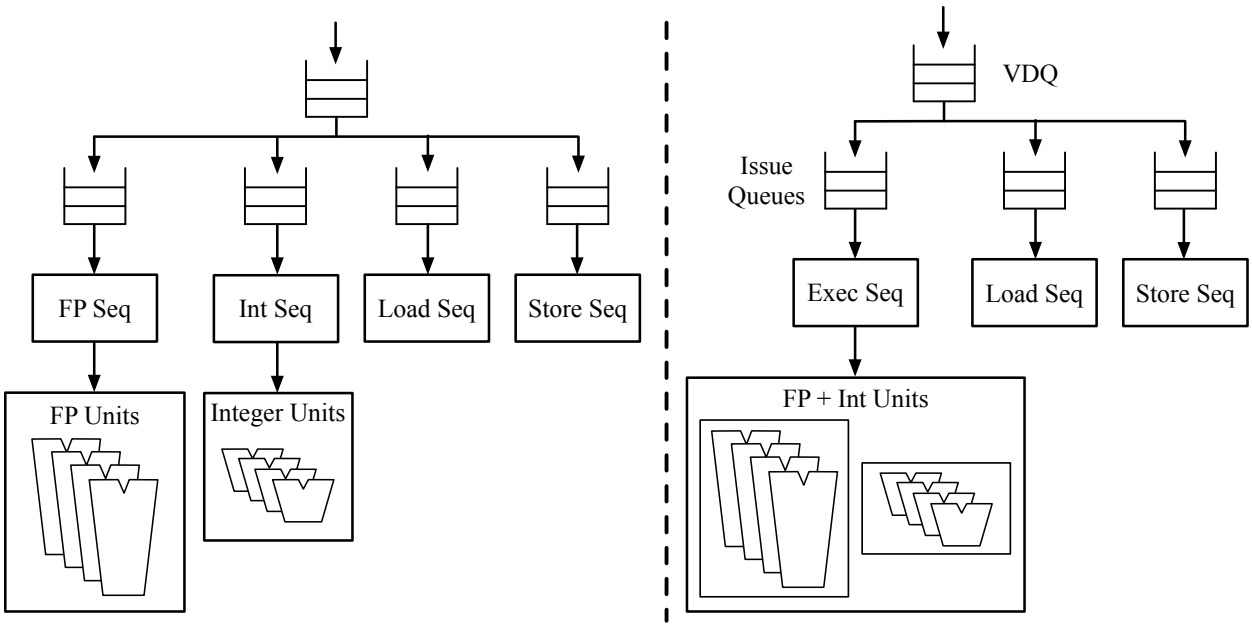


Figure 3.5: Saturn vector backend configuration block diagrams [132].

possible concurrent execution, since each sequencer can operate independently. The trade-off is additional area cost and hazard checking complexity. Saturn can generate different combinations of issue queues, sequencers, and functional units, as pictured in Figure 3.5. The lower area configuration, on the right, unifies the floating-point and integer vector functional units under a single issue queue and sequencer. The higher area configuration, on the left, splits the floating-point and integer functional units into separate issue queue and sequencing paths, allowing for concurrent operation. Having a modest number of sequencers is typically sufficient for good performance since DSP kernels are often dominated by FMA or MAC-type operations interleaved with loads, stores, and possibly permutation operations.

The sequencers are responsible for driving the various functional units within a particular execution unit. Sequencers operate on one vector instruction at time and check that an operation is safe to issue by ensuring there are no data hazards or structural hazards. Figure 3.5 shows that there is a load sequencer, store sequencer, and one or more arithmetic sequencers. The load and store sequencers are the simplest, as they primarily track just a single vector operand destination or source, respectively. When a vector memory instruction is sent to the VLSU, a corresponding operation is sent to the Saturn backend. The load and store sequencers then coordinate the movement of data to and from the VRF when the VLSU is ready. The execute sequencers are the most complex, as they must track the most vector operands simultaneously. For example, a vector-vector FMA instruction must track three source vector registers to be read and one destination vector register to be written, matching one of the source vector registers. A permute sequencer is responsible for helping to manage permutation instructions and operations that require arranging data in an irregular

manner in the vector backend. For vector slide, register gather, and compress instructions, the sequencer fetches the data and buffers it to be sequenced by an execute sequencer.

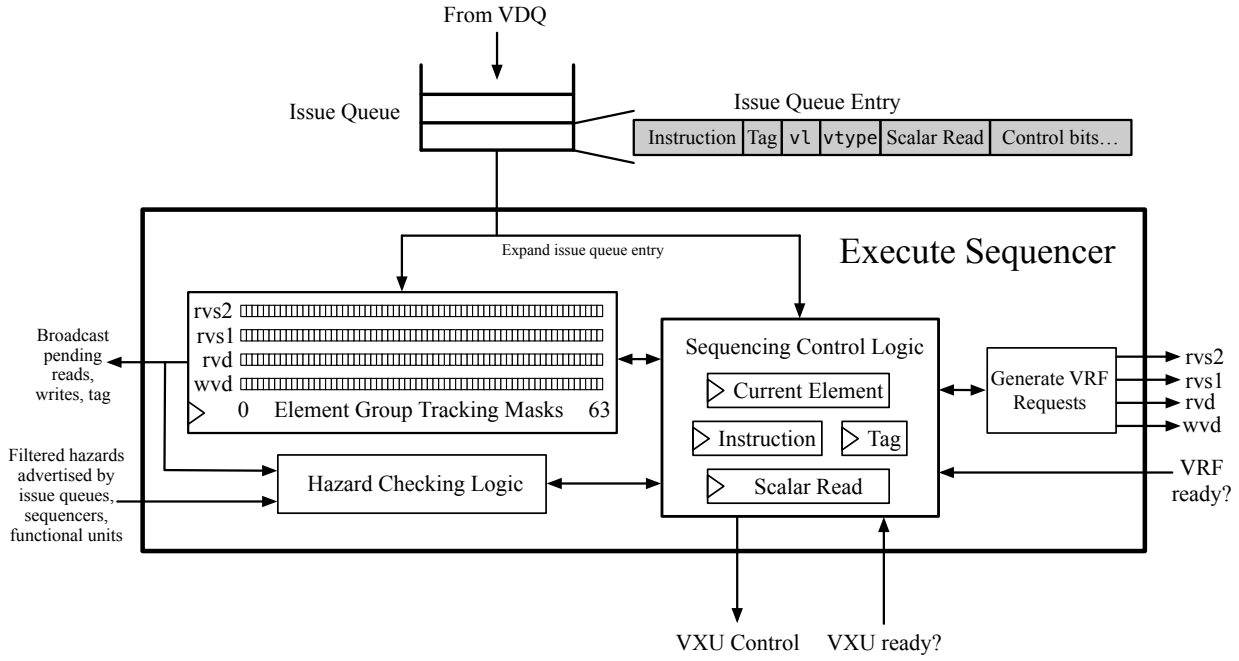


Figure 3.6: Execute sequencer block diagram for a Saturn instance with `VLEN/DLEN=2`.

Figure 3.6 shows the block diagram of an execute sequencer in Saturn, along with an expanded issue queue entry. The issue queue entry contains the instruction and its metadata, including scalar register file reads and generated control bits. This entry is expanded by the sequencer into various salient control signals. The element group tracking masks are 64 bits wide, corresponding to each of the element groups in the VRF when the `VLEN` to `DLEN` ratio is 2. The proper bits will be set true to indicate pending reads and writes to element groups according to the instruction type, `vtype`, `vl`, and vector register specifiers. The sequencing control logic considers data hazards advertised by other portions of the vector backend as well as structural hazards, such as those on VRF read and write ports. The current element register and element group tracking masks are updated as the beats of execution are sequenced. The sequencer orchestrates the control and data movement for the execution unit, or VXU, which can also backpressure the sequencer in the case of non-fully pipelined functional units.

Since the sequencers consider the value of an instruction's vector length register, `vl`, they do not need to process all possible beats of execution when the vector length is trimmed sufficiently such that `DLEN`-wide results would be unused. This allows for more optimized execution of vector-length-agnostic kernels. When a sequencer issues the final beat of a vector instruction, it can immediately accept a new instruction from its issue queue and

begin sequencing it on the very next cycle. Assuming there are no hazards, instructions can be sequenced back-to-back with zero dead time. This implies that all hazards must be resolved and the operands read from the VRF and propagated to the proper functional unit all in one cycle. Executing with zero dead time is critical for short-vector machines. Any dead time would severely limit performance and efficiency, particularly when operating on short application vector lengths.

Recall that the base unit of granularity in the machine is the `DLEN`-wide element group, which represents a slice of a given architectural vector register. Using the currently sequencing vector instruction's operands and its `vl` and `LMUL`, a sequencer expands the information that was held in its issue queue entry into bit masks of width of the number of element groups. The number of element groups can be found from $32 \times (VLEN/DLEN)$, since there are 32 architectural registers, each subdivided into `DLEN`-wide chunks. The bits corresponding to the element groups that will be read from or written to are set high and the rest are set low.

To ensure there are no data hazards, all portions of the backend that maintain information about an in-flight vector operation must be checked. The issue queues and other sequencers advertise the element group read and write bit masks for each currently valid vector instruction slot. The functional units also expose a bit mask for the outstanding writes of in-flight instructions. These must be checked, since different functional units may have different pipeline depths, which could otherwise allow a conflicting younger instruction to write back before an older instruction that was issued earlier. The sequencer's own issue queue need not be checked since these instructions are always dequeued in-order. The tags associated with each vector instruction is used to disambiguate the relative ages of the operations, enabling or disabling the inclusion of a particular bitmask in a hazard check. Vector operands are read at the time of issue from the sequencer, so the functional units only need to advertise their outstanding writes.

Vector data hazards are more specifically resolved by each sequencer as follows:

- **RAW:** Older writes in other issue queues and sequencers and outstanding writes in the functional units must be checked for conflicts against read operands. Note that the age of the in-flight operations in the functional units need not be checked because those operations would not have been issued if there was a data hazard against an older operation.

- **WAR:** Older reads in other issue queues and sequencers must be checked for conflicts against current write operands. The operations in-flight in the functional units need not be checked since operands are read at the time of issue to an execution unit.

- **WAW:** Older writes in other issue queues and sequencers and outstanding writes in functional units must be checked for conflicts against current writes. An optimization to these checks could be made if all pipelined functional units driven by a sequencer had the same pipeline depth. Since all pipelined operations have a fixed latency, WAW

hazards would not be possible against outstanding pipelined writes driven by the same sequencer and so would not need to be checked.

Structural hazards must also be considered. The VRF arbitrates read port conflicts, which can backpressure a sequencer. Gaining access to a VRF write port is handled in two different ways, one for pipelined functional units and one for all other cases. Once issued, pipelined operations execute and write back their results with a fixed latency. This is by design, so as to avoid the physical overhead of distributing backpressure logic throughout all of the functional unit pipelines. When a pipelined operation is issued, the destination VRF bank write port is reserved, such that it will be available after a number of cycles corresponding to its pipeline depth. The VRF arbitrates access to a bank's write port for other VRF writers, always giving priority to the pipelined units. While this gives other units a lower effective priority, forward progress is guaranteed by the sequencing mechanism, and when there is an operation waiting for that result, it will quickly get access.

When all hazards are resolved, operands are read from the VRF and issued to the correct functional unit. The corresponding bit in the sequencer's element group mask is set low to indicate that it no longer needs to be tracked within the sequencer. The sequencers will issue an operation as soon as all its operands are ready and all other hazards are resolved. Since hazards are tracked at the element group granularity, vector chaining can occur at a granularity finer than the width of an architectural vector register. Thus, instructions in different sequencers can overlap execution, which can improve utilization. Note that there are no direct bypasses for vector data. All chaining between vector operations occurs through the VRF.

The data hazard tracking scheme in Saturn as described here does not avoid implementing a relatively non-scalable, distributed scoreboard. The hardware complexity will grow quadratically as more independent sequencers are added to the system. While this is a valid concern, there are mitigating factors that allow this scheme to match short-vector machines like Saturn well. The number of sequencers in the system is fairly small, so while the scheme will not scale well to many independent agents, it does not have to. As noted earlier, the issue queues need not be very deep and the functional unit pipeline depths are fairly shallow in a short-vector machine. This limits the complexity contribution to the hazard checking from those portions of the machine. Furthermore, the width of the hazard checking bit masks is determined by the number of element groups, not the architectural vector length or datapath width on their own. Since we are typically implementing short-vector machines with short base chimes, the hazard tracking mask bit width is limited. We find that the hardware complexity of the hazard tracking scheme does not limit the maximum clock frequency of the machine when pushed through the physical design flow.

Saturn features a unified VRF. The VRF is constructed from flip-flops, fully synthesized and placed-and-routed (PnR). Nothing precludes the design from utilizing a more custom memory to reduce the area and energy costs of the VRF, aside from additional engineering effort. In Saturn, the logical and physical byte mapping for a given architectural register match each other.

The VRF is banked, currently configurable from 1 to 4 banks. Increasing the number of banks increases the number of possible simultaneous accesses from the sequencers and functional units. This comes at an additional hardware cost since each bank must connect to all accessing agents. We find that a modest number of banks is sufficient for good performance without incurring too much overhead. The width of a single bank in bits is `DLEN` (i.e. the width of an element group). Thus, there are $32 \times (VLEN/DLEN)$ entries in the VRF, given that there are 32 architectural vector registers. Each bank has three independent read ports and one write port. This is sufficient for a sequencer for an FMA or MAC-type instruction to read all its operands from a bank in a single beat. The banking of the VRF and allowing pipelined operations to read all vector operand data in a single beat means that overheads from structures like operand queues can be avoided without sacrificing performance.

The vector functional units are where the actual arithmetic operations are executed. These units are capable of operating on all RVV 1.0-defined element widths (8b, 16b, 32b, 64b). In Saturn, there are two major types of functional units:

- **Pipelined functional units (called PipelinedFunctionalUnit)** operate at a fixed latency given its parameterized pipeline depth. Pipelined functional units are `DLEN`-wide, allowing a full element group of work to be executed every beat, with some exceptions. Pipelined functional units provide no backpressure to the sequencers. This is possible since structural hazards on the target VRF write port are checked ahead of time. When the operation has propagated through the functional unit pipeline, the result is written back to the VRF.
- **Iterative functional units (called IterativeFunctionalUnit)** operate on one element of a vector register at a time. Iterative functional units typically have variable execution latencies or consume too much area given their typical usage to implement in a spatial manner. Once the result is complete, the output element is written back to the VRF and the unit can immediately accept the next input.

Table 3.1 summarizes most of the Saturn functional units and their structure.

3.4 Saturn Evaluation

Here, we present some performance, power, and area (PPA) evaluation results for Saturn to demonstrate its potential and validate some of its key ideas.

To measure performance, we run a series of cycle-accurate RTL simulations for several Saturn designs generated from the Chisel implementation. The Chipyard framework [1] is used to generate the SoC DUT, which includes an instance of the Saturn vector unit and a last-level cache (LLC). The simulated vector units are integrated with an instance of a dual-issue Shuttle core in a manner as described in the previous sections. The LLC has four banks and has a total capacity of 512 KB. The VLSU fetches data from the LLC in the generated SoC, which can sustain a throughput of up to `DLEN` bits per cycle. The LLC is

Table 3.1: Saturn functional unit summary.

Functional Unit	Saturn Type	Implementation
Integer add/sub/max/min	Pipelined	1-stage array
Shifts	Pipelined	2-stage array
Bitwise	Pipelined	1-stage bitwise array
Integer Divide	Iterative	Elementwise divide unit
Integer MAC	Pipelined	3-stage array
Permutes: slides, gathers, compress	Pipelined	1-stage, writes back results from permute unit
FMA (+ variants)	Pipelined	4-stage array
FP Divide	Iterative	Elementwise
FP convert, compare	Pipelined	2-stage array

warmed up such that the data is resident. The performance results are shown with Saturn parameterized with sensible values, such as modest queue depths, as described earlier in this chapter.

Figure 3.7 shows performance results on SGEMM kernels with swept dimensions for two Saturn configurations, assuming an operating frequency of 1GHz. The two configurations are V256D128, a machine with a vector length of 256b and a datapath width of 128b, and V512D256. Figure 3.8 shows the corresponding datapath utilizations results for the same kernels. The `LMUL` in these kernels is set to four. These results show some of the advantages of the Saturn implementation of a short-vector machine. Both configurations are able to achieve high utilizations even at small matrix dimensions. Both reach greater than 90% of the maximum machine performance across most of the swept dimension sizes. As expected, the V256D128 machine achieves a higher utilization more quickly as it does not require quite as long application vector lengths to saturate its functional units.

We also run a physical evaluation of Saturn to demonstrate its implementability and to understand the costs of various portions of the machine. To do this, we ran several instances of Saturn through a physical design flow using the Cadence suite of EDA tools. We use the Intel16 process technology, a 16nm-class FinFET process. The results shown here are with Saturn synthesized and placed-and-routed at 800 MHz, meeting timing at the pessimistic slow-slow corner.

Figure 3.9 shows the area results of the scalar unit and three instances of Saturn. This plot shows several important results. First, the area of the scalar core is a significant portion of the total size of the whole machine. This helps demonstrate the compactness of the vector unit itself. The scalar unit could be scaled down further in size, reducing the total area of the machine while retaining the benefits of the Saturn implementation. These results show that a complete and performant implementation of the RVV 1.0 specification, including all of its features, can be implemented compactly. Next, the area of the vector machine is dominated

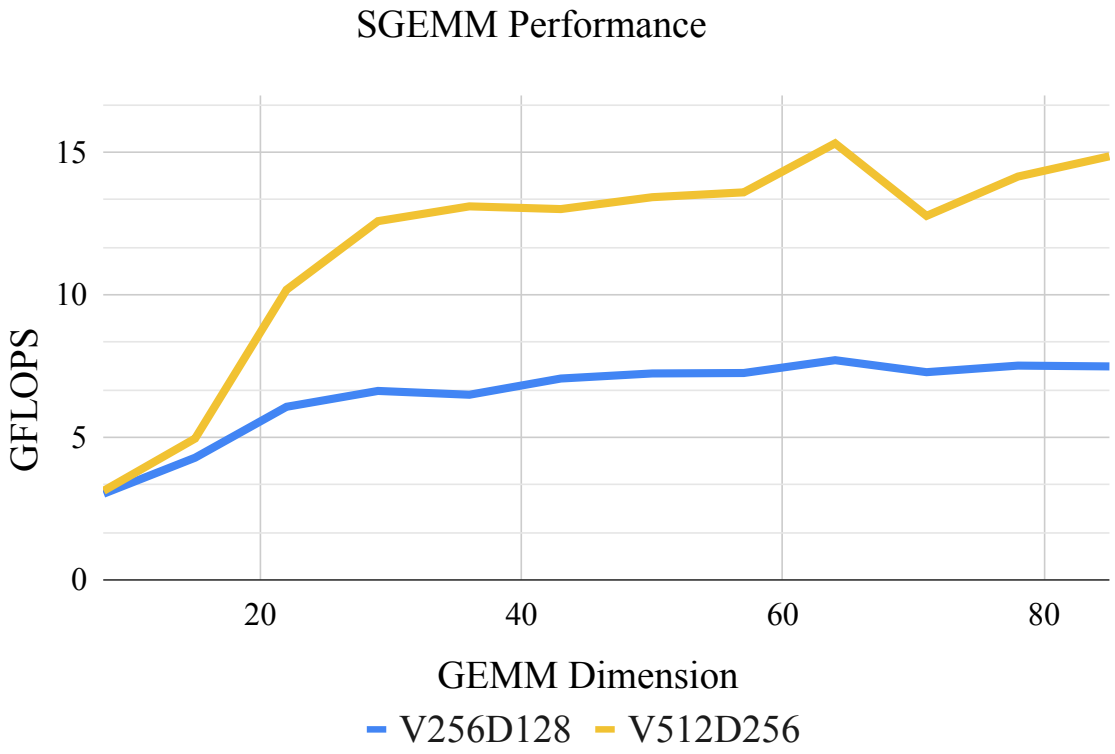


Figure 3.7: Saturn performance on SGEMM.

by the VRF and the functional units, in particular the FMAs and integer MAC units. The cost of the sequencers and issue queues, the heart of the mechanisms behind Saturn and the short-vector paradigm, are small in comparison. Finally, examining the results of the different design points, we see the area costs of various portions of the machine scale as expected. The functional unit cost roughly doubles when the datapath width doubles. The VRF doubles in size with the architectural vector length. The area cost of the VRF here is limited by the available memory generators, with the area expected to decrease with a more optimized, compiled memory. The cost of the sequencers and control logic only grows modestly across these design points.

Figure 3.10 show the power results for the same Saturn configurations when executing a high-utilization SGEMM kernel (dimensions of 64) at a clock frequency of 800MHz. Similar to the area results, the power of the scalar core is a significant portion of the total power. The power consumed in the VRF and the functional units roughly doubles when the `VLEN` and `DLEN` are doubled, respectively, as expected. The VRF and the FMAs dominate the

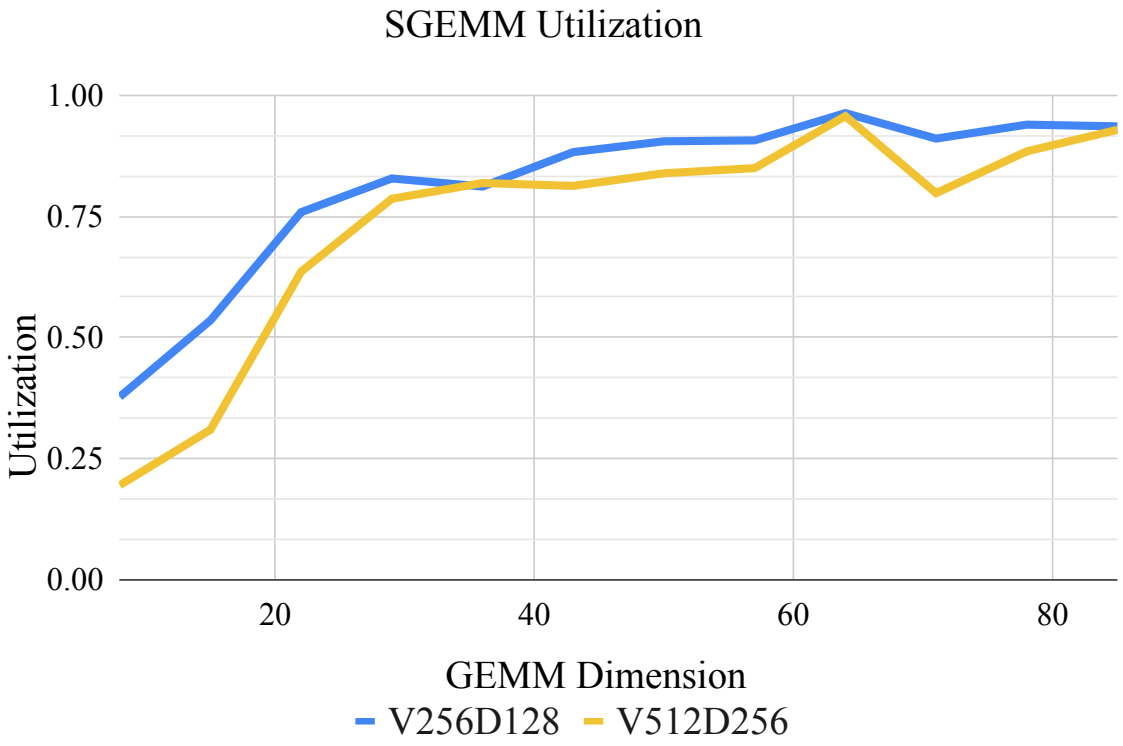


Figure 3.8: Saturn datapath utilization on SGEMM.

power cost of the vector unit hardware, with the remainder being made up of many small contributions from various modules, such as the sequencers and issue queues. The inputs to the inactive functional units are zero-gated, so their power contribution is largely limited to leakage power. Once again, this shows the minimal cost of the key microarchitectural mechanisms behind Saturn's implementation of a short-vector machine. Considering the scalar unit, L1 caches, and vector unit for the V512D256 design, Saturn achieves an energy efficiency of 121 single-precision GFLOPS/W.

Figure 3.11 shows an annotated layout of a V512D256 Saturn design, as evaluated in this section. The design has been placed-and-routed (PnR) into a block of 0.8mm by 0.8mm. We can see the layout results of the fully synthesized and PnR'd VRF in the center of the design.

These results demonstrate the efficacy of Saturn's implementation of a short-vector machine across a variety of problem sizes. The physical results show that Saturn is spending area and power mostly in the VRF and functional units, rather than in the vector control

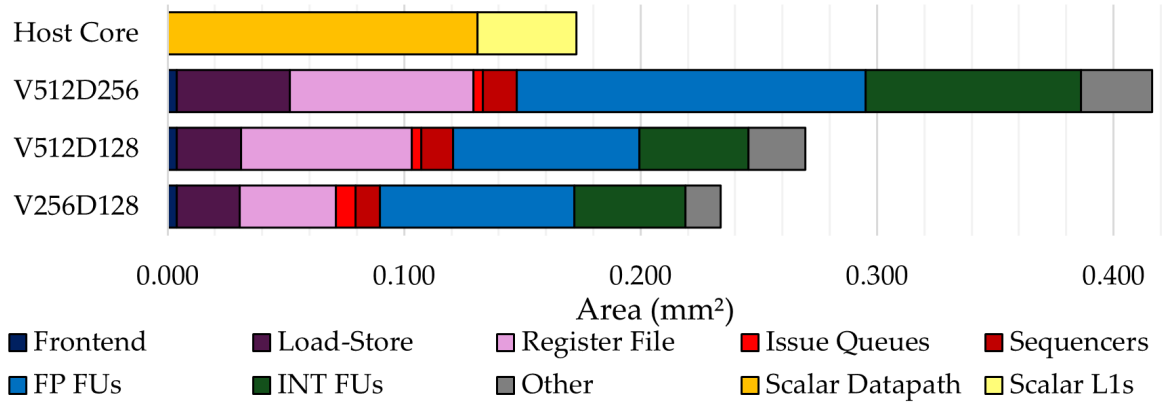


Figure 3.9: Saturn area results for several configurations [132].

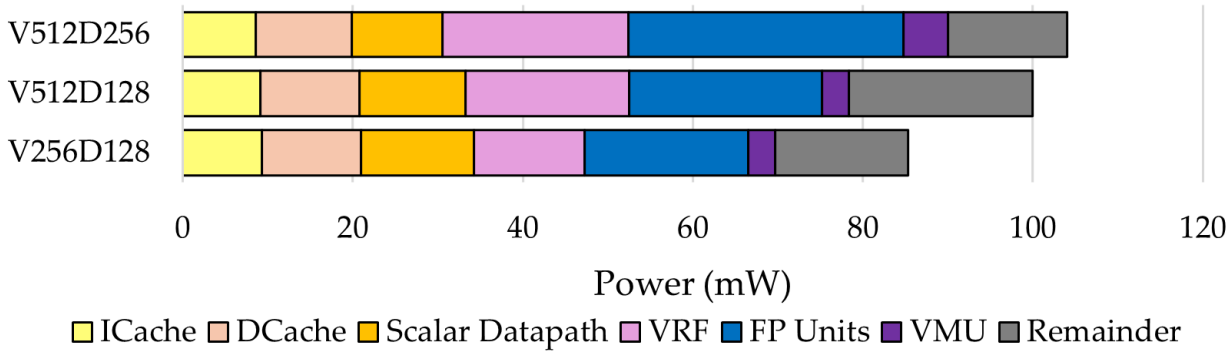


Figure 3.10: Saturn power results for several configurations [132].

mechanisms. This bodes well for Saturn as its key components do not limit its implementation at these design points. The costliest portions of the design can be optimized further with more advanced design techniques. For instance, the VRF can be implemented out of a more area- and power-efficient memory than simply synthesized and placed-and-routed flip-flops. This would further improve the energy and area efficiency of Saturn implementations without changing its core structures.

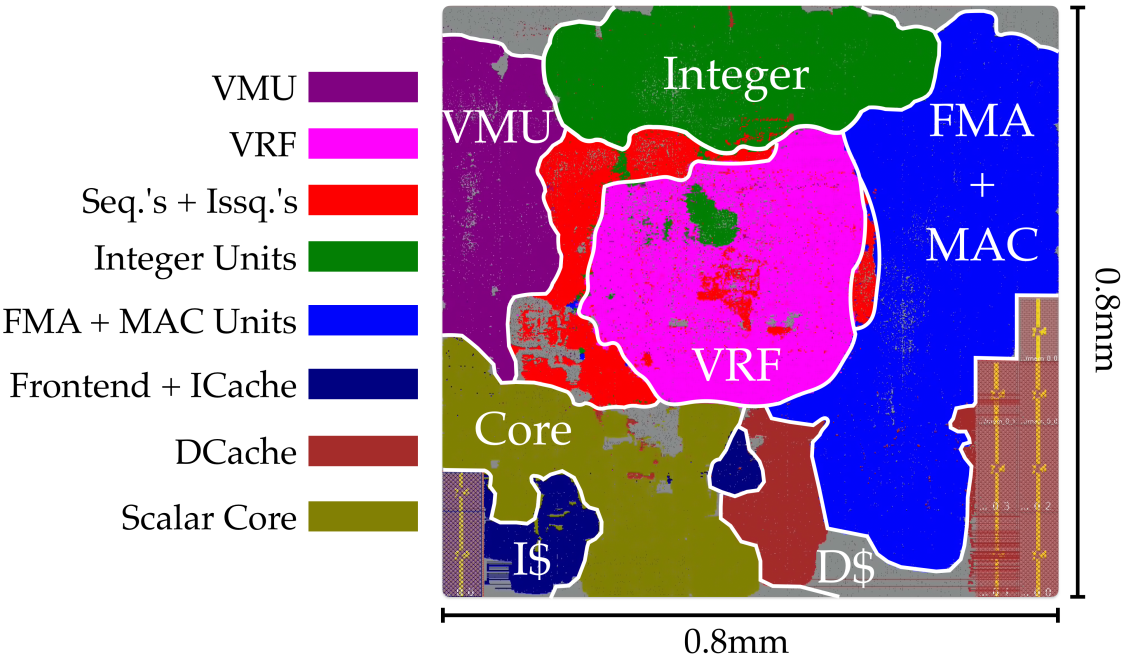


Figure 3.11: Annotated Saturn PnR results for a V512D256 configuration [132].

3.5 Related Work, Summary, and Conclusions

Related Work

Spatz is an academic vector machine targeting short vector lengths [25]. Spatz aims to be energy efficient by eliminating as much control overhead as possible while implementing most features of the RVV 1.0 specification. This leads to a very compact implementation. Similarly to Saturn, it implements a unified, banked VRF, but is set at two banks. Spatz manages all vector operations through a single controller that tracks hazards through a scoreboard. In Saturn, independent sequencers dynamically schedule operations, leading to limited out-of-order execution in addition to chaining all data hazards. Spatz chooses to stall scalar memory operations while waiting for vector memory instructions and vice-versa, while in Saturn, scalar and vector memory operations are disambiguated and only stalled on potential hazards.

Arm MVE, or Helium, is the vector extension for the Arm Cortex-M series of processors [5]. Helium targets short-vector implementations for DSP and ML applications, though it leans toward an embedded design point. Cortex-M processors that implement Helium can execute their 128-bit vectors over several beats, allowing for concurrent execution of arithmetic and memory operations with a limited form of chaining. Notably, there is a notion of "beats" of execution built into the architecture, in which typically 128-bit vectors may be executed on a 32-bit or 64-bit datapath. Similar in spirit to Saturn, this mix of

spatial and temporal execution is meant to improve datapath utilization with a minimal scalar core to improve efficiency.

Much of the more recent academic works in vector machines have focused on long-vector implementations. These machines tend to be targeted towards HPC and high-performance machine-learning applications. Hwacha [70][69], Ara [92][91], and AraXL [94] are all decoupled, long-vector machines that leverage deep temporal execution of long vectors. They feature high capacity, lane-distributed, SRAM-based vector register files for scalability. Vitruvius+ [81] and RISC-V^2 [89] are vector units that implement register renaming.

There are a number of commercial vector units in the short-vector machine design space, although most of their design details are unknown. These include the SiFive X-series of vector machines [111], the Andes NX27V [122], and the Semidynamic Avispado with vector unit integration [110]. The SiFive P470, P670, and P870-D are examples of out-of-order short-vector machines [112].

Summary and Conclusions

Short-vector machines can provide flexible and efficient multimedia and embedded machine-learning processing for mobile SoCs across a variety of design points. They leverage economical hardware features to efficiently sequence short chimes spatially and temporally without requiring high instruction throughput. Short-vector machines also support a vector-length-agnostic programming model and features like vector register grouping by implementing a modern scalable vector ISA. This is in contrast to other DSP core paradigms that rely on expensive, high-IPC scalar units or VLIW frontends that require extensive static scheduling or simply sacrifice performance in favor of a minimal scalar unit that integrates narrow, tightly coupled P-SIMD datapaths. These qualities of short-vector machines have positive implications for PPA, energy efficiency, and code size across a wide swath of kernels with both small and large problem sizes.

The Saturn vector unit project demonstrates how to build a generator for compact, efficient short-vector machines, while being feature compliant with the RVV 1.0 specification. Saturn implements a DAE-style architecture, limited out-of-order execution of vector operations, and first-class support for vector register grouping. This leads to vector operation execution with zero dead time, granular chaining across all data hazards, and tolerance to memory latencies. Saturn can achieve high utilization of its functional units and memory resources for applications with both short and long vector lengths, leading to high performance and energy efficiency. In this chapter, we showed simulation and PPA evaluation results for Saturn to demonstrate this. A silicon implementation of Saturn will be presented in Chapter 5, showing measured results for performance and efficiency.

To push the energy efficiency of these short-vector machines further for particular application domains, we must turn to specialization, the focus of the next chapter. With Saturn, we have a platform for exploring design techniques for specialization quantitatively.

Chapter 4

Techniques for Specializing Short-Vector Machines

Short-vector machines can be deployed as DSP cores in mobile SoCs, as they can be implemented compactly and efficiently process a wide range of multimedia workloads of both large and small problem sizes. However, when we examine the PPA design space of different parameterizations of a baseline short-vector machine, such as the Saturn vector unit, we find that it is quite coarse. Since DSP cores often have strict PPA constraints, this type of trade-off space means that performance may have to be sacrificed for the sake of reduced area, or vice versa. Furthermore, given the importance of multimedia workloads in mobile SoCs, these DSP cores must be as efficient as possible. Processing flexibility is achieved at the cost of energy efficiency compared to fixed-function solutions, so it is critical to work to close this gap. To explore flexible and more efficient short-vector machines, we turn to domain specialization.

This chapter discusses design techniques for the domain specialization of short-vector machines, extending a baseline template machine with new hardware features and novel vector instructions. The primary focus is on techniques that remain broadly applicable within a domain, rather than highly customized solutions for absolutely minimizing the energy cost of singular kernels.

Section 4.1 discusses the baseline parameterization of the Saturn vector unit and the resultant PPA design space, building on the results of Chapter 3 and motivating the need for domain specialization. It also motivates the use of Saturn as a platform for specialization.

Section 4.2 states the goals of domain specialization in this context and discusses how the characteristics of mobile multimedia workloads can be leveraged. It then taxonomizes various techniques for specializing short-vector machines.

Sections 4.3 and 4.4 present design techniques for specialization at the microarchitectural level, showing how these techniques can extend Saturn and discussing their benefits and limitations.

The remaining sections discuss the design of specialized vector instructions in the context of this thesis. They then present a set of novel vector instructions that extend the baseline

RVV 1.0 ISA to improve Saturn's efficiency for multimedia workloads and enable new design trade-offs. These instructions improve the baseline energy efficiency by capturing kernels in higher dimensions, reducing the amount of data shuffling through the VRF, and enabling higher compute throughput.

4.1 Baseline Short-Vector Machines

As outlined in Chapter 3, the Saturn vector unit is a parameterized generator. When generating Saturn vector unit instances, the designer can set several important high-level aspects of the machine, such as the vector length, `VLEN`, and the datapath width, `DLEN`. These parameters will set the area cost of the VRF and the functional units. The designer can also set a number of low-level parameters, like the sequencer configurations, the depths of the issue queues and buffers in the VLSU, and functional unit pipeline depths.

The low-level parameters in Saturn are quite important for the performance of the machine. If the various buffers in the vector backend are not deep enough, the machine will not be able to support a sufficient number of in-flight operations and will frequently stall. However, as noted in Chapter 3, we empirically find that these queues and buffers need not be particularly deep. Beyond a certain point, performance in most cases will no longer improve by making them deeper. The configuration of the sequencers can be set to a few sensible defaults, and the functional unit pipeline depths are often set to be as shallow as is practical given clock frequency targets. As a result, the portions of the machine controlled by these parameters only contribute a small portion of the overall area and power of the design. Examining the Saturn PPA results presented in Chapter 3, the vector length, datapath width, and the configuration of the scalar core dominate the power and area results.

To reiterate, DSP cores in mobile SoCs are typically constrained by performance and area requirements, trying to meet these targets while striving to be as energy efficient as possible. When parameterizing a Saturn-based short-vector machine for a particular deployment, to increase the performance significantly, we must increase the `DLEN` (and also likely the `VLEN`). In other words, **large changes in performance require large changes in power and area**.

From the Saturn PPA results presented in Chapter 3, changing the machine from a `V256D128` configuration to a `V512D256` configuration roughly doubles the performance at an increase of about 45% in power and 75% in area for the vector unit. Looking at the resultant PPA design points using the baseline parameterization of Saturn, we find that the design space is quite coarse. Mobile SoC design is all about making trade-offs to achieve an efficient result, so this coarse design space presents a challenge. Thus, we want to design ways to enable a richer set of trade-offs in the short-vector machine design space. For this, we turn to domain specialization, since adding specialized features can enhance performance and efficiency without scaling the size of the entire vector unit. Beyond enabling new trade-offs, domain specialization is a critical technique for improving energy efficiency.

In this chapter, we will not discuss the role of voltage scaling in energy efficient short-vector machine design, but it is still important to mention. Voltage scaling is an effective technique for improving the energy efficiency in CMOS designs [28]. Voltage and frequency scaling play important roles in trade-off analysis when optimizing designs at the circuit and microarchitectural levels [9][76]. While outside the scope of these thesis, voltage and frequency scaling techniques can be applied on top of the techniques presented here.

4.2 Short-Vector Machines as Templates for Domain Specialization

The need for energy efficient processing for multimedia applications and embedded ML in mobile SoCs drives the design of specialized processing [49]. A short-vector machine is an effective template for flexible, 1-dimensional vector computation in mobile SoCs, but they can be extended to further improve their efficiency. Domain specialization can increase performance and efficiency by matching the capabilities of a processor to the arithmetic and dataflow characteristics of a particular application domain. Our goal is to design cost-effective mechanisms that extend short-vector machines to improve their efficiency and enable new PPA trade-offs by leveraging domain-specific workload characteristics. The following sections will present design techniques for achieving this while retaining processing flexibility and composing neatly with the baseline machine and vector programming model.

Recalling the discussion on metrics for evaluating DSP cores in Chapter 2, we will primarily focus on techniques aimed at improving energy efficiency and enabling new silicon area trade-offs. The high-level goals of this work can be summarized as follows:

- To select and parameterize a baseline design.
- To enhance energy efficiency when executing applications in a particular domain while retaining processing flexibility.
- To enable greater vector DSP core deployment flexibility by creating new PPA design space trade-offs, instead of simply relying on the baseline core parameters.
- To leverage the key features of short-vector machines and to retain compatibility with the vector programming model, enabling kernel portability and empowering performance programmers.

To understand the techniques for specializing short-vector machines, we'll first discuss ways in which workloads can be characterized. First, we can look at high-level workload characteristics. One way of classifying a particular kernel is according to its arithmetic or operational intensity, a measure of the amount of data reuse [23]. A particular processor can be characterized by its balance of compute throughput and memory bandwidth and can be viewed graphically using a roofline model [128][67]. These pieces of information, taken

together, determine if a kernel is compute-bound or memory-bound when deployed on a particular processor.

For memory-bound kernels, the effective memory bandwidth of the processor must be increased to improve performance. To do this, the memory interface of the short-vector machine can be widened, which must be backed up by a sufficiently high bandwidth memory hierarchy. In the DSP core domain, banked TCMs are often used to provide high bandwidth, fixed-latency memory accesses. Scatter-gather optimized TCMs are also frequently designed to significantly improve memory bandwidth for kernels that rely on indexed accesses, such as sparse kernels, lookup tables, and lens distortion correction [31][17].

Compute-bound kernels have a surplus of arithmetic intensity that can be converted into higher performance, and potentially better energy efficiency, through the deployment of additional compute resources without increasing the memory bandwidth. For example, embedded machine-learning workloads are often dominated by 2D multilayer convolutions, GEMMs, and fully connected layers. 2D multi-layer convolutions, as commonly found in convolutional neural networks (CNNs), tend to have very high levels of data reuse, helping to explain the great success of many architectures at efficiently accelerating them [40]. GEMM kernels have an arithmetic intensity that scales with the dimensions of the matrices, while convolutions have an arithmetic intensity that scales with the size of the convolution kernel. Non-batched fully connected layers map to a matrix-vector multiplication kernel, which has low data reuse that does not scale with the problem size.

As discussed in Chapter 2, many operations in image processing have low arithmetic intensities, but successive operations in a given processing pipeline can be fused together to ameliorate this. These dense, high-arithmetic intensity kernels are prime targets for efficient acceleration due to their compute characteristics and their general importance. In this chapter, we will present techniques for adding useful compute resources to a baseline short-vector machine to target such workloads.

Another method of characterizing a target domain is to classify its computational and data communication patterns [8]. By classifying workloads and identifying their important computational patterns, we can design vector primitives that more efficiently capture them while composing with the baseline machine. Simply increasing the vector length and datapath width in a short-vector machine does increase the maximum compute throughput, but yields diminishing returns for overall performance and struggles to help with short application vector lengths.

Common computational patterns can be leveraged to extend the baseline microarchitecture or instruction set. Common strategies for doing so include the design of fused and compound instructions, which combine multiple successive operations together and execute multiple related operations concurrently, respectively [105].

Many mobile multimedia and embedded ML workloads are inherently multidimensional. Capturing computations in dimensions higher than 1D vectors can yield significant improvements in performance and energy efficiency. We will explore specialized vector instructions that target these compute patterns in the following sections.

Why use short-vector machines as a platform for specialization?

We have now discussed the key motivations for specializing short-vector machines, but we might take a step back and ask why choose short-vector machines as a platform for this at all? Chapter 3 argued for the use of short-vector machines in the DSP core context, given that they provide flexible yet efficient data-parallel processing for multimedia workloads. With Saturn, we have demonstrated that short-vector machines can achieve high hardware resource utilization across a wide range of problem sizes while being implemented compactly. While machines that achieve lower utilizations of their resources can still benefit from domain specialization, short-vector machines are well-positioned to capitalize on additional specialized hardware resources, since they should be able to use them effectively.

Many existing aspects of the short-vector machine can be leveraged. The dynamic sequencing mechanisms in the vector backend and the support for vector register grouping means that specialized functional units can be efficiently driven alongside existing ones. As we shall see later in this chapter when discussing new vector instructions to accelerate convolutions, dynamically extending the maximum vector length also has the benefit of reducing the cost of vectorizing problems horizontally across the vector.

Domain-specific accelerators leverage specialized storage structures to minimize data movement and maximize data locality. The unified VRF in Saturn can serve as this local storage. Specialized vector instructions can combine data permutation and computation into a single instruction to align vector data within the datapaths rather than in separate vector registers. This reduces the energy cost of VRF reads and writes.

In short, we have demonstrated that Saturn can generate efficient and compact short-vector machine instances. While the generated design space is coarse, it is primed to leverage domain specialization to improve efficiency and enable new trade-offs. We want to make sure we're optimizing a design that is already capable and efficient, rather than simply optimizing a design that is a low-performing baseline.

Opportunities for Improved Efficiency in Short-Vector Machines

We can first examine where in our baseline short-vector machine there are opportunities to improve energy efficiency. Fundamentally, the Saturn vector unit does not rely on a high-IPC scalar core, but we can still reduce its overheads. By designing specialized instructions that encode more operations, we can amortize the cost of instruction fetch, decode, and sequencing. This can be captured by analyzing the dynamic instruction count required for a particular kernel. Domain specialization can also potentially allow the designer to reduce the cost of the scalar core at implementation time. By designing features that further reduce the vector unit's reliance on scalar performance, we might be able to reduce the hardware resources devoted to the scalar unit.

The VRF is another significant contributor of power and area in short-vector machines. Therefore, it is desirable to amortize the energy cost of each VRF read and write. This might be done by fusing multiple steps of computation together or folding data permutation

together with computation. Capturing a pattern common in a domain (e.g. a convolution-type operation in image-processing workloads) with a specialized primitive can significantly reduce the amount of data shuffling required to and from the VRF and vector memory accesses.

The functional units themselves also consume a significant amount of power and area, in-particular the FMA and MAC units. Defining specialized functional units that exactly match the computation required in a particular kernel may provide savings by reducing the overhead of sequencing an operation on more general-purpose functional units.

We found in Chapter 3 that the circuitry required for sequencing and control in the vector backend (e.g. the sequencers, issue queues, etc.) is not a significant consumer of the overall area or power. Thus, they are not an important target for optimization. Instead, due to their low cost, we might be able to implement additional sequencers and control hardware that drive domain-specialized features to improve efficiency.

Domain Specialization Taxonomy

The space of design techniques for specializing short-vector machines is wide, but we taxonomize it as follows:

- **Microarchitectural Techniques:** The baseline microarchitecture can be extended with new features that increase performance and energy efficiency. These are changes that are invisible at the architectural level, but can have significant implications for PPA and how vector kernels should be optimized. Thus, these techniques lead to *implicit* specialization, since they do not make any particular vector instruction more expressive.

- **Specialized Vector Instructions:** Specialized, scalable vector instruction assists can be designed to extend the baseline vector ISA. This represents *explicit* domain specialization, as the instructions explicitly encode a new, ideally more efficient, operation. We will focus on new instructions that are designed to assist in the computation of a particular class of kernels and are compute primitives, rather than inflexibly computing a large set of tasks with a single instruction. Specialized vector instructions can be further segmented into two categories, those operating on existing architectural state and those that add new architectural state. The former strategy solely uses the VRF as its local storage, while the latter adds additional addressable state that might include specialized buffers or accumulators.

- **Custom Accelerator Integration:** A short-vector machine may be integrated with one or more domain- or application-specific accelerators. This represents a much more specialized strategy in which the short-vector machine is used as a baseline for general-purpose vector compute, while the most intensive operations are offloaded to the attached accelerators. The design of "magic" instructions that might execute hundreds of operations can yield huge efficiency gains [45]. The trade-off is the inflexibility of

these datapaths. This strategy is not a primary focus of the design techniques presented in this chapter, but we will discuss how a short-vector machine can be extended with such operations while still interfacing with the vector unit.

The following sections describe design techniques for specializing short-vector machines, presenting several microarchitectural techniques and a set of specialized vector instructions designed by examining important computational patterns in mobile multimedia and embedded ML workloads. These design techniques leverage the characteristics of short-vector machines and are meant to naturally compose with the baseline machine and vector programming model such that they can remain scalable and portable across different DSP core design points.

4.3 Vector Backend Sequencer Configurations

The typical short-vector machine configuration that we have been discussing thus far has a balance between its maximum memory bandwidth and compute throughput for fully pipelined operations (e.g. MACs, FMAs). In other words, the width of the memory interface equals the width of the pipelined functional units (recall the `DLEN` parameter). One way to increase the maximum compute throughput of the machine is to add more arithmetic sequencers. In the Saturn vector unit, these sequencers are configurable.

A typical baseline Saturn configuration, as depicted on the left in Figure 4.1, has several independent sequencers, including for floating-point and integer operations. This allows for concurrent execution of these different types of vector operations. If we want to increase the maximum throughput for a particular operation, such as FMAs, we can generate a configuration as depicted on the right in Figure 4.1. This configuration has multiple floating-point sequencers and datapaths, effectively doubling the maximum throughput for FMA operations without doubling the `VLEN` or `DLEN` parameters. This configuration pays for the area cost of the additional functional units and sequencing logic without increasing the size of the VRF or other functional units. Workloads with sufficiently high arithmetic intensities can take advantage of this capability.

There are several ways in which these sequencers and their issue queues can be implemented in the Saturn vector unit. For an additional sequencer meant to execute a unique set of operations, it should have its own issue queue. Then the sequencer can execute the buffered operations independently, allowing for more concurrent execution of vector operations across the different sequencers. Recall that operations within an issue queue execute in-order. If sequencers with disjoint operations were to share the same issue queue, then vector instructions matching each sequencer would need to be scheduled in the instruction stream in turn. If the operations were not properly interleaved, then the non-matching sequencer would be blocked until the matching sequencer popped one or more vector operations off the issue queue. In area-minimal Saturn configurations, unifying sequencers would likely be preferable to sharing issue queues in this manner.

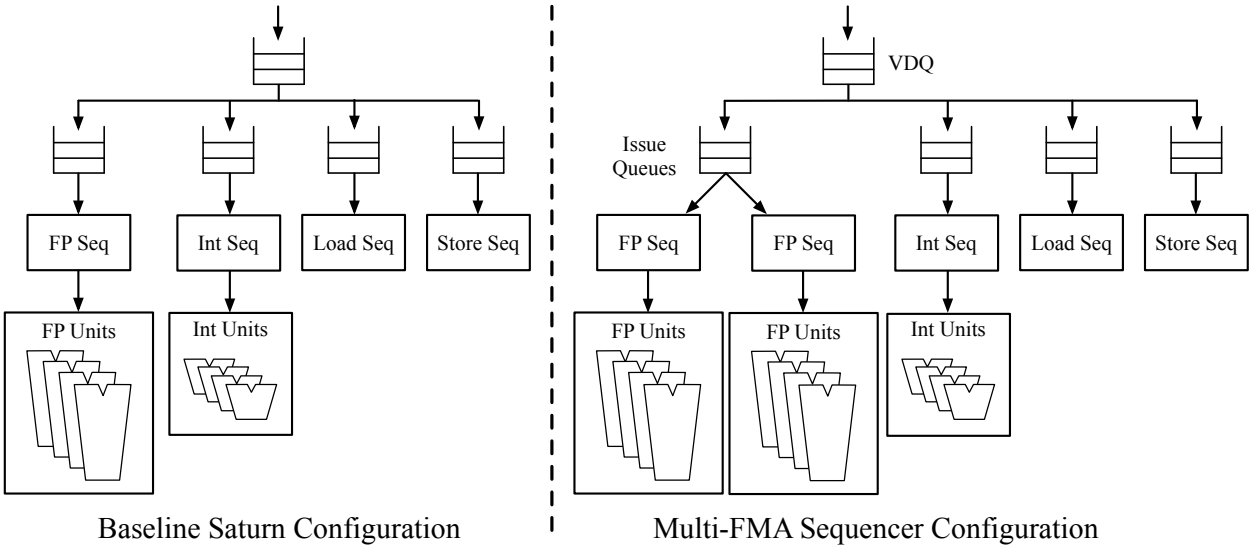


Figure 4.1: Baseline and multi-FMA sequencer configurations of Saturn.

For sequencers that share an operation type, such as in the case of instantiating multiple FMA sequencers, it is preferable to share an issue queue, as pictured on the right in Figure 4.1. In this configuration, each sequencer can simply begin executing the next available instruction when it is finished sequencing its previous instruction. If they were instead to have separate issue queues, then operations leaving the `VDQ` would need to be distributed to each issue queue in a load-balanced manner. Otherwise, each sequencer may run out of useful work. This load-balancing would be difficult in the face of dynamic system behaviors that may temporarily block the execution of particular vector operations. Enabling the sharing of the issue queue among multiple sequencers requires only minor logic modifications, including simply prioritizing one of the sequencers when both are ready to accept a new operation on the same clock cycle.

An additional complexity is the impact of structural hazards on the VRF. For sequencers to be doing useful work, there must be sufficient VRF banks to enable each sequencing agent to independently access the read and write ports. VRF read and write bandwidth could also be increased by increasing the number of independent read and write ports on each bank, but that will not be considered further here due to the high physical cost of doing so. If there is a bank conflict between two sequencers, there is arbitration logic to give access to one of the sequencers on that particular cycle. The other sequencer will then likely gain access on the subsequent cycle. In normal operation, after this initial cycle delay, the sequencers can then be accessing offset banks such that they can proceed conflict-free between each other for the remainder of their execution. Adding more independent sequencers does incur an area cost for connecting to the VRF access crossbars.

Figure 4.2 shows the performance results of executing an SGEMM kernel, sweeping the

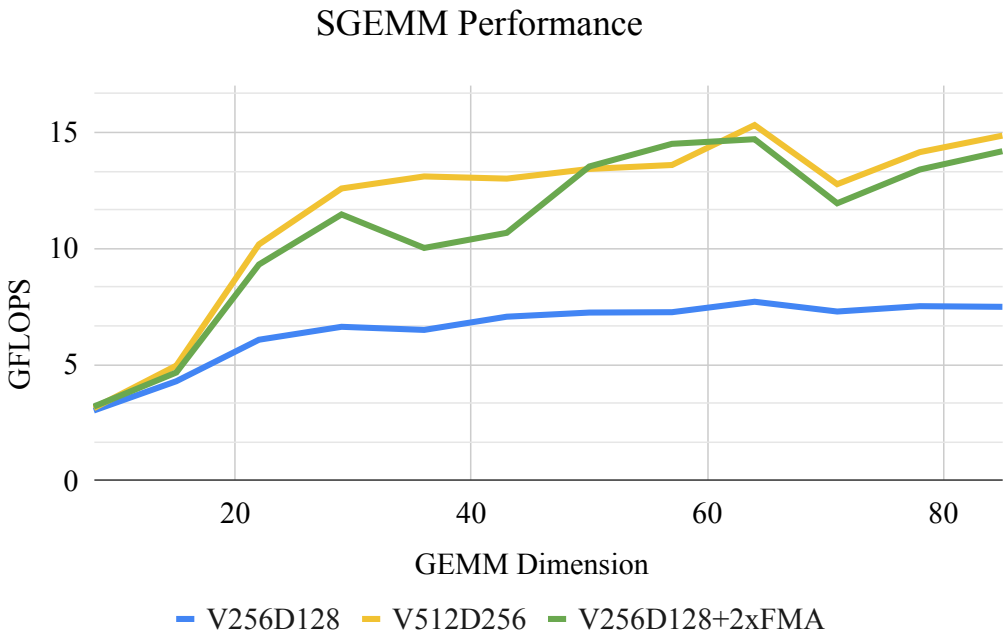


Figure 4.2: Saturn multi-FMA sequencer performance.

matrix dimensions, on a baseline `V256D128` Saturn configuration, a `V512D256` configuration, and a `V256D128` configuration with two FMA sequencers, `V256D128+2xFMA`. We can see that the performance of the two sequencer configurations largely matches that of doubling `VLEN` and `DLEN` across much of the range of GEMM dimension sizes, even exceeding it for certain problem sizes.

Figure 4.3 shows the normalized area costs of the same Saturn configurations when implemented in the Intel16 process technology. The configuration with multiple FMA sequencers has a roughly 25% area overhead compared to the baseline, while the `V512D256` configuration has a roughly 80% area overhead. This means that the multiple FMA sequencer configuration can create an effective trade-off in short-vector machine implementations if the target workloads can leverage the additional compute throughput without a corresponding increase in memory bandwidth.

This approach has several important limitations. As noted previously, because it increases the compute throughput without increasing the memory bandwidth, this configuration does not help kernels with low arithmetic intensities. While it is a very general-purpose specialization feature, it will only help with kernels with sufficient surplus data reuse that can be leveraged.

This approach can also be limited by the application vector lengths and the vector in-

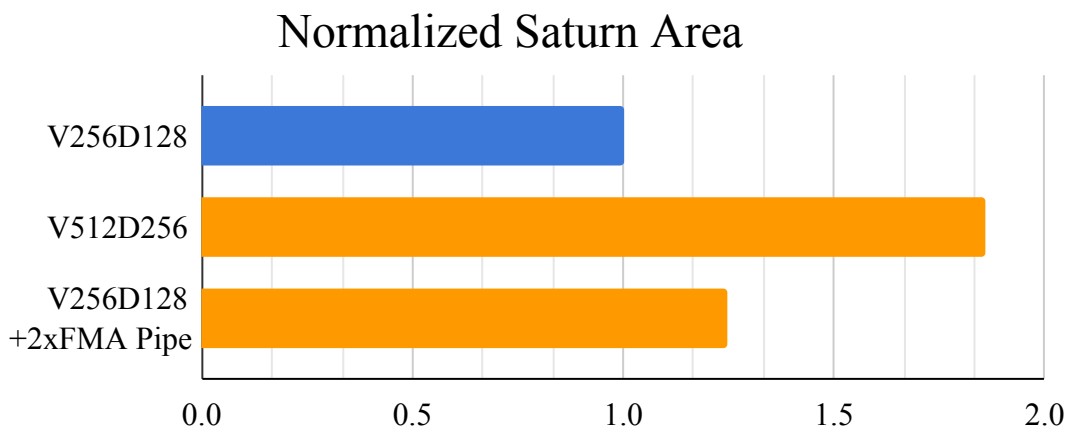


Figure 4.3: Saturn multi-FMA sequencer normalized area cost.

struction issue rate. Recall that in Saturn, the scalar core issues up to one vector instruction per cycle to the vector backend. For multiple FMA sequencers to overlap execution with high datapath utilizations, the scalar core must feed the backend with operations at a sufficiently high rate or the application vector lengths must be long enough such that vector register grouping can be used to keep each sequencer busy.

Finally, adding a second FMA sequencer yields a very limited power difference compared to doubling the width of the machine, such that the overall energy efficiency is not meaningfully increased in comparison to the larger baseline machine configuration. While the VRF is not increased in size, having multiple sequencers running at full rate means that it is being accessed twice as frequently. The number of FMA units is doubled, so the power contribution of those units will be similar to those in the wider machine. This is a purely microarchitectural technique, so the number of vector instructions executed does not change compared to the baseline machine configuration. Thus, there is no reduction in dynamic instruction count, and the costs of instruction fetch and the scalar core are not reduced or amortized by encoding a greater number of operations per instruction. In terms of physical implementation, while we have seen the cost of the sequencing logic is minimal in terms of the overall machine, this approach does incur overheads in the hazard checking and VRF access logic on top of the cost of the additional sequencer and functional units.

Despite these limitations, the results show that augmenting a baseline short-vector machine with additional sequencers and functional units can be effective for increasing the machine throughput with a lower area cost than doubling the entire vector machine width. This is particularly useful for target domains that have sufficiently high arithmetic intensities such that the functional units can be highly utilized.

4.4 Vector Operation Fusion

Macro-op fusion is a common microarchitectural technique deployed in many microprocessors, capable of improving performance [26][114]. Macro-op fusion can also be applied in the short-vector machine context by identifying patterns of instructions that can be effectively combined together. Doing so has the potential to increase performance and improve energy efficiency by reducing the amount of shuffling of data in the VRF. In Saturn, the fusion opportunities could be detected in the frontend of the machine or in the vector backend itself. The frontend decoupling queue of the vector unit, which receives all vector instructions in order, can be augmented to detect sequences of operations that may be fused together by comparing the operations, vector register destinations, and instruction metadata.

A simple, demonstrative example is to enable effectively non-destructive scalar-by-vector FMA-type operations. The RVV 1.0 specification only defines destructive FMA operations, which conserves instruction encoding space.

```
// LMUL = 4
vmv.v.v v0, v8          // Vector register move
vfmacc.vf v0, ft0, v4   // FMA instruction with matching destination
```

Figure 4.4: Example of fusable vector move and FMA instructions.

In an instruction sequence like in Figure 4.4, we might fuse the vector register move instruction with the following FMA instruction to have an effectively 3-source FMA with a separate destination. This can help vector register allocation in certain vector kernels, possibly reducing the need for unrolling some loops. For instance, in a vectorized software pipelined separable convolution kernel with only destructive FMAs in which the horizontal and vertical convolutions are fused, the inner loop must be unrolled `K-1` times, where `K` is one dimension of the convolution kernel. This is required to place the intermediate convolution results in the correct vector register for subsequent iterations, which could otherwise be avoided with non-destructive FMA instructions. While this may not have a performance impact, it can force a significant expansion of static code size. Vector register moves can be used to avoid unrolling the loop, but actually executing the vector move operations costs execution cycles and energy. By fusing this pattern of instructions, as in Figure 4.4, the vectorized separable convolution could avoid this unrolling without actually paying the costs of the vector move operation beyond its own code size contribution and instruction fetch.

The vectorization of convolution operations presents another opportunity for fusion. Convolutions are highly data-parallel but require the alignment of the operands across the neighborhood of input samples or pixels. Figure 4.5 shows a 1D convolution operation on the left (note that the convolution filter elements are labeled according to their order of computation, not according to their arrangement in the mathematical convolution kernel) and one method

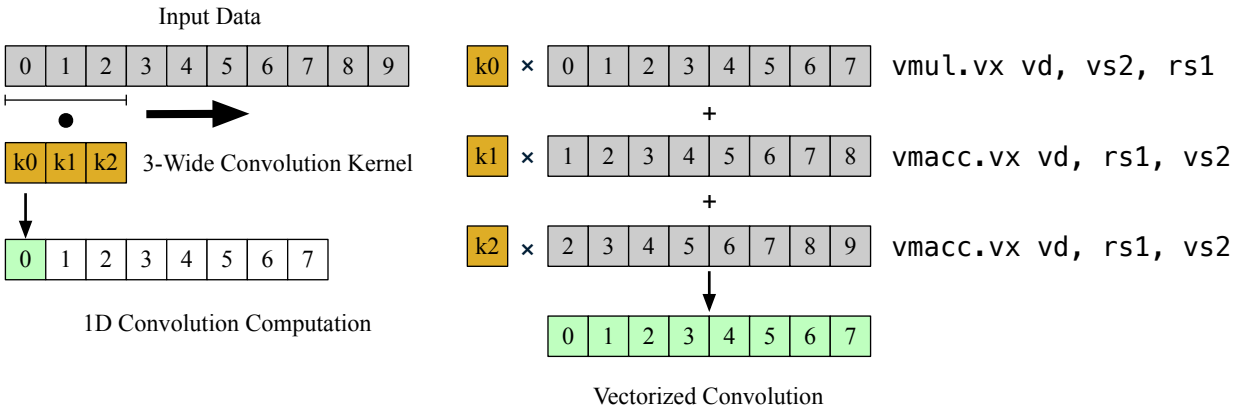


Figure 4.5: Convolution vectorization example using offset vector loads to align the data.

```
// LMUL = 2
// Scalar bookkeeping code is omitted
vle32.v v0, (a0)        // load 32b input data
vle32.v v2, (a0)        // a0 + 4 (offset by 4B)
vle32.v v4, (a0)        // a0 + 8 (offset by 8B)
vmul.vx v8, v0, t0      // multiply by first weight
vmacc.vx v8, t1, v2     // MAC with second weight
vmacc.vx v8, t2, v4     // MAC with third weight
```

Figure 4.6: Offset load-based convolution code snippet.

for vectorizing it. There are several ways to vectorize a direct convolution computation. A number of offset vector loads, equal to the kernel width, into separate vector registers can be used. A single vector load with a series of slide permute operations can also be used. These implementations are shown in the code snippets in Figures 4.6 and 4.7. These vectorization strategies can achieve high utilization of the vector functional units, but they require redundant loads of data from memory or redundant permute operations, as well as the storage of redundant data in the VRF.

An alternative vectorization strategy, the dual of the first slide-based convolution we showed, slides the destination vector down by one element after each MAC operation, as shown in Figure 4.8. The kernel weights are multiplied in reverse order to achieve the correct result. Since the slide of the destination vector matches the destination of the MAC operations, we can fuse the slide and the MAC operation and achieve the same result[1]. By doing this, we can more naturally capture the flow of the convolution, increase the effective

[1]Credit to Albert Ou for the original idea to combine the vector slide with a MAC or FMA operation in

```
// LMUL = 2
// Scalar bookkeeping code is omitted
vle32.v v0, (a0)            // load 32b input data
vmul.vx v8, v0, t0          // multiply by first weight
vslide1down.vx v2, v0, x0   // slide input down by one element
vmacc.vx v8, t1, v2         // MAC with second weight
vslide1down.vx v4, v2, x0   // slide input down by another element
vmacc.vx v8, t2, v4         // MAC with third weight
```

Figure 4.7: Convolution with sliding input code snippet.

```
// LMUL = 2
// Scalar bookkeeping code is omitted
vle32.v v0, (a0)            // load 32b input data
vmul.vx v8, v0, t2          // multiply by third weight
vslide1down.vx v8, v8, x0   // slide input down by one element
vmacc.vx v8, t1, v0         // MAC with second weight
vslide1down.vx v8, v8, x0   // slide input down by another element
vmacc.vx v8, t0, v0         // MAC with first weight
```

Figure 4.8: Convolution with sliding accumulator code snippet.

throughput of the computation, and reduce the amount of data shuffling and VRF writes, thus saving energy.

Vector operation fusion has some inherent limitations. It does not directly provide static code size savings. While it can capture some fixed patterns, such as the combination of arithmetic and data movement in the convolution example, it is more difficult to capture complex patterns. Thus, the possible savings from fusion alone are limited.

4.5 Adding Specialized Vector Instructions

The design of domain-specialized vector instructions can enable significant gains in performance and energy efficiency beyond those possible from the previously described microarchitectural techniques alone. Carefully designed instructions can succinctly encode complex operations that can leverage workload characteristics more effectively than the base vector

the convolution dataflow.

ISA. We will focus on specialized vector instructions that are amenable to implementation in short-vector machines.

The design of specialized vector instructions must consider several aspects of the vector machine. First, the interaction of the instruction with the base ISA and how software is written must be considered. Vector instruction semantics can capture a particular computation in a more natural fashion by encoding a new type of operation or a combination of computation and data movement. This can reduce dynamic instruction count and static code size. The instruction should fit within the programming model of the machine such that it can be used seamlessly within an optimized vector program. By encoding an instruction in a similar fashion to existing instructions, the overheads of its implementation can be reduced.

Second, the underlying implementation must be considered. Designing specialized functional units to directly match the computation encoded in a new instruction can greatly increase performance and energy efficiency. An instruction that encodes a greater number of operations can be implemented in a manner that increases the effective compute throughput of the machine. For instructions that encode a combination of computation and data movement, the implementation can reduce the amount of data shuffling that would otherwise be required, reducing the total number of VRF reads and writes.

We define several tenets of instruction design that we will use to guide the design process of extending our short-vector machine template:

- The instructions should be motivated by the characteristics of the target domain. Energy efficiency can be improved by designing more expressive vector instructions that better capture important computation and dataflow patterns in the target workloads.

- The instructions should be scalable, like the underlying machine. This enables the continued portability of vector-length-agnostic code, and ensures that the designed instruction is applicable across a variety of DSP core PPA design points. It also allows for the writing of efficient and concise kernels by conforming to the vector programming model.

- The instructions should be designed as primitives to maximize utility. The benefits of highly specialized "magic" instructions that might operate for many cycles are well known [45]. However, that is not the focus of this work. We are looking to design new instructions that capture salient, domain-specialized compute patterns that can extend a baseline short-vector machine template in a cost-effective manner.

- The instructions and their implementation should leverage the efficient mechanisms of the baseline short-vector machine. Furthermore, the implementation should not impose excessive PPA overhead on the baseline machine. For example, we will work to design instructions that do not require additional VRF bank read and write ports since they are relatively expensive and degrade overall efficiency when not utilized effectively.

4.6 Accelerating Multidimensional Workloads

Many mobile image-processing and machine-learning kernels are inherently multidimensional. Neighborhood kernels in image processing, such as convolutions, typically require a reduction across multiple vectors to compute a single output vector. In machine learning, 2D multilayer convolutions, as well as other convolution variants, often require reductions of vectors across multiple dimensions. Input channels are convolved with sets of filters, with those results being combined to compute the output channel data. Furthermore, many machine-learning primitives can be mapped to Basic Linear Algebra Subroutines (BLAS)-type operations, namely matrix-matrix multiplications (GEMM) and matrix-vector multiplications (GEMV) [34][30]. Accelerating these GEMM and GEMV operations are critical for efficient machine-learning computation. The importance of these workloads motivates extending a short-vector machine with instructions that provide efficient multidimensional primitives.

In recent years, P-SIMD and vector ISAs have been extended to target GEMM kernels, largely driven by the demand for machine-learning workload performance [84][58]. These are largely complete ISA extensions, defining additional architectural state and typically coupled with a baseline vector extension for 1D processing. Others have worked to define various multidimensional abstractions to efficiently capture multidimensional memory access and compute patterns [130][63]. Some P-SIMD ISAs, like Arm Neon, have defined matrix instructions that implicitly define 1D registers to contain a series of small matrices that can be multiplied together [4]. There have also been works to extend existing scalable vector ISAs with new matrix instructions that operate on tiles of data held in the VRF, with matrix operations being sequenced onto the existing vector functional units [109][93].

The inner loops of an efficient vectorized matrix multiply kernel consist of scalar-by-vector MAC instructions accumulating into a vector holding a portion of the `C` matrix. The loops advance along the shared `K` dimension of tiles of the `A` and `B` matrices, accumulating into a tile of `C` held across several vector registers. We can define a specialized vector instruction to accelerate this computation to extend the baseline Saturn vector unit, in turn accelerating and increasing its efficiency for mobile multimedia and embedded machine-learning workloads.

Multi-Vector Multiply-Accumulate Instruction

The baseline Saturn vector unit is able to achieve high datapath utilizations across a broad range of problem sizes for matrix multiplication and convolution kernels. Thus, in order to further improve performance, we must be able to add additional useful functional units to the machine. As we've seen, increasing the `VLEN` and `DLEN` can improve performance, but comes at a high area cost, and adding a second sequencer for MAC operations can improve performance, but has some critical limitations. Thus, we'd like to explore a family of vector instructions that can improve performance and energy efficiency, while remaining highly utilized, at a reasonable area cost.

We define the `vmvmacc` instruction, and its floating-point counterpart `vfmvmacc`, as a "multi-vector multiply-accumulate" operation. The key idea of this instruction is to effectively fuse a sequence of multiple MAC operations into a single instruction. The `vmvmacc` instruction leverages the multidimensional nature of important multimedia kernels to execute two scalar-by-vector operations along the reduction dimension at once, accumulating into the same vector destination register. This means the instruction is effectively a 2-input dot product.

Following the tenets of instruction design we outlined, this instruction requires no additional VRF read ports. The baseline RVV MAC and FMA instructions read three vector registers for the vector-vector variants, or two vector registers and one scalar value from the scalar register file for scalar-vector variants. The `vmvmacc` instruction instead reads three vector registers and one scalar register. Because this instruction is encoding two scalar-vector operations, the two scalar values are packed into the lower bits of a single 64b scalar register specified by the instruction. This scalar value can be initially loaded into the register file from memory with a double-wide scalar load. The additional vector register source is implicitly defined as the next consecutive vector register after the specified `vs2` register. Thus, this instruction can be encoded similarly to its 1D MAC instruction counterpart without requiring additional instruction fields. The following is the definition of one variant:

```
vmvmacc.vvx vd, rs1, vs2 #vd[i] += (rs1[0]*vs2[i])+(rs1[1]*(vs2+1)[i])
```

Figure 4.9 shows a high-level view of the operation of the instruction, with the packed scalar elements from `rs1` splatted and multiplied against their respective `vs2` source vectors, and accumulating into `vd`. Note that this instruction still composes neatly with the vector register grouping feature in RVV. The instruction behavior respects `LMUL > 1` and the implicitly encoded `vs2+1` vector register will map to the canonical consecutive vector register (i.e. starting at the vector element group that is at `vs2+(LMUL*VLEN/DLEN)`).

Figure 4.10 depicts this mapping for several `LMUL` settings by representing each vector register as one block. This means the `vmvmacc` instruction can leverage the beneficial features of vector register grouping as implemented by the Saturn vector unit.

The `vmvmacc` instruction comprises a family of instructions that operate on both integer and floating-point values. These instructions can also easily be extended to support widening variants, just as the FMA and MAC instructions in the base RVV 1.0 spec do [104]. We restrict this instruction to only be valid for input element widths of `<=32b` because a single scalar register is only able to hold two values of width 32b. This could be overcome by reading two consecutive scalar registers for the scalar coefficients.

Implementation

To implement the `vmvmacc` instruction in Saturn, only minor modifications to the machine frontend and sequencing logic need to be made. The sequencers are modified to account for passing along the scalar register read bits as well as all three vector register reads. For

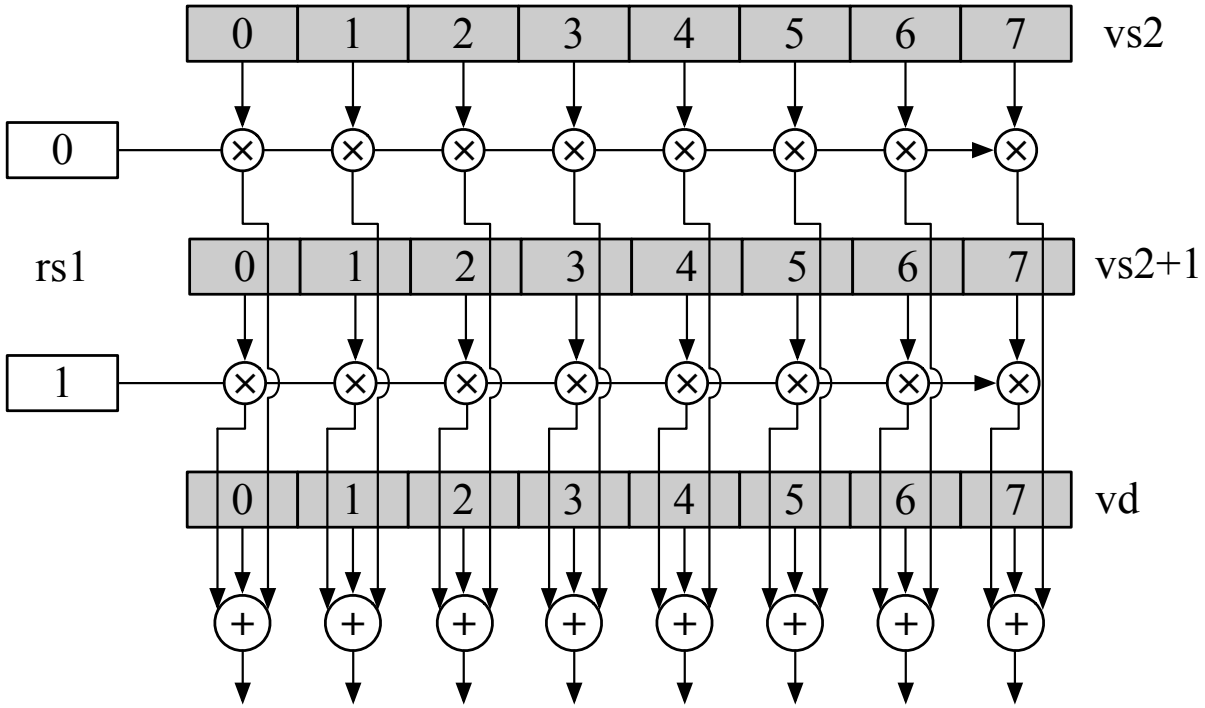


Figure 4.9: `vmvmacc` instruction operation.

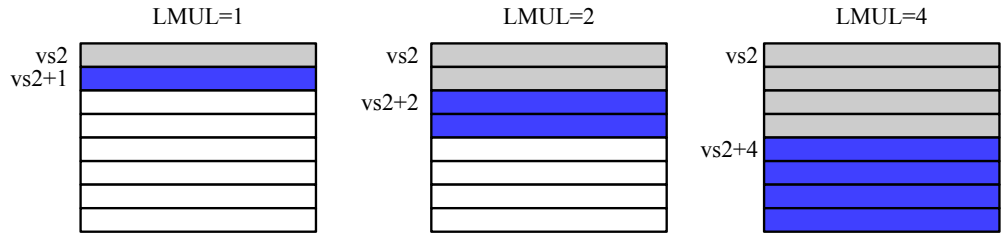


Figure 4.10: Mapping of source vector registers for the `vmvmacc` instruction for several `LMUL` settings.

hazard tracking, the bit masks in the vector sequencers that encode which vector element groups will be read from are modified to advertise read hazards on both `vs2` and `vs2+LMUL`.

The actual functional unit comprising the dot product operation capability is implemented as a `PipelinedFunctionalUnit` in Saturn. Some of the MAC or FMA hardware could be shared with the existing 1D functional units in the machine. The decode matching circuits in the execution unit are modified to accept the `vmvmacc` instruction. The sequencer orchestrates the three vector register reads from the VRF and passes along the scalar register bits to the proper functional unit. The proper scalar values are extracted from the scalar register bits, according to the element width, and are broadcasted to be multiplied against

```
// LMUL = 4
vmacc.vx v0, t0, v16
vmacc.vx v4, t2, v16
vmacc.vx v0, t1, v20
vmacc.vx v4, t3, v20
```

```
// Equivalent version
// t0 and t2 have packed values
vmvmacc.vvx v0, t0, v16
vmvmacc.vvx v4, t2, v16
```

Figure 4.11: Snippet of GEMM inner loop with and without the `vmvmacc` instruction.

their corresponding vector register reads.

The operation is executed in a fully pipelined manner and will be written back to the VRF after a number of cycles determined by the parameterized functional unit pipeline depth. Again, note that this instruction does not require any additional VRF read or write ports, it simply uses the hardware resources already available in the machine.

This instruction can provide significant performance and energy efficiency gains for dense, multidimensional workloads in several ways. By capturing more operations in a single instruction than a baseline MAC or FMA instruction, the overhead of the scalar core is amortized. The dynamic instruction count and the overall cost of instruction fetch, decode, and sequencing is reduced. Furthermore, it allows us to scale up the number of useful functional units in the vector backend. By doing so, this instruction is executing more operations per VRF read and write. Recall from Chapter 3 that the VRF accounts for a significant portion of the vector backend power. By reducing the number of total VRF read and write operations, the total energy cost is reduced. A quantitative analysis on the PPA impact is presented later in this chapter.

Further Extensions

A natural way to extend the `vmvmacc` instruction concept would be to operate on more than two `vs2` vectors at once to increase the maximum compute throughput further. This could be done by increasing the width of the dot products and accumulating into a single vector register. In Saturn, this strategy would still only require a single write port but would require additional read ports to run at full throughput, or a more complex scheduling mechanism to read out more vector element groups over multiple cycles, with the need for additional operand buffers. It would also likely deepen the pipeline latency of the functional units, which could lead to increased vector register pressure. An alternative would be to execute a multidimensional primitive in an outer product-type dataflow. However, this would generate more VRF writes per instruction, possibly requiring additional VRF bank write ports.

To further improve the performance and efficiency of multidimensional workloads, we could also define a set of vector tile load instructions for loading multiple unit-stride vectors from a tile of a matrix. A single instruction would encompass multiple vector load instruc-

tions, reducing the dynamic instruction count This strategy may also enable more efficient scheduling and management of the load operations by the VLSU. Similar ideas have been proposed to improve the efficiency of multidimensional workloads for other vector processors [93][109].

This section has focused on dense multidimensional computation, but sparse computation is also very important, particularly in the embedded ML context. Sparsity can be leveraged to reduce the costs of data movement. In particular, structured sparsity, often in the form of 2:4 or 3:4 sparsity, can be efficiently leveraged in data-parallel architectures due to its regularity by generating index bits ahead of time that align the sparse weights with the correct dense input elements [82]. The `vmvmacc` instruction concept can be extended in a similar fashion, reusing its register read scheme and dot product functional units. Sparse weight indices could be used to select two vectors from a tile of four or more consecutive vector registers in the VRF to multiply against two packed sparse weights and accumulate into a single destination. For example, in the 2:4 sparsity case, two vectors from a tile of four vectors would be selected using two 2b indices. The selected data would then execute as a two-input dot product, much like the baseline `vmvmacc` instruction. While this would add additional hazard tracking complexity to some of the sequencing mechanisms in Saturn, it could be advantageous if the generated short-vector machine were to target embedded machine-learning workloads.

In this section, we've discussed the design and implementation of a family of instructions to accelerate multidimensional kernels in short-vector machines. The design of the `vmvmacc` instruction fits the previously discussed tenets of instruction design by being flexible, scalable, and fitting within the structure of the machine efficiently. There are several complementary instructions and variants that could be implemented to further improve efficiency in a broader range of workloads.

4.7 Accelerating Convolutions

Convolutions, and kernels that follow the neighborhood operator pattern, are ubiquitous in mobile multimedia and embedded ML workloads. High-arithmetic intensity 2D multi-layer convolutions make up the backbone of much of the computation of CNNs. In the image-processing domain, 2D convolutions and 2D separable convolutions make up a significant portion of the computation of many workloads. Linear filtering is used to smooth images and extract edges, such as with 2D spatially separable box filters, Gaussian kernels, and Sobel kernels. These operators are used as building blocks in image-processing pipelines.

For example, unsharp masking is an image sharpening technique that involves blurring the image, subtracting the blurred image from the original, scaling the difference, and adding it back to the original image [119]. Depending on the exact workload parameterization, the blurring step, typically a spatially separable kernel, may make up over 75% of the operations in the pipeline.

Convolutions are found throughout image-processing-based feature-detection pipelines. In the Harris corner detection algorithm, the image is filtered with kernels that emphasize edges to find potential corners in the image [46]. In SIFT, the scale-space generation step involves blurring the image with a set of Gaussian filters, repeating this step several times after downsampling the image [74]. SIFT, and other feature-detection pipelines, contain other processing steps that follow the neighborhood operator pattern. SIFT looks for potential keypoints by finding local extrema in neighborhoods of pixels after computing the difference-of-Gaussian from the image pyramid previously described.

Convolutions can be mapped onto short-vector machines efficiently, achieving high datapath utilizations. However, given that they often make up a large share of the computation in image-processing workloads, it is important to design strategies for improving their performance and energy efficiency.

In our discussion on vector operation fusion, we explored various strategies for handling the alignment of the vector data for convolutions and how redundant vector loads and data shuffling operations represent an inefficiency. This shortcoming of 1D SIMD machines has been identified in previous works, leading to the design of specialized convolution machines and convolution-focused ISA extensions. The Convolution Engine leveraged the convolution dataflow pattern common to many kernels in image processing to build specialized datapaths that more efficiently handle the sliding window pattern [95]. SLIDEX is another work that defines a set of instructions for accelerating sliding window kernels on P-SIMD machines, with the window width controlled by an architectural register [121].

The following sections present a family of vector instructions for extending short-vector machines to more efficiently target convolutions.

Convolution Instructions

We define the `vconv2` and `vconv4` instructions, and their floating-point counterparts `vfconv2` and `vfconv4`, as 2-wide and 4-wide vector convolution instructions, respectively. The key idea of these instructions is to more efficiently vectorize convolutions by aligning windows of data directly in the datapath instead of using multiple vector loads or slide operations.

The instruction definition for `vconv2` is:

```
vconv2.vx vd, rs1, vs2 #vd[i] = (rs1[0]*vs2[i])+(rs1[1]*vs1[i+1])+vd[i+2]
```

Figure 4.12 depicts the functionality of the `vconv2` instruction. The vector register `vs2` holds the input data for the convolution. The scalar register `rs1` holds two packed values in the least significant bits, which are used to hold the convolution kernel weights. The result of the scalar-by-vector multiplication is accumulated into the destination vector register, `vd`, that is implicitly shifted left by two elements. This is done to align the summation properly such that convolutions with kernels wider than the encoded instruction window width (e.g. 2 or 4) can still be computed efficiently by tiling across multiple instructions.

For example, a 5-wide convolution kernel can be computed by executing a normal MAC instruction for the 5th weight, followed by a `vconv4` instruction for the other four weights.

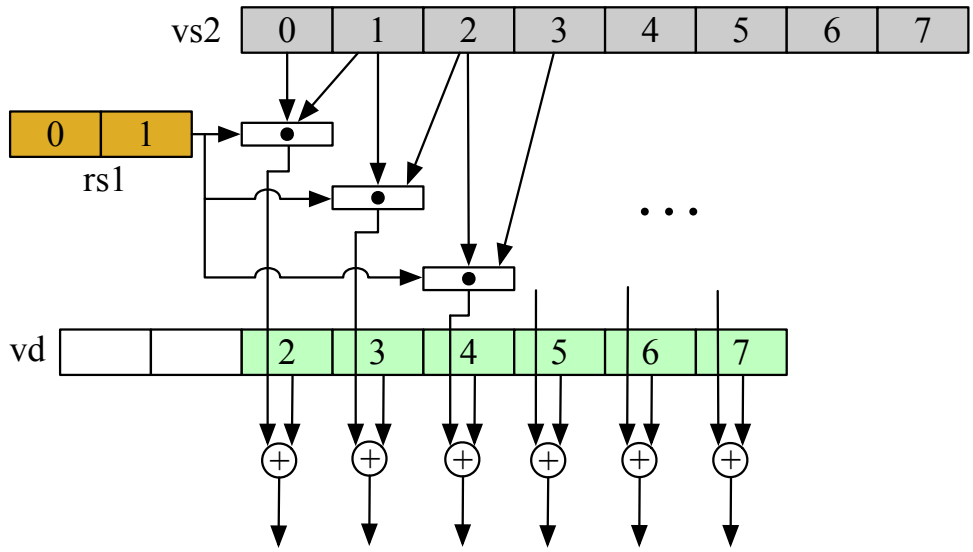


Figure 4.12: `vconv2` instruction functionality.

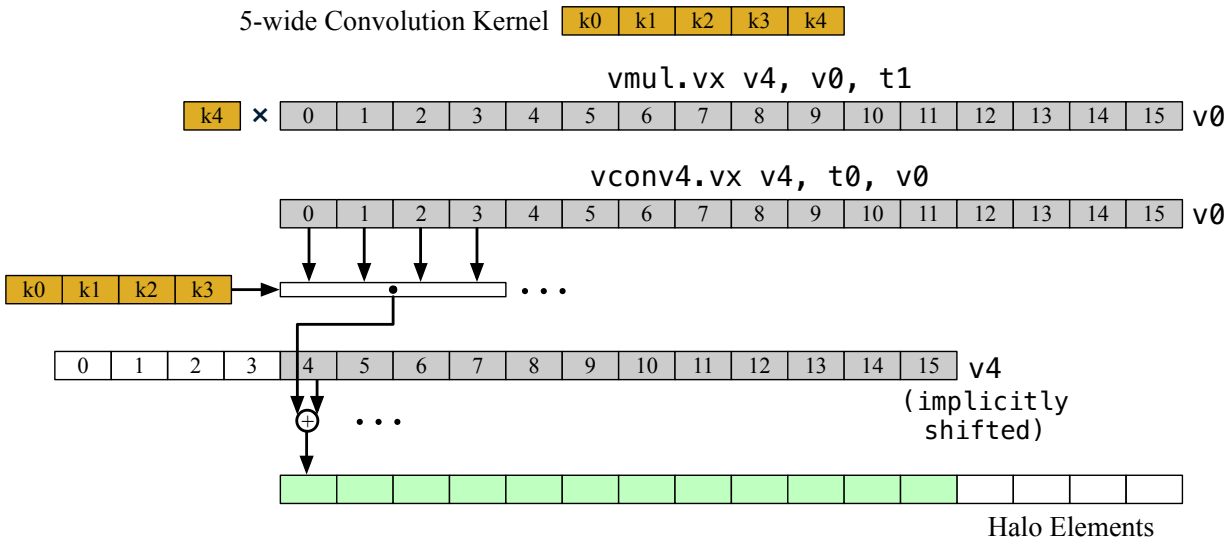


Figure 4.13: Tiled convolution using the `vconv4` instruction.

Figure 4.13 shows this operation. Similarly, it could be computed using a single MAC instruction and a sequence of two `vconv2` instructions. Thus, the convolution is executed using a single vector load and a series of these convolution instructions, rather than using multiple vector loads or slide operations to align the data. Figure 4.14 shows code fragments comparing code snippets using `vconv2` and `vconv4` to compute 3-wide and 5-wide convolutions.

```
// LMUL = 4
// Scalar bookkeeping code is omitted
// Convolution kernel of width 3
// vconv2                   // vconv4
vle16.v v0, (a0)            vle16.v v0, (a0)
vmul.vx v4, v0, t1          vconv4.vx v4, t0, v0  // v4 initialized to 0
vconv2.vx v4, t0, v0

// Convolution kernel of width 5
// vconv2                   // vconv4
vle16.v v0, (a0)            vle16.v v0, (a0)
vmul.vx v4, v0, t2          vmul.vx v4, v0, t1
vconv2.vx v4, t1, v0        vconv4.vx v4, t0, v0
vconv2.vx v4, t0, v0
```

Figure 4.14: Convolution with `vconv2` and `vconv4` instructions.

The `vconv*` instructions utilize one scalar register read and two vector register reads. Thus, it uses the same number of register sources as a normal scalar-by-vector MAC or FMA instruction and can be encoded similarly.

Using the `vconv*` instructions, as well as the baseline slide-based vectorization approach, leads to "losing" `KW-1` elements off the end of the vector, where `KW` is the width of the convolution kernel. This is because the convolution halo elements are not contained within the source vector, `vs2`. The baseline multi-load convolution vectorization does load in these extra halo elements and so a full vector of results is computed per iteration. While losing elements is a drawback, the number of lost elements is constant with the width of the kernel, so it can be amortized over longer vectors (i.e. by increasing `VLEN` at design time or by leveraging vector register grouping with `LMUL>1` at runtime). Furthermore, convolution kernel sizes are typically small in image processing when computed directly (e.g. 3, 5, 7), limiting the number of lost elements. There are other strategies that may ameliorate this issue, as discussed at this end of this section.

Convolution kernels with a width narrower than the width of the encoded instruction window can be executed in a single instruction. For example, with a kernel width of 3, the operation can be executed with a single `vconv4` instruction that packs the 3 elements into the least significant bits of `rs1` and includes zero bits where the fourth element would be such that it does not contribute to the result.

For 2D convolutions (separable and non-separable), as are common in image-processing workloads, the `vconv*` instructions can be used to efficiently execute the horizontal portions of the convolution, followed by normal MAC or FMA instructions to compute the vertical

portion across multiple vector registers. In certain cases, the `vconv*` instructions could be sequenced with the `vmvmacc` instructions described in the previous section to accelerate the vertical portion of the convolution.

By operating on groups of offset elements across each vector, these convolution instructions can increase the maximum performance of the machine for convolution operations when implemented to run at full throughput. For the `vconv2` and `vconv4` instructions, the maximum number of operations per cycle when computing convolutions is `2*(DLEN/SEW)` and `4*(DLEN/SEW)`, respectively, trimming to `2*((DLEN/SEW)-1)` and `4*((DLEN/SEW)-3)`, at the tail of processing a vector. While these throughput increases over the baseline machine will not be achieved in practice since they do not accelerate all of the computation in a 2D convolution, they still provide a significant processing advantage.

Similar to the `vmvmacc` instructions, this type of instruction comprises a family of instructions, including for both integer and floating-point input datatypes. The instruction window widths can be designed and implemented as the designer sees fit, but we focus here on `vconv2` and `vconv4` since they strike a balance between utilization and performance. Using a wider window (e.g. 8-wide) could lead to higher performance in some cases, but would likely be significantly underutilized in most cases. Widening variants of these instructions can be defined to meet the numerical requirements of a given application domain. Since the `vconv4` instruction requires four elements to be packed into a scalar register, it is limited to input data widths of 16b and 8b. The instruction could be extended to read from multiple scalar registers to support 32b inputs as well.

Implementation

The required changes to the decode and control circuitry in Saturn to support the `vconv*` instructions are minimal since they use the same type and number of register sources as a scalar-by-vector MAC or FMA instruction. The actual arithmetic units themselves are similarly straightforward to implement, particularly for the integer variants, as they are a set of 2- or 4-wide dot products that overlap across the input data. The destination halo elements at the end of the vector (i.e. the halo elements whose indices exceed the length of the vector) are replaced by zeroes when input to the arithmetic units.

The shift of the destination vector requires special handling because it computes on elements that cross a vector element group boundary. These adjacent element groups are resident in consecutive banks of the VRF. This is not wholly compatible with the baseline Saturn `PipelinedFunctionalUnit`, which requires the operations to be fire-and-forget once the vector operands are read from the VRF and sent to the execution unit. The `vconv*` instructions require reading from two banks of the VRF to get all the data for a single operation. The read from the subsequent bank is not guaranteed to occur in the very next cycle as it may be blocked by a data hazard or structural hazard. This can cause a misalignment of the data and thus incorrect results. However, we still want pipelined execution with zero dead time and without requiring additional VRF read ports.

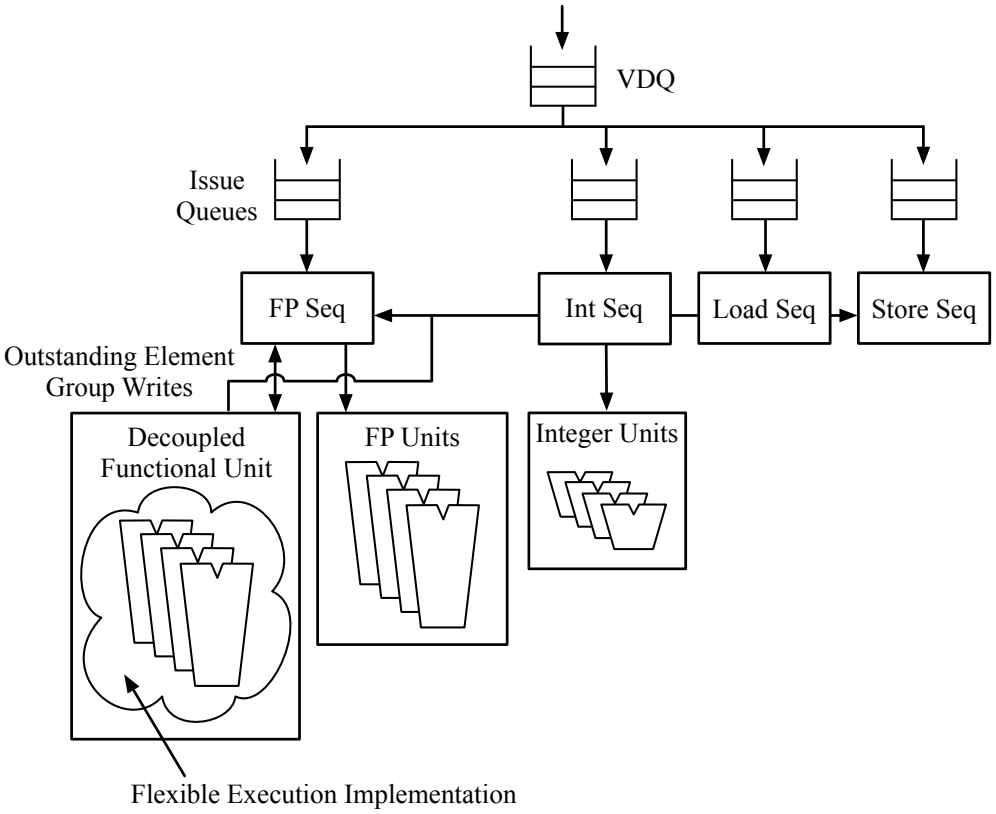


Figure 4.15: Decoupled functional unit integration in Saturn. The decoupled functional unit can be implemented in a more flexible manner than the pipelined functional units and advertises its outstanding element group writes to the sequencers for the purposes of hazard checking.

To implement this, we extend the Saturn vector unit with a new type of functional unit, called a *decoupled* functional unit. The `DecoupledFunctionalUnit` gives greater flexibility in implementation compared to the baseline functional unit types in Saturn. The module accepts beats of `DLEN`-wide execution from a sequencer across a decoupled interface. It maintains a bitmask of outstanding writes to vector elements groups that are in-flight within the functional unit. This bitmask is advertised and is used by the sequencers to detect data hazards. On writeback to the VRF, the functional unit arbitrates for access and the corresponding bit in the bitmask representing the outstanding writes is lowered. As long as this bitmask is maintained correctly, the internals of the functional unit may be implemented in any way, without being tied to the strict execution requirements of the baseline `PipelinedFunctionalUnit` or `IterativeFunctionalUnit`. Figure 4.15 depicts this high-level structure.

In our implementation of the `vconv*` instructions, the data proceeds fully pipelined down the functional unit as long as either the next beat of data will be issued on the subsequent

cycle or the end of the vector has been reached (i.e. reached `vl`). This way, the overlapping windows of elements across vector element groups can be implemented by grabbing the overlapping elements from the very next beat of vector read data. If the next beat of data is not issued on the next cycle for whatever reason, the pipeline stages within the functional unit are stalled until it is available. If all beats of the instruction are issued back-to-back, then the operation of this functional unit looks effectively the same as a `PipelinedFunctionalUnit`. With this implementation, it can still maintain pipelined operation, zero dead time, and still allows for chaining at the element group granularity. There are other ways in which this type of instruction could be implemented in Saturn, but this way introduces minimal overhead and avoids invasive changes in other parts of the machine.

This convolution instruction family can improve performance and energy efficiency in several key ways. By more naturally matching the convolution dataflow, it obviates the need to align the data explicitly using redundant vector loads or shift operations. This significantly reduces the number of VRF reads and writes required and enables more efficient usage of the VRF storage. Vectorizing with neighborhoods of elements across the input vectors also allows the machine to scale to more functional units that accelerate the convolution computation. Encoding more operations per instruction amortizes the costs of the scalar unit and vector backend sequencing. Some selected PPA results for these instructions are shown in the following section.

Further Extensions

The convolution instructions presented in this section again remain scalable and fit within the structure and programming model of our baseline short-vector machine. There are some instruction variants and complementary instructions that could enable further gains in efficiency. Recall that the proposed convolution instructions lose a fixed number of vector elements due to not computing with the halo elements. For 1D convolutions and tiled 2D convolutions, this may be ameliorated by defining the convolution instructions to compute the halo elements using the first elements of the `vs2+1` vector register, which would need to have the proper input elements resident. This would allow the convolution instructions to compute an entire vector of results together, similar to the multi-load convolution vectorization.

Since the `vconv*` instructions only use two vector register reads, this concept could be combined with the `vmvmacc` instruction to create a multi-vector convolution instruction. This instruction would read three vector registers and would execute two vertical steps of a 2D convolution at once. This would allow the machine to scale to even more compute throughput for multidimensional convolutions without requiring additional read and write ports on the VRF. However, it would be a less flexible primitive than the base convolution instructions.

While this section focused on accelerating convolution operations, other neighborhood operators that follow the same pattern of computation could also be targeted. This can include computations like comparison operations used for finding the minimum and maximum

in windows of pixels or the sum-of-absolute-differences operations common in dense image stereo correspondence workloads.

We used the `DecoupledFunctionalUnit` to implement a pseudo-pipelined functional unit to align the vector elements properly, but it has the potential for many more uses. Because it enables a less restricted implementation of a vector functional unit in Saturn, it can be used to extend a baseline short-vector machine with highly specialized and complex functional units that may offload a significant amount of application-specific compute. It could be used to interface to one or more accelerators, which may be required to meet the performance and energy efficiency demands in a particular DSP core deployment. The Saturn instance would provide a baseline of general-purpose vector compute, while sequencing data to the attached datapath and maintaining hazard controls with the outstanding write bitmask from the `DecoupledFunctionalUnit`.

4.8 Specialized Vector Instruction Results

To evaluate the proposed specialized vector instructions, we first implement them in Spike[2]. We then write kernels against them to ensure their correctness. We then implement the hardware support for the instructions in RTL using Chisel in the Saturn vector unit generator, as described in the previous sections. We can then co-simulate the Spike functional simulation with a cycle-accurate RTL simulation of the extended Saturn instance with the new instructions, using Synopsys VCS, to check the RTL functionality. From the RTL simulations, we gather performance results. These results are gathered using a similar SoC harness to that described in Chapter 3. Saturn is integrated with a dual-issue in-order core and the L2 cache is warmed up such that the data is resident. The PPA results come from a post-synthesis evaluation of the RTL implemented in the Intel16 process technology using the Cadence suite of EDA tools.

Figure 4.16 shows the performance of several variants of Saturn running a 32b integer matrix multiply kernel, sweeping the matrix dimensions. The operating clock frequency is 1 GHz. The machine instances are all identical aside from their `VLEN`, `DLEN`, sequencer configurations, and the implementation of the `vmvmacc` instruction. The baseline machine is the `V256D128` instance. The `V256D128+2xMAC` configuration implements a second MAC unit sequencer, as described in the Vector Backend Sequencer Configuration section, and the `V512D256` configuration doubles the architectural vector length and the datapath width of the baseline. The `V256D128+vmvmacc` configuration extends the baseline machine with an implementation of the `vmvmacc` instruction.

The machine implementing `vmvmacc` shows similar or superior performance results for this kernel compared to doubling the width of the machine or adding an entire additional sequencer and execution unit, particularly at small problem sizes. The other configurations may be able to increase their performance at the smaller matrix sizes through more extensive, problem-specific static scheduling at the expense of code size. The `V512D256` does not provide

[2]https://github.com/riscv-software-src/riscv-isa-sim

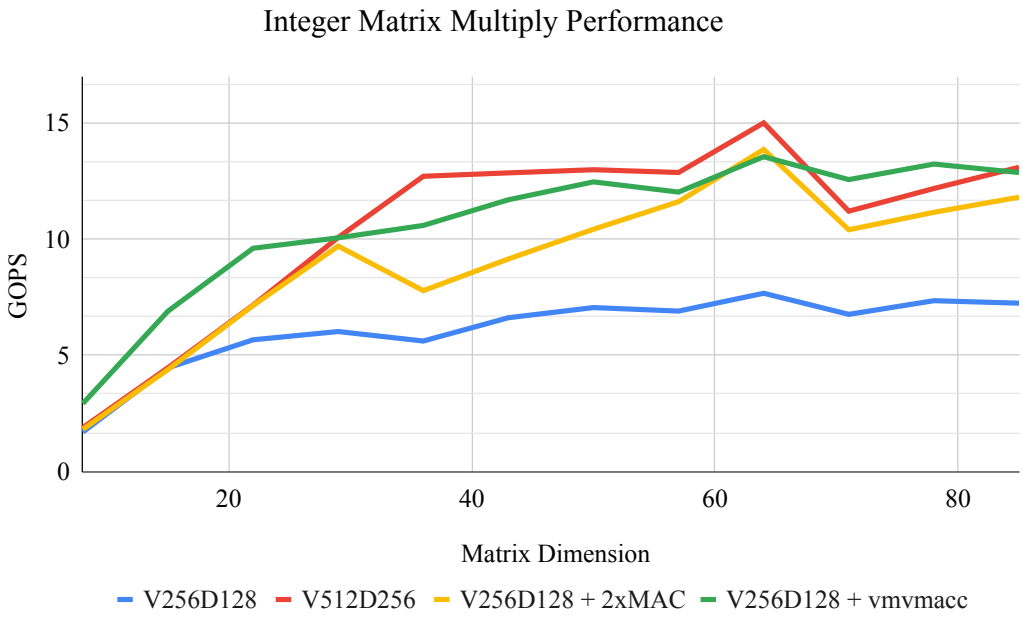


Figure 4.16: Comparison of matrix multiplication performance of Saturn instances, including the implementation of the `vmvmacc` instruction.

a performance improvement in this regime since the application vector lengths are too short to take advantage of the additional resources. The `vmvmacc` instruction is able to leverage the short-vector machine sequencing and provide increased throughput.

Figure 4.17 shows the normalized silicon area evaluation for the same Saturn instances. All of the design points are normalized to the baseline `V256D128` configuration. The area cost of the `vmvmacc` instruction represents only a modest increase over the baseline, less than the multi-sequencer configuration and far less than doubling the vector length and datapath width. The area overhead largely is just a result of the additional cost of the functional units, since this instruction type was designed to enable increased performance while reusing as much of the baseline machine as possible.

Figure 4.18 shows the normalized energy efficiency (`GOPS/W`) results for the same Saturn instances. Extending the baseline with the `vmvmacc` instruction results in an energy efficiency uplift of over 60%, significantly larger than those of the other evaluated instances. The `vmvmacc` instruction is able to significantly reduce the number of total VRF reads and writes and reduces the dynamic instruction count of the kernels by almost 50%. These PPA results suggest that the `vmvmacc` instruction is effective for improving machine performance and energy efficiency for matrix multiplication-based workloads in an area-efficient manner.

To evaluate the specialized convolution vector instructions, we target a 5x5 2D convolution kernel with 16b integer elements operating on an image large enough to avoid

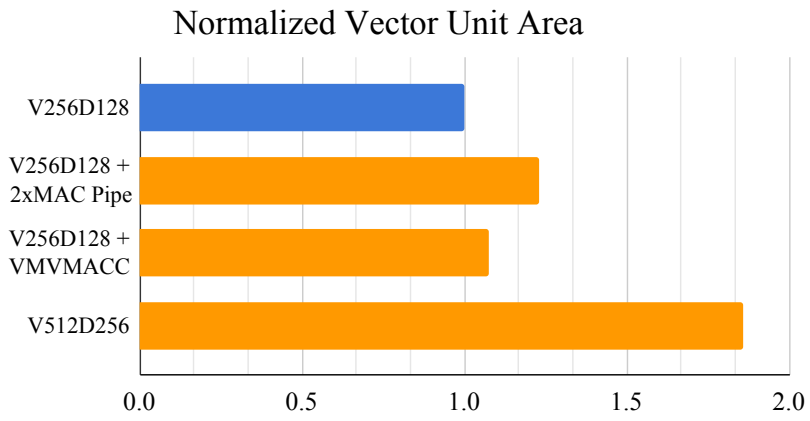


Figure 4.17: Normalized area results for the `vmvmacc` instruction implementation in the Intel16 process technology.

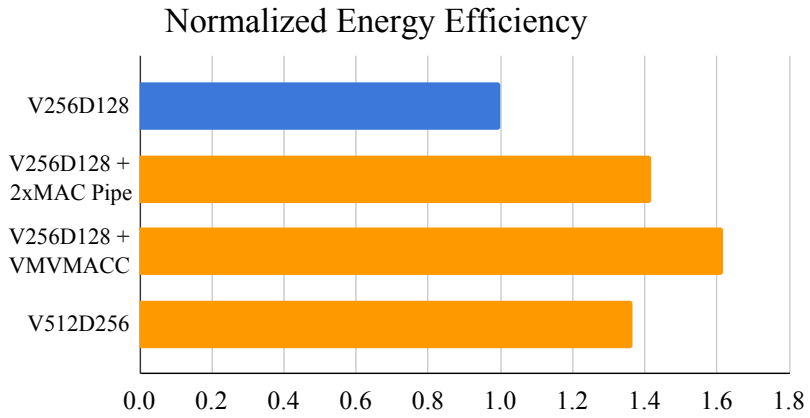


Figure 4.18: Normalized energy efficiency results for executing a 64x64 matrix multiplication with the `vmvmacc` instruction.

significant performance differences that may arise from computing on short application vectors. Figure 4.19 shows the kernel speedup results of several machine configurations over a baseline `V256D128` machine. The `V512D256` configuration doubles the baseline `VLEN` and `DLEN`. The configuration with `vmaccslide` implements a convolution instruction variant that fuses a MAC instruction with a single element shift, similar to the operation fusion strategy discussed earlier in this chapter. The `vconv2` and `vconv4` configurations implement the corresponding instructions described in Section 4.7.

The `V512D256` provides a roughly 2x performance increase over the baseline. This is

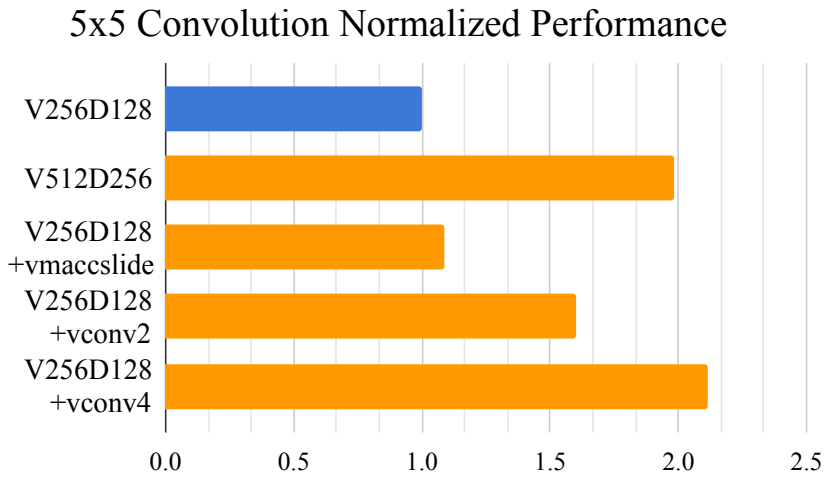


Figure 4.19: Specialized convolution instruction normalized performance comparison for a 5x5 2D convolution.

expected given that this kernel can saturate the functional units. The `vconv2` implementation provides a roughly 1.7x improvement over the baseline. The `vconv4` implementation outperforms even the `V512D256` configuration. It also leads to a roughly 40% reduction in dynamic instruction count compared to the baseline.

Figure 4.20 shows the normalized energy efficiency results for a 5x5 2D convolution for the same machine variants. Again, the `vconv4` variant, as well as the other convolution instruction variants, exhibit significant gains in energy efficiency by more naturally capturing the convolution dataflow and reducing the amount of data shuffling required. The number of VRF reads and writes is significantly reduced and the costs of instruction fetch and sequencing are amortized by encoding more operations per instruction.

Application Evaluation

To understand the impact of the convolution instructions in a representative multimedia application, we vectorize an image sharpening pipeline based on unsharp masking [119]. The unsharp masking computation we implement is summarized as follows, where g is the output image, f is the input image, h is a spatially separable filter convolved with the input image, and γ is a scaling factor:

$$g = f + \gamma(f - (h \star f)) \tag{4.1}$$

The unsharp mask pipeline consists of a separable convolution, followed by a series of element-wise vector operations. We produce three versions of the pipeline: a baseline, one that utilizes the `vconv2` instruction, and one that utilizes the `vconv4` instruction. Note

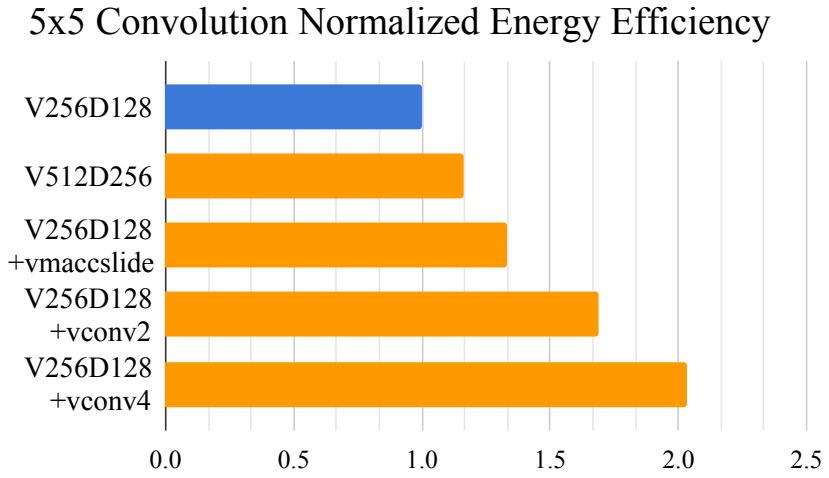


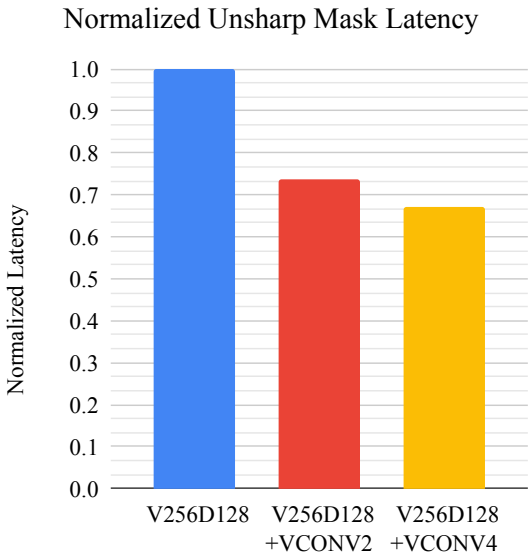
Figure 4.20: Specialized convolution instruction normalized energy efficiency results for a 5x5 2D convolution.

that separable convolutions are less computationally intensive than the 2D convolutions evaluated in Figures 4.19 and 4.20. Due to this, the convolution instructions accelerate a smaller portion of the overall computation but still remain effective.

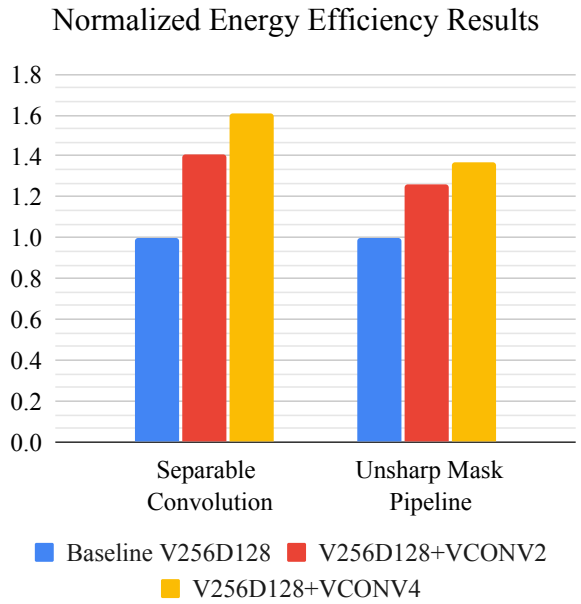
Figure 4.21a shows the latency of the pipeline, with and without the convolution instructions, normalized against the performance of the baseline `V256D128` Saturn configuration. Adding the `vconv2` and `vconv4` instructions results in a reduction in pipeline latency by 26% and 33%, respectively. These results are effectively an exercise in evaluating Amdahl's Law. As more element-wise vector operations are added to the pipeline, the overall performance impact of the convolution instructions is lessened. In this case, the convolution instructions are able to deliver a speedup of up to 1.5x over the baseline.

Figure 4.21b shows the energy efficiency results for the same pipelines, again normalized to the baseline `V256D128` Saturn configuration. We compare the energy efficiency results for the full unsharp mask pipeline, as well as just the separable convolution portion of the pipeline, side by side. As expected, the energy efficiency gains from the convolution instructions are higher when evaluating the separable convolution alone. The difference in energy efficiency gains between adding the `vconv4` and the `vconv2` instructions also decreases.

Figure 4.21b demonstrates the impact of the makeup of the target application on the energy efficiency improvements from the proposed convolution instructions. In this case, implementing the `vconv4` instruction results in a more energy efficient design for both the separable convolution kernel and the unsharp mask pipeline. As the proportion of computation spent in convolution decreases, there will be a point in which `vconv4` provides little or no benefit over `vconv2`, no longer justifying the additional area cost. In turn, at a certain point, `vconv2` will provide little benefit over the baseline, making the baseline the best de-



(a) Normalized latency of the unsharp mask pipeline.



(b) Normalized energy efficiency comparison.

Figure 4.21: Performance and energy efficiency of the unsharp mask pipeline for Saturn variants with and without the specialized convolution instructions implemented.

sign point. Figure 4.21b also shows the value in designing instructions with several variants: architects can select which to implement depending on the PPA constraints of a particular short-vector machine design and the computational composition of the target domain.

This type of analysis can be applied to other proposed specialized instructions for extending short-vector machines. Based on these results, and due to the general importance of the convolution operation, the instructions proposed here can provide a performance and energy efficiency benefit across a broad range of short-vector machine deployments in mobile SoCs.

4.9 Summary

In this chapter, we explored design techniques for specializing short-vector machines, ranging from microarchitectural techniques to the design of new vector instructions. By leveraging domain-specific computational patterns and the structure of short-vector machines, we can significantly improve the energy efficiency and performance of key workloads in a flexible, scalable, and cost-effective manner. The PPA results of the specialized instruction implementations demonstrate that they are effective mechanisms for enabling new short-vector

DSP core design points, gaining comparable or superior performance and efficiency compared to simply doubling the vector length and datapath width, at modest area costs. The design techniques presented here can be applied to other domains.

To deploy efficient specialized short-vector machines, we must also consider how they are integrated into a larger SoC context and how they can be physically designed. The following chapter discusses a reusable physical design framework and a silicon implementation of an eight-core vector processor SoC.

Chapter 5

Silicon Implementation and Design Methodologies

We have so far discussed the design of short-vector machines and techniques for their specialization. We now turn to techniques for their evaluation and implementation. When designing specialized processing systems, rapid and accurate physical evaluation is critical for design space exploration (DSE) and optimizing silicon implementations. This capability is even more important in the context of modern silicon design, in which chip design costs are extremely high and domain specialization plays a vital part in achieving higher performance and energy efficiency.

Therefore, it is valuable to study ways in which physical design (PD) productivity can be increased so as to enable chip architects to create better chips more quickly. Furthermore, better PD flows and techniques may lower the barrier to entry into chip design, aiding efforts in improving silicon design education and empowering small teams of engineers to build more complex systems.

Previous work has already identified some key strategies for enabling more agile chip design flows. An agile hardware design approach has been proposed to enable small teams to implement complex, efficient microprocessors [71]. The proposed approach emphasizes the importance of taping-in complete designs and incrementally adopting features, underlain by the use of generators in the design process. Chipyard [1] has been maintained as a complete framework for full stack SoC design, evaluation, and implementation by leveraging RTL generators, RTL simulation and emulation infrastructure.

This chapter presents contributions to Hammer[1], a framework for generating PD flows, with the aim of enabling a more agile chip design process and maximizing the reuse of design effort. In particular, this chapter will discuss Hammer's PPA evaluation capabilities, which have been critical for optimizing the Saturn designs presented earlier, as well as other designs. This chapter will then present the Cygnus chiplet, a 16mm^2 octa-core RISC-V vector processor for DSP applications, taped out in the Intel16 process technology using

[1]Hammer has evolved significantly since its original publications [126][129].

Hammer[59]. Cygnus is the first silicon demonstration of the Saturn vector unit and achieves over 90% datapath utilization on matrix multiplication and convolution kernels and best-in-class energy efficiencies of 414 GOPS/W and 109 GFLOPS/W on INT8 and FP32 matrix multiplication kernels, respectively.

Section 5.1 discusses the design tenets and key details of the Hammer framework. Section 5.2 presents an evaluation of Hammer, including data from tapeouts that have used Hammer. Section 5.3 discusses the use of Hammer in the optimization and implementation of the Saturn vector unit, including improvements to its flows for simulation and power analysis. Section 5.4 presents the Cygnus DSP chiplet, which includes eight instances of the Saturn vector unit across two design points.

5.1 Hammer: a Reusable VLSI Flow Generator

Hammer is an open-source physical design flow generator that aims to maximize design reuse and enable agile physical design flows [126]. It is not itself a CAD tool, but rather wraps around existing CAD tools, both proprietary and open-source. It is written in Python and generates Tcl scripts to drive the underlying CAD tools, while managing all the necessary collateral passed between steps of the flow. Hammer is designed to be modular and emphasizes encoding designer intent to improve PD productivity and flow reuse. A key aspect of Hammer is abstraction: the idiosyncrasies of a particular CAD tool, from a given vendor, or process technology is abstracted as much as possible. For each step of a PD flow, a generic interface is exposed to maintain this modularity, aiming to decouple them from design-, tool-, and technology-specific information. These sections give an overview of the structure and use of Hammer, along with an evaluation of its efficacy [72].

The following are the key design principles of Hammer:

- **Separation of Concerns**: Hammer aims to separate the construction of the flow from the details specific to any particular process technology, tool, or design. This enables composability and the generation of process technology-, tool-, and design-specific flow steps from a common interface.
- **Standardization**: Hammer defines a standardized schema for flow constraints and design collateral. This facilitates the interchange of data between flow steps. Importantly, this also helps codify designer intent, enabling PD design effort to be reused across different tapeouts. It also enables other parties to understand the reasoning behind how a particular aspect of the flow was previously designed and utilized.
- **Modularity**: Hammer's abstractions and codified schema enable different tools and process technologies to be plugged in for the same design. It also facilitates the sharing of effort across different flows, enhancing PD flow reuse.
- **Incremental Adoption**: Hammer provides an interface for designers to integrate custom solutions with a baseline generated flow seamlessly. This allows Hammer-

generated flows to remain flexible and allows new features to be used and tested before being integrated back into the main framework in a reusable manner.

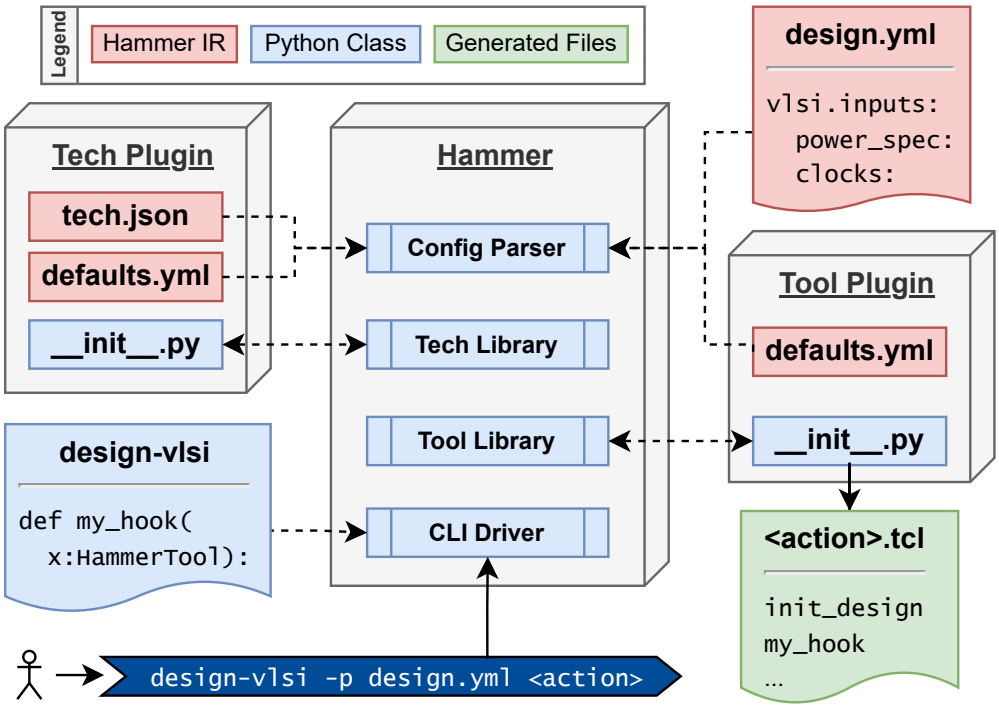


Figure 5.1: Hammer framework block diagram [72].

These design principles have driven the development of Hammer and resulted in the framework pictured in Figure 5.1. The key features of Hammer itself and its inputs are outlined as follows:

- **Core Hammer** comprises the baseline Hammer framework, including the driver, which generates a flow by combining information from the user-specified input design, tool plugins, and technology plugin. The Tcl to drive the CAD tools and configure them is generated. A Make fragment is also generated to help the designer run the flow easily from the command line. Core Hammer includes all of the infrastructure for parsing and checking inputs and defines the baseline flow that tool- and technology-specific plugins can draw from.

- **Hammer IR** is the standardized definition of the Hammer schema. Hammer IR is serialized as YAML or JSON. Hammer IR files are used as inputs to the top-level flow and for passing settings and collateral between flow steps. While the top-level flow is generated at the start, additional Hammer IR keys can be included to augment

or override existing configuration settings. Hammer also supports some lightweight metaprogramming for its IR, which allows some Hammer IR to modify other pieces of IR. This functionality can be used to manage paths to collateral, concatenate the contents of the same key specified in different locations together, and more.

- **Technology plugins** contain all of the information specific to a particular process technology. The technology plugin defines references to PDK collateral (e.g. Verilog models, LIB, etc.) and includes some information about the technology stackup, which can be used in higher-level Hammer methods. Technology plugins also include custom configuration keys and methods that define PDK-specific behavior. Typically, as a technology plugin in Hammer is brought up and matures, more technology specific functionality and flow fixes are integrated, improving flow reuse and reducing designer effort in subsequent tapeouts.
- **Tool plugins** contain all the information specific to a particular CAD tool. This includes all of the steps to drive the tool along with any custom configuration settings that may be required. Tcl templates are defined in the tool plugins, which are completed based on flow configuration setting when the flow is generated. By separating all of the tool-specific concerns into a plugin, different tool plugins can be used with the same flow with just the change of a top-level configuration parameter.
- **Hammer hooks** are the escape that allows designers to customize their flow beyond what baseline Hammer supports. Hooks are methods that can be used to inject custom Tcl into the flow. The designer can specify the hook be injected before, after, or in place of any existing flow step. Furthermore, hooks can be used to remove a step from the flow if it is not necessary for whatever reason. As previously noted, this interface allows for the incremental adoption of new features before they are generalized and upstreamed into core Hammer.
- **Design inputs** are the design-specific settings in the form of Hammer IR. This includes references to the actual design collateral, as well as the flow settings themselves, which may differ from the default settings defined in core Hammer.

When beginning to use Hammer for a particular evaluation or tapeout, the user will typically first set the initial input design keys, select the particular tool and technology plugins to use, and modify the default flow steps as needed. Hammer can then be invoked from the command line, either directly or by using the Make snippet that Hammer generates. The user can launch an entire flow at once using this Make snippet, which will automatically run all the required underlying Hammer actions. Alternatively, the user can specify a particular flow step to stop at or start from depending on what is needed and which steps have been previously run. The default Hammer flow is flat, executing a linear pass through synthesis and PnR for the entire design. However, this becomes untenable as tool runtimes explode as the size of the design grows. Thus, Hammer includes support for a bottom-up hierarchical flow, as will be discussed in Section 5.5.

Hammer is integrated into the Chipyard framework [1] and serves as the driver for the VLSI backend during full-stack SoC development. A wide variety of Hammer tool plugins have been developed, including for both proprietary and open-source backends. Similarly, Hammer technology plugins have targeted both closed- and open-source process technologies, and include both planar and FinFET processes.

5.2 Hammer Evaluation

While the efficacy of a framework such as Hammer over a baseline flow or competing framework may be difficult to measure directly, we can present both empirical evidence and several quantitative metrics. Hammer has been used as the DSE and evaluation framework for many projects. One empirical example is with the Gemmini project [41]. Gemmini is a systolic array generator targeting deep learning applications implemented in Chisel. During Gemmini's development, Hammer was used to rapidly evaluate various design points and experiment with different floorplans and other physical parameters. PnR results showed that certain Gemmini configurations were untenable in terms of physical realizability that more cursory physical evaluations would not have revealed. These results provided valuable insights, highlighting the importance of a tool like Hammer for enabling architects to get rapid and realistic PPA feedback for new designs. Hammer was used throughout the development of the Saturn vector unit to similar effect, as is discussed in the following section.

Quantitatively, we can estimate the efficacy of Hammer through proxy-metrics such as the percentage of unique lines of code between the Hammer flows used in different tapeouts and the design effort for a tapeout, measured in person-months of work. The data presented here is gathered from chips with a wide diversity of designs and target process technologies. Figure 5.2 shows the number of unique lines of code used in tapeouts across time as Hammer matured and plugins were developed.

The first three chips featured in Figure 5.2 were taped out in the same 16nm process, with Eagle being taped out in the year before the two subsequent chips. We can see that thousands of lines of code were added between these tapeout with roughly a quarter of all lines of code being unique. These tapeouts were completed during a time of tremendous development in Hammer itself and its plugins. The following chips were then able to get significant reuse out of the existing Hammer framework and contribute to it such that the downward trend of unique lines of code continued. These chips were taped out in process nodes spanning from 12nm to 130nm, by designers with varying levels of tapeout experience. The last few chips on the chart required only a small percentage of unique, design-specific code to build their flows, pointing to Hammer's efficacy for facilitating design reuse across different tapeouts.

Table 5.1 provides more specifications of the chips from Figure 5.2 and includes statistics about the person-months of work required to tape out each chip. Again, we can see a downward trend in overall design time and effort across the tapeouts over time as Hammer matured and gained richer functionality. The most recent tapeouts that have used Hammer

Table 5.1: Comparison of chips using Hammer [72].

	Eagle	**Eagle-X**	**GPS**	**Hydra Spine**	**ARGO**	**OsciBear**	**HD Binary Core**
Description	9-core RISC-V SoC	22-core RISC-V SoC	GPS receiver SoC	MU-MIMO baseband SoC	RISC-V SoC for ML	Bluetooth SoC	HDC processor
Foundry node	16nm	16nm	16nm	22nm	12nm	28nm, SKY130	28nm
Die Area (mm^2)	24	56.3	24	32	16	0.33, 7.6	4
Signoff frequency	1.05 GHz	1.05 GHz	500 MHz	2 GHz	1.1 GHz	50 MHz	—
Hierarchy levels	3	3	1	3	2	1	1
Unique LoC	32.9	38.7	27.1	11.1	8.6	0.8, 0.7	3.4
Person-months	22	10	6	5	4	8, 1	8

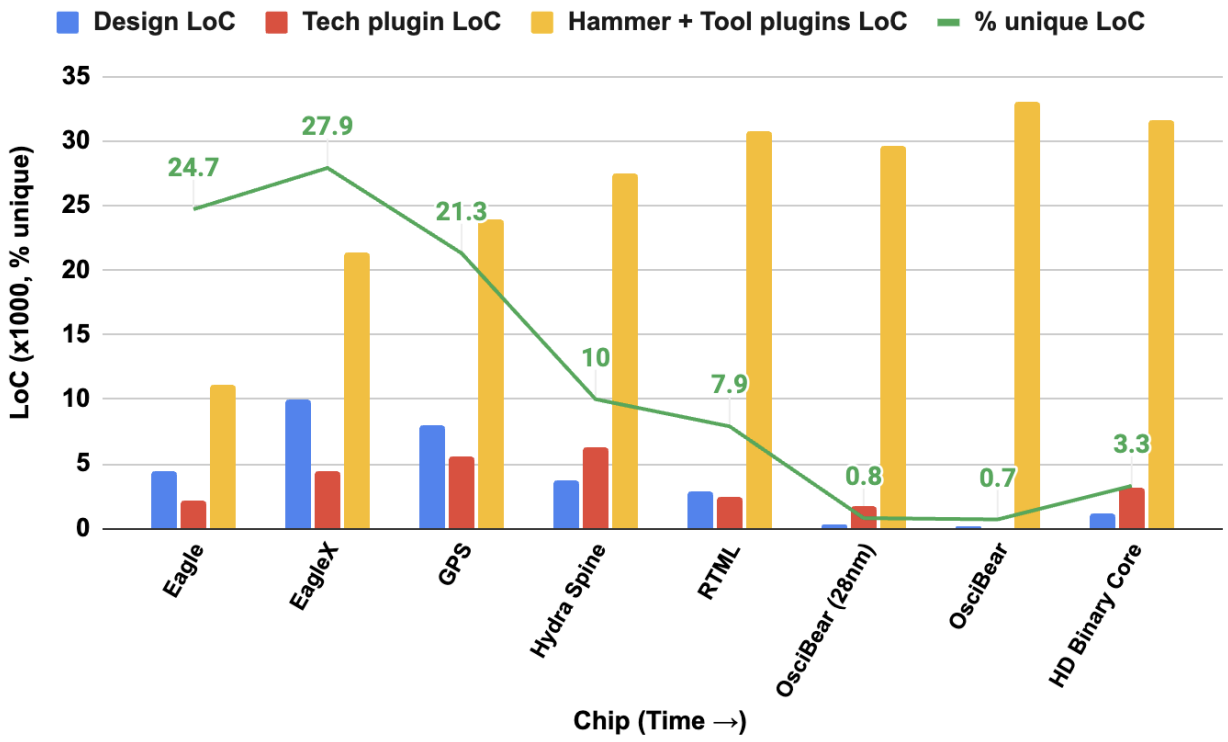


Figure 5.2: Hammer evaluation of design reuse across various chips [72].

have largely been able to avoid heavy use of design-specific Hammer hooks. This is in contrast to the earliest chips featured in Figure 5.2 and Table 5.1, which had to make heavy use of flow customizations before upstreaming functionality back into Hammer.

Related Work

Reference flows provided by CAD tool vendors and foundries provide a starting point for working with a given CAD tool or silicon process technology. These flows typically consist of Tcl script templates which can then be customized to a particular design. While reference flows provide critical tool- and technology-specific information when starting out, they are often difficult to use and incomplete for what is needed for an entire tapeout. Since a given reference flow is inherently tied to a particular process technology and tool flow, it is difficult to re-use work across different tapeouts because there is limited modularity and no separation of concerns.

In contrast, Hammer enforces modularity to improve design reuse, while still allowing for design-specific flow customization through the use of hooks. Empirically, using reference flows in a non-structured manner leads to the introduction of many "magic numbers" and bespoke flow options that are difficult to decipher or use in subsequent flows without a

centralized, structured framework. Hammer and its plugins serve as a repository for design-abstracted flow parameters and knowledge by encoding designer intent. Reference flows are often used in conjunction with Hammer when starting out with a new tool or technology, but are quickly left behind as Hammer's support for that target matures.

SiliconCompiler [88] and mflowgen [24] are the most comparable open-source flow tools to Hammer. These flow tools have strengths in their flow graph management and PPA metrics extraction capabilities. However, they do not codify design inputs as clearly as Hammer IR and do not have features analogous to Hammer IR's metaprogramming or Hammer's hooks.

5.3 Saturn Implementation and Evaluation

Hammer was a critical tool during the development of the Saturn vector unit by enabling the rapid gathering of PPA feedback when varying design parameters. As the design evolved and began to take shape, Hammer was used to push Saturn instances through the implementation flow. Post-synthesis and post-PnR timing results were fed back into the RTL design to fix critical paths, including informing the pipelining of control paths and the default settings for vector functional unit pipeline depths.

As an example of this physical feedback process, early on in Saturn's design, we found that the critical path ran through the `DLEN`-wide vector integer adder. This functional unit must support input data widths of 8b, 16b, 32b, and 64b, ideally reusing hardware to save area. The integer adder unit was originally implemented as a series of byte-wide adders that could be chained together to handle the wider input data. This datapath was reimplemented to consist of a single wide adder with an additional bit inserted between each byte group to propagate or kill carries depending on the data width setting, a technique commonly used in vector arithmetic unit implementations. This optimization allows the synthesis tool to better optimize the arithmetic circuits, reducing the delay and making this circuit no longer on the critical path.

Silicon area results gathered using Hammer informed the optimization of certain modules to reuse hardware more efficiently. Parameters were built into Saturn that allow the user to trade off performance for area on vector instructions that may be less critical in a particular deployment. Physical implementation results also demonstrated the feasibility of some key design features in Saturn. As discussed in Chapter 3, the sequencing mechanism in the Saturn vector backend requires the sequencer blocks to resolve hazards based on the vector operations in other sequencers, issue queues, and functional units. Physical design results demonstrated that these scheduling mechanisms in typical Saturn configurations had no impact on the critical path in the design and added minimal hardware overhead.

Earlier on in Hammer's development, the primary focus of the framework was on implementation flows, namely synthesis and place-and-route (PnR). As the use of Hammer evolved to also support DSE more efficiently, the need for integrating richer simulation and power analysis capabilities became apparent. Support for RTL and gate-level simulation, primarily with Synopsys VCS, and support for power analysis, primarily with Cadence Vol-

tus and later Joules, was integrated into Hammer. These capabilities were leveraged in the development and evaluation process for Saturn and the proposed instructions described in Chapter 4.

Hammer currently supports driving RTL and gate-level simulations, and has plugins for Synopsys VCS and Cadence Xcelium. Figure 5.3 shows a snippet of Hammer's input keys for simulation. The user can define simulation options through this interface. While there seem to be many Hammer keys to configure the simulation flow, most of them either do not typically change much between designs or are automatically generated by a particular Hammer step. For example, lightweight scripts in Chipyard generate effectively all of these input keys automatically for Hammer based on the design itself and a set of statically encoded simulation options that need not change much over time. In addition to the user-defined input files, Hammer automatically includes Verilog files from the PDK defined in the technology plugin, such as functional models for SRAMs, by using the Hammer IR "append" meta-directive.

Model files for any extra design libraries defined by the user are also included, such as for mixed signal blocks. The `sim.inputs.level` controls if the simulation should be run with RTL or gate-level options, while Hammer facilitates piping the outputs from the desired flow step (e.g. RTL design, post-synthesis, post-PnR) to the simulation. Initially, flags were simply included to dump waveforms from the testbench, but the plugin was later extended to be capable of generating Switching Activity Interchange Format (SAIF) switching activity files based on start and stop triggers for use in power analysis flows[2]. Exposing this thorough API means that flexibility is granted for users who may need to significantly modify the simulation flow, though most of the options can be configured once and are handled automatically.

Cadence Voltus was the first power analysis tool supported in Hammer with a plugin. Voltus is integrated into Hammer to provide post-place-and-route power analysis capabilities. As DSE and energy efficiency analysis became more critical use-cases for Hammer, it was clear that support for a faster turn-around-time (TaT) power flow was needed. Support for Cadence Joules was added, a tool capable of estimating power from an RTL or gate-level design. Joules can provide active power analysis using waveforms or switching activities dumped from simulation and can estimate the impact of post-PnR non-idealities arising from routing parasitics and the clock tree. With the integration of Joules, the Hammer power analysis API was extended to have richer functionality for specifying and generating power reports. Since, the Joules plugin has been further improved with additional functionality and ease-of-use[3].

Figure 5.4 shows a portion Hammer's input keys for specifying power analysis parameters and driving power analysis tools. These keys define a standard interface for specifying the design to analyze, including the input files or design database. Hammer automatically manages piping design collateral from the output of various implementation flow steps to

[2]Credit: Colin Schmidt

[3]Credit: Nayiri Krzysztofowicz

```
sim.inputs:
  top_module: null # Top RTL module for simulation
  input_files: [] # Input files for simulation including verilog sources,
  ↪ testharness/testbench, etc.
  options: [] # Permissive options for the simulator generation
  defines: [] # Define options passed when generating the simulation executable
  compiler_cc_opts: [] # C compiler options used when generating the simulation
  ↪ executable
  compiler_ld_opts: [] # C linker options used when generating the simulation
  ↪ executable
  timescale: null # Simulation timescale
  benchmarks: [] # Benchmarks to run with the simulator executable
  parallel_runs: 1 # Maximum number of simulations to run in parallel.
  tb_dut: "dut" # Hierarchical path to the top level instance of the "dut" from
  ↪ the testbench.
  tb_name: "TestDriver" # Name of the simulation testbench/test driver module
  level: "rtl" # Simulation mode for simulation at different levels of the VLSI
  ↪ flow (rtl, syn, par)
  all_regs: "" # Path to a list of all registers in the design passed from
  ↪ synthesis or place-and-route
  seq_cells: "" # Path to a list of all sequential standard cells in the design
  ↪ passed from synthesis or place-and-route
  sdf_file: null # SDF File
  gl_register_force_value: 0 # Simulation force value used for register
  ↪ initialization for gate level sims
  timing_annotated: false # Turns time annotated sim on or off
  execution_flags: [] # Flags to run with the simulator
  execution_flags_prepend: [] # Flags that are to be pre-pended to the simulator
  ↪ execution flags in "execution_flags"
  execution_flags_append: [] # Flags that are to be appended to the simulator
  ↪ execution flags in "execution_flags"
  execute_sim: true # Determines whether or not the simulation executable that is
  ↪ generated with the above inputs will actually be executed with the given
  ↪ flags or if the executable will just be generated in the specified run
  ↪ directory.
```

Figure 5.3: Hammer simulation input keys.

the power analysis flow steps. The step of the flow to run power analysis from, such as on the input RTL or post-synthesis, is controlled by the `power.inputs.level` key, similar to the Hammer simulation input keys. The `waveforms` and `saifs` keys allow a user to run active power analysis using the switching activity from a simulation, including over a particular time window. The `extra_corners_only` key is meant to give the user more control over which corner the analysis is done. It would typically be used to analyze just the typical-typical corner with nominal supply voltage and temperature, rather than at a setup or hold corner. The `report_configs` key defines a fine-grained interface for specifying a set of power reports for the design. This includes defining which input waveforms to use, setting the time window over which to run the power analysis, which module to report power for, and more.

While the faster TaT of tools like Joules trade off some accuracy in comparison to post-PnR analysis, they are particularly useful in the context of evaluating the results of domain-specialization. In this context, the relative result between two different designs is often more informative than the absolute accuracy of either result.

All together, these extensions of Hammer's simulation and power analysis functionality provide the ability for designers to rapidly evaluate design PPA from the same framework that will be used for an eventual silicon implementation. This leads us to the tapeout of Cygnus, the first large-scale tapeout of the Saturn vector unit and the first demonstrator of Saturn working in silicon, all designed using Hammer.

5.4 Cygnus DSP Chiplet

Cygnus is a 16mm^2 octa-core RISC-V vector processor SoC taped out in the Intel16 process technology. Cygnus aims to provide efficient computation for DSP and image-processing workloads, such as those commonly found in mobile multimedia applications. To achieve this, Cygnus integrates eight short-vector machines in the form of instances of the Saturn vector unit, implementing the RVV 1.0 specification. These short-vector machines are designed to be flexible and efficient, capable of achieving high datapath utilizations on a wide variety of kernels with both short and long application vector lengths.

Cygnus makes use of a "big/little"-type architecture, integrating four "big" Saturn instances, each with a V512D256 configuration, and four "little" Saturn instances, each with a V128D64 configuration. This architecture can efficiently compute a broad range of mobile multimedia workloads by mapping them onto the SoC according to their computational characteristics. For instance, highly parallel, homogeneous kernels can be mapped in a completely data parallel fashion with rate-balanced vector code executing concurrently on both the big and little vector machines. Multimedia application pipelines that are made up of a series of heterogeneous tasks of varying operations and application vector lengths can be mapped in a pipeline parallel fashion across the big and little vector machines to maximize energy efficiency.

Each of the vector core tiles in Cygnus contains a tightly coupled memory (TCM), which serves as a software-managed scratchpad. These TCMs enable high-bandwidth, efficient local

```
power.inputs:
  top_module: "ChipTop" # Top RTL module for power analysis
  database: null # The path to a synthesis or place-and-route database of the
  ↪  design to be analyzed
  tb_name: "TestDriver" # The name of the test bench from the simulation.
  tb_dut: "dut" # The name of the device under test (DUT) from the simulation.
  spefs: [] # The paths of all spef (extraction) files needed for the design.
  waveforms: [] # A list of paths to waveforms to be used for dynamic power
  ↪  analysis.
  saifs: [] # A list of paths to SAIF activity files to be used for dynamic power
  ↪  analysis.
  extra_corners_only: false # If overridden to ``true``, the power tool will
  ↪  report for only the extra MMMC corners, saving runtime.
  input_files: [] # A list of the paths to the design inputs files (HDL or
  ↪  netlist) for power analysis
  sdc: "" # SDC input file
  report_configs: [] # Optional: Power report configs that are run in addition to
  ↪  the average power reports in power.inputs.waveforms
  # examples:
  #   report_configs: [{waveform_path: "/path/to/fsdb", module: "chiptop",
  ↪  levels:3, start_time: "0ns", end_time: "1ns",
  ↪  toggle_signal:"/ChipTop/clock", num_toggles:1, frame_count:1000,
  ↪  report_name: "my_fsdb_report"}]
  #   report_configs: [{waveform_path: "/path/to/fsdb", inst:
  ↪  "ChipTop/system/tile_prci_domain/tile_reset_domain_tile", interval_size:
  ↪  "1ns", output_formats: ["plot_profile"]}]
  level: "rtl" # Power analysis mode for different levels of the VLSI flow (rtl,
  ↪  syn, par)
  start_times: [] # Optional: A list of analysis start times corresponding to
  ↪  each waveform used for dynamic power analysis.
  end_times: [] # Optional: A list of analysis end times corresponding to each
  ↪  waveform used for dynamic power analysis.
```

Figure 5.4: Hammer power input keys.

memory accesses for the vector machines, bypassing the SoC's last-level cache (LLC). These TCMs also facilitate the streaming of data between the vector core tiles, helping applications to leverage the chip's big/little architecture.

All together, these features make Cygnus a highly flexible and efficient platform for computing mobile multimedia applications, suitable for demonstrating the efficacy of short-vector machines in the mobile SoC context. Cygnus is the first demonstration of implementations of the Saturn vector unit working in silicon. Cygnus demonstrates high vector unit utilizations of over 90% when executing GEMM and convolution kernels, achieving a leading 414 GOPS/W on an INT8 GEMM kernel and 109 GFLOPS/W on an SGEMM kernel.

Chip Motivation and Background

DSP and image-processing workloads make up critical multimedia applications run on mobile SoCs. These domains require significant processing flexibility and contain widely varying application vector lengths [64][38][99]. Long-vector machines are flexible and can achieve great performance, efficiency, and utilization results while tolerating memory latencies when application vector lengths are sufficiently long [12]. However, application vector lengths are frequently short in mobile multimedia workloads, making short-vector machines more suitable for this domain. Cygnus integrates short-vector machines via instances of the Saturn vector unit, which we have demonstrated in simulation to be capable of achieving high utilization and efficiencies on both short and long application vector lengths.

As discussed in Chapter 2, image-processing applications are often made up of a pipeline of many operations computed in succession. Feature-detection pipelines, such as SIFT [74], are frequently structured with a frontend that denoises or generates a new representation of the image, consisting of operations applied across the image, and a backend that tests, refines, and describes image keypoints. The frontend stages tend to consist of compute-heavy convolution operations, among others, working with relatively long application vector lengths. These frontend algorithmic steps are applied largely uniformly across the image and multiple stages can often be fused together to improve data locality for more efficient execution. The backend stages tend to be less vectorizable and work on much shorter application vectors centered around each keypoint.

The big/little architecture paradigm has been employed as a method for leveraging processor heterogeneity in SoCs to build systems that are both performant and efficient [65][66][133]. In a similar spirit, the big/little vector architecture of Cygnus aims to match the heterogeneity of these mobile multimedia applications to be energy efficient. It can be advantageous to integrate memories that allow the cores executing different tasks in a pipeline to communicate directly, rather than through a lower level of the cache hierarchy. At the same time, a big/little architecture can also be suitable for targeting more homogeneous, highly data parallel workloads, such as a large matrix multiplication, by splitting up the workload in a balanced fashion across the big and little vector machines.

While not a focus of this chapter, Cygnus also lays the groundwork for further research projects in the chiplet domain. The chiplet design pattern has recently been used in com-

puting systems to make them more economical or to realize more complex systems than are possible with a single die. Chiplets have gained much attention in recent years with the emergence of advanced packaging technologies and 2.5D and 3D integration. As a chiplet targeting DSP applications, Cygnus may be interconnected with another Cygnus die to scale out the system (as pictured in Figure 5.8) or with a different type of chiplet, such as one targeting ML applications to create a heterogeneous chiplet system. To this end, Cygnus has a custom chip-to-chip TileLink interface and a streamlined version of UCIe [32], UCIe-lite, connected to GPIOs, which may be used to investigate the design of chiplet-based systems.

Cygnus Architecture

Figure 5.5 shows the overall architecture of Cygnus and its component blocks. Cygnus integrates eight RISC-V vector cores, compliant with the RVV 1.0 specification, generated from the Saturn vector unit generator. We can see the big/little vector architecture of Cygnus, with four big vector cores, with a `VLEN` of 512b and a `DLEN` of 256b, and four little vector cores, with a `VLEN` of 128b and a `DLEN` of 64b. The big vector core tiles each connect to the system interconnect individually, while the little vector core tiles are arranged together in a cluster, with the entire cluster connecting to the system interconnect. The choice of arranging the little vector cores in a cluster was made to enable higher-bandwidth communication between them and to simplify the physical implementation at the top-level of the design.

The top-level system interconnect consists of a 2D mesh NoC, of width 128b for data, that connects all the major subsystems together. Cygnus integrates a 512KB LLC with four banks. Figure 5.5 shows the arrangement of the LLC cache banks and the I/O peripherals. Each cache bank connects to the NoC in addition to the memory subsystem interconnect.

The chip has multiple channels for external communication with the test host, including UART and JTAG, in addition to a serial TileLink port. As previously mentioned, Cygnus also has a custom chip-to-chip digital interface, as well as some analog and mixed-signal test circuits for UCIe and DDR interfaces. The chip also integrates a foundry-supplied PLL for on-die clock generation, in addition to a top-level pin to clock the chip externally.

Figure 5.5 shows that each vector core tile comprises a scalar unit, an instance of the Saturn vector unit, and a 32KB TCM. Each scalar unit also includes private L1 instruction and data caches. Each vector unit's memory pipeline is backed by the LLC, via the system interconnect, and a TCM. Integrating a TCM in each compute tile allows the vector machines to operate on tiles of data without accessing the shared LLC, aside from initial fetches and final stores. This can enable improved performance and energy efficiency by providing high-bandwidth, uncached memory accesses for each vector unit and reducing pressure on the top-level memory system.

The short-vector machines integrated by Cygnus provide the flexible, yet efficient, computation required by SoCs targeting a broad range of mobile multimedia processing applications, as described in detail in Chapter 3 of this thesis. The decoupled vector backend and load-store unit in Saturn helps each compute tile tolerate memory latencies. This is impor-

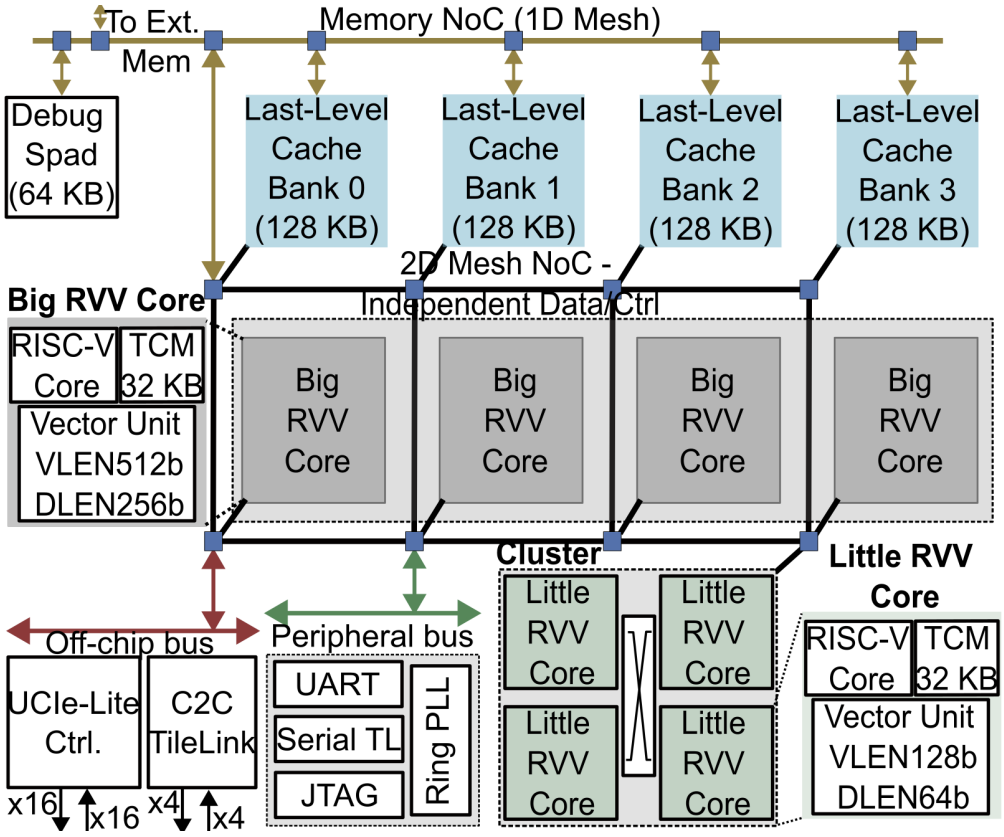


Figure 5.5: Block diagram of the Cygnus DSP chiplet architecture [59].

tant in the SoC context in which the memory latency from the LLC to each vector unit must be tolerated to achieve good performance in many cases. The sequencing mechanisms that enable limited out-of-order execution in the Saturn vector backend and the efficient support for the vector register grouping feature in RVV also help each vector core tile in Cygnus be performant and efficient.

The big vector core tiles integrate a dual-issue, in-order scalar unit. While the Saturn vector unit strives to reduce its reliance on the instruction throughput of its scalar unit, a higher IPC scalar can aid vector performance particularly when application vector lengths are short. A dual-issue scalar unit was selected as a balance between increased performance and increased scalar unit power and area overheads. Moving to wider in-order scalar units would deliver significantly diminishing returns along with increase scalar unit overheads.

For vectorizable kernels with long application vector lengths, it is expected that the big vector cores will achieve significantly better performance and energy efficiency than the little vector cores. This is because the energy costs of instruction fetch, the scalar unit, and control logic are amortized over longer vectors while still achieving high datapath utilizations. For particularly short application vector lengths, the utilization of the resources in the vector

backend will decrease and the energy efficiency will drop as the longer `VLEN` and `DLEN` may provide little benefit in this regime. In these cases, the kernels may more suitably map to the little vector cores, which are capable of achieving higher datapath utilizations on shorter application vector lengths than the big vector cores. This demonstrates one aspect of the Cygnus architecture for leveraging a big/little short-vector architecture for efficiently computing mobile multimedia workloads.

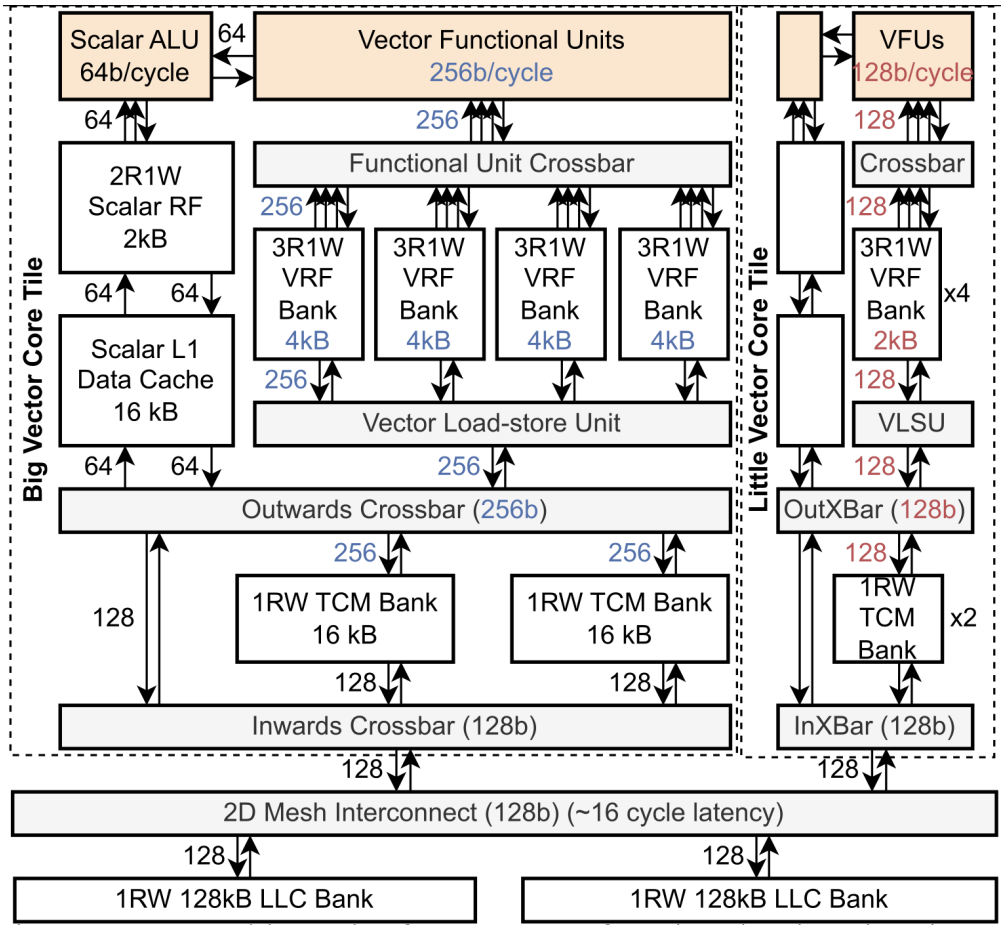


Figure 5.6: Block diagram of the Cygnus memory hierarchy [59].

Figure 5.6 shows a block diagram of the memory hierarchy of Cygnus, from the tile-level connecting out to the LLC via the 2D mesh NoC. Maximizing data reuse in the VRF with tiling or kernel fusion techniques can improve performance and energy efficiency by reducing the amount of time and energy spent moving data through the memory hierarchy. Every vector core on Cygnus is parameterized to have a VRF with 4 banks, each with 3 read ports and 1 write port. Recall that the VRF in Saturn is constructed out of flip-flops. Parameterizing Saturn to have 4 banks helps allow the various vector execution units

and vector memory units to access the VRF with fewer access conflicts, improving overall performance and efficiency.

The private scalar L1 data cache in each tile caches data sourced from the scalar register file by vector-scalar instructions, in addition to data for any other scalar code. The L1 caches are backed by the LLC through the 2D mesh NoC.

Figure 5.6 also shows the integration of the 32KB TCM in every tile. Each TCM is made up of 2x16KB banks of 1RW SRAM. These TCMs are uncached, software-managed scratchpad memories. While using the TCM adds some code complexity for explicitly managing the movement of data (rather than implicitly to and from the LLC), it can provide high-bandwidth, low-latency vector memory accesses local to each tile. Each bank is the width of the vector unit's `DLEN` so as to match the maximum memory bandwidth of the vector load-store unit. The TCM can serve as an intermediate level of the memory hierarchy between the VRF and the LLC. Each vector core can tile data in the VRF and the TCM to maximize the reuse of data within each vector core compute block. Results can be stored out to the LLC or streamed to a TCM local to a different vector core when they are finished. This is particularly valuable since, due to physical constraints, the top-level NoC bandwidth is somewhat limited, and so the TCM provides higher-bandwidth and lower-latency memory accesses than would otherwise be possible for all vector cores accessing the LLC without an additional level of the memory hierarchy. These accesses are also more energy efficient than fetching data from the LLC. Vector workloads can use a mix of cached and uncached memory accesses. For instance, for vector-scalar operations, the vector operand data may be fetched from the TCM, while scalar operand data may be fetched from L1 data cache. We will show some examples of this working in the following chip measurements section.

Beyond just the big/little aspect of the overall chip architecture, the flexibility provided by integrating multiple vector machines means that workloads can be mapped in a data parallel or pipeline parallel fashion. Large GEMMs and convolutions, for instance, can be split across the chip, with each vector core working on an individual part of the problem. These workload slices can be sized appropriately to rate-match the throughput of the big and little cores to better utilize all of the vector datapaths. In contrast to this data parallel mapping, workloads can also be mapped in a pipeline parallel or task parallel fashion, leveraging the integration of the TCMs with each vector core tile. Since each TCM is globally addressable, data can be streamed directly between vector core tiles using them, with each working on one or more stages of a compute pipeline. This feature can be leveraged to efficiently compute workloads like the aforementioned feature-detection pipelines onto Cygnus, mapping each portion of the pipeline onto the big and little vector cores according to which is more favorable.

The Saturn vector unit generator and Hammer were leveraged to explore and evaluate a space of designs for Cygnus. The highly parameterized nature of Saturn was used to generate the design points for the big and little vector units to evaluate. The final configurations of V512D256 and V128D64 were selected based on a balance of performance and physical constraints. Other Saturn parameters, such as the issue queue depths, were also selected during this process.

Hammer was used to drive the physical evaluation of the design space of the big and little vector tiles. The area of a Saturn instance is driven largely by the configuration of its scalar unit and the selection of the `VLEN` and `DLEN` parameters. Thus, those were the primary parameters of consideration when evaluating these instances given the chip area allotted to the compute cores. We found that a V512D256 configuration for the big vector cores allowed us to implement four of them, along with the required system interconnect routing, while comfortably meeting the timing targets that the top-level design required. In the area remaining from the big vector cores and the LLC, we found that we could implement four V128D64 vector units in a cluster configuration. The TCM sizes were selected to maximize their capacity within each tile while meeting the area constraints imposed by the top-level floorplan of the chip. Table 5.2 compares some of the high-level parameters and the resultant silicon area of the big and little vector tiles.

Table 5.2: Comparison of Big and Little Vector Units

	Big Vector Tile	**Little Vector Tile**
VLEN (bits)	512	128
DLEN (bits)	256	64
TCM Size (KB)	32	32
Area (mm^2)	0.82	0.26

Taking all these factors together, Cygnus is designed, from the SoC-level down to the implementation of each compute tile, to efficiently target DSP and image-processing applications while remaining highly flexible.

Cygnus Implementation

The Cygnus die has a total area of $16mm^2$, 4mm by 4mm, and is implemented in the Intel16 process technology. Each big vector core tile is 0.810mm by 1.008mm, about $0.82mm^2$, and each little vector core tile is 0.5mm by 0.510mm, about $0.26mm^2$. Each tile has a density of a bit over 60%, post-clock tree synthesis. To simplify the implementation, the core portion of the chip is all clocked synchronously, with clock crossings placed appropriately at the chip's I/O interfaces.

The design was implemented in Chisel within the Chipyard framework. Hammer was used to drive the entire digital PD flow for the chip. Due to the complexity of the design, Hammer's hierarchical flow capabilities were used. As previously mentioned, Hammer currently enables a bottom-up hierarchical implementation flow. Sub-blocks of the SoC are designated to be implemented hierarchically and are given a set of floorplan and timing constraints by the designer. Each sub-block is pushed through the entire implementation flow and the post-PnR boundary timing characteristics are realized. These implemented sub-blocks are then passed to a final run that synthesizes and places-and-routes the top-level of the design. This

flow can significantly reduce CAD tool runtimes for a design like Cygnus as common blocks can be implemented once and then stamped out multiple times at the top-level.

In Cygnus, each big vector core tile and each little vector core tile were leaf nodes in the generated Hammer flow and thus were implemented once at the block-level and replicated four times at the top-level. Each tile could meet timing at a clock frequency of upwards of 750MHz in the pessimistic slow-slow corner, but physical implementation challenges at the top-level left the final signoff frequency of the chip a bit lower.

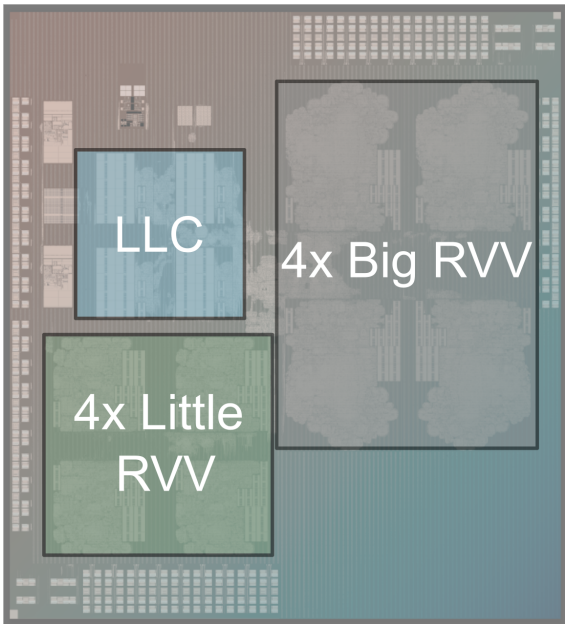


Figure 5.7: Cygnus GDS rendering with block overlays [59]. The Big RVV tiles have `VLEN=512b` and `DLEN=256b`. The Little RVV tiles have `VLEN=128b` and `DLEN=64b`. The LLC is 512KB and comprises four banks.

Figure 5.7 shows a rendering of the Cygnus SoC GDS at tapeout[4]. Overlayed are the locations of the four big vector tiles and the little vector tile cluster, along with the LLC. The open space on the die was originally budgeted for mixed-signal blocks for chiplet communication, which were moved to a subsequent tapeout. Cygnus was implemented in the Spring semester of 2024, with the tapeout in May[5].

[4]GDS Credit: Ken Ho

[5]I led the configuration and physical design of the Saturn vector core tiles on Cygnus and their interconnection at the top-level. I also assisted on the top-level design and chip finishing steps. Vikram Jain was the overall Cygnus chip lead, managing the top-level of the chip, verification, and chip signoff. Jerry Zhao, Kevin Anderson, Ken Ho, Yufeng Chi, Ella Schwarz, and Harrison Liew all played critical roles in the tapeout process. The first vector code confirming that the Saturn cores were working on the chip ran on Friday, December 13th, 2024, a few days after the packaged chips were received.

Cygnus Measurements and Results

In the testing setup, two Cygnus chips are co-packaged on an organic substrate as shown in Figure 5.8. The package is fastened into the testing board, which in turn plugs into two separate FPGA boards as pictured in Figure 5.9.

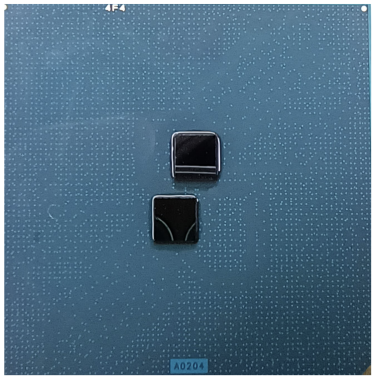


Figure 5.8: Cygnus package [59].

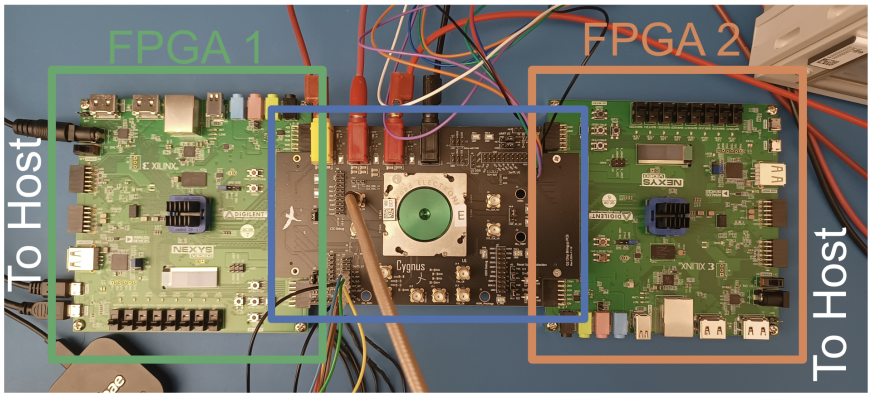


Figure 5.9: Cygnus test setup [59].

Using two FPGAs simplified the testing process, as each FPGA board can individually connect to its corresponding Cygnus chip. The FPGA is then connected to a host PC. Figure 5.10 shows the shmoo plot for a measured Cygnus die. The chip is functional up to a clock frequency of 1GHz at a core voltage of 0.94V and can operate at a supply voltage of 0.6V with a 350MHz clock frequency.

Cygnus is the first chip to demonstrate the Saturn vector unit working in silicon. While prior physical design experiments had demonstrated that these Saturn configurations were physically implementable, this chip makes the design silicon proven and supports one of the original Saturn project goals of targeting ASIC implementations. Furthermore, this allows

us to physically measure energy efficiency and performance in a real SoC to benchmark these Saturn instances.

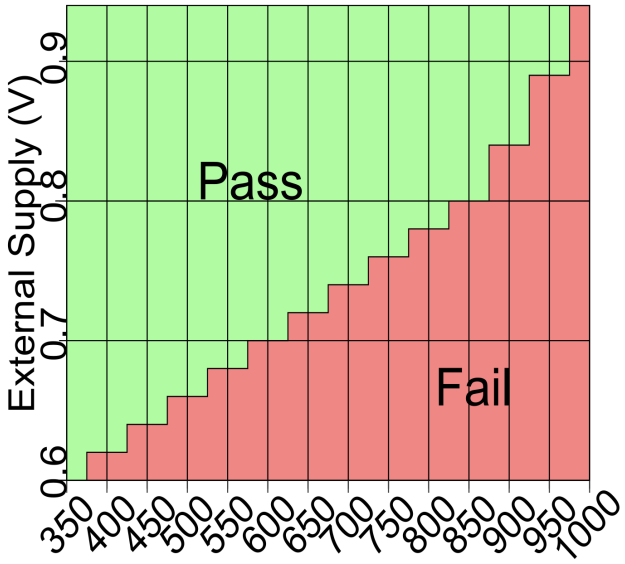


Figure 5.10: Cygnus shmoo plot [59].

Figures 5.11 and 5.12 show the measured performance and efficiency for executing an SGEMM kernel and an 8-bit integer IGEMM kernel on one of each of the big vector cores and little vector cores on Cygnus. For a 64x64 SGEMM and a 128x128 IGEMM, the big vector core achieves 40 GFLOPS/W and 154 GOPS/W at the most efficient points, respectively. The little vector core achieves 12 GFLOPS/W and 47 GOPS/W on the SGEMM and IGEMM at the most efficient points, respectively. Each core achieves greater than 90% datapath utilization in these experiments.

The GEMM kernels utilize standard performance programming techniques, such as loop unrolling and software pipelining. This helps validate the design intent behind each Saturn vector core and usage of the TCMs. The Saturn design helps tolerate memory latencies in the system, with the sequencing mechanism in the vector backend able to keep the functional units busy. The TCMs help alleviate issues with top-level interconnect bandwidth. In the GEMM kernels, the values from the A matrix are accessed as scalars, fetched from the L1 data cache after being initially fetched from the LLC. Because the vector unit is decoupled from the scalar core, these accesses can run ahead. The B and C matrices are fetched from the TCM. Tiling techniques are also used to keep intermediate results resident in the vector register file to minimize data movement and maximize efficiency.

Figures 5.13 and 5.14 show the results for the SGEMM and IGEMM kernels, respectively, when all vector cores on the chip are used together. In this case, a 192x192 SGEMM and a 384x384 IGEMM are deployed. The work for these matrices is divided across the big and little vector cores to roughly rate-balance the throughput. At the most efficient points,

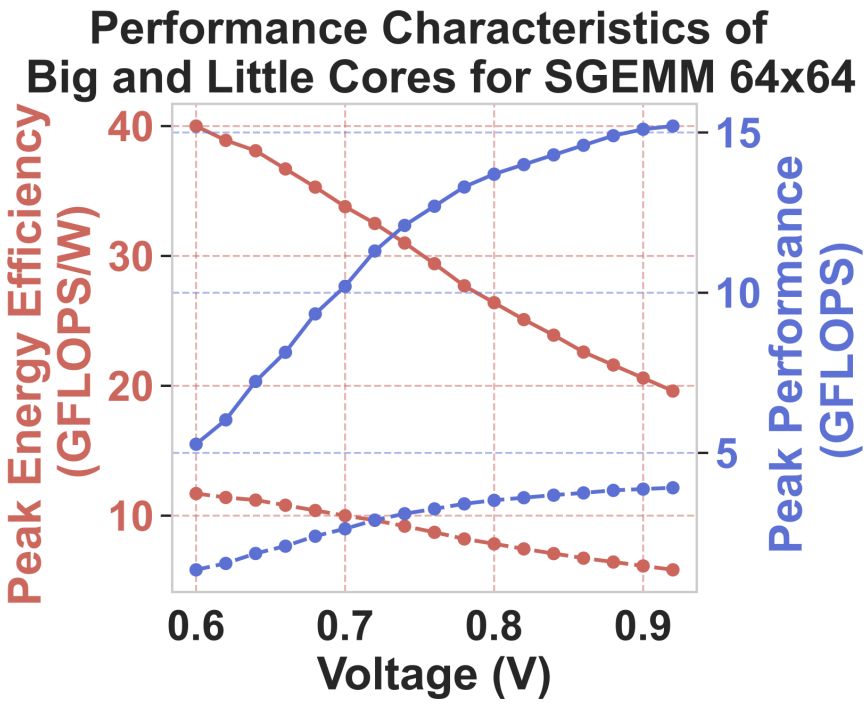


Figure 5.11: Vector core 64x64 SGEMM results [59].

Cygnus is capable of achieving 109 GFLOPS/W for the SGEMM kernel and 414 GOPS/W for the IGEMM kernel. When compared against other state-of-the-art vector processors and chip implementations, Cygnus achieves best-in-class efficiency [59].

Figure 5.15 shows the efficiency and performance results when mapping other kernels to the big and little vector cores. This demonstrates the flexibility of the system to map a wide range of workloads. Figure 5.16 shows the performance of the big and little vector cores when sweeping SGEMM kernel size. As expected, the little vector core achieves higher utilizations at smaller kernel sizes, but both cores are able to achieve high utilizations quickly as the problem size increases, as would be expected from the design intent and results for the Saturn vector unit discussed in Chapter 3.

Cygnus Conclusion

Cygnus, at the SoC level, demonstrates the efficiency and flexibility granted by a big/little vector architecture. The big and little cores can all work together on the same problem for maximum top-line compute throughput or can operate in a pipeline parallel fashion for compute pipelines that may map to a wider or narrower vector unit more efficiently. This is underlain by the integration of the TCMs per-vector core tile, which allows for more efficient local memory accesses, facilitates the streaming of data between cores, and alleviates top-

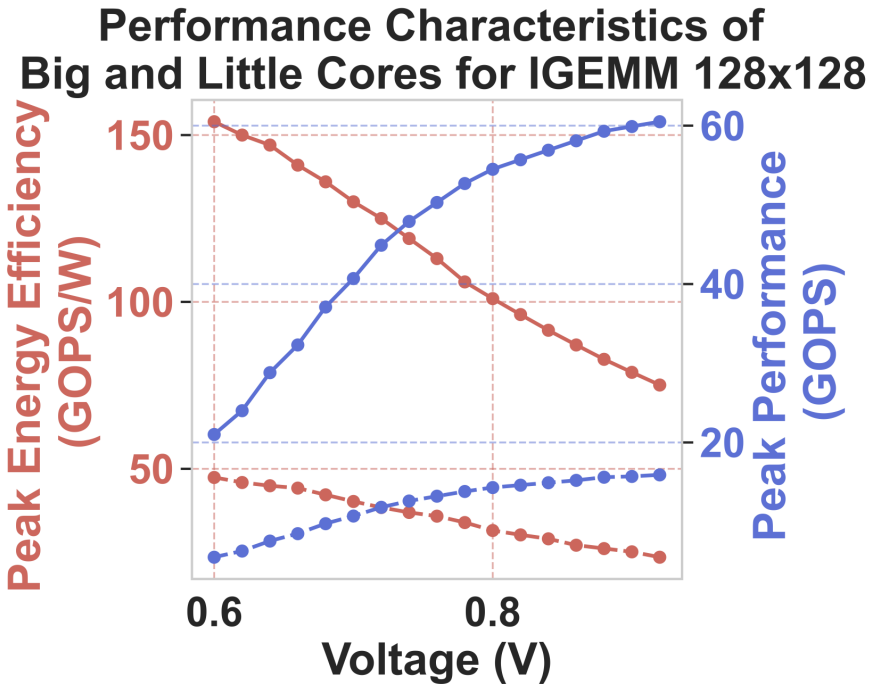


Figure 5.12: Vector core 128x128 IGEMM results [59].

level interconnect throughput limitations.

Cygnus is the first demonstration of the Saturn vector unit working in silicon. Not only does this show Saturn's physical realizability, but the results help validate the ideas presented earlier in this thesis. The measured results from Cygnus show that short-vector machines can efficiently execute a wide variety of kernels with both short and long application vector lengths. Each core is able to achieve high datapath utilizations on GEMM and convolution kernels, contributing to the top-line energy efficiency results. Since we know that the area overhead from the control mechanisms in the Saturn vector backend is fairly marginal, these results also demonstrate favorable area efficiency characteristics. These are all critical results for DSP-type cores that are integrated into mobile SoCs for multimedia applications, as described previously.

Finally, Cygnus becomes another chip in the now long line of silicon designed using the Hammer framework. Once again, a small team was able to come together to design a complex SoC in a short period of time using Hammer. This was enabled in part by Hammer facilitating the reuse of design effort from previous tapeouts, including past investments in the Intel16 technology plugin and many CAD tool plugins. At the block-level, Hammer had already facilitated the physical optimization of the Saturn vector unit, such that it did not lie on the critical path in the final tapeout.

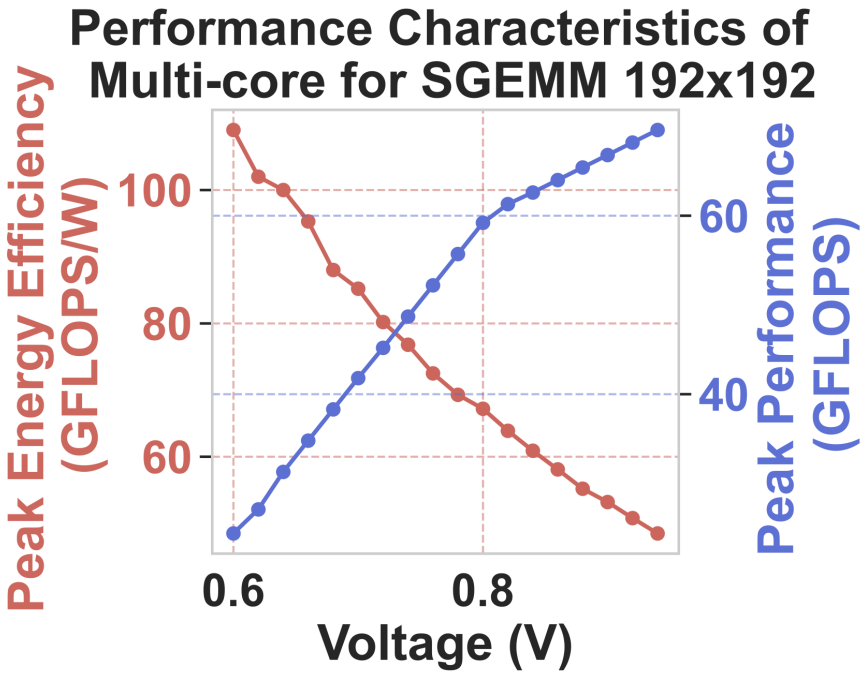


Figure 5.13: Cygnus multi-core 192x192 SGEMM results [59].

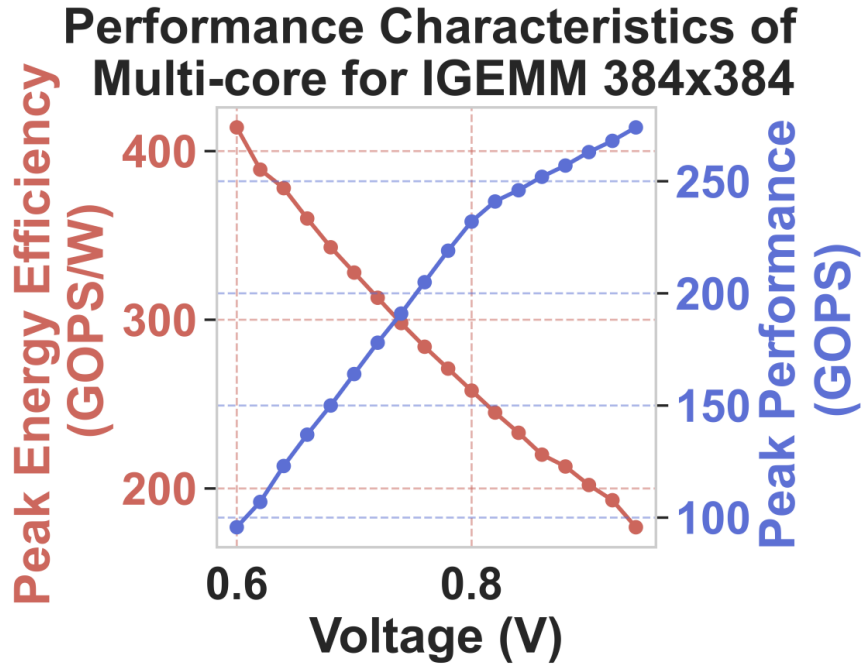


Figure 5.14: Cygnus multi-core 384x384 IGEMM results [59].

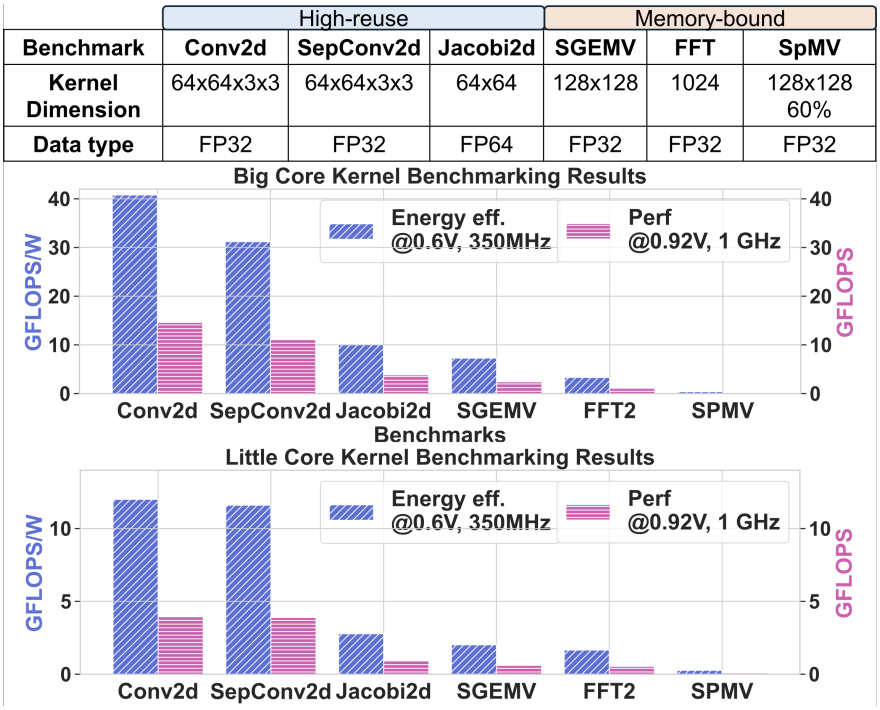


Figure 5.15: Performance and energy efficiency results for the big and little vector core tiles executing kernels with varying arithmetic intensities [59].

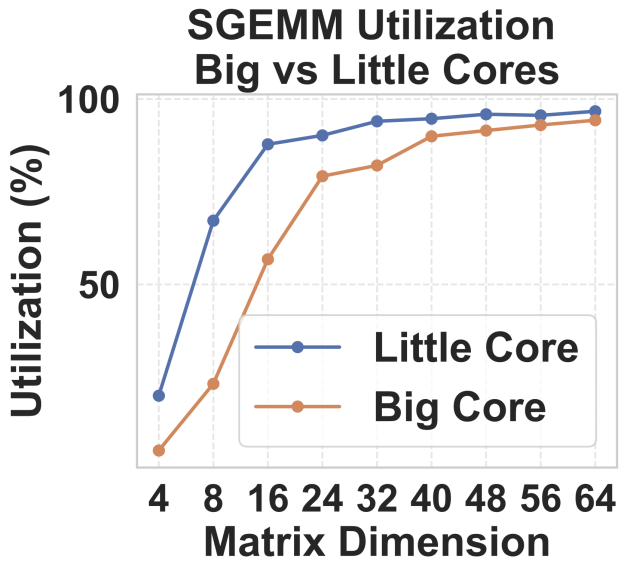


Figure 5.16: Utilization of the big and little vector core tiles executing an SGEMM across a sweep of square matrix sizes. The little vector cores achieve a higher utilization at smaller problem sizes, demonstrating how utilizing the little vector cores may be advantageous [59].

Chapter 6

Conclusion

The need for flexible yet efficient processing for multimedia applications in constrained deployments drives the integration of DSP cores in mobile SoCs. Traditional DSP core design paradigms have been tremendously successful, but the continued slowing of silicon scaling and growth in algorithmic breadth and complexity necessitates continued innovation. The emergence of modern scalable vector ISAs, such as RVV and SVE, presents the opportunity for these cores to be designed based on an efficient short-vector machine template. Such a DSP core platform has the potential to be highly energy efficient and area efficient across a variety of design points, underlain by cost-effective hardware mechanisms that enable high resource utilization without high instruction throughput. Furthermore, this paradigm can enable software compatibility, and potentially performance portability, across DSP core instances while being programmer friendly. These are particularly valuable characteristics given that a single SoC, or class of SoCs, may integrate the same DSP core multiple times, parameterized to fit particular performance and area constraints.

The Saturn project makes the argument for designing these DSP cores as short-vector machines. Short-vector machines can efficiently execute a wide range of multimedia kernels with both short and long application vector lengths by leveraging a decoupled design, aggressive sequencing mechanisms in the vector backend, and efficient support for vector register grouping. Saturn, being highly parameterized, also demonstrates how this type of design can be scaled up and down to fit the needs of a particular deployment while running the same vector-length-agnostic code with high resource utilization.

Domain specialization is critical for energy efficiency in silicon design today, and so it is important to understand how these DSP cores can be specialized. The techniques presented in this thesis show that short-vector machines can be specialized using economical, low-overhead mechanisms without sacrificing the flexibility of the baseline machine. Designing new, scalable vector instructions for short-vector DSP cores is one of the most effective techniques for domain specialization and gives kernel writers new tools to implement efficient multimedia applications.

Not only is what we design important, but also how we implement and evaluate it. Designing specialized short-vector machines hinges on the ability of architects to rapidly

implement, evaluate, and iterate based on actionable PPA feedback. Thus, the development of new physical design methodologies that enable architects to realistically evaluate design feasibility, efficiency, and trade-offs is vital. This will only increase in importance as the challenges and costs of silicon design grow. The Hammer framework facilitates design reuse and rapid DSE through the separation of concerns, modular design, and codifying designer intent. The Cygnus DSP SoC leveraged Hammer and demonstrated the Saturn vector unit working in silicon. The utilization, performance, and efficiency measurements from this chip help to validate the arguments made in favor of deploying short-vector machines as DSP cores.

Summary of Contributions

This thesis makes the following contributions:

- A review of mobile multimedia applications and domain-specialized processing design patterns in mobile SoCs.
- An analysis of the use of scalable short-vector machines as signal processors and their implementation challenges.
- Contributions to the Saturn vector unit, a parameterized Chisel generator for short-vector machines implementing the RVV 1.0 specification.
- The proposal and analysis of techniques for extending short-vector machines to specialize them for a particular multimedia application domain to enhance their energy efficiency and enable new PPA trade-offs.
- The design and evaluation of novel vector instructions that are flexible and scalable, suitable for integration with short-vector machines to improve efficiency on multimedia and embedded machine-learning applications. The proposed instructions result in performance gains of over 1.75× and 2× and energy efficiency gains of over 1.6× and 2× for integer matrix multiplication and convolution kernels, respectively, compared to a baseline short-vector machine instance with the same architectural vector length and base datapath width.
- Contributions to Hammer, a modular physical design framework that aims to maximize design reuse, enabling agile physical design flows and the rapid gathering of PPA feedback and design space exploration.
- The tapeout of multiple instances of Saturn vector DSP cores in the Cygnus DSP SoC, a 16mm^2 octacore vector processor in the Intel16 technology, demonstrating the viability of silicon implementations of Saturn and measured energy efficiency results. Cygnus can achieve over 90% datapath utilization on matrix multiplication and convolution kernels and measured energy efficiencies of 414 GOPS/W on INT8 matrix multiplication and 109 GFLOPS/W on FP32 matrix multiplication kernels.

Future Work

- The domain specialization techniques described in this thesis focused primarily on mobile multimedia applications, including image processing and embedded machine learning. These methodologies and techniques can be extended to address other domains including communications, audio processing, and other machine-learning workloads. This may reveal interesting trade-offs for the deployment of vector DSP cores in other mobile SoC subsystems. This could include new microarchitectural techniques, new scalable vector instructions, and support for new datatypes.

- The custom vector instructions presented were designed manually by considering the target application, programming model, microarchitecture, and physical design. A framework could be designed to automatically extract important sequences of vector operations that may be fused or targeted with a new set of functional units. This could enable the more rapid design of specialized short-vector machines. Previous works have designed such flows to target traditional DSP cores, which could be extended to target short-vector machines.

- The Saturn vector unit generator could be enhanced to improve its baseline energy efficiency and area efficiency. For example, the vector register file may benefit from implementation as a custom memory or latch array, rather than being purely synthesized with flip-flops.

- The work in this thesis primarily focused on techniques for domain specialization at the short-vector machine-level, but future work could be done at the system-level, building on the work described here. Exploring specialized vector DSP multicore architectures, building on the work of Cygnus, and more advanced accelerator integration is an important direction to address the challenges of efficiently running image-processing and machine-learning workloads in mobile SoCs.

Bibliography

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21.

[2] Apple. *Apple Introduces M4 Chip*. Available: `https://www.apple.com/newsroom/2024/05/apple-introduces-m4-chip/`. 2024.

[3] Apple. *Deploying Transformers on the Apple Neural Engine*. Available: `https://machinelearning.apple.com/research/neural-engine-transformers`.

[4] Arm. *Arm Architecture Reference Manual for A-Profile Architecture*. 2024.

[5] Arm. *Armv8-M Architecture Reference Manual*. 2015.

[6] Krste Asanovic. "Vector Microprocessors". PhD thesis. 1998.

[7] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. "The Rocket Chip Generator". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), pp. 6–2.

[8] 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. Dec. 2006.

[9] Omid Azizi, Aqeel Mahesri, Benjamin C Lee, Sanjay J Patel, and Mark Horowitz. "Energy-Performance Tradeoffs in Processor Architecture and Circuit Design: a Marginal Cost Analysis". In: *ACM SIGARCH Computer Architecture News* 38.3 (2010), pp. 26–36.

[10] 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: *Proceedings of the 49th Annual Design Automation Conference*. 2012, pp. 1216–1225.

[11] David H Bailey. "A High-Performance FFT Algorithm for Vector Supercomputers". In: *The International Journal of Supercomputing Applications* 2.1 (1988), pp. 82–87.

[12] Fabio Banchelli, Marta Garcia-Gasulla, and Filippo Mantovani. "Batched DGEMMs for Scientific Codes Running on Long Vector Architectures". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2024, pp. 17–31.

[13] George H Barnes, Richard M Brown, Maso Kato, David J Kuck, Daniel L Slotnick, and Richard A Stokes. "The ILLIAC IV Computer". In: *IEEE Transactions on Computers* 100.8 (1968), pp. 746–757.

[14] Batcher. "Design of a Massively Prallel Processor". In: *IEEE Transactions on Computers* 100.9 (1980), pp. 836–840.

[15] Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanovic. "Cache Refill/Access Decoupling for Vector Machines". In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE. 2004, pp. 331–342.

[16] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "Surf: Speeded Up Robust Features". In: *Computer Vision–ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I 9*. Springer. 2006, pp. 404–417.

[17] BDTI. *Next-Gen Cadence Tensilica Vision Processor Core Claims Big Performance, Energy Consumption Gains — Berkeley Design Technology, Inc.* Available: `https://www.bdti.com/InsideDSP/2015/12/17/Cadence`.

[18] Jeff Bier. *Jeff Bier's Impulse Response—Are DSPs Dying? — Berkeley Design Technology, Inc.* BDTI Article. Available: `https://www.bdti.com/InsideDSP/2013/06/13/JeffBierImpulseResponse`.

[19] MS Brown and S Kim. "Understanding Color and the In-Camera Image Processing Pipeline for Computer Vision". In: *Proc. IEEE/CVF Int. Conf. Comput. Vis. Tutorial*. 2019, pp. 1–247.

[20] John S Brunhaver II. "Design and Optimization of a Stencil Engine". PhD thesis. Stanford University, 2015.

[21] Cadence. *Compute IP*. Available: `https://www.cadence.com/en_US/home/tools/silicon-solutions/compute-ip.html`.

[22] John Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6 (1986), pp. 679–698.

[23] Steve Carr and Ken Kennedy. "Improving the Ratio of Memory Operations to Floating-Point Operations in Loops". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.6 (1994), pp. 1768–1810.

[24] Alex Carsello, James Thomas, Ankita Nayak, Po-Han Chen, Mark Horowitz, Priyanka Raina, and Christopher Torng. "mflowgen: A Modular Flow Generator and Ecosystem for Community-Driven Physical Design". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 1339–1342.

[25] Matheus Cavalcante, Domenic Wüthrich, Matteo Perotti, Samuel Riedel, and Luca Benini. "Spatz: A Compact Vector Processing Unit for High-Performance and Energy-Efficient Shared-L1 Clusters". In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 2022, pp. 1–9.

[26] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V". In: *arXiv preprint arXiv:1607.02318* (2016).

[27] Ceva. *Ceva - Leading Licensor of Signal Processing IP*. Available: `https://www.ceva-ip.com/`.

[28] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. "Low-Power CMOS Digital Design". In: *IEICE Transactions on Electronics* 75.4 (1992), pp. 371–382.

[29] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, et al. "An Overview of Core Coding Tools in the AV1 Video Codec". In: *2018 Picture Coding Symposium (PCS)*. IEEE. 2018, pp. 41–45.

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

[31] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications". In: *IEEE Micro* 34.2 (2014), pp. 34–43.

[32] Debendra Das Sharma, Gerald Pasdast, Zhiguo Qian, and Kemal Aygun. "Universal Chiplet Interconnect Express (UCIe): An Open Industry Standard for Innovations With Chiplets at Package Level". In: *IEEE Transactions on Components, Packaging and Manufacturing Technology* 12.9 (2022), pp. 1423–1431.

[33] 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* 9.5 (1974), pp. 256–268.

[34] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. "A Set of Level 3 Basic Linear Algebra Subprograms". In: *ACM Transactions on Mathematical Software (TOMS)* 16.1 (1990), pp. 1–17.

[35] Roger Espasa and Mateo Valero. "Decoupled Vector Architectures". In: *Proceedings. Second International Symposium on High-Performance Computer Architecture*. IEEE. 1996, pp. 281–290.

[36] Jennifer Eyre and Jeff Bier. "The Evolution of DSP Processors". In: *IEEE Signal Processing Magazine* 17.2 (2000), pp. 43–51.

[37] Paolo Faraboschi, Geoffrey Brown, Joseph A Fisher, Giuseppe Desoli, and Fred Homewood. "Lx: A Technology Platform for Customizable VLIW Embedded Processing". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 2000, pp. 203–213.

[38] Gianluca Frison, Dimitris Kouzoupis, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. "BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization". In: *ACM Transactions on Mathematical Software (TOMS)* 44.4 (2018), pp. 1–30.

[39] Geekbench. *Geekbench 6 - Cross-Platform Benchmark*. Available: `https://www.geekbench.com/`.

[40] Hasan Genc. "Enabling Full-Stack DNN Accelerator Design and Evaluation on Synthesizable Hardware". PhD thesis. EECS Department, University of California, Berkeley, Aug. 2024.

[41] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 769–774.

[42] Ricardo E Gonzalez. "Xtensa: A Configurable and Extensible Processor". In: *IEEE micro* 20.2 (2000), pp. 60–70.

[43] Matthias Gries and Kurt Keutzer. *Building ASIPS: The MESCAL Methodology*. Springer Science & Business Media, 2005.

[44] Monica Gupta. *Google Tensor is a milestone for machine learning*. Google Article. Available: `https://blog.google/products/pixel/introducing-google-tensor/`. 2021.

[45] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. "Understanding Sources of Inefficiency in General-Purpose Chips". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: Association for Computing Machinery, 2010, pp. 37–47.

[46] Chris Harris, Mike Stephens, et al. "A Combined Corner and Edge Detector". In: *Alvey Vision Conference*. Vol. 15. 50. Citeseer. 1988, pp. 10–5244.

[47] David Harris and N Weste. "CMOS VLSI Design". In: *ed: Pearson Education, Inc* (2010).

[48] Khalid Al-Hawaj, Tuan Ta, Nick Cebry, Shady Agwa, Olalekan Afuye, Eric Hall, Courtney Golden, Alyssa B Apsel, and Christopher Batten. "EVE: Ephemeral Vector Engines". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 691–704.

[49] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture". In: *Commun. ACM* 62.2 (Jan. 2019), pp. 48–60.

[50] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[51] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. Appendix E. 2017.

[52] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. Appendix H. 2017.

[53] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14.

[54] Mark Horowitz, Elad Alon, Dinesh Patil, Samuel Naffziger, Rajesh Kumar, and Kerry Bernstein. "Scaling, Power, and the Future of CMOS". In: *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.* IEEE. 2005, 7–pp.

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

[56] Chenming Hu. *Modern Semiconductor Devices for Integrated Circuits*. 2010.

[57] Christopher J Hughes. "Exploiting Data Parallelism with SIMD Execution". In: *Single-Instruction Multiple-Data Execution*. Springer, 2015, pp. 9–26.

[58] Intel. *Intel®Architecture Instruction Set Extensions and Future Features*. 2025.

[59] Vikram Jain, Daniel Grubb, Jerry Zhao, Kevin Anderson, Ken T. Ho, Yufeng Chi, Ella Schwarz, Krste Asanovic, Y. S. Shao, and Borivoje Nikolic. "Cygnus: A 1 GHz Heterogeneous Octa-Core RISC-V Vector Processor for DSP". In: *2025 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. IEEE. 2025.

[60] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, Thai Nguyen, Ramesh Chukka, Ken Shiring, Koan-Sin Tan, Mark Charlebois, William Chou, et al. "MLPerf Mobile Inference Benchmark: An Industry-Standard Open-Source Machine Learning Benchmark for On-Device AI". In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 352–369.

[61] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[62] Leela S. Karumbunathan. *Nvidia Jetson AGX Orin Series*. NVIDIA Technical Brief. 2022.

[63] Alireza Khadem, Daichi Fujiki, Hilbert Chen, Yufeng Gu, Nishil Talati, Scott Mahlke, and Reetuparna Das. "Multi-Dimensional Vector ISA Extension for Mobile In-Cache Computing". In: *arXiv preprint arXiv:2501.09902* (2025).

[64] Alireza Khadem, Daichi Fujiki, Nishil Talati, Scott Mahlke, and Reetuparna Das. "Vector-Processing for Mobile Devices: Benchmark and Analysis". In: *2023 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2023, pp. 15–27.

[65] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. "Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 2003, pp. 81–92.

[66] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance". In: *Proceedings of the 31st Annual International Symposium on Computer Architecture*. ISCA '04. München, Germany: IEEE Computer Society, 2004, p. 64.

[67] Hsiang-Tsung Kung. "Memory Requirements for Balanced Computer Architectures". In: *Journal of Complexity* 1.1 (1985), pp. 147–157.

[68] Monica Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. 1988, pp. 318–328.

[69] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and K Asanovic. "The Hwacha Microarchitecture Manual, Version 3.8". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-263* (2015).

[70] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and K Asanovic. "The Hwacha Vector-Fetch Architecture Manual, Version 3.8. 1". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262* (2015).

[71] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. "An Agile Approach to Building RISC-V Microprocessors". In: *IEEE Micro* 36.2 (2016), pp. 8–20.

[72] Harrison Liew, Daniel Grubb, John Wright, Colin Schmidt, Nayiri Krzysztofowicz, Adam Izraelevitz, Edward Wang, Krste Asanović, Jonathan Bachrach, and Borivoje Nikolić. "Hammer: A Modular and Reusable Physical Design Flow Tool". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 1335–1338.

[73] Kun-Shan Lin, G.A. Frantz, and R. Simar. "The TMS320 Family of Digital Signal Processors". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1143–1159. DOI: 10.1109/PROC.1987.13867.

[74] David G Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60 (2004), pp. 91–110.

[75] Eric Mahurin. "Qualocmm® Hexagon™ NPU." In: *HCS*. 2023, pp. 1–19.

[76] Dejan Markovic, Vladimir Stojanovic, Borivoje Nikolic, Mark A Horowitz, and Robert W Brodersen. "Methods for True Energy-Performance Optimization". In: *IEEE Journal of Solid-State Circuits* 39.8 (2004), pp. 1282–1293.

[77] Dejan Marković and Robert W Brodersen. *DSP Architecture Design Essentials*. Springer Science & Business Media, 2012.

[78] MediaTek. *MediaTek — MediaTek Dimensity 9300*. Available: https://www.mediatek.com/products/smartphones/mediatek-dimensity-9300.

[79] MediaTek. *White Paper: The All-Big-Core CPU Architecture of MediaTek Dimensity 9300*. MediaTek White Paper. 2024.

[80] Sachin Mehta and Mohammad Rastegari. "MobileViT: Light-Weight, General-Purpose, and Mobile-Friendly Vision Transformer". In: *arXiv preprint arXiv:2110.02178* (2021).

[81] Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, et al. "Vitruvius+: an Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications". In: *ACM Transactions on Architecture and Code Optimization* 20.2 (2023), pp. 1–25.

[82] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. "Accelerating Sparse Deep Neural Networks". In: *arXiv preprint arXiv:2104.08378* (2021).

[83] Gordon E Moore. "Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, pp. 114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35.

[84] José E Moreira, Kit Barton, Steven Battle, Peter Bergner, Ramon Bertran, Puneeth Bhat, Pedro Caldeira, David Edelsohn, Gordon Fossum, Brad Frey, et al. "A Matrix Math Facility for Power ISA (TM) Processors". In: *arXiv preprint arXiv:2104.03142* (2021).

[85] Debargha Mukherjee, Jingning Han, Jim Bankoski, Ronald Bultje, Adrian Grange, John Koleszar, Paul Wilkins, and Yaowu Xu. "A Technical Overview of VP9—The Latest Open-Source Video Codec". In: *SMPTE Motion Imaging Journal* 124.1 (2015), pp. 44–54.

[86] Huy Nguyen and Lizy Kurian John. "Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology". In: *Proceedings of the 13th international conference on Supercomputing*. 1999, pp. 11–20.

[87] Tetsuya Odajima, Yuetsu Kodama, Miwako Tsuji, Motohiko Matsuda, Yutaka Maruyama, and Mitsuhisa Sato. "Preliminary Performance Evaluation of the Fujitsu A64FX using HPC Applications". In: *2020 IEEE International Conference on Cluster Computing (Cluster)*. IEEE. 2020, pp. 523–530.

[88] Andreas Olofsson, William Ransohoff, and Noah Moroze. "A Distributed Approach to Silicon Compilation". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 1343–1346.

[89] Kariofyllis Patsidis, Chrysostomos Nicopoulos, Georgios Ch Sirakoulis, and Giorgos Dimitrakopoulos. "RISC-V 2: a Scalable RISC-V Vector Processor". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5.

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

[91] Matteo Perotti, Matheus Cavalcante, Renzo Andri, Lukas Cavigelli, and Luca Benini. "Ara2: Exploring Single-and Multi-Core Vector Processing with an Efficient RVV 1.0 Compliant Open-Source Processor". In: *IEEE Transactions on Computers* (2024).

[92] Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. "A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V 1.0 Vector Processor Design". In: *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2022, pp. 43–51.

[93] Matteo Perotti, Yichao Zhang, Matheus Cavalcante, Enis Mustafa, and Luca Benini. "MX: Enhancing RISC-V's Vector ISA for Ultra-Low Overhead, Energy-Efficient Matrix Multiplication". In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2024, pp. 1–6.

[94] Navaneeth Kunhi Purayil, Matteo Perotti, Tim Fischer, and Luca Benini. "AraXL: A Physically Scalable, Ultra-Wide RISC-V Vector Processor Design for Fast and Efficient Computation on Long Vectors". In: *arXiv preprint arXiv:2501.10301* (2025).

[95] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. "Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 2013, pp. 24–35.

[96] Qualcomm. *Compare Snapdragon Processors — Snapdragon Series Comparison and Latest Snapdragon List — Snapdragon Chipset Ranking — Qualcomm*. Available: `https://www.qualcomm.com/products/mobile/snapdragon/smartphones/comparison`.

[97] Jan Rabaey. *Low Power Design Essentials*. Springer Science & Business Media, 2009.

[98] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *ACM Sigplan Notices* 48.6 (2013), pp. 519–530.

[99] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. "A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 17.4 (2020), pp. 1–30.

[100] B. Ramakrishna Rau. "Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops". In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 63–74.

[101] Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham. "Pixel Visual Core: Google's Fully Programmable Image Vision and AI Processor for Mobile Devices". In: *Proc. IEEE Hot Chips Symp.(HCS)*. 2018, pp. 1–18.

[102] Yi Ren, Chenxu Hu, Xu Tan, Tao Qin, Sheng Zhao, Zhou Zhao, and Tie-Yan Liu. "Fastspeech 2: Fast and High-Quality End-to-End Text to Speech". In: *arXiv preprint arXiv:2006.04558* (2020).

[103] Alejandro Rico, Satyaprakash Pareek, Javier Cabezas, David Clarke, Baris Ozgul, Francisco Barat, Yao Fu, Stephan Münz, Dylan Stuart, Patrick Schlangen, Pedro Duarte, Sneha Date, Indrani Paul, Jian Weng, Sonal Santan, Vinod Kathail, Ashish Sirasao, and Juanjo Noguera. "AMD XDNA NPU in Ryzen AI Processors". In: *IEEE Micro* 44.6 (2024), pp. 73–82.

[104] *RISC-V "V" Vector Extension*. Available: `https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf`. Sept. 2021.

[105] Chris Rowen. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*. Pearson Education, 2008.

[106] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. "ORB: An Efficient Alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. Ieee. 2011, pp. 2564–2571.

[107] Richard M. Russell. "The CRAY-1 Computer System". In: *Commun. ACM* 21.1 (Jan. 1978), pp. 63–72. DOI: `10.1145/359327.359336`.

[108] Samsung. *Samsung Exynos Processor*. Available: `https://semiconductor.samsung.com/processor/`.

[109] Colin Schmidt. "Extending Temporal-Vector Microarchitectures for Two-Dimensional Computations". PhD thesis. EECS Department, University of California, Berkeley, Aug. 2021.

[110] Semidynamics. *Semidynamics Vector Unit - Only 100% customisable RISC-V Vector Unit*. Available: `https://semidynamics.com/en/technology/vector-unit`.

[111] SiFive. *SiFive Intelligence X380*. Available: `https://www.sifive.com/cores/intelligence-x380`.

[112] SiFive. *SiFive Performance Family*. Available: `https://www.sifive.com/cores/performance`.

[113] R. Simar and R. Tatge. "How TI Adopted VLIW in Digital Signal Processors". In: *IEEE Solid-State Circuits Magazine* 1.3 (2009), pp. 10–14. DOI: `10.1109/MSSC.2009.933438`.

[114] Sawan Singh, Arthur Perais, Alexandra Jimborean, and Alberto Ros. "Exploring Instruction Fusion Opportunities in General Purpose Processors". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 199–212.

[115] James E Smith. "Decoupled Access/Execute Computer Architectures". In: *ACM SIGARCH Computer Architecture News* 10.3 (1982), pp. 112–119.

[116] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. "The ARM Scalable Vector Extension". In: *IEEE Micro* 37.2 (Mar. 2017), pp. 26–39.

[117] Wonyong Sung and Sanjit K Mitra. "Implementation of Digital Filtering Algorithms Using Pipelined Vector Processors". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1293–1303.

[118] Synopsys. *Synopsys Processor IP Solutions — Synopsys*. Available: `https://www.synopsys.com/designware-ip/processor-solutions.html`.

[119] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer Nature, 2022.

[120] Tuan Ta, Khalid Al-Hawaj, Nick Cebry, Yanghui Ou, Eric Hall, Courtney Golden, and Christopher Batten. "big.VLITTLE: On-Demand Data-Parallel Acceleration for Mobile Systems on Chip". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 181–198. DOI: `10.1109/MICRO56248.2022.00025`.

[121] Raúl Taranco, José-María Arnau, and Antonio González. "SLIDEX: Sliding Window Extension for Image Processing". In: *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2023, pp. 332–334.

[122] Andes Technology. *RISC-V:NX27V*. Available: `https://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v/`.

[123] Rafika Thabet, Ramzi Mahmoudi, and Mohamed Hedi Bedoui. "Image Processing on Mobile Devices: An Overview". In: *International Image Processing, Applications and Systems Conference*. IEEE. 2021, pp. 1–8.

[124] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. "Evaluating Programmable Architectures for Imaging and Vision Applications". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13.

[125] Pablo Vizcaino, Georgios Ieronymakis, Nikolaos Dimou, Vassilis Papaefstathiou, Jesus Labarta, and Filippo Mantovani. "Short Reasons for Long Vectors in HPC CPUs: a Study Based on RISC-V". In: *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 1543–1549.

[126] Edward Wang, Colin Schmidt, Adam Izraelevitz, John Wright, Borivoje Nikolić, Elad Alon, and Jonathan Bachrach. "A Methodology for Reusable Physical Design". In: *2020 21st International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2020, pp. 243–249.

[127] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. "Overview of the H. 264/AVC Video Coding Standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (2003), pp. 560–576.

[128] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[129] John Wright. "Physically Aware Design of Generated Systems-on-Chip". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2022.

[130] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. "SIMD2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation Beyond GEMM". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 2022, pp. 552–566.

[131] Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, Kevin Anderson, Borivoje Nikolic, and Krste Asanovic. "Instruction Scheduling in the Saturn Vector Unit". In: *arXiv preprint arXiv:2412.00997* (2024).

[132] Jerry Zhao, Daniel Grubb, Miles Rusch, Tianrui Wei, Kevin Anderson, Borivoje Nikolic, and Krste Asanović. *The Saturn Microarchitecture Manual*. Tech. rep. UCB/EECS-2024-215. Dec. 2024.

[133] Yuhao Zhu and Vijay Janapa Reddi. "High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2013, pp. 13–24.