Extending Temporal-Vector Microarchitectures for Two-Dimensional Computations

Colin Schmidt

Electrical Engineering and Computer Sciences
University of California, Berkeley

August 12, 2021

Extending Temporal-Vector Microarchitectures for Two-Dimensional Computations

by

Colin Schmidt

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Krste Asanović, Chair
Borivoje Nikolić
Robert C. Leachman

Summer 2021

Extending Temporal-Vector Microarchitectures for Two-Dimensional Computations

Abstract

Extending Temporal-Vector Microarchitectures for Two-Dimensional Computations

by

Colin Schmidt

Doctor of Philosophy in Computer Science

University of California, Berkeley

Krste Asanović, Chair

Modern computing is shaped by technology trends, like a slowing Moore's law and lack of Dennard scaling, as well as application trends, like mass application of machine learning. Technology has constrained modern computer architectures to focus on energy-efficiency in order to improve, battery life, total cost of ownership, and even performance. Emerging deep-learning applications require computation volumes that increase exponentially and yet change in structure substantially every few years. One solution for both of these problems is specialized programmable architectures, that can adapt to new applications while specializing for the commonalities, and thus improving energy-efficiency.

This thesis presents a set of two-dimensional architecture extensions for Hwacha an existing vector-fetch architecture designed to improve energy-efficiency on two-dimensional computation while remaining fully programmable. This thesis discusses the constraints modern CMOS process technologies place on such an architecture, and describes several silicon implementations of similar architectures. Finally, this thesis presents the physical implementation of such extensions and their realized energy-efficiency gains on select applications.

To Sloka, who always believed in me and made the long nights worth it.

Contents

List of Figures

List of Tables

Acknowledgments

The work in this thesis would not have been possible without collaboration with many students, staff and faculty at U.C. Berkeley. I would like to thank everyone who contributed time, ideas, or support to me throughout this process.

First, and foremost, I would like to thank my advisor, Krste Asanović, for his belief in me from the beginning to the end. His technical advice on vector architectures is second to none, and he provided me with expert guidance throughout my time at Berkeley. I'd also thank to thank Krste for giving me the freedom to explore so many different aspects of computer science, and his grand visions of projects that span the discipline. I would also thank to thank Bora Nikolic who, although not technically an advisor, played a crucial role in much of my work. In particular, his entrusting me with leading large chip projects, coaching me through copius chip paper submissions, and always encouraging me to find the thesis chapters in the work I was doing. Finally, I'd like to thank Robert Leachman as the final member of my thesis committee for his advice and support throughout this process.

I'd like to also specially thank Yunsup Lee and Ben Keller for being mentors to me in my Ph.D. work and in life. Yunsup brought me into the world of vectors, tapeouts, and making things work. Ben showed me how to build chips, manage a team, and do actual science. Without the guidance of both of them I would not be the researcher and individual I am today.

I've worked on several large projects during my time at Berkeley, each instrumental to my growth and an important part of the journey of my Ph.D. The Rocket-Chip generator was my first introduction to the team of amazing graduate students in the ASPIRE lab. All of the students I interacted with in the ASPIRE lab made it a welcoming, engaging, and exciting place to work, but some of the students I spent more time with deserve special mention. Andrew Waterman always provided excellent technical direction, personal guidance, and a friendly ear. Henry Cook helped me envision more expansive and radical projects, and show me what graduate student life could be. Scott Beamer demonstrated the power in writing papers, and that the path through Krste's group didn't always need to be microarchitecture. Chris Celio showed me that dedication to a single project can lead stellar results.

Simultaneously to the Rocket-Chip project, I began work on the fourth version of the Hwacha vector-fetch architecture and implementation. This project was lead by Yunsup but was truly a team effort with Albert Ou and myself, and led to a lot of the work during my Ph.D. The scope of this team project showed me that coming to Berkeley for the collaborative environment was an excellent decision. Getting to learn from Yunsup's in-depth knowledge of previous vector architectures and designs was priceless. Albert's detailed and methodical microarchitecting, and immense knowledge of unix systems and utilities always made me strive for better from my own designs and scripts.

The first pair of test-chips I contributed more than verification and testing to were Hurricane-1 and -2. The leadership of Ben Keller showed me the benefits of organization and I always strove to manage the Eagle chips half as well as Ben did the Hurricane chips. Palmer Dabbelt's never ending desire to write and re-write software to solve intensly complex

problems directly led to the tool building work I did after the Hurricane project. Howard Mao always had an impressive desire to write RTL whenever the larger projects architected themselves into a corner, and was a huge help.

The Eagle and Eagle-X projects were my first foray into more of a leadership role and I couldn't have done it without all of the team members. Neither chip would have been possible without the tireless work of John Wright. We spent many late nights on calls together debugging chip, flow, and tool issues. John was an indispensible partner and I'm glad we were able to barely hold each others sanity together throughout both chips. Zhongkai Wang and Eric Chang were excellent partners on the analog side of the Eagle project. Sean Huang was helpful during and after the tapeout process, and continues to help with getting Eagle-X tested.

The Hammer project was critical to getting the Eagle project out the door successfully, and I hope the tool itself can live on at Berkeley and beyond. Edward Wang, the lead designer of Hammer showed excellent leadership and technical vision for the core principles of Hammer. John and I wouldn't have been able to contribute nearly as much physical design expertise without Edward's thorough and flexible design. Daniel Grubb, and Harrion Liew have admirably taken up the mantle of Hammer development and I'm excited to see them use it for future tapeouts and papers.

The latest project I've been involved with at Berkeley is Chipyard. Chipyard is designed to integrate all of the previous projects in an easy to use manner to make test-chips like Eagle and Eagle-X easier for those outside Berkeley to build, test, and verify. Obviously many students are involved in this project but I'd like to thank a few specially. I've worked with Jerry Zhao since he was an undergrad and he has shown nothing but dedication, intelligence and amazing execution throughout the many projects he has worked on. It has also been great to work with Abe Gonzalez, and see him learn to manage an entire tape out himself. David Biancolin has shown the consistency and commitment to his ambitious goals that has helped drive me to the finish line. Albert Magyar has always been a friend, a great discussion partner, and an excellent cube-mate. Sagar Karandikar's leadership of the Firesim project has shown me how to manage a wildly popular open-source project, and has always been generous with his time. I'd also like to thank the system administrators in both the ADEPT/ASPIRE lab, Kostadin Ilov, and the BWRC, Brian Richards, for dealing with all of my requests and keeping everything running smoothly even when I was working feverishly to overload them. The process of graduating with a Ph.D. is not a simple one but the administrative staff of the lab and the department have made it smooth and painless. Thank you to Roxana Infante, Tamille Chouteau, Ria Briggs, and Jean Nguyen for all their assistance in helping me reach the end of this journey.

Graduate school has been a long journey for me and I would not have been able to succeed without the love and support of my friends and family. I've found many new friends on the west coast but I was lucky enough to bring one of my best all the way from Cornell. Adam Izraelevitz, has grown from an early lab partner to an amazing friend. I relish any opportunity to work together with him and deeply admire his thoughtful and caring response to any situation. I'm also grateful for all the other friendships I've made and continued during graduate school. They've helped me keep a balance between life and work even when all I can think about is research.

My entire family has always been extremely supportive of me and I know are waiting excitedly yet patiently for this work to be done. My parents have always been believed in me, and encouraged me endlessly to pursue my education. I think without the drive enstilled in me by them from a young age this journey would have been nearly impossible. Thank you so much for everything. My brother was the instigator of my entire computing career. By showing me his school projects he got me interested in programming, and the love and support he has shown me throught all of my schooling has been wonderful. I'd also like to thank my wife Sloka's family for their support and their ceaseless interest in understanding even the most complex portions of my work. Finally, I'd like to thank my wife Sloka. Our journey together to california wasn't planned in advance but it makes me feel like the luckiest man on earth every day. Thank you for dealing with the late nights working and writing, and for making sure I got the job done in the end. Thank you for all of your love and support.

Chapter 1

Introduction

Computing has seen a rapid and fundamental shift in the past decade, away from single companies dictating the direction of the entire industry, and towards a more meritocratic space where ideas are easier than ever to prove out and the number of companies producing custom hardware has rapidly expanded. This is the result of many factors but the largest factors are likely the end of Dennard scaling in the mid 2000s, and more recently, a potential slowing of Moore's law. The cost of producing a new design in cutting edge nodes [61] and the reduced gains of moving to new nodes has encouraged a proliferation of designs on technology nodes several steps behind.

In addition to the underlying technology changes, the rise of open source hardware designs, in particular RISC-V, has led to a revitalization of explorative architectures. Designers now have the ability to build on top of existing hardware and software stacks, leveraging the work of large communities to make their designs more effective and usable. This time and cost savings allows for more focus on innovation rather than focus on catching up to the state of the art.

Lastly, the near-singular focus of modern applications on machine learning has enabled more specialized designs to proliferate. Deep learning is now being applied to nearly all applications in computing and requires orders of magnitude more computation than previous methods. Applications of deep learning are thus required to be highly concerned with performance, and as a consequence, energy efficiency. Designers then can direct their specialization at a relatively small problem, deep learning, and still see wide adoption across applications and also a strong need for their improvements in efficiency.

This chapter describes the recent trends that have lead to this shift, how computer architects have adapted, and the role programmable data-parallel architectures are playing in this transformation. This thesis provides a particular method, and example, of building programmable architectures designed to exploit these industry trends and future directions. Finally, section 1.3 outlines the contributions of this thesis, and the remaining chapters.

1.1 Energy Efficiency, Specialization, and Programmability

The slowing, and eventual end, of Dennard scaling means that energy efficiency is now the most important metric for a new design or architecture. In any power limited setting, mobile handset, desktop computer, server rack, the energy efficiency has become a proxy for performance. With a fixed power budget increased performance can only occur alongside an increase in performance per watt. And now all settings have become power limited, even supercomputers, as the cooling and power distribution problem for individual components continues to be a difficult problem to solve in the general case.

This change in focus for systems from pure performance to performance per watt has had a great impact on computer architecture. There must now be a continuing focus on getting every operation possible out of a given set of transistors. Specialization, where a design reduces the set of computations it supports, naturally eliminates large sets of operations that are no longer possible. The elimination of these other operations greatly improves the efficiency of the design as long as it is handling appropriate computations. A combination of distinct designs each focused separate types of computation can then compose into a system with higher overall energy efficiency.

As a consequence the number of specialized designs or hardware block in common products has proliferated dramatically in the last decade [78]. The integration of these diverse blocks can cause many problems that reduce the efficiency gains from adding additional blocks. Improving the interconnect between the blocks, or the mechanism by which data is shared between blocks, can alleviate some of these issues. However, a different approach, in which blocks are combined or new blocks are proposed that maintain some diversity in computational support via software, can also improve on efficiency of the system. In addition, these flexible designs also address the fundamental challenge of specialization, by enabling the applications supported to differ or change in ways that aren't known at design time.

As the advanced nodes have become more expensive and design cycles become longer the lack of flexibility has become a larger disadvantage for fixed-function accelerators. Applications continue to change faster as computers are integrated into ever more aspects of life. The rapid change in applications becomes critical to system design when the applications also demand a large amount of computation. In these cases, e.g. machine learning, a fixed function design's efficiency is needed but the rigidity would make designs obsolete soon after they are available. As a result, there has been a focus on programmable accelerators for those workloads that demand high computing power.

1.2 General-Purpose Data-Parallel Programmable Specialization

The primary work of this thesis is related to these programmable accelerators, their development, and implementation. The approach is essentially a middle-ground straddling the extreme demands for efficiency of modern applications, and the rapid development of new and unique applications which necessitates flexibility. These architectures should not be fully general or the benefits will be lost, so a common relatively large domain is computation on multi-dimensional data. This application domain covers not only deep learning, but also image and video processing, and scientific compute.

Many different implementations of such extensions exist by using various data-parallel architectural paradigms as their baselines. The work in this thesis utilizes the vector-fetch paradigm as its baseline rather than the more common paradigms recent commercial designs have been built with. The decoupled and runtime configurable nature of the vector-fetch architecture enables the desired specialization, for two primary reasons. First, the decoupling ensures that well written software is already built not to expect low latency communication with the data-parallel portions of the code. Expecting a larger latency enables more flexibility in performing longer running specialized tasks without hindering common software applications. Second, the runtime configuration of vector-fetch architectures ensures that programmers are already setting up the machine based on their current application. This setup code can then be reused or replaced to enable certain specialization features or other energy-efficiency improvements. Finally, a temporal microarchitecture for the vector-fetch paradigm can further enable specialization by giving the freedom to process the multiple dimensions of the data over time without significantly changing the normal control flow of the design.

1.3 Thesis Outline

The confluence of these factors leads to this thesis' central hypothesis, that a temporal vector-fetch microarchitecture would be an excellent candidate for multi-dimensional specialization on top of an existing programmable data-parallel architecture.

- Chapter 2 provides background on popular data-parallel architectures, comparing features and goals of each architecture. This includes representative code samples and a brief analysis of the architectures' programming models.

- Chapter 3 describes the history and current state of two-dimensional and some multi-dimensional architectures. A brief summary of multi-dimensional applications with a focus on machine learning is also provided.

- Chapter 4 presents the challenges associated with building data-parallel architectures in modern advanced technology nodes. An analysis of the constraints placed on such

architectures and techniques to mitigate or address these constraints is presented. The chapter also includes a description of the Hwacha architecture, microarchitecture, and the design decisions made during its implementation.

- Chapter 5 accounts several silicon test-chip implementations of the Hwacha architecture. The technologies and methodologies used to realize these implementations are presented. In addition, a description of physical design challenges for each chip and their solutions is included. Finally, a summary of results and measurements for each design is presented.

- Chapter 6 describes a two-dimensional extension to the Hwacha architecture, and how it would interact with the architecture's constraints. Also included is the encoding, operation, and a code sample for each of the new instructions. The chapter concludes with a brief discussion of possible future extensions and their design considerations.

- Chapter 7 presents the implementation of the two-dimensional extension for both the microarchitecture and the physical design aspects. A brief discussion of the previous potential future extension's implementation is also included. Finally, the chapter concludes with a presentation and analysis of the performance and energy-efficiency of different operations on the baseline and extended implementation.

- Chapter 8 concludes the work by summarizing the contributions of the thesis and providing a set of topics suitable for future exploration of two-dimensional programmable architectures.

Chapter 2

Data-parallel Architectures

A classical approach to improving performance in computer architecture is to focus on parallelism and attempt to do as much work as is reasonable at once. This exploitation of parallelism is often categorized into instruction-level parallelism (ILP), data-level parallelism (DLP), thread- or task-level parallelism (TLP) [38].

These types of available parallelism in computer programs has fostered many specific design paradigms, from ILP sparking very-long instruction-word (VLIW) and out-of-order execution (OoO) machines, to TLP spawning simultaneous multi-threading and many core processors, and DLP has proven no different. The primary data parallel design paradigms can be classified as fixed-width packed single-instruction multiple-data (SIMD), many-threaded single-instruction multiple-thread (SIMT), and variable-length vector. Nearly all computer systems manufactured today include some component designed for DLP. Even small deeply embedded processors often have small SIMD or digital-signal processing (DSP) units [100], small consumer devices like cell phones or tablets include SIMD units, graphics processing units (GPUs), and lately machine learning accelerators [3], server class processors have high performance SIMD implementations sometimes with on-chip GPUs [4], and high-performance compute clusters include even higher performance SIMD or vector units[99].

This chapter identifies characteristics common to each of these DLP paradigms, enabling a comparison and an understanding of the design-space. Sections 2.1, 2.2, and 2.3 describe the three primary paradigms from above, some of their implementation trade-offs, their primary use cases, and how they are intended to change over implementation iterations. The chapter continues by giving the same treatment to some of the newer and less common paradigms. Section 2.4 covers another newer paradigm, implementation-time variable-length vector packed-SIMD, which is exemplified by ARM's scalable vector extensions (SVE). Section 2.5 describes the latest data-parallel architecture the RISC-V vector extension and how it mixes some of the above paradigms. And finally, section 2.6 describes an additional more uncommon paradigm, vector-fetch, which is used by Hwacha the baseline architecture for the extensions described in the remaining chapters.

In order to discuss the differences in these architectural paradigms and how it affects their programming models a single example code will be compared across each of them. This frag-

```
void csaxpy(size_t n, bool c[], float a, float x[], float y[])
{
  for (size_t i = 0; i < n; ++i)
    if (c[i])
      y[i] = a*x[i] + y[i];
}
```

Figure 2.1: CSAXPY as C code with a single vectorizable loop.

```
csaxpy_scalar:
stripmine_loop:
        lb      t0, 0(a1)           % Load c[i]
        beqz    t0, skip            % branch around work
        lw      t1, 0(a3)           % Load x[i]
        lw      t2, 0(a4)           % Load y[i]
        fma     t2, t1, a2, t2      % y[i] += a * x[i]
        sw      t2, 0(a4)           % Store y[i]
skip:
        add     a1, a1, 1           % update c pointer by 1 byte
        add     a3, a3, 4           % update x pointer by 4 bytes
        add     a4, a4, 4           % update y pointer by 4 bytes
        subi    a0, a0, 1           % update remaining elements
        bnez    a0, stripmine_loop  % continue loop
        ret
```

Figure 2.2: CSAXPY kernel mapped to a scalar architecture.

ment implements a simple linear algebra routine, and common benchmark CSAXPY, that multiplies a scalar a times a vector x, of length n, and sums that result with another vector y, this multiplication and summation are conditional on a third vector c. A non-vectorized implementation of this loop is presented in Figure 2.2, and all following implementations will follow the same calling convention. The number of elements n is passed in the first argument register $a0$, a pointer to the byte-packed condition vector c is passed in $a1$, the scalar value a is passed in $a2$, and the two vectors x and y are passed in $a3$ and $a4$ respectively. The scalar implementation is relatively straightforward, branching around the loads and math based on the condition, and unconditionally updating all the pointers and loop count.

There exist more optimized implementations of this loop using unrolling and or software pipelining but they would vary based on the specific implementation. One of the benefits of many of these data-parallel implementation strategies is that a relatively simple code can scale it's performance based on the underlying architecture, and so the codes presented

below maintain this style of straightforward implementation. In addition, rather than using the assembly syntax of any particular architecture from the paradigms, and be forced to introduce new syntax for each code example, the code presented below is in a generic assembly syntax highlighting only the important distinctions.

2.1 Fixed-Width Packed-SIMD Architectures

This architectural paradigm is one of the earliest data-parallel architectures [24], and one of the most popular due to Intel's adoption of it for their data-parallel extensions. The core idea of this paradigm is to enable the user to reference subdivisions of a larger register, hence the packed moniker, in computations. This is a popular methodology because it initially allowed the use of registers that were already available and simply added parallelism to the handling of the register's bits. Over time, many implementations have added dedicated registers for this purpose as they grew in size past the natural word length of general purpose processors. Functional units such as multipliers scale quadratically with input bit-width while the number of subdivisions of a register only scales linearly so adding the required number of functional units to process an entire register's subdivisions at once can be smaller than the functional unit that uses all of the registers bits as a single input. These additional functional units allow for an increase in operations per cycle assuming the computation can make use of smaller, sub-register, data types. Additional instructions to handle these wider registers and common DLP operations are often added to make it easier to map high-level code to the architecture.

Figure 2.3 shows CSAXPY written for a fixed-width packed-SIMD architecture. Because the architecture has a fixed width of four words for each register, there is a fragment of code, elided above, at the end of the loop that would handle the remaining one, two, or three elements. This fixed width also shows up in the specific opcodes used, and in the register names. In addition, because the architecture does not support interaction between the scalar registers and the vector registers the user must explicitly replicate the scalar value into a vector register. Recently fixed-width SIMD architectures adopted full predication [27], and so are able to avoid some extra processing that has been unavoidable in the past.

In order to understand how a data parallel architecture will evolve over time it is useful to imagine how a current algorithm or code would need to be changed for new implementations. For example, what would happen if a new implementation were to be built with more or less processing power available. In the packed-SIMD approach if an implementation wants to perform more operations per cycle the instruction-set-architecture (ISA) will need to be extended to include a new set of opcodes for the wider registers and increased operations. This slow but constant increase in opcodes as a packed-SIMD architecture evolves creates several issues. Old binaries that are run on newer machines will not take advantage of the improved performance, and will need to be recompiled or potentially rewritten. Also because new opcodes continue being added they will tend to be longer in bit-width, the latest Intel extension AVX512 required an extra byte of prefix. This prefix $EVEX$ is 4 bytes compared

```
csaxpy_simd:
    slli    a0, a0, 2                   % Scale number of elements to number of bytes
    add     a0, a0, a3                  % Calculate end address for loop body
    vsplat4 v4w0, a2                    % Fill a vector register with replicas of a
stripmine_loop:
    vlb4    v4w1, (a1)                  % Load the condition vector c[i:i+3]
    vcmpez4 vp0, v4w1                   % Fill a predicate register based on c
!vp0 vlw4    v4w1, (a3)                 % Load x[i:i+3] under the c predicate
!vp0 vlw4    v4w2, (a4)                 % Load y[i:i+3] under the c predicate
!vp0 vfma4   v4w1, v4w0, v4w1, v4w2 % y[i:i+3] += a * x[i:i+3] under the c predicate
!vp0 vsw4    v4w1, (a4)                 % Store y[i:i+3] under the c predicate
    addi    a1, a1, 4                   % Increment c pointer by 4 elements
    addi    a3, a3, 16                  % Increment x pointer by 4 elements
    addi    a4, a4, 16                  % Increment y pointer by 4 elements
    bleu    a3, a0, stripmine_loop
% handle edge cases
% when (n % 4) != 0 ...
    ret
```

Figure 2.3: CSAXPY kernel mapped to the packed-SIMD assembly programming model.

to the shorter two and three byte VEX prefix used by AVX and AVX2. [45]. These prefixes encode operand and operation type but do not include the arguments so each instruction will be significantly longer.

Overall, the packed-SIMD paradigm is popular but has several downsides, some of which could be avoided if legacy codes could be left unsupported. The remaining paradigms all have mechanisms to avoid this central problem of directly encoding the width and number of elements in the architectures opcodes.

2.2   Many-Threaded SIMT Architectures

Another highly prevalent architecture is the many-threaded SIMT architecture implemented in some GPUs, including Nvidia's standalone GPUs, ARM's mobile GPUs, and AMD's integrated GPUs. This architecture was initially designed to enable per-pixel operation and so is focused on independent control of each thread. This design lends itself to an efficient and easy to scale physical design, given a large set of independent problems. The engines that handle threads will be replicated many times on a single chip and so the physical design of these units can be very detailed and well optimized, when compared to a more monolithic packed-SIMD engine that is only a portion of a general purpose processor. Physical design and scaling have often lead to SIMT architectures being one of the first designs on new

```
csaxpy_simt:
  mv    t0, tid          % Retrieve the current thread id
  bgeu  t0, a0, skip     % Branch around work for fringe elements
  add   t1, a1, t0       % Setup c pointer
  lb    t1, (t1)         % Load c[tid]
  beqz  t1, skip         % Branch around work for masked elements
  slli  t0, t0, 2        % Scale elements to bytes
  add   a3, a3, t0       % Setup x pointer
  add   a4, a4, t0       % Setup y pointer
  lw    t1, (a3)         % Load x[tid]
  lw    t2, (a4)         % Load y[tid]
  fma   t0, a2, t1, t2   % y[tid] += a * x[tid]
  sw    t0, (a4)         % Store y[tid]
skip:
  stop
```

Figure 2.4: CSAXPY kernel mapped to the SIMT assembly programming model.

technology nodes and therefore one of the least-expensive options in terms of $GFLOPS/W$ which is a component of total cost of ownership and therefore cost [73].

Figure 2.4 shows CSAXPY written for a SIMT architecture. The SIMT architecture relies on independent threads and a scheduler distributing these many threads to identical resources so the implementation looks similar to the scalar implementation. The largest differences are the lack of a loop, because the parallelism is implicit, and in the presence of a branch based on thread id to avoid executing any tail elements. One major drawback with this model is that the loads and stores are not expressed as vector loads in the code. Instead the microarchitecture is responsible for coalescing these individual loads and stores into more efficient operations for the memory system if the performance is to be recovered. In addition, the address calculations for these loads and stores is done in each thread and must be materialized into each thread's register file. This code example does not show the outer control processor code that would setup and launch this code onto the pool of SIMT execution units. This portion of code is highly variable per implementation and often includes privileged code, as the SIMT units are often exposed as devices. As with the edge case code for the packed-SIMD paradigm, the hiding of this code an optimistic view of the SIMT paradigm but focuses the comparison on the data-parallel portions of code.

The benefit of this many-threaded model is that with so many threads the microarchitecture is free to use each pipelines resources for any number of other threads if one becomes stalled on memory or another long latency operation. This does mean that in order to operate efficiently SIMT architectures often require large amounts of work to be schedule at once, as no individual thread is designed to be able to execute to completion without stalling.

This paradigm partially addresses the issue of direct bit-width encoding but the indepen-

```
csaxpy_tvec:
stripmine_loop:
      vsetvl  t0, a0              % Set vector length for loop
      vlb     vv0, (a1)           % Load vl elements from c
      vcmpez  vp0, vv0            % Setup predicate for c[i:i+vl]==0
!vp0 vlw      vv0, (a3)           % Load y[i:i+vl] under c predicate
!vp0 vlw      vv1, (a4)           % Load x[i:i+vl] under c predicate
!vp0 vfma     vv0, vv0, a2, vv1   % y[i:i+vl] += a * x[i:i+vl] under c predicate
!vp0 vsw      vv0, (a4)           % Store y[i:i+vl] under c predicate
      add     a1, a1, t0          % Update c pointer by elements completed
      slli    t1, t0, 2           % Convert number of elements to bytes for x and y
      add     a3, a3, t1          % Update y pointer by elements completed
      add     a4, a4, t1          % Update x pointer by elements completed
      sub     a0, a0, t0          % Reduce remaining number of elements
      bnez    a0, stripmine_loop
      ret
```

Figure 2.5: CSAXPY kernel mapped to the traditional vector assembly programming model.

dent nature of the threads still incurs some redundant processing and is unable to represent the uniform nature of DLP operations. The remaining paradigms are able to resolve both this representation issue and the direct encoding problem by using a programmatic number of elements to be processed.

2.3   Traditional Vector Architectures

Another type of data-parallel architectures that have been around for a long time are traditional long vector machines pioneered by Cray in supercomputers in the 1980s [74]. These architectures are characterized by run-time variable-length vectors, often taking multiple cycles for a single operation to complete, hence the long vector distinction. Currently these architectures are less common but do still show up in supercomputers [99] as at their origin. Eventually architectures like the IBM VF [16] added programmatic vector-lengths to this paradigm, which enables different physical implementations of the same ISA can run the same binary code. This enables a designer to produce many more design points along the performance spectrum without linearly increasing the amount of software needed for the designs. Longer vector architectures can also be more tolerant of memory system latency and have more freedom in their pipeline microarchitecture since the number of instructions need per application-level operation is greatly reduced.

Figure 2.5 shows the same CSAXPY kernel implemented on a hypothetical traditional vector architecture. Since this model allows for an explicit, but runtime varied, number of

elements to be processed at once it avoids several issues from the fixed-width packed-SIMD and SIMT designs. The initial setting of vector length vl sets the number of elements for the remainder of the loop. This setting can be an arbitrary value so the need for code to handle the tail of the vector is eliminated. In addition, because the vector length is explicit there is no need for extra management code or structures to figure out how many threads to launch as in the SIMT designs. The explicit parallelization also means that the memory operations explicitly encode the access patterns without needing something to recover and coalesce the accesses. Unlike the previous two paradigms there is no extra code needed for this function to run completely and correctly, which shows another benefit of this paradigm.

One issue with this model is that there are often a few extra instructions inside the loop to setup vector length, but this is offset by the lack of tail code. In addition, depending on the constraints of the specific ISA, the programmer may be able to assume a certain vector length is always available and for short application vectors avoid some of these extra instructions. Another issue is that modern compiler infrastructure is mostly used to dealing with fixed-width parallelization, or has special code for SIMT machines and uses the GPU runtime code to manage the variable amount of parallelism [64] [43]. And lastly without additional ISA features it can be difficult to fully utilize the register file particularly with small bit-width data types or mixed-precision computations.

This paradigm is able to avoid the fixed encoding, and redundant computation problems of the previous two models but still leaves room for improvement with regards to mixed-precision computations. A solution that solves all of these problems elegantly is not widely accepted and the final three paradigms discussed each take a different approach to work towards such a solution.

2.4   Implementation-time variable-length vector packed-SIMD Architecture

This paradigm introduces variable-length vectors that unlike the traditional vector architectures do not vary at runtime but vary at implementation time, and is embodied by the new ARM vector architecture SVE. Each implementation is allowed to have a different total number of bits in each vector register. The register size variance is quantized to 128 bits in order to restrict the ISA design and to make software easier to write, and has an upper limit in the architecture of 2048 bits. In order to provide the flexibility for arbitrary-length vectors the architecture provides predication and many specialized operations to set and update these predicate registers.

Figure 2.6 shows the CSAXPY kernel mapped to an implementation-time variable-length vector packed-SIMD architecture. The code shows the hybrid nature of the paradigm by needing to refer to subsets of vector registers with the .b and .w suffix, but not explicitly encoding the number of elements each iteration of the loop will process. The use of predicates as the means for programmatic vector length requires multiple named predicate registers and

```
csaxpy_sve:
      mov     t0, 0                    % Initialize i
      whilelt vp0, x0, a0              % Initialize vl predicate for indexes < n
!vp0 splat    v2.w, a2                 % Splat scalar into vector reg under vl predicate
stripmine_loop:
!vp0 vlb      v0.b, (a1)               % Load elements from c under vl predicate
!vp0 vcmpez   vp1, v0.b                % Setup predicate for c[i:i+vl]==0 under vl predicat
!vp1 vlw      v0.w, (a3)               % Load vl elements from y under c predicate
!vp1 vlw      v0.w, (a4)               % Load vl elements from x under c predicate
!vp1 vfma     v0.w, v0.w, v2.w, v1.w % y[i:i+vl] += a * x[i:i+vl] under c predicate
!vp1 vsw      v0.w, (a4)               % Store vl elements of y under c predicate
      incvb   a1                       % Update c pointer by elements completed
      incvw   a3                       % Update y pointer by elements completed
      incvw   a4                       % Update x pointer by elements completed
      incvw   t0                       % Update remaining number of elements
      whilelt vp0, t0, a0              % Update vector length predicate
      bnempty vp0, stripmine_loop      % Continue loop if there are more elements
      ret
```

Figure 2.6: CSAXPY kernel mapped to the implementation-time variable-length vector architecture.

a set of dedicated predicate generation instructions like *whilelt*. These predicate instructions are essentially substitutions for the *vsetvl* instructions of the traditional vector paradigm. Because the paradigm does not expose the number of elements explicitly in a scalar register, special increment instructions are included for different data widths, and branching must be based on the predicate register.

The primary benefit of this architecture is the ability to retain the sub-register SIMD programming model while enabling implementations to scale to different performance points. This scaling also enables more portable software that can maintain performance across implementations.

Unfortunately, there are still a few issues effectively using all the bits in these wide registers. When mixing different width data types extra instructions will often be needed to split the wider vector into multiple registers, and then align the narrower vector with more splits. In the code example above if the result were meant to be accumulated into a double-precision register the instruction count more than doubles for the inner loop.

This is evidence that despite approaching a better solution this paradigm still has potential for improvement. The last two architectures discussed move closer to the traditional vector style but with better support for mixed precision and sub-word parallelism.

```
csaxpy_rvv:
stripmine_loop:
    vsetvl  t0, a0, e8,m1      % Set element to bytes
    vlb     v1, (a1)           % Load vl elements from c
    vcmpez  v0, v1             % Setup v0 as a mask for c
    vsetvl  x0, a0, e32,m4     % Set element to words keeping SEW/LMUL constant
!v0 vlw     v1, (a3)           % Load y[i:i+vl] under c predicate
!v0 vlw     v2, (a4)           % Load x[i:i+vl] under c predicate
!v0 vfma    v1, v1, a2, v2     % y[i:i+vl] += a*x[i:i+vl] under c predicate
!v0 vsw     v1, (a4)           % Store y[i:i+vl] under c predicate
    add     a1, a1, t0         % Update c pointer by elements completed
    slli    t1, t0, 2          % Convert number of elements to bytes for x and y
    add     a3, a3, t1         % Update y pointer by elements completed
    add     a4, a4, t1         % Update x pointer by elements completed
    sub     a0, a0, t0         % Reduce remaining number of elements
    bnez    a0, stripmine_loop
    ret
```

Figure 2.7: CSAXPY kernel mapped to the RISC-V vector extension programming model.

2.5   RISC-V Vector Architecture

The RISC-V vector extension (RVV) has a new architectural model that is similar to traditional vectors with some elements of the fixed width packed-SIMD architectural paradigm. This discussion is based on the 1.0 version of the specification which has changed dramatically from the early versions, over the few years of specification history.

The general programming model is still one of explicit runtime configurable vector length. However, rather than the location of these elements being opaque to the programmer the location of the bits are made explicit, as with the previously discussed architectures. This allows for using the same registers as different bit-width values without clearing the machines configuration. The vector unit is still configured before operation, but it is lightweight and does not cause any register state to change. The main components of the configuration are the current element width (SEW), and the vector length multiplier (LMUL), which determines the number of physical registers present in a register group. Rather than encoding the width of elements explicitly in the instruction opcode as in all previous architectures, encoding space is saved by using the ahead of time configuration (SEW) to specify the element bit-width for most operations. There are many exceptions to this rule especially with respect to memory operations, but most arithmetic operations do not explicitly encode the bit-width of operations. Some arithmetic operations consume or produce elements of a twice as wide or twice as narrow bit-width relative to the current element width, which along with changing the current element width allow for relatively efficient mixed precision code.

Figure 2.7 shows the CSAXPY kernel mapped to the RISC-V vector extension. The noticeable differences in this sequence are the additional *vsetvl* instructions needed before operations on different data widths, and the use of a regular vector register $v0$ as the predicate. The extra configuration instructions allow for the computation instructions to be encoded without an explicit data type. The vector length multiplier allows for wider data types to use more architectural registers while maintaining mixed-precision computation by matching the ratio of multiplier to element width.

The primary benefit of this architecture over the previous designs is that it enables more complete use of the vector register file with the vector length multiplier, without using additional instruction bandwidth. This architecture still allows for programmatic vector length changes and predication for more complex control flow, maintaining most of the benefits of the previous architectures. Considering the code change from above where the accumulator is now double precision, the addition of widening operations means that only a single instruction needs to be added to the loop.

One downside of this architecture is that the data layout for the vector register file is exposed which puts some limits on the microarchitecture. This is a clear trade-off to allow the kinds of sub-register SIMD along with mixed-precision operation in a small amount of encoding space. In addition, the programmer's model of execution is more complex than the original three paradigms due to the mixing of explicit and implicit element widths and potential non-obvious configuration constraints.

Overall, the RISC-V vector extension design provides a very attractive architecture for general purpose data-parallel execution. It provides good code efficiency, expressive primitives, and allows for many differently performant implementations. The last data-parallel architecture to discuss provides similar features but makes some concessions on code size to provide more room to experiment with new architecture ideas, and is thus the basis for the work in the later chapters.

2.6   Vector-Fetch Architecture

The vector-fetch architectural paradigm [56] is a non-standard scheme that moves the traditional vector instructions out of the scalar control thread and into separate blocks, unlike all of the previous paradigms discussed. The goal of this change is to more effectively decouple the vector operations from the scalar operations. This decoupling allows the scalar core to run ahead calculating addresses and loading scalar data and resolving control flow that will be needed for future iterations of the vector execution loop. One crucial idea that allows the decoupling to occur is that several of the setup instructions including the *vsetvl* instruction exist in the scalar control stream and the vector unit can calculate the available vector length regardless of the code block that will be sent later.

In the vector-fetch paradigm, the control thread is responsible for setting up the vector unit by configuring its registers (*vsetcfg*), moving scalar data and addresses into the vector register file (*vmcs*, *vmca*), setting the vector length (*vsetvl*), and launching the worker

```
csaxpy_hwacha:
    vsetcfg t0                   % Set up vector register for number and precision
    vmcs vs1, a2                 % Move the scalar a to vector unit
stripmine_loop:
    vsetvl  t0, a0               % Set vector length for loop
    vmca va2, a1                 % Move &x to vector unit
    vmca va0, a3                 % Move &y to vector unit
    vmca va1, a4                 % Move &c to vector unit
    vf      csaxpy_vf            % Launch vector-fetch block
    add     a1, a1, t0           % Update number of elements completed
    slli    t1, t0, 2            % Convert number of elements to bytes for x and y
    add     a3, a3, t1           % Update x pointer
    add     a4, a4, t1           % Update y pointer
    sub     a0, a0, t0           % Update number of remaining elements
    bnez    a0, stripmine_loop   % Continue if needed
    ret


csaxpy_vf:
    vpset vp0                    % Initalize predicate register
    vlb vv2, (va2)               % Load c[i:i+1]
    vcmpez vp0, vv2              % Setup predicate for c[i:i+vl]==0
!vp0 vlw vv0, (va0)             % Load vl elements from y under c predicate
!vp0 vlw vv1, (va1)             % Load vl elements from x under c predicate
!vp0 vfma vv0, vv0, vs1, vv1    % y[i:i+vl] += a * x[i:i+vl] under c predicate
!vp0 vsw vv1, (va1)             % Store vl elements of y under c predicate
    vstop
```

Figure 2.8: CSAXPY kernel mapped to the vector-fetch programming model.

thread's code block (vf). The worker thread contains the memory, compute, and predicate operations. These worker thread instructions are fetched from memory, decoded, and then executed in an independent unit.

Figure 2.8 shows the same CSAXPY kernel implemented on the vector-fetch architecture. It contains two sets of instructions: one set for the control thread, and a second set for the vector worker thread. The control thread instructions are similar to a traditional vector architecture with the addition of the scalar and address data moves, and the vector-fetch launch. The worker thread instructions are also very similar but with a *vstop* instruction at the end to denote the end of the worker thread. After the stop, the vector unit is idle until another vector-fetch block is issued and could be clock-gated or potentially power-gated.

This paradigm also supports the same binary to have extended vector lengths on new im-

plementations, and additionally supports configuring the number and data-widths of vector registers This vector register configuration allows for efficient use of the register file as well as straightforward mixed-precision computation. Changing the summation array to double-precision does not increase the number of instructions in the loop at all, as the configuration instruction is the only one that changes.

The obvious downsides of this model are the increased code size, and the required moves for scalar data used in the vector unit. In addition, the separate stream of vector execution will increase the latency for short vector sequences more commonly used in non data-parallel code.

However, the vector-fetch architectures separation of the vector unit and loose encoding allow for easy experimentation with new architectural or microarchitectural ideas. In addition, an open-source implementation of the entire vector-fetch design significantly reduces the overhead of establishing a baseline architecture for further experiments. Overall, the vector-fetch architecture provides an excellent platform to propose and evaluate more data-parallel extensions.

Chapter 3

Multi-Dimensional Vector Applications and Architectures

The definition of DLP does not restrict the parallelism to single dimensional operations and is often used for dense linear algebra where the dimensionality of data can be very high. Fortunately, all these high-dimension operations can be decomposed into one dimensional vector operations or even scalar operations, and so can be computed on even the simplest of hardware. The downside to this decomposition is an inherent loss of context. When these computations are broken down into their elements and run individually on a piece of hardware the pattern that is expressed by the higher-level operations is lost. The hardware no longer be aware that it is processing rows of a matrix or slices of a tensor. These patterns, when exploited, can provide efficiency and performance improvements in myriad ways, from not materializing intermediate results to generating fewer memory requests. This lure of efficiency and performance has fostered many research projects, products, and architectures to move the data-level parallelism abstractions above simple one dimensional operations.

This chapter describes the fundamental components of some popular multi-dimensional applications, how these applications can be mapped to hardware, and what types of hardware has been designed for these applications. Section 3.1 discusses some of the original array-based SIMD designs that often included a 2D interconnect, but were widely varied in the programming models and supported operands. Section 3.2 describes the linear algebra routines that eventually standardized what operations data-parallel architectures were expected to provide and perform well on and are the basis of most multi-dimensional compute. Section 3.3 discusses the core set of operators that have been used to build most of the prevalent machine learning architectures, and how these relate to the linear algebra routines.

With these applications in mind and the knowledge that they continue to demand more computing resources year after year, much time has been spent developing custom architectures or additional functional units in existing architectures that can accelerate these applications. Section 3.4 concludes the chapter by discussing the different extensions that have been explored and how they impact the architecture and programming model.

3.1 Origins of Two-Dimensional Architectures

Single-instruction multiple-data architectures and two-dimensional architectures have a long shared history. An early branch of designs was taking advantage of the physical world to lay out physically distributed components in a grid or array. These architectures had their physical structure built into their programming model with nearest-neighbor connections, and the applications envisioned for them. This specialization also extended to the processing elements (PEs) initially with bit-serial implementations being very popular as it allowed applications to map more efficiently to the array regardless of its bit-width. As with many designs constructed during the early decades of computing compatibility between different designs was non-existent, but eventually standard word-length processing elements became popular, presaging large networked computers as in data-centers and supercomputers. As with many computing designs increasing levels of device integration leads to more PEs being packed into each package and the 2D grid was replicated in on chip connections. These machines paved the way for both the modern fixed function processing arrays, the more general purpose 2D architecture extensions, and even reconfigurable gate arrays, and processing in memory.

An early 2D machine, the Illiac-IV, was designed to tackle many of the applications mentioned at the start of this chapter, linear algebra, scientific computing and signal processing [12]. At the time explicit parallelism was rare, but the designers recognized it would help increase the computing power of the machine such that their large target applications would be tractable. Having been designed before large scale on-chip integration was possible the physical structure of the machine matches the logical structure, a homogeneous grid of processing elements with connections to its neighbors. This introduced several ideas that eventually became common for SIMD architectures. The Illiac-IV is capable of using predicates based on elements as control flow. The addressing scheme is also based on a local PE register, making its addressing computations similar to those of SIMT machines. However, it avoids some of the redundant storage and computation of the SIMT paradigm with a constant broadcast bus that each PE can use directly, which is more similar to the traditional vector designs. Finally, there is was lot of focus on decoupling the control unit for the PEs which is similar to the vector-fetch paradigm. All of these one-dimensional data-parallel concepts are augmented by the 2D interconnect between PEs. These direct connections enable more complex algorithms such as matrix inversion to proceed directly on the PE without higher level coordination [57]. However the 2D organization is mostly used to speed up long routing as most algorithms use the memory as the 2D structure holding an array but spread columns among the PEs in a more linear fashion.

Other early machines experimented with the connectivity of the array and the bit-width of the PEs, to improve utilization. The Distributed Array Processor (DAP), reduces the bit-width of each PE to a single bit, and adds several other features to address low utilization scenarios [72]. The limits that apply to array processors are still present in today's architectures and include: non-uniform memory access for non-local storage; small problem sizes underutilizing the array; idle time during setup due to I/O, configuration, or loading;

and edge cases in computation due to reductions or boundary conditions leaving most of the array idle. The bit-serial approach of the DAP aims to help problem size array matching. The lower cost of a bit-serial PE enables more memory to be attached to each PE, allowing for fewer slow non-local accesses, and more problems to fit entirely in the system removing the need for reconfiguration. An additional neighbor connection, half way across a row and column, increases the speed of data transfer among the PEs. Because not all data movement in 2D applications is regular, Amdahl's Law warns that if these irregular movements take a very long time they can still dominate array performance. The DAP's organization is also programmable enabling different organizations of PEs to be changed during execution. These optimizations show the architecture striving to increase efficiency by reducing the idle periods of the array, which will continue to be an issue for future 2D or array processors.

Another bit-serial approach but more in line with modern array processors and the Illiac-IV is the Massively Parallel Processor (MPP), which makes only a few modifications besides once again scaling up the peak processing power [13]. MPP returns to 4-way nearest neighbor connectivity, but maintains the programmable topology from the DAP. Of interest for future 2D machines is a global reduction tree that reduces the underutilization involved in a reduction computed directly on the array. There is also more attention paid to loading data into the array and it can mostly proceed in parallel with the processing, an important aspect of all data-parallel workloads especially those operating on large portions of data like 2D grids.

Another unique approach to a 2D processor that was targeted more directly towards artificial intelligence tasks was the Connection Machine [39]. The connection machine is again bit serial at each PE, with 4 nearest neighbor connections, and an or-reduction tree. The most important change from an architecture perspective is the great increase in dimensionality of the non-direct connections. The CM-1 prototype connection machine has a 12-dimensional network between the processing elements. This network takes a variable number of cycles to traverse but enables much higher overall bandwidth compared to the simple two dimensional network of the nearest neighbor connections. Managing connectivity between elements in a data-parallel architecture is a balancing act between the expense of the network which scales with the machines parallelism and the expressiveness provided by the additional connections. The connection machine pushed this expressiveness considerably further than previous bit-serial machines.

Unfortunately, all of these early designs suffer from difficult programming models where they needed to invent a whole new set of abstractions to provide programmers with something usable for general software development. The MasPar MP-1 moves towards a more usable and common set of interfaces while maintaining the massively parallel SIMD architecture [15]. It adopts a RISC-like instruction set for the PEs with standard data types but still with an instruction broadcast mechanism such that it operates similar to the SIMT paradigm. The PEs also have registers that are accessible by sub-words, bits, bytes, half words, words, and double words, in a similar manner to future packed-SIMD architectures. It is explicitly designed to support high-level languages like C and be more easily programmable, a trend that continues with future array processors. The MP-1 also adds indexed loads and stores to

enable mapping more applications efficiently, such as indirection or lookup tables. The architecture is designed to be scalable as well, such that each set of PEs added also adds memory and communication bandwidth keeping the machine balanced. Although this doesn't go as far as ensuring old binaries can run on scaled up architectures it is still progress towards the performance portability that DLP architectures strive for. Finally, the MP-1 increases the neighbor connections to 8-way and more importantly adds an all-to-all network for the PEs that although slower than the neighbor communication still greatly reduces the average distance of nodes.

As VLSI integration increases in the mid 1990s these SIMD array processors begin to divide into two groups, microprocessors where an entire, often 1D, SIMD processor fits onto a single chip [5], and smaller processors in large networked configurations [83]. We see both of these trends emerge in the MasPar MP-1 because it has both sub-word SIMD registers just as in Intel MMX from 1996 [70], and each PE is a RISC-like processor with a three hop global network between them. Both of these groups of highly parallel architectures begin to adapt their programming models towards more standard software and targeting from high level languages, which is aided by the previous decade's creation and study of the basic linear algebra subprograms (BLAS) as discussed in the next section.

3.2   Multi-dimensional Applications and Libraries

While these SIMD array processors were being developed, built, and used for a wide variety of applications, one of these application domains at the base of much multi-dimensional compute, linear algebra, was being standardized with high-level software APIs. The initial set of proposed basic linear algebra subprograms (BLAS) included only single dimensional operations [62], but still provided an API for higher level software to conform to and hardware to implement efficiently. Over the next decade higher dimensional BLAS routines were developed until level-3 BLAS in 1990 [28]. Shortly after this, a specific set of BLAS functions were combined into a benchmark suite LINPACK which is now used to identify the highest performing computers on the TOP 500 list [29]. This standard set of subprograms is often provided by vendor specific extremely well tuned software packages such as MKL for Intel CPUs and CuBLAS for Nvidia GPUs.

Many different application domains have come to rely on these routines and the software is often built on top of BLAS APIs. A large consumer of the TOP500 compute is scientific computing focused on aiding scientists by modeling or simulating real world physical systems. These models can be so large that they will only fit or be computationally tractable on supercomputers, and are mostly responsible for driving the development of supercomputers. As mentioned before supercomputers have focused on DLP from the beginning with SIMD, vectors, and SIMT architectures making up nearly all of the TOP500.

Other smaller applications more appropriate for a workstation also rely on multi-dimensional compute. Computer-aided design tools, image and video processing and rendering, and engineering simulations all make use of highly parallel often multi-dimensional compute [84] [20].

Recently artificial intelligence and machine learning have become key applications not only at the server level of a data center, but also increasingly at the edge on handheld or embedded devices. Using neural networks for machine learning is not new and many of the SIMD array processors described earlier in this chapter were built with these applications in mind. Initially these machine learning applications were built purely on BLAS but as interest grew the diversity of operations in the networks proliferated and most high performance networks are now implemented using specialized libraries or frameworks. This diversity and its affect on hardware design is outlined below and leads to some of the software changes presented later in the chapter.

3.3   Deep Neural Network Operations

Neural networks have a long history but had a resurgence in the last two decades as the larger labeled datasets became available, along with low-cost high performance GPUs, enabling the networks to become deeper, improving accuracy while retaining reasonable training times [63]. This increase in popularity was also helped and contributed to by a large expansion of the available software and frameworks for using and developing deep learning models, which are discussed later in this chapter.

These networks are named as such because they are represented as a computational graph of different operators, sometimes called layers. A typical deep learning network has many operators that are based on BLAS or are similar in data access and computational patterns to BLAS. These operators include convolutions, element-wise functions, and pooling or subsampling operations, and operate on tensors of many sizes and dimensions. In computer vision applications these intermediate results are often three-dimensional with the image's 2D structure being maintained while the third dimension grows to include more features of those locations. But in other applications of deep neural networks the intermediate results can have even higher dimensionality. It has been shown generally that deeper networks are able to perform better on many of these tasks so the complexity of the networks and correspondingly the number of operations needed to make an inference with them has increased much faster than Moore's law [2].

Fortunately, most of these operations are located in relatively few distinct types of operations and often ones with relatively high arithmetic intensity, so are very amenable to special purpose hardware and other acceleration techniques. The largest contributor to operation count in most networks are the convolutional layers which are often framed as the BLAS-3 operation GEMM by replicating the input data and weights into a larger matrices [21]. The most common hardware strategy for these layers are systolic arrays, which have their own rich history [60], but have also been implemented in a more programmable fashion [22]. This does not increase the number of operations needed but does increase data movement and so there is also research on performing some convolutions directly [30].

Another set of extremely common set of layers in the networks are reductions, such as sub-sampling and pooling. These require more data movement relative to the number of

operations but the data needed has often recently been operated on or produced and can be updated processed before being written back to memory. The acceleration technique here is simply to fuse this operation into the previous operation as the number of inputs for each reduction are usually only a handful of elements and less data can be transferred to memory if the reduction is done first [1]. The last set of common layers are simple element-wise operations. These are often accelerated with attached vector processing pipelines, sometimes with special functional units for specific non-linearity operations. In order to reduce the memory bandwidth requirements on these operations that lack arithmetic intensity they are often fused to other convolutional layers as with the reductions above [47].

Popular network designs change several times a year, but are most often still predominantly composed of the above groups of operations. The number of papers written per year is growing exponentially and shows how fast the field is moving [46]. Because the network of the moment changes frequently and mostly contain combinations of common operations a general programming language implementation or compiler are over-engineered for the problem. Instead most deep learning users and developers work with DSLs or frameworks that have internal ability to generate code or specialize it for hardware targets specifically designed for machine learning.

3.4   Modern Two-Dimensional Architecture Extensions

The early array SIMD processors fell out of fashion, as described in section 3.1, but as we reached a new level of integration, the potential slowing of Moore's law scaling, and the rise in popularity of 2D applications like DNNs, a new wave of research and designs focused on 2D data parallelism emerged. There are two primary features of interest in classifying any given design, programmability and coupling. The programmability feature spectrum scales from a single application code turned into an ASIC similar to high-level synthesis to a general purpose processor with a fully exposed data path. The coupling spectrum spans from distinct devices with their own boards, memory and interfaces to functional units built into another processor's pipeline.

The simplest of these devices are the decoupled and fixed function accelerator devices. Some of these devices are truly fixed function, being designed to only solve a single task and having a fixed structure [11]. But most commonly decoupled devices retain some programmability because driving them remotely would have significant latency.

Eyeriss is a standalone accelerator with a spatial grid of PEs and a limited amount of programmability [23]. Its PEs are focused on CNN layer processing and so can only be configured from off-chip to do computations related to that goal. It does provide a general interconnect between PEs and so given sufficient time and desire one could imagine programming it for other data-parallel or data flow style computations, much as the early SIMD array processors were repurposed.

The first three versions of the Google TPU also operates as its own device with limited programmability, it is only capable of a subset of TensorFlow operations and no general compute, and as with Eyeriss it must be managed by another host computer [51]. The primary compute density and 2D capabilities in the TPU are located in a large systolic array designed for matrix-matrix multiplication. Newer versions of the TPU have added more 1D compute capabilities in the form of a large vector unit which enables many more layers to map efficiently to it, increasing its programmability [50]. Because of the nature of Google's business model there are no details on how it is programmed but it presumably could be programmed to perform other data parallel workloads.

Simba provides another design point in the standalone SIMD array processor space [79]. Simba's distinct feature is that it was designed as a chiplet and so one or more chips could be integrated into a single package with another more general purpose device. The PEs in Simba are more closely aligned with DNN layers but uses vector engines instead of a systolic array so could potentially be reprogrammed for other applications more easily.

Other accelerators are designed to be integrated on chip with general purpose processors like the Convolution Engine(CE) [71]. The CE is designed for a more general purpose compute including 2D convolutions popular in deep learning but it can perform them in the direct fashion. It does use an exotic data size of 10-bits throughout, but exposes its operations via an extended set of instructions and can even be programmed from C code with intrinsics. Because of this specialization for direct convolutions it is able outperform standard packed-SIMD and approach the efficiency of a custom implementation for several image processing benchmarks.

NVIDIA's tensor core is even more tightly integrated than the CE, as it operates as a functional unit inside of the GPU cores [68]. It supplies a set of fixed matrix-multiplication routines that increase efficiency by reducing data movement to and from the register file, operating on more data in parallel, and avoiding intermediate results. The sizes of the matrices supported by the tensor cores are limited to a few fixed sizes based, and the data types supported, although initially limited to lower precisions, have been expanded in subsequent generations. The designs rigidity in terms of sizes and data types causes a lot of non-linearity in performance and increases the burden on the programmer to carefully optimize there code.

Intel's vector neural network instructions (VNNI) are very tightly coupled to the general purpose core and appear as a sub component of the vector portion of the core [59]. Similarly to the tensor cores VNNI instructions have limited data types supported and as with all packed-SIMD instructions fixed data sizes. The instructions added are essentially a hardware implementation of a series of operations already provided by the latest AVX-512 extension. The efficiency gains again come from the reducing register file accesses, avoiding materializing intermediate results, and potentially performing more operations at once.

The architecture described in chapters 6 and 7 is designed to more closely line up with the programmability of the VNNI where operations that could be executed previously are simply executed more efficently with fewer instructions, while the integration is designed to be slightly more decoupled to enable some additional optimizations that a fixed latency functional unit would be unable to perform.

Chapter 4

Implementing Vector Architectures in Silicon

Implementing a vector architecture in silicon can be challenging because different sections of the design present different challenges for the microarchitecture and physical design. Some of these sections share similarities with other high-performance architectures like the high memory bandwidth required for GPUs or the large and high bandwidth register files present in out-of-order processors. This chapter discusses these features, their constraints on the design and from the technology, and how the Hwacha architecture chooses to address these microarchitectural and physical design concerns.

First, section 4.1 will enumerate several modern technology trends that impact the design and implementation of current SoCs. Then, section 4.2 discusses, each in their own subsection, the principal components of a vector microarchitecture and their implementation choices. Next, section 4.3 describes the details of the Hwacha architecture and microarchitecture. Finally, section 4.4 elaborates the different implementation choices made in the microarchitecture and how these were informed by the previous trends and design constraints.

4.1 Modern Technology Constraints on Digital Design

In modern deep sub-micron processes using FinFETs the density and delay of logic gates is rarely the largest constraint on a design [36]. Many new patterns in chip and system design have been developed in order to continue to be able to utilize all the dense logic for useful work. These techniques include silicon interposers with large high bandwidth memory interfaces integrated in package, such as high-bandwidth memory (HBM) [82], and many large fixed-function low-interconnect functional units for offloading tasks from programmable units [78]. These techniques attack three of the primary constraints in modern processes, off-chip interconnect, on-chip interconnect, and on-chip storage.

Off-chip interconnect is usually bounded by two limiting factors, the length of the perime-

ter of a chip which will always scale slower than the number of transistors in the area, and the signaling technology at the interface. There is work attempting to mitigate each of these limiting factors. Off-package interconnects, with traditional interfaces and materials with copper wires on PCBs do not scale in the same way as transistors, and so have become a bottleneck over time. [98] Some techniques are exploring how to use more central areas of the chip to interface with the outside like HBM, Advanced Interface Bus (AIB), and the more general technique of through silicon via (TSV). Most of these techniques also need to address the signaling technology and material problem because the medium they communicate through may be a package or perhaps only a TSV. The differences in requirements of these different mediums and distances between the compute die and other devices allows for a redesign of the signaling techniques. Other techniques focus on continuing to improve traditional off-package electrical communication or switching to novel off-package communication with optical interconnects.

Moving off-chip interconnect from inter-package to intra-package is a common solution currently and is frequently used to bring DRAM even closer to the processor, in terms of latency and bandwidth. Other alternative DRAM organizations that can only be integrated in package, like HBM, provide further performance improvements[52]. These memory in package techniques have large trade-offs between performance, cost, and power. The cheapest system designs that require DRAM will continue to have off-package memory as the additional packaging step will always cost something, and currently cost quite a lot. Systems that want either higher performance or lower power usage will move memory in package, because the reduction in wiring length both increase performance and enables less power consumption in the signaling. Systems that want high performance still have two options. The absolute highest performance designs will be using HBM memory at large expense but many other high performance systems will simply integrate high speed conventional DRAM or potentially the higher bandwidth graphics DDR (GDDR) in package[26]. Finally, the high performance systems that prioritize power efficiency will likely integrate low power DDR (LPDDR) as in the recent Apple M1 processor[34]. These integration techniques enable better scaling than traditional off-chip interconnect at a higher cost but also open the ability to integrate more than just memory and an SoC into a single package.

These packaging technologies are also currently being explored to build more complex systems-in-a-package (SiPs) out of small chips often called chiplets. Chiplets are attractive because they allow for larger scale integration with benefits, in terms of power and performance, approaching that of on-chip integration but without the problems associated with large die size and integrating different processing technologies. Very large chips become increasingly hard to guarantee high yield since each device represents a possible failure point, and the variation across the chip increases potentially reducing overall system performance. These designs often require explicit on-chip redundancy and ability to adapt to the variation, such as the ability for high core count Intel processors to have cores disabled differently based on which cores pass testing. Directly integrating DRAM onto a processor die is possible with embedded DRAM (eDRAM) but eDRAM performs worse than off-chip DRAM, and still increases the die size again potentially causing yield problems[33]. Memory in package

has led the way for chiplets because DRAM is incredibly popular and has a standard interface with multiple companies developing compatible and interchangeable products, while non-memory chiplets are still rather distinct with several companies developing standards and methodologies in parallel. Intel's chiplet effort is focused on using the embedded multi-die interconnect bridge (EMIB) packaging solution with the advanced interface bus (AIB) as a standard physical layer interface that many chiplets, both logic and memory, could implement[65][44]. If the complexities of packaging and testing these multi-chip SiP designs can be addressed and standards can be adopted for more than just memory, chiplets have the potential to greatly improve the performance of single package systems.

Despite all of the work on multi-chip packages there will still always be a need for fast electrical signaling whether it is off-package or potentially in-package. Analog designers have continued to improve traditional differential electrical signaling achieving bitrates of up to 112Gbps on a single wire in the latest technology nodes[87]. Others have experimented with single ended signaling, including AIB, to improve bandwidth per pin utilization especially when the data transfer will only be in package[80]. These techniques are focused on bridging the gap between off-package and on-chip interconnect by adjusting to the unique challenges of in-package interconnect like shorter lengths and the different substrate's electrical properties. Some designers focus on novel signaling architectures that work well across all shorter range interconnects including long distances on-chip, most in-package connections, and short off-package connections[89]. In addition, improving conventional off-chip interconnect to have energy use proportional to the consumed bandwidth is an active area of research and can help always-on systems or those systems with widely varying workloads like mobile SoCs[10]. Some researchers are even looking to move away from electrical signaling and are testing the possibilities of moving off-chip signaling to optical fibers.

Optical connections are often used for communicating over long distances at high-bandwidth, comprising nearly all of the internet's backbone links. Moving the optical connections inside of the data center will require advances in switching, laser power, and connection technologies. There have been recent advances in having optical connections interface directly with an SoC [85]. Extensions of this work for the SiP future have looked at combining optical connections and chiplets[91]. Overall it is yet to be seen if optical connections can truly make the leap to a commercial product, but the improvements in performance could be a large step forwarded for extremely high performance systems, especially as traditional metal wires continue to see diminishing gains. Even less common than optical off-chip interconnect are integrated wireless interconnects that could offer an alternative solution to communicating with the outside world. Wireless transmitters appear in many systems currently but are almost always confined to separate packages. However some work has been done in attempting to integrate these components in-package[69].

Like off-chip interconnect, on-chip interconnect also scales less aggressively than transistor performance and can become a limiter in overall system design [18]. Due to a myriad of physical effects traditional on-chip wires will continue to scale more slowly than the transistors that drive them, and so different techniques, materials, and interconnect architectures are being explored. In addition, to improving the on-chip interconnect its affect can also be

limited by adopting smaller local regions of interconnect and increasing the number of globally asynchronous locally synchronous (GALS) regions. Without improving the interconnect designs will be forced to increase latency or reduce frequency, both of which can significantly reduce the benefits of on-chip integration.

The most straightforward approach to improving on-chip interconnect given fixed geometries is to change the composition of the wires adjusting their metallurgy [90]. Intel plans to use cobalt wires in their $10nm$ technology to improve interconnect resistance and longevity [8]. Even more exotic materials, like graphene, might be used in place of traditional metals for some or all layers of interconnect [93]. Different dielectrics are also being explored including removing the dielectric altogether resulting in air gaps between the interconnect [32].

The above techniques are mostly focused on the local and intermediate interconnect but with large chips the cross-chip connections might be better suited to SerDes [67]. These designs begin to overlap with the off-chip interconnect at this point, needing transmission lines and complex signaling methodologies. If no repeaters are needed the interconnect across the chip will be similar to the in-package connections. These technologies are limited to large chips since they are often designed for distances greater than $5mm$. However, at these distances they can offer more efficient higher speed transmission than traditional global interconnect with repeaters [58]. Once again, as with off-chip interconnect some researchers are exploring using wireless connections even at these small distances, but this currently consists mostly of architecture work with few if any implementations.

The third primary constraint on cutting-edge chip design, on-chip storage, has scaling characteristics most similar to dense logic. Because on-chip storage consists primarily of minimum-sized transistors, static random-access memory (SRAM) cells have continued to scale in modern non-planar processes. However, more effort is required to lower SRAM minimum operating voltage due to the large number of minimum-sized devices required for operation, and the threshold voltage fluctuation these minimum-sized devices experience [19]. As discussed in subsection 4.2, the primary constraint of on-chip storage then becomes the wiring or on-chip interconnect required to access and utilize the array. The constraints this applies on the microarchitecture will be discussed further in the referenced subsection.

Despite the conventional scaling of SRAMs only slightly slowing compared to standard logic, there is ongoing work to provide alternatives, although no current technologies avoid the wire limitations. Truly 3D integrated circuit technologies are being used to split SRAM cells across multiple layers of active devices. These additional layers often add extra functionality potentially useful for in-memory computation [42]. Other on-chip memory technologies are focused on non-volatility like resistive random-access memory (RRAM), spin-transfer torque magnetic random-access memory (STT-MRAM), and eDRAM. RRAM in the same technology node can be more dbout 30% more dense than SRAM but trades speed for non-volatility [53] [49]. STT-MRAM has approximately the same density as RRAM, while eDRAM has 70% higher density than even RRAM [95] [37]. These technologies are not targeted at replacing traditional on-chip SRAMs and so will not have a large effect on processor and SoC designs. Fortunately, traditional SRAMs are scaling well and only end up being

limited by on-chip wiring concerns and generic timing and throughput issues.

4.2   Vector Architecture Design Constraints

Given the current technological trends vector architectures and microarchitectures need careful design to avoid exacerbating the bottlenecks of off-chip interconnect and on-chip interconnect. The off-chip interconnect effects are primarily related to the memory system the vector processor is integrated with and will be discussed in relation to that. On-chip interconnect is more pervasive in its constraints and requires a more wholistic design view to fully understand its impact.

One of the main benefits of a vector architecture is the ease with which their throughput can be scaled up or down depending on application needs. This scalability is primarily driven by the explicitly independent nature of most computation done on the vector architecture. Therefore, designs must be very attentive to what communication patterns are allowed between the data elements. A conceptually efficient way to think about scaling a vector microarchitecture is by identifying the internal structure that will be replicated to improve throughput. This structure is normally called a lane, and design decisions often rely on understanding how they affect the size of an individual lane as well as how they affect the complexity of each additional lane added. These concerns are unlikely to be separable but thinking about these two axes during design is a useful way to frame the problem.

Local interconnect can be limited in a vector microarchitecture by the careful design of the lane and the instructions that are allowed to cross these lane boundaries. Reducing this communication allows the dense interconnect to be limited in scope to a smaller on-chip area, but has wholistic affects on the microarchitecture. Specifically the register file, memory system, and obviously reduction or other cross lane networks are impacted significantly by the amount of cross-lane communication. Off-chip interconnect requirements can be reduced in a vector microarchitecture by increasing the size of the vector register file, implementing expressive memory operations, and increasing on-chip caches. Older vector processors that span multiple chips or are made of discrete components have very different design decisions and so won't be discussed. Instead, this section focuses on fully-integrated vector processors and how technology trends might affect future implementations.

In order to analyze these effects on a vector architecture implementations it is useful to go through the microarchitectural components independently. Consequently, this section breaks a generic vector architecture implementation into three core components, the vector register file, the interconnect and functional units, and the memory system.

Register Files

The vector register file implementation is a core design decision that affects the rest of the microarchitecture will process instructions and data. In the past, vector machines with flip-flop based register files have spent a large portion of their area and the design effort on their

register files due to the abundance of ports and total storage size [6]. An alternative style of vector machines built with SRAM-based register files, such as Hwacha, can alleviate this bottleneck at the cost of scheduling and control logic needed to manage the highly banked register files which are used to restore the register file throughput. Understanding the trade-offs between these and other design points requires understanding the available methods for building a register file.

Unlike scalar architectures, vector machines rely on multiple operations being explicitly encoded in an instruction and so require larger registers to source and sink these parallel operations. The register files for vector architectures are often much larger than a scalar design, and in long-vector architectures can be many times larger, even approaching the size of a scalar architecture's level one cache. In order to provide that amount of storage there are several common design patterns followed by modern vector machines. At the highest end of performance out-of-order vector designs implement the register file in, potentially custom, flip-flop arrays to enable high cycle times[41]. In the middle of the performance continuum are long-vector machines with SRAM-based register files, often banked and with multiple lanes to increase throughput above their highest frequency counterparts[99]. At the low end of absolute performance a vector design can share resources with a scalar core, or operate directly out of memory, but these designs points are uncommon outside of the deeply embedded space where other less flexible forms of SIMD execution, like VLIW, are more popular.

The design space for these register files is quite complex since it spans many dimensions: throughput, latency, capacity, area, and power. Determining the overall energy efficiency of register files is also complex as there are many different design options for both the bit-cells and the periphery and the design decisions are also impacted by the architecture's design [101]. Non-planar technologies, like FinFETs, will make some of these techniques difficult as the number of replicated structures forces tight alignment that now must match the FinFET grid. True multi-ported designs can still be made with both standard cell and bit-cell-array based designs but requires custom design or a compiler[31]. Figure 4.2 shows a summary table of the common register file implementation techniques, and shows a rough estimation of how they scale with capacity, throughput, and number of ports.

The most common choice for on-chip memories of significant size are standard one-cycle SRAMs with either a single combined read-and-write port or a dedicated-read and dedicated-write port. The characteristics of these memories are enumerated in the first row of Table 4.2. These memories are often designed for density as they will be occupying a large portion of most SoCs and so they have the best capacity scaling of any techniques. Conversely, this focus on density causes their latency to be much worse, since the bit-lines are not only driven by minimum-sized transistors but also require sense amplifiers to spend time reading the small voltage change from each cell before outputting the correct data value. This use of peripheral circuitry is the primary reason these memories have a high base area cost. The port cost of these memories is very high because they are fixed blocks that cannot include more than two ports without explicit duplication of entire memories. Standard bit-cell-array based memories do not have a combinational read path and so can be somewhat difficult to

Type of Memory	Latency	Capacity Scaling	Base Area	Port Cost	Integration Effort	Custom Design
Standard 6T/8T 1-cycle SRAM	Slow	Best	High	High	Average	No
Custom bit-cell RAM	Average	Good w/o extra ports	High	Average	High	Yes Analog
Standard-cell RAM	Good	Average w/o extra ports	Low	Average	Low	No
Custom standard-cell RAM	Best	Average	Low	Low	Average	Yes Digital

Table 4.1: Comparison table of scaling for various register file implementation techniques.

integrate since the microarchitecture must be able to provide the read address a cycle before the data is needed. However, these memories do not require any custom design and so can be used freely in generic designs that do not assume more than an RTL description and a reasonable technology library.

The second row of Table 4.2, describes what can be built if a designer is capable of doing the custom analog design required to construct a bit-cell and periphery circuitry with more complex properties. Some of these common properties are adding extra read or write ports, increasing the frequency of accesses, and increasing resilience to errors or low-power states. This requirement of custom design makes these memories only usable by larger teams with experience across the chip design spectrum, making integration difficult. If this experience is available however, custom bit-cell-array based memories can improve on almost all other aspects recorded in the table. The most notable changes for processor design are the ability to add physical ports to the same memory or improve the frequency to match that of deeply pipelined processors. The custom memories still retain the high base area from periphery circuitry. The standard multi-porting techniques cause the area of the memory to scale quadratically with the number of ports so the capacity scaling for custom memories can be good as long as the total number of ports is still reasonable [88].

The final two memory techniques are based on standard-cell storage arrays, such as flip-flops or latches, and are sometimes known as standard-cell memories (SCMs). These memories are based on larger single bit storage elements and so do not need nearly as much periphery circuitry which has consequences on all other aspects of the memories. Not having to use sense amplifiers to recover the stored values reduces latency of reads significantly. In addition, flip-flops often come in many drive strengths and threshold voltages which enables automatic synthesis and place and route tools to match the timing of the memory to that of the surrounding circuitry. This matching allows the frequency, area, energy trade-off for

each memory to be explored more completely with little overhead to the designer. The primary downside to SCMs is their relatively large size and greatly reduced capacity scaling. Typically only very small flip-flop based memories, with less than 1 kilobit of total state, are smaller than bit-cell-array based designs [66]. Adding additional ports to an SCM is possible without custom design, but because they have larger storage arrays the area scaling with ports is worse than for a bit-cell-array based design due to the increased wire lengths. If a designer is able to use custom design with an SCM they can produce extremely high performance designs, but if the designer is willing to put in such efforts they would often be better served with the custom design of an SRAM to reduce the cost of extra capacity. However, for many applications the speed of SCMs and ease of integration will out weigh the higher cost to meet overall performance targets.

The decision on what register file implementation technique to use is not made in a vaccum, and the constraints of a vector architecture often push the design towards a few well worn paths. Vector machines that have a relatively small total vector register file capacity, hereafter described as short vector machines since the resultant maximum vector length will be short, are often coupled with higher frequency cores such as out-of-order designs. The out-of-order design benefits from tight integration with the vectors so wants them to operate at the same speed as other pipelines to avoid any clock crossing latencies. Low latency is also important in out-of-order designs as it allows instructions to retire earlier increasing the effective size of the issue window and reduces the cost of misspeculation. In addition, trying to restart a vector operation without losing the misspeculated work requires more complex dependency tracking, which would most likely be eliminated by or subsumed into the ability to issue smaller portions of a vector operation. This causes out-of-order designs to issue whatever small unit of vector work can be scheduled and completed relatively quickly, potentially the smallest addressable register file unit. Breaking the vector operations into smaller pieces of work encourages the register file to be accessible in smaller units making it more similar to a short-vector register file. The out-of-order hardware will often be able to hide the latency incurred by splitting a long vector operation into multiple short vector instructions. In addition, the smaller operations can often expose more parallelism by allowing distinct portions of dependent vector operations to proceed in parallel, in a scheme called vector chaining. There will be a benefit to this out-of-order machines having longer vector registers in the reduction of instruction fetches but this could be partially addressed by loop buffers or other general high-performance processor microarchitectures. In addition, it is hard to estimate how the additional dependency tracking hardware will scale with the longer vector registers and so a more detailed implementation study would be needed to find the optimal design point. Overall, out-of-order designs should be able to perform well with short high frequency vector register files, and most commercial implementations tend to have vector lengths around half a cache-line.

Another point in the design space of vector microarchitectures are those with very large total vector register file capacity, deemed long-vector machines since they will have much longer maximum vector lengths. These designs are often focused on aggregate throughput and energy efficiency than peak throughput and so run at lower frequencies. The reduced fre-

quency requirement allows bit-cell-array based SRAMs to be used which gives these designs much better capacity scaling. Given a bit-cell-array based register file design, the long-vector machines will need to make a trade-off between the control complexity of increased banking or custom design with more ports per bank to achieve the desired bandwidth. The exact amount of bandwidth required of the register file is also a microarchitectural decision that contributes to both peak performance and average performance. Building more ports per bank can allow higher utilization of functional units during highly parallel instruction sequences but common sequences will be unlikely to use all ports, similar to large issue widths in super-scalar designs. One commercial example of a long-vector design is the SX-aurora which has a register file of 128KiB[99]. This large register file is spread across 32 lanes and 8 banks such that each bank is 4Kibit which allows for relatively high-speed SRAMs. In their 16nm FinFET process these SRAMs are likely to be the limiting factor in the core frequency of 1.6GHz, but they also are unlikely to need custom design as lower threshold voltage SRAMs are probably able to achieve this frequency. NEC does not reveal their exact storage technology but assuming each of these banks is a one-read and one-write 8T SRAM, or are one-read-write, then they have a total of eight reads with a possible consumption of eleven reads if all of the functional units were active at once. In this case it is unlikely that the design would need a two read one write custom SRAM given the extra area that would require and the overprovisioning of register file read bandwidth. Avoiding custom register file design is a great advantage, for smaller volume designs, of this class of long-vector machines.

Long-vector designs with SRAM-based register files also tend to be temporal designs where portions of these long vectors execute over time. This is a result of the low port count on standard SRAM instances and even more so from the ratio of read bandwidth to capacity. This ratio will be several binary order of magnitudes different between the SCM and SRAM-based register files. Temporal execution increases the control complexity required to manage these vector operations but usually less so than having full out of order execution seen in the short vector SCM-based designs. The regular access patterns of vector operations can simplify the register file address generation during temporal execution making it even easier to build long-vector designs with banked standard SRAM designs.

The vector register file design has far reaching impacts on the overall microarchitecture and the choice of implementation technique limits the achievable design points. Designing around the slower but denser SRAM based register files allows for long-vector machines with a focus on energy efficiency and throughput. On the other hand using faster SCM-based register file designs moves the machine into the short vector design space where there is a focus on latency and tight coupling with scalar code. The available technologies and their trade-offs push designs into two families of designs of vector machines, temporal long-vector designs, and single-beat short vector designs. The impact of these two families on the other two major components of a vector design is analyzed in the next two subsections.

Functional Units, Interconnect, and Wiring

Spatially, the central portion of a vector microarchitecture are usually the functional units. However, despite their central location, the functional units are the portion of the architecture that has continued to scale well with successive technology nodes and therefore are often the most flexible in terms of design. The register files, memory units, caches, and off-chip memory system all are designed around being able to feed these relatively cheap functional units. And one of the advantages that vector architectures enable is the ability to more easily replicate these functional units to take advantage of the explicit parallelism. Because these components are so cheap the designer often focuses more on how they connect to the rest of the system including the register file and memory interface, especially for functional units that require touching elements from multiple lanes. However, this decision is not purely microarchitectural, like the register file, because the number and frequency of operations that require cross-lane or intra-element communication greatly depends on the vector instruction set.

Assuming the design is intended as a general-purpose vector architecture some form of cross-lane or intra-element operations will be almost necessary. Without supplying these operations the users will be forced to do any element shuffling or reductions on the scalar portion of the design aggravating the Amdahl's law problem of data-parallel compute. There are two facets of cross-lane communication instructions, expressiveness and expensiveness. The expressiveness facet relates to how many use cases will be covered by each instruction or how many instructions will need to be used to implement an application or use case. The expensiveness facet is related to how much area and power these additional instructions will add to the microarchitecture. Designing a vector architecture requires careful consideration of these facets, especially if the architecture is going to be scalable and support a varied number of lanes. On one end of spectrum, cross-lane communication could be limited to occurring through memory via scatter and gather memory operations. Or if the architecture supports masking, through a very simple instruction sequence like element shift and mask. Both of these approaches will require many cycles and potentially many instructions but will be able to reuse aspects of the data path that are already present. On the other end of the spectrum, general register-register permute operations require additional datapath connections or functional units to accomplish at speed, but can accomplish most cross-lane communication patterns in a single instruction. As with most problems in computer architecture, the ideal design is some middle ground that enables most data-parallel programs to execute efficently with few instructions on the vector architecture while not squandering the efficiency of the design by including rarely used connections and data-paths for cross-lane communication.

A more detailed analysis of cross-lane operations finds that they can be classified into explicit and implicit communication. The explicit communication is easiest to identify as it arises from the instructions specifically designed and added to the architecture for cross-lane communication, such as register-register permutes and reductions. The implicit communication is much more nuanced and a design must be analyzed wholistically to determine

how much implicit cross-lane communication will occur. In particular, implicit cross-lane communication is often a design decision and different microarchitectures can have different amounts of implicit cross-lane communication for the same architecture. This trade-off is related to other microarchitectural organization, such as the width of the replicated lane, the width of each register file bank, the number of memory access ports, etc.

The design families discussed in the register file section provide a useful basis for discussion of different cross-lane communication strategies as well. Long-vector designs with their inherent temporal structure are often designed with built-in latency that can allow cross-lane communication to be multi-cycle. This additional latency is often hidden by its overlap with other latency in the design while still allowing full throughput. For explicit cross-lane operations, the long-vector designs have a few other issues in addition to the scheduling of register file accesses and use of cross-lane routing resources. Because the long-vector family usually doesn't have all elements in flight at once, there will need to be some module that aggregates the elements over multiple cycles. Permutations of elements can be handled without extra storage but other types of aggregation can require extra buffering to perform at all or perform at a reasonable rate of speed. Because of the potentially very long vector lengths any extra storage requirement that scales with maximum vector length will be prohibitive for long-vector architectures. For implicit cross-lane operations, the primary design consideration for long-vector architectures is where the elements will reside in the register file and which functional units they will use. Explicit cross-lane operations will always need to use additional resources, but the goal of the design for implicit cross-lane operations should be to eliminate them as much as possible. Keeping the size of each lanes register file bank matched to the widest possible operand in the architecture can eliminate implicit cross-lane operations from mixed precision computations. Figure 4.3 shows an example mapping that supports this elimination, by carefully placing all matching elements, regardless of a precision, in the same lane. To avoid wasting register file storage, or reintroducing implicit cross-lane communication, each lane now needs to hold a number of continuous elements equal to the maximum number of elements, of minimum precision, that fit in a single addressable entry of the vector register file bank.

Some long-vector architectures focus more on the cross-lane operations in order to allow for a more MIMD like design that can vectorize a wider set of codes efficiently. These designs often have a specific architectural feature that makes the explicit cross-lane operations more powerful or pervasive than the standard cross-lane operations. In the SCALE vector-thread architecture, for example a special cross-lane network that is directly accessible as instruction operands enables a different programming model with independent vector-thread lanes to still collaborate on cross-lane work [56]. These types of designs tend to move away from the pure SIMD approach of vector architectures, and while interesting will not be considered further in this chapter.

On the other hand, short-vector designs will not have much latency or cycle time to hide cross-lane communication. The finer read granularities of SCMs, typically used in short-vector designs, can be utilized along with their higher read bandwidth to potentially reduce the hardware required for efficient cross-lane communication. In addition, the fact

that an entire vector operation can be accomplished in one pass means that most often all the data needed for a cross-lane operation will be in-flight at once. Having all the data in-flight at once enables reordering or permutation as desired without extra steps or special buffers. For explicit cross-lane operations, small vector machines will similarly categorize these instructions by the amount of cross-lane communication and implement each category differently. Enabling the vector register file to be read from at different addresses for different element positions can enable simple merges to be very fast, while a full crossbar will still be required for more complex permute patterns [40]. The microarchitecture can also use latency as a degree of freedom for these operations allowing less aggressive register file and crossbar implementations. For implicit cross-lane operations, short-vector designs have fewer concerns because having all data in flight at once means the only issue is the datapath connections that will need to exist if the implicit operation is to be supported at all.

Current technology scaling continues to provide faster and smaller transistors while approaching limits in wire resistivity putting pressure on the interconnect design even inside of a processor. Vector designs are carefully designed to avoid losing the explicit data-parallelism advantage of highly replicable functional units to the potentially large cross-lane interconnects. A large portion of this interconnect occurs between the vector register file read and the vector register file write, but a portion of this wiring and another large factor affecting performance of the design is in the memory system. The next section will look at how different memory system implementations interact with current technology trends and the two families of vector designs being discussed.

Memory System Requirements

The memory system of vector designs is critical to its performance because without the ability to get data into and out of the register file all the functional unit bandwidth could be wasted. Fortunately, as mentioned before, the SRAMs that most caches are built out of are still scaling well and so the limits on their size are primarily based on the interconnect and wiring needed to move the data to and from them. The on-chip memory system is not the only concern however, as most applications will not fit entirely on chip especially those that are highly data-parallel and so the off-chip memory interface is also of prime concern. This focus on off-chip interfaces and the memory wall is why there has been a trend toward tighter integration of higher performance DRAM modules. Extremely high performance data-parallel architectures like GPGPUs have used HBM and DDR in package to narrow the gap between off-chip bandwidth and on-chip bandwidth. This subsection will discuss how these on- and off-chip interface design decisions can be made with respect to the two families of vector designs previously outlined.

The most critical level of memory hierarchy after the vector register file is the cache that the vector design makes direct requests of. This may not necessarily be the first level cache as that may only be accessible by the scalar cores. For short-vector designs the high frequency of their SCM based register files will want to operate with a similarly fast cache and so most often fetch data directly from an L1 data cache. In addition, the tight integration of short-

vector designs lends itself to tracking the memory dependencies of the scalar and vector units in the same location and therefore a unified load-store queue fetching from a L1 cache makes this process simpler. The shorter vectors also ensure that multiple vector registers are likely to be able to reside in a smaller L1 cache without generating excessive capacity conflicts. Finally, because these vectors are short the will span only one or two cache lines making the translation lookaside buffer (TLB) access and dependency tracking easier to handle in the short cycle times required by an L1 integration. On the contrary, long-vector designs will often have vector memory operations that bring in potentially kilobytes of data and so would easily overrun the L1 forcing a high rate of misses and worse latency than a direct request to the second level of caching. In addition, the frequency of long-vector designs is often matched to slower SRAM based register files and so there is even less benefit to making requests directly out of a high frequency L1 cache.

Another issue that often correlates with the design families are the vector-scalar memory ordering requirements. Short-vector designs will usually include significant memory disambiguation hardware to facilitate their tight integration with the vector unit. This hardware can often be shared with the scalar-scalar memory ordering hardware that an out-of-order design will already need to keep its scalar memory operations latency low. Because the vectors are short they may only represent a handful of scalar-sized memory operations and so the additional hardware required to also track the vector operations will be relatively less costly. On the other hand, the synchronization between a long-vector design's memory operations and its accompanying scalar processor is assumed to be less frequent. This makes the extra cycles required for the L1 to fetch these recently used vector data out of the L2 acceptable. If a long-vector design wants to avoid this latency it will need to include a large amount of memory disambiguation hardware due to the large data regions the long vectors are capable of addressing. This expense encourages long-vector designers to either be more conservative in their orderings simplifying the disambiguation, or follow the previous less frequent synchronization model.

In both design families, however, it is unlikely for an entire working set to fit in the level of caching that is directly accessed and so the remaining levels of on-chip hierarchy are designed similarly for both families. For the outer levels of the cache hierarchy the vector unit will have two primary access patterns, either the data will be a portion of a block that will be reused by inner caches and the vector register file, or the data is being streamed into the vector register file and won't be accessed again before being flushed off chip. The blocking access method primarily occurs during N^3 type algorithms that utilize multiple levels of blocking to ensure the maximum reuse of data is occurring at each level. This reuse means that well designed machines can use an appropriately sized register file and first level cache access bandwidth to ensure these algorithms balance the memory and compute requirements for low arithmetic intensity kernels or are compute bound for higher intensity kernels. The outer levels of cache then only need to be designed following traditional average memory access time minimization techniques to be useful for the blocking access pattern. The stream access method however requires and consumes as much bandwidth as the memory hierarchy can supply up to the maximum supported by the inner levels. In order to achieve this high

bandwidth at reasonable cost the outer levels will need to be highly banked especially as the outer levels tend to be accessed by multiple processors potentially each with their own vector unit.

Highly banked memory systems are not unique to vector designs but the pressure that explicitly data-parallel architectures like vectors put on a memory system is often much more than what a similarly sized scalar or even super-scalar design is capable of. Short-vector designs are less resilient to bank conflicts based on the length of vectors, but the tighter integration with out-of-order scheduling hardware enables them to handle the bank conflicts better. Long-vector designs on the other hand are most frequently requesting data that spans multiple cache lines and not infrequently multiple pages. Spanning multiple cache lines allows the accesses to proceed in parallel in the face of bank conflicts, but excessive conflicts can still reduce throughput. Because a bank conflict and consequential stall from a vector memory access blocks an entire cache line worth of data processing the lost performance from vector bank conflicts is much higher than a single scalar access that only blocks one data processing operation. This increased cost and desire for high peak throughput in general has led to a lot of research into avoiding vector cache bank conflicts [77].

Unfortunately, highly banked caches run into several issues mostly related to the physical design of such systems. Each bank of the cache adds an end point to the crossbar between the vector memory unit's access ports and the cache. The size and bi-sectional bandwidth of this crossbar can quickly become an issue as will be discussed in the next chapter. In addition, increasing the number of banks while keeping the total area used by the cache constant will reduce the overall storage provided. This leads to a trade-off in number of banks and cache capacity or power, that usually restricts inner level caches to be only moderately banked [86].

Finally the off-chip memory system is much more expensive to bank as each additional off-chip memory channel requires separate signaling hardware and a separate set of off-chip ports. Despite this high cost many high performance chips will include several memory channels as many applications will require a working set larger than the capacity of on-chip caches, or a single off-chip memory. The in-package memory system improvements like HBM and in-package DRAM provide more bandwidth relative to their on-chip area. Given a vector design's ability to consume memory bandwidth they, and other highly parallel machines, favor these in-package solutions or having numerous traditional off-chip memory channels. Older vector designs sometimes had direct off-chip access from the vector machine but modern process technologies make that unwise as the latency gap between on-chip and off-chip accesses are so large that average memory access time would be much better served with an on-chip cache.

The memory system, as one half of the roofline model [96], restricts peak performance in a wide range of applications and enables efficient blocking even with arithmetically intense applications. It therefore requires careful design, and yet is tightly coupled with technology constraints and physical design restrictions for both short- and long-vector designs. The next two sections, discuss how the Hwacha long-vector architecture and design addresses not only the constraints of the memory system but also the register file and functional units.
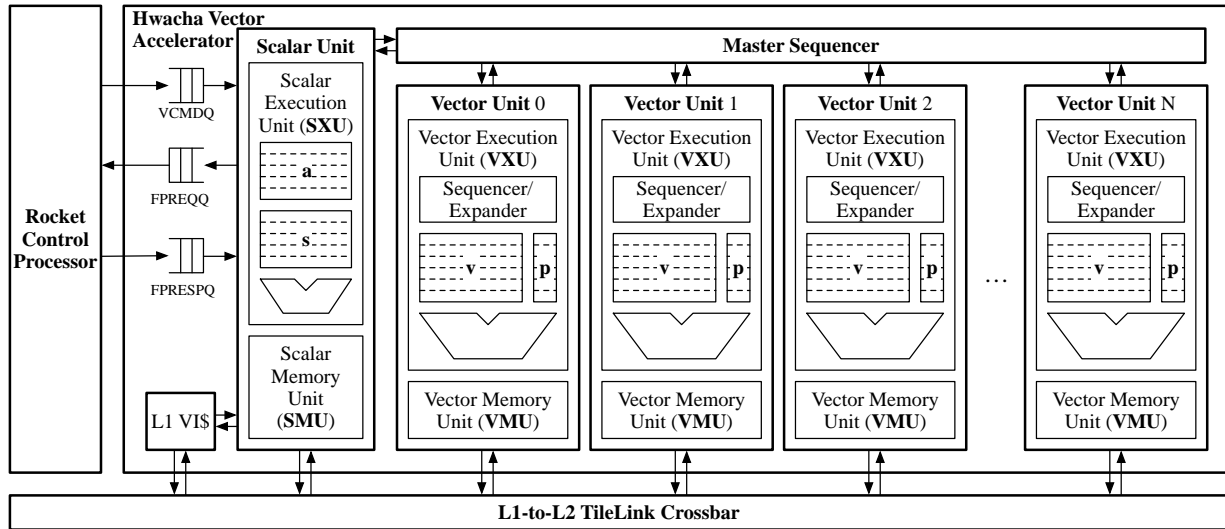
Figure 4.1: A high level diagram of the Hwacha microarchitecture.

4.3   Hwacha Architecture Details

The Hwacha architecture is an explicitly-decoupled vector-fetch architecture, as described in section 2.6, designed as a custom general-purpose data-parallel extension to the RISC-V scalar ISA, distinct from the RISC-V vector extension. The explicit decoupling arises from the separation of vector data-processing instructions from scalar instructions and vector control instructions. These data-processing instructions, or worker-thread instructions, are never fetched by the control processor. Instead a pointer to each vector-fetch block is sent to the vector unit which independently fetches these instructions from memory. This explicit decoupling enables more overlap of computation, control, and memory operations within the overall system when running applications with data-parallel sections that can be accelerated with Hwacha. An additional architectural detail that enables decoupling without many synchronization or loss-of-decoupling events are the address registers. These registers are only writable by the control thread which enables the vector memory unit to begin fetching data based on those addresses without needing to check whether the scalar unit has updated them or whether they correspond to a different vector fetch block.

The Hwacha microarchitecture is shown at a high level in Figure 4.1. At the far left is the control processor, which handles the scalar RISC-V instructions and the Hwacha control-thread instructions. In most implementations, the control processor is an in-order 5-stage pipeline but more recent implementations have used an out-of-order super-scalar control processor. This processor is always running ahead of the vector unit executing control instructions that will eventually be received by the accelerator. Sometimes when the commands are simple, or the vector length is short, the two will be nearly in sync and at other times several vector fetch blocks will be in flight at the same time. These control-thread

instructions are dispatched to the vector command queue (VCMDQ) which is a simple FIFO queue that is consumed by Hwacha's scalar unit.

The scalar unit is responsible for handling all vector control instructions and some worker-thread instructions. There are two classes of control instructions, those that require a response to the control-thread, which are mostly configuration instructions, and those that do not require a response. By design, the control instructions that require a response take only a single cycle to execute once at the head of the command queue, and so unblock any control processor code waiting to use the response. The control instructions that don't require a response are either moving data to the scalar or address registers (`vmca`, `vmcs`) or launching vector-fetch blocks. The data-movement instructions must be buffered until the current vector-fetch block finishes after which they can be handled at a rate of one per cycle. The vector-fetch blocks require the scalar unit to fetch instructions from the vector-instruction cache, which will translate the address potentially refilling its TLB or refilling the data from the outer memory system. Once fetched it executes these worker-thread instructions serially, handling any purely scalar computations on its own in-order pipeline including scalar memory operations that are handled by a separate scalar memory unit (SMU) that coordinates with the vector memory unit to avoid memory consistency issues. For vector-vector or vector-scalar worker-thread instructions, the scalar processor decodes and issues these to the master sequencer along with any scalar data needed. In addition, vector memory worker-thread instructions are also issued directly to the vector memory unit (VMU).

The master sequencer is responsible for keeping track of the status of all in progress vector worker-thread instructions, and ensuring they retire in program order. It is also responsible for determining the static dependencies between instructions. Hwacha vector instructions must respect program order dependencies for each element of the vector, such as read-after-write, write-after-read and write-after-write. This allows for different elements of different vector instructions to be in flight at the same time as long as they are either independent, or for earlier elements than the dependent instruction. The master sequencer is only tracking the static architectural register dependencies and delegates the element-order portion of dependency tracking to the lane sequencers. Instructions are tracked as a series of micro-operations which, when all executed, complete a whole instruction. This finer granularity allows for more parallelism in the machine, at the cost of larger instruction-dependency-tracking hardware.

Each Hwacha lane has its own independent lane sequencer which accepts instruction issue at the same time as the master sequencer. As the lanes are operating independently, each lane sequencer must keep track of how many elements it has left to complete for each instruction. It also tracks microarchitectural dependencies in the execution unit such that no queue's are overrun and no other structural hazards occur. This dependency tracking also involves scheduling the micro-operations that each vector instruction is composed of. Because the register file is banked, each micro-operation takes multiple cycles to complete and these overlapping multi-cycle operations require careful scheduling to avoid interference. Once the lane sequencer determines a valid operation to schedule it will send it to the expander. The expander is simply a series of shift registers with some logic to populate

and manage them correctly for different micro-operations. The expander directly drives the control signals from these shift registers into the vector execution unit (VXU).

The VXU is primarily a datapath module that includes the vector register file, predicate register file, and functional units. Almost all of the VXU executes in a fire-and-forget manner where control signals flow through to drive multiplexers and other control circuitry from outside of the VXU. The multiple cycles of control signals from the expander are pipelined through each bank of the register file to correctly sequence each operation for each bank. As mentioned before the vector register file is built out of 1-read 1-write SRAMs and so has limited read and write bandwidth. On the other hand the predicate register file is built from flip-flops and so has multiple read-ports and write-ports which are implemented relatively cheaply because of the small number of bits in the predicate register file.

The last high-level component of the microarchitecture is the vector memory unit (VMU), which is responsible for tracking the status of memory operations as well as executing them. The three types of loads and stores supported by the microarchitecture are unit-stride, constant-stride, and indexed or scatter-gather. The microarchitecture tracks the outstanding requests with a pair of bit vectors allowing it to overlap the issuance of a second load or store with the completion of the prior operation. The data going to and coming from the VXU is sent through queues to enable some buffering for the low read and write bandwidth of the register file.

The next section will discuss how this microarchitecture and its design decisions relate to the previous section. Specifically the vector register file, cross-lane connections and functional units, and the memory subsystem will be analyzed for their benefits to and limits on the overall machines efficiency and performance.

4.4   Hwacha Microarchitecture Implementation Choices

In order to fully understand the microarchitecture of Hwacha, for the purposes of analysis, another layer of abstraction must be peeled back. Figure 4.2 shows a detailed diagram of the replicable lane of the Hwacha architecture.

In the upper left of the figure are the lane sequencer and expander, both described in the previous section. The lane sequencer has many detailed data structures attached to each entry but their details are not necessary to understand the design decisions of the microarchitecture so are omitted. The sequencer considers all micro-operations for scheduling but prioritizes the eldest micro-operation if possible. Most of the complexity of scheduling these multi-cycle operations on the hardware is encapsulated in the lane sequencer and the feedback it gets from the expander. The expander is directly below the lane sequencer, and as mentioned previously contains the hardware to expand a scheduled micro-operation into the requisite multi-cycle steps. These steps are stored in shift-registers arranged by microarchitectural resource used and the occupancy of these shift-registers is exported to
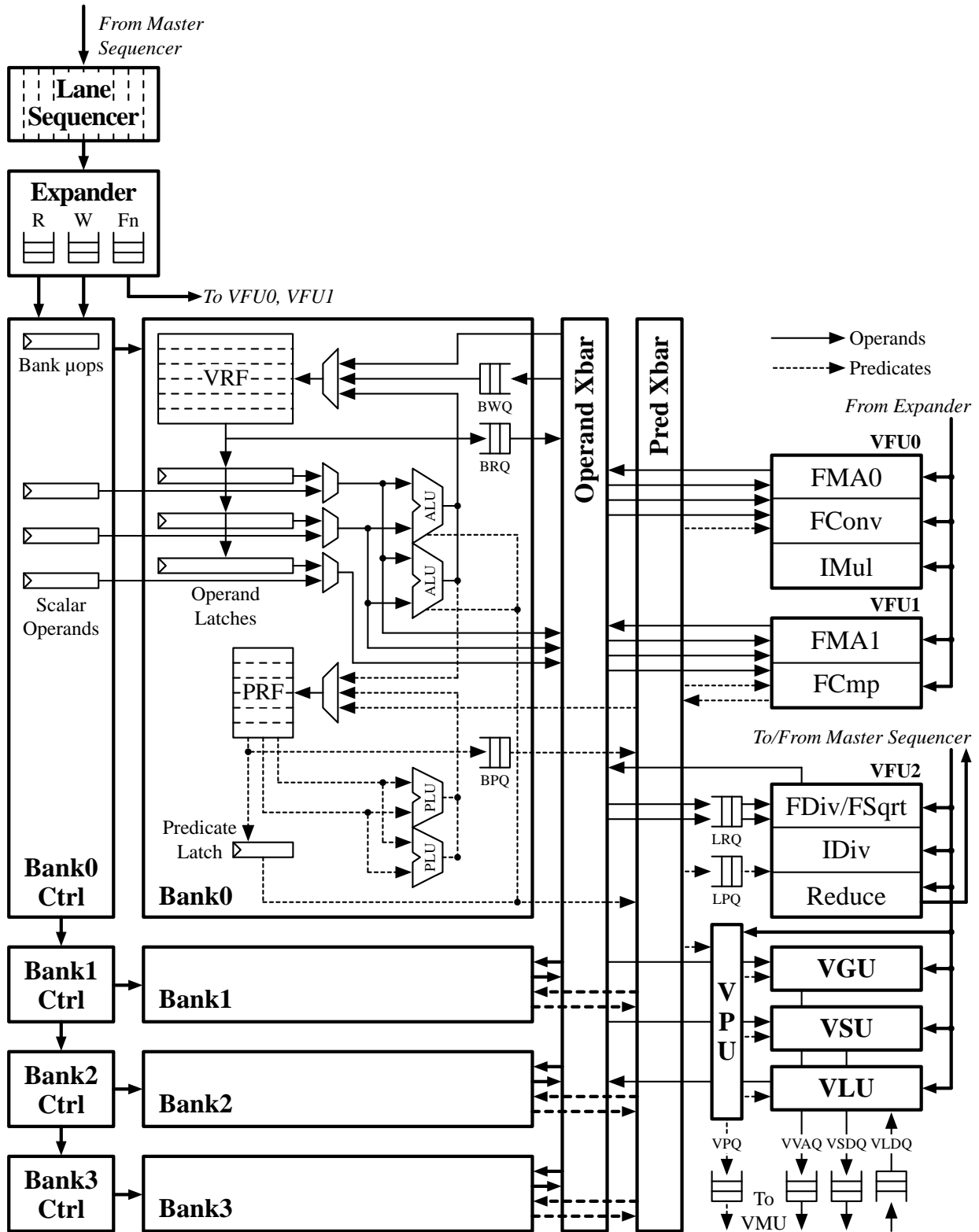
Figure 4.2: A detailed diagram of Hwacha lane.

the lane sequencer for conflict-free scheduling. The head element of these shift-registers are sent out to the rest of the lane, including the first bank and the functional units.

The banks are immediately below the expander in the figure. There are four banks in the normal configuration of the Hwacha microarchitecture, with the first bank being expanded to show its detail. The left side shows the bank control logic which turns micro-operations into bank control signals and acts as a pipeline register for the sequential ordering of bank operations. The bank itself is arranged around the one-read one-write SRAM-based vector register file bank, and most of the hardware deals with how operands are read out and written into this structure. The limited read bandwidth of the vector register file forces the microarchitecture to include operand latches located directly below the VRF. These operand latches maintain the values read out of the VRF until they are consumed by computation. The latch output is multiplexed with scalar operands to allow for the seamless vector-scalar operations supported by the Hwacha architecture. The output of these multiplexors is either consumed by the per-bank ALUs or sent to the operand crossbar for the shared functional units. The per-bank ALUs are simple integer-only functional units capable of two input logical operations, shifts, adds, comparisons, and a few miscellaneous bit manipulations. Below these is a similar set of register files, latches, and ALUs for the predicates. The predicate register file is made of flops but still uses operand latches to prepare predicates for the shared functional units to simplify the control logic for the crossbar. Each bank also contains a predicate logic unit (PLU) which can compute any arbitrary three-input logic function. Finally on the right hand side of the banks are a few different queues. All three, the bank write queue (BWQ), the bank read queue (BRQ), and the bank predicate queue (BPQ), interact with the vector memory unit and help reduce the stalls from the variable consumption and production of the outer memory system.

The operations that require the shared functional units, including memory operations, are sent over the operand crossbar, which routes the data to one of the three vector functional units (VFU) or the memory functional units. The three functional units have different processing capabilities as shown in the figure. There are two fused-multiply-add units (FMA), one floating-point conversion unit (FConv), one floating-point comparison unit (FCmp), one floating-point divide and square-root unit (FDiv/FSqrt), one integer multiply unit (IMul), one integer divide unit (IDiv), and one reduction unit (Reduce). These units are split up amongst the functional units by intuition to enable the highest degree of parallelism on common workloads. As the largest supported datatype in the architecture is only 64 bits and the vector register file is 128 bits wide each of the processing units are actually spatially subdivided into identical smaller units. Furthermore because Hwacha supports reduced and mixed-precision arithmetic down to 8-bit integer and 16-bit floating point the processing units are replicated for the smaller bit-width operations to enable full throughput on these smaller data-types. In addition the predicates are fed into the functional units to inhibit computations for elements that are disabled due to predication.

In the bottom right of the figure the portions of the VMU that live in the VXU are shown. These functional units are mostly for record keeping, buffering, and preparing data for consumption by the vector memory unit. In order to decouple the unpredictable latency of
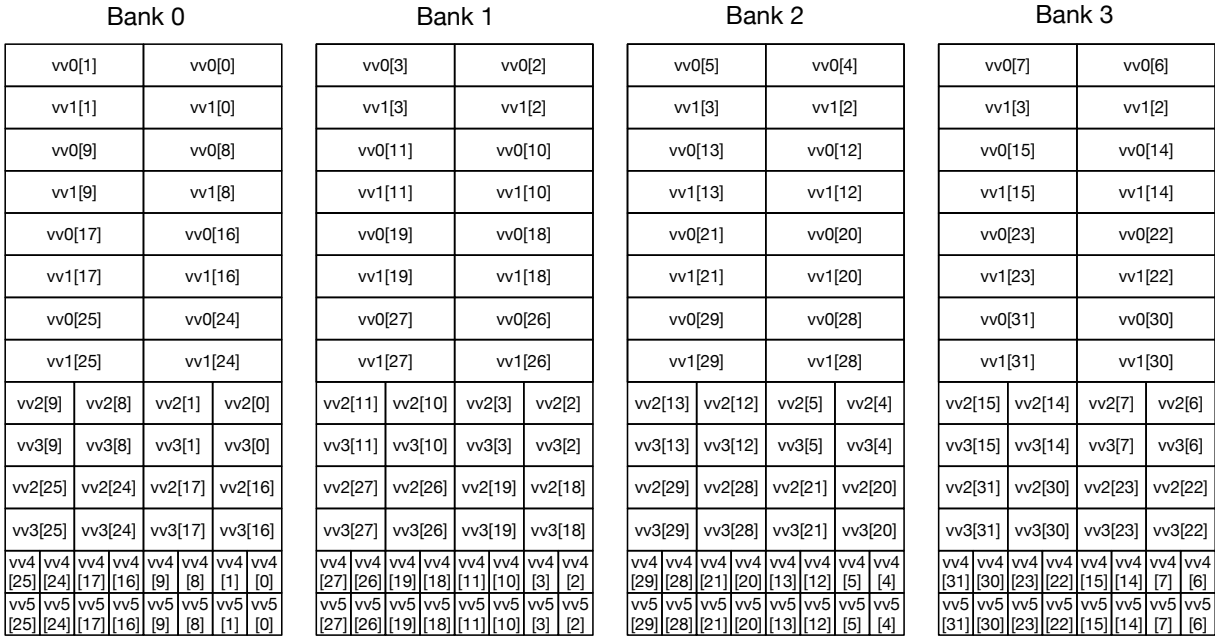
Bank 0			
vv0[1]		vv0[0]	
vv1[1]		vv1[0]	
vv0[9]		vv0[8]	
vv1[9]		vv1[8]	
vv0[17]		vv0[16]	
vv1[17]		vv1[16]	
vv0[25]		vv0[24]	
vv1[25]		vv1[24]	
vv2[9]	vv2[8]	vv2[1]	vv2[0]
vv3[9]	vv3[8]	vv3[1]	vv3[0]
vv2[25]	vv2[24]	vv2[17]	vv2[16]
vv3[25]	vv3[24]	vv3[17]	vv3[16]
vv4[25] vv4[24] vv4[17] vv4[16] vv4[9] vv4[8] vv4[1] vv4[0]			
vv5[25] vv5[24] vv5[17] vv5[16] vv5[9] vv5[8] vv5[1] vv5[0]			

Bank 1			
vv0[3]		vv0[2]	
vv1[3]		vv1[2]	
vv0[11]		vv0[10]	
vv1[11]		vv1[10]	
vv0[19]		vv0[18]	
vv1[19]		vv1[18]	
vv0[27]		vv0[26]	
vv1[27]		vv1[26]	
vv2[11]	vv2[10]	vv2[3]	vv2[2]
vv3[11]	vv3[10]	vv3[3]	vv3[2]
vv2[27]	vv2[26]	vv2[19]	vv2[18]
vv3[27]	vv3[26]	vv3[19]	vv3[18]
vv4[27] vv4[26] vv4[19] vv4[18] vv4[11] vv4[10] vv4[3] vv4[2]			
vv5[27] vv5[26] vv5[19] vv5[18] vv5[11] vv5[10] vv5[3] vv5[2]			

Bank 2			
vv0[5]		vv0[4]	
vv1[3]		vv1[2]	
vv0[13]		vv0[12]	
vv1[13]		vv1[12]	
vv0[21]		vv0[20]	
vv1[21]		vv1[20]	
vv0[29]		vv0[28]	
vv1[29]		vv1[28]	
vv2[13]	vv2[12]	vv2[5]	vv2[4]
vv3[13]	vv3[12]	vv3[5]	vv3[4]
vv2[29]	vv2[28]	vv2[21]	vv2[20]
vv3[29]	vv3[28]	vv3[21]	vv3[20]
vv4[29] vv4[28] vv4[21] vv4[20] vv4[13] vv4[12] vv4[5] vv4[4]			
vv5[29] vv5[28] vv5[21] vv5[20] vv5[13] vv5[12] vv5[5] vv5[4]			

Bank 3			
vv0[7]		vv0[6]	
vv1[3]		vv1[2]	
vv0[15]		vv0[14]	
vv1[15]		vv1[14]	
vv0[23]		vv0[22]	
vv1[23]		vv1[22]	
vv0[31]		vv0[30]	
vv1[31]		vv1[30]	
vv2[15]	vv2[14]	vv2[7]	vv2[6]
vv3[15]	vv3[14]	vv3[7]	vv3[6]
vv2[31]	vv2[30]	vv2[23]	vv2[22]
vv3[31]	vv3[30]	vv3[23]	vv3[22]
vv4[31] vv4[30] vv4[23] vv4[22] vv4[15] vv4[14] vv4[7] vv4[6]			
vv5[31] vv5[30] vv5[23] vv5[22] vv5[15] vv5[14] vv5[7] vv5[6]			

Figure 4.3: A detailed diagram of Hwacha's vector data and predicate register files.

the outer memory system from the predictable and regular latency of the execution pipeline these units contain buffers and flow control for those buffers to ensure none are overrun and put minimal back-pressure on the schedule of execution. At peak throughput in a four bank system the VXU could handle an instruction mix ratio of three two-input arithmetic operations, or three-input with one scalar input operations, to two load-store operations at full throughput. The sequencer may not be able to store this many instructions but a workload with this ratio will approach peak throughput as the sequencer contents vary between regions of higher or lower arithmetic intensity.

The remainder of this chapter explains how the three core components of vector architectures, from section 4.2, are implemented in the Hwacha microarchitecture, described above, and how these interactions affect the design and its efficiency.

Register File

As described above, Hwacha's vector register file is composed of four banks of simultaneous one-read and one-write SRAMs, with a data layout that depends on the current configuration. In order to get adequate throughput with this relatively low bandwidth register file it is critical to carefully arrange the data layout for minimal conflicts. Figure 4.3 shows the layout of data elements from different register widths in each bank of the register file. In this figure, *vv*0 and *vv*1 are 64-bit data registers, *vv*2 and *vv*3 are 32-bit registers, *vv*4 and *vv*5 are 16-bit registers the smallest configurable size in Hwacha. The elements for the widest

supported data type are simply striped across the banks, which allows for consecutive cache lines to be written directly into each SRAM bank. This also presumably matches a traditional outer memory system that optimizes for consecutive memory address access. For the smaller data types the elements included in each bank are chosen such that the same element index across all vector registers reside in the same register file bank. Not aligning the elements to the same banks would have meant a large increase in implicit cross-element communication, as any mixed precision operations would need data from multiple banks. In addition, the control logic responsible for scheduling this communication would be complicated as it would have to track multiple banks as the source of operands rather than in the current system having each bank contain all operands and outputs. The index and offset are calculated at configuration time allowing the indexing calculations that are preformed for each micro-operation to be straightforward repeated additions. The end result of this layout is that most operations are able to simply read the different width operands directly out of each bank and operate on them with minimal data movement.

The exact width and depth of the register file SRAMs were decided based on common technology patterns and diminishing returns of long vector lengths respectively. Most technologies only provide relatively narrow simultaneous one-read one-write so the reasonable powers of two are 64, 128, or 256. The wider SRAMs will have slower access times and can limit frequency of the overall design significantly, as the microarchitecture is primarily designed with the register file as the critical path. In addition, a larger physical SRAM width increases the minimum vector length required to fill each row of the different banks due to the packing constraints above. Operations with vector lengths that do not fill all banks have reduced register file bandwidth and will proceed at a slower effective rate. Even at smaller physical SRAM widths the above issue can present itself with small data-width vectors. For example, 16-bit registers require 32 elements in a 128-bit 4-bank register file to fully utilize the register file bandwidth compared to the 8 elements needed for the largest data width of 64-bits. On the other hand, smaller physical register widths will also lower the overall efficiency of the design by causing a relatively higher energy cost per bit read out of the register file, due to the increasing overhead of periphery circuitry. In addition, another factor in the SRAM width decision is how to match the bandwidth between the aggregate register file bandwidth, and the functional unit and the memory system bandwidth. The implications of these factors will be discussed in their related sections below. Overall a width of 128-bits balances the physical concerns, with the application mapping concerns, and the overall energy efficiency of the design but other sizes for different design points are certainly possible.

The predicate register file, despite being a SCM-based, also has some constraints on its design. Early microarchitectural implementations of Hwacha included each potential read port as a dedicated port in the design. This caused several congestion issues and the read ports are now implemented in a similar multiplexed manner as the vector register file, although with more total ports. The predicate register file mirrors the 16-bit register layout in the vector register file with eight single-bit entries per row of the predicate register file. This enables the same limited communication between elements that the vector register file

layout allows, as well as the ability to read out the maximum number of predicate elements needed in one cycle from a single row.

The freedom to make all these layout choices and optimize for technology constraints is a result of architectural decisions that focus on energy efficiency. The primary architectural choice is making the vector register file opaque and non-transferable between machine configurations. This architectural choice is pervasive and effects many portions of the architecture and microarchitecture. The register file being opaque means that accessing a register using a different data-width or type than it was configured with, also known as type punning, is not guaranteed to have consistent results and thus effectively heavily discouraged. This allows the different width registers to be packed non-uniformly but optimally for their own width, avoiding wasted register file space, or slow small data-width operations. The non-transferable state aspect of the architecture allows the register file to be laid out optimally for the current set of configured registers without needing to move or save the data from the previous configuration. However, this freedom is not without trade-offs. Applications that require type punning, or would reconfigure the register file while still needing to maintain its contents, won't map well to the architecture. There will be an associated loss of energy efficiency on these applications but the design expects them to be rare and the benefits to other applications that do map well to outweigh those losses.

The vector register file design choices are not made in a vacuum and are affected by other microarchitectural decisions in the design. The next two subsections discuss both how technological constraints apply to the functional units and memory subsystem, as well as how these two design components affect and are affected by the vector register file.

Interconnect and Functional Units

Figure 4.4 shows the internal structure of the operand crossbars and the connections to the mixed-precision functional units. In the upper left is the expanded interface coming from the first bank. Each operand latch is exposed to the crossbar to potentially be selected between at each destination which correspond to each input of the functional units. These three operand latches may contain data that is packed 64-bit, 32-bit, or 16-bit elements, but the crossbar operates on data at the same width as the physical vector register file banks, 128-bits. A more expensive crossbar could be implemented to allow for some or all cross-element interactions, but, without a detailed study of the potential benefits, the large number of increased multiplexers at the output of the crossbar would greatly increase the energy consumed for standard vector operations.

Each output port of the crossbar is consumed by at least two functional units with one output port being consumed by three functional units. In the upper right of the figure one of these functional units, integer multiplication (IMul), is expanded to show the subdivision of the inputs into the smaller subdivided functional units. The output ports of the crossbar are represented by the multiplexers in the center of the figure and show the sharing of outputs between different functional units. The design decision of which functional units to attach to each output is similar to issue slot allotment to functional units in an out-of-
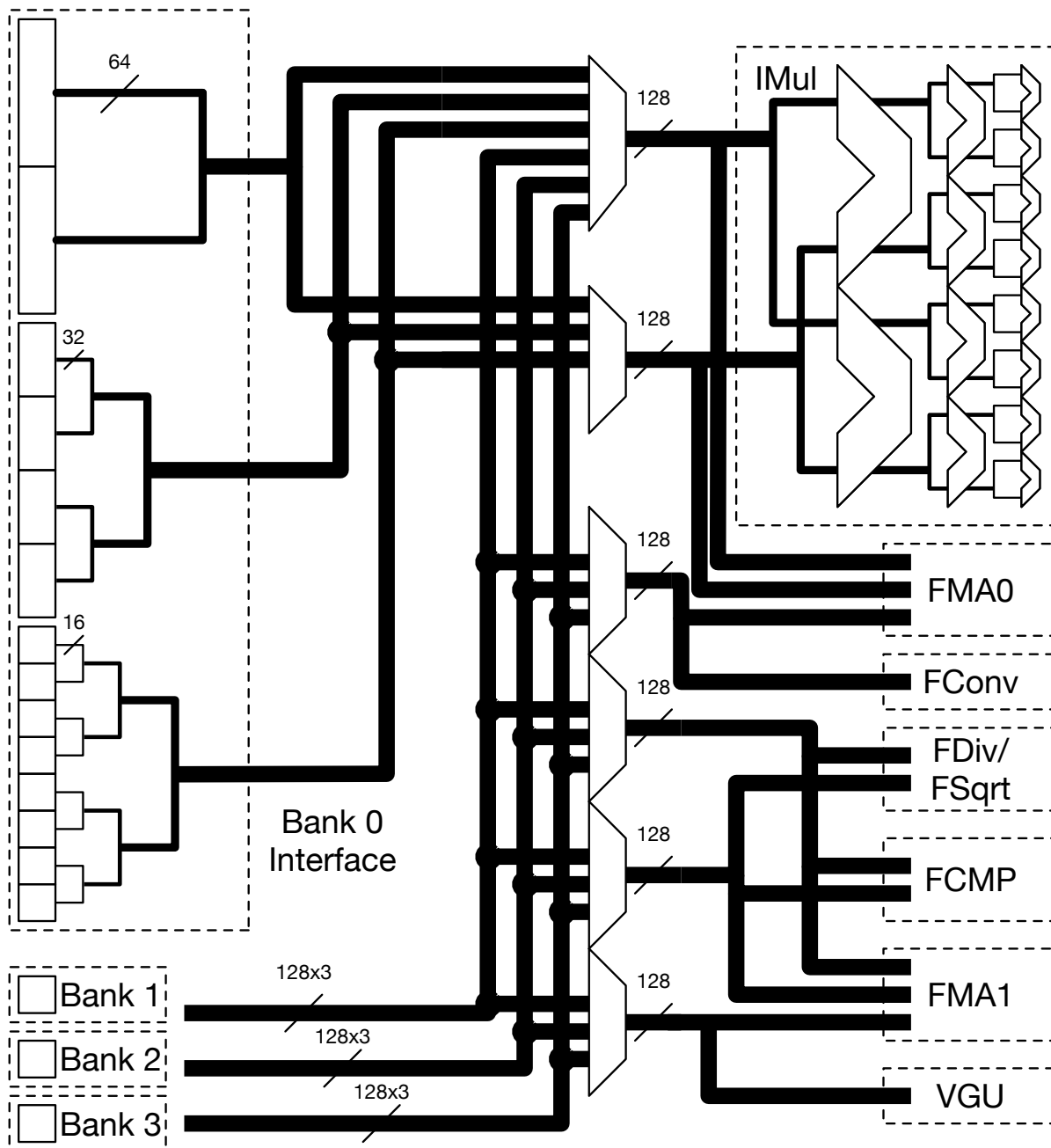
Figure 4.4: A detailed diagram of Hwacha's shared functional units.

order architecture. Depending on the proportion of instructions expected in common code certain functional units are more or less likely to be needed at the same time. Carefully spreading the most common functional units across distinct output ports of the crossbar results in minimal structural hazards, while minimizing overprovisioning. It is possible to fully eliminate structural hazards on the crossbar ports by making each functional unit input a fully independent output of the crossbar, but this greatly increases the size of the crossbar. The bi-section bandwidth of the crossbar is a good approximation for the size of the crossbar, and in Figure 4.4 the total bi-section bandwidth is $128\,bits/latch \times 3\,latches/bank \times 4\,banks \times 6\,ports$ or $9216\,bits/cycle$. At typical frequencies this will be well over $9\,Tbps$, compared with an independent port per functional unit which instead requires $128 \times 3 \times 4 \times 14$ or $21504\,bits/cycle$ or $21.5\,Tbps$. This additional cost will rarely be used and can never be fully utilized as the number of inputs, 12, is fewer than the number of outputs, 14. Despite local interconnect scaling relatively well with technology nodes the cost of the crossbar in power and area still should be minimized by careful selection of which ports are shared among functional units. The result path is omitted from this figure as it has a simpler organization with each functional-unit output being capable of arriving at any bank for a total bisection bandwidth of $128 \times 4 \times 6$ or $3072\,bits/cycle$. This crossbar will be much more likely to be fully utilized and has a much lower bandwidth requirement already so further reducing it at the cost of performance is not productive.

The final subsection will discuss on how the memory system is impacted by technological constraints and how it influences the register file and functional unit microarchitecture.

Memory System

Figure 4.5 shows the internal structure of a single lane's vector memory unit responsible for making requests for data from the outer memory system. The left-hand side shows the issue unit, Ibox, which is responsible for managing the control flow through the different parts of the vector memory unit. The issue unit of the VMU is driven separately from the execution unit with both being directly connected to the frontend of Hwacha that is responsible for decoding and issuing instructions. The issue unit can track up to two operations at a time allowing it to overlap the completion of the one operation with the request phase of another. The predicate handling and address generation are pipelined to maintain throughput without long critical paths, and the figure uses numeral suffixes to denote pipeline stage. In addition, to provide elasticity for these pipelines, each stage is composed with queues that enable each stage to continue to operate for longer in the face of back-pressure from the outer memory system. The predicate handling portion, PBox0 and PBox1, of the VMU receives the predicates from the vector execution unit after being read out of the banks. The first stage determines which addresses will need to actually be affected, skipping ahead when possible for large groups of unset predicates. The second stage aligns and expands the input predicates, that are on an element level, to the byte granularity of the outer memory interface. The address-generation pipeline has three stages, with the first two being aligned with the predicate stages. The first address-generation stage is address translation which

uses a fully associative eight-entry TLB. This is followed by an alignment stage that coalesces the elements which can fit in a single outer memory request. The last stage of the address pipeline corresponds with two other units in the figure, the store box and load box, which are responsible for moving the data between the memory system and the execution lane. These three units all communicate with the outer memory system almost directly. A small interface adapter sits between the rest of the VMU and the outer memory system and is responsible for marshalling the different pieces of data onto the single outgoing memory channel.

The biggest impact that the memory interface has on the rest of the design is that its access width and bandwidth drive the overall machine data consumption and production balance. The main effect of technology constraints on the VMU is a limit on the practical size of access width and realizable total bandwidth. The memory access width is matched to the physical width of the register file which allows for the common case of unit-stride loads or stores, without predication, to proceed at full rate without any stalls when at the largest element size. The element width of the load-store unit affects this process due to the element packing in the register file. The additional buffering on the inputs to the VMU needs to be deep enough to allow for the repacking into the native memory format. The end result of this bandwidth balancing is that the vector register file still has more read and write bandwidth than the memory interface but can overlap with computation to fully utilize the bandwidth of both subsystems.

Hwacha supports several different access patterns to main memory at varying levels of performance and efficiency. The most efficient access pattern for the VMU and for nearly all vector designs is the unit-stride load or store. In this case each vector element is accessed from consecutive memory addresses. This allows for many optimizations such as fewer translations, easy address calculations, and effective use of memory access width. These combine to allow the VMU to achieve full bandwidth utilization with minimal control overhead on unit-strided accesses. Constant-stride accesses are those with a fixed address gap between each vector element and this adds additional complexity to translation and effective use of memory access width. The VMU is still able to use the straightforward repeated addition strategy used for address generation that was used for unit-stride access, but with a larger addend although both come from immediates in the instruction bits. The fact that the stride may not be a power of two unlike the unit-stride accesses complicates determining which elements require a new address translation, and in general will cause more translations that a unit stride, increasing the energy spent on these accesses. In addition, depending on element size and stride, few or no elements will share the same memory-access-width bytes and so many more requests will be made to the outer memory system. Finally, the scatter-gather memory operations are the least efficient because even the address generation is without constraint. Each address now requires a full virtual address-width addition as well as another vector data element to be read from the register file per element accessed. Because there is no way to know where this new address will be, each and every element will now require a separate translation and a separate request to the outer memory system. The cost of these less regular access patterns is high enough that most programs will avoid them whenever possible to achieve peak performance. Therefore, the VMU is designed to only support a sin-

From VPU

From Master Sequencer

From VGU

From VSU

To VLU

VPQ

VVAQ

VSDQ

VLDQ

→ Operands

⇢ Predicates

Ibox VMU Issue

PBox0 address mask generation

ABox0 address translation

PPQ

VPAQ

PBox1 store mask generation

ABox1 coalescer

PAQ

ABox2 metadata generation

SBox store aligner

LBox vector load table

Request

Store Data

Meta Data

Tag

Load Data

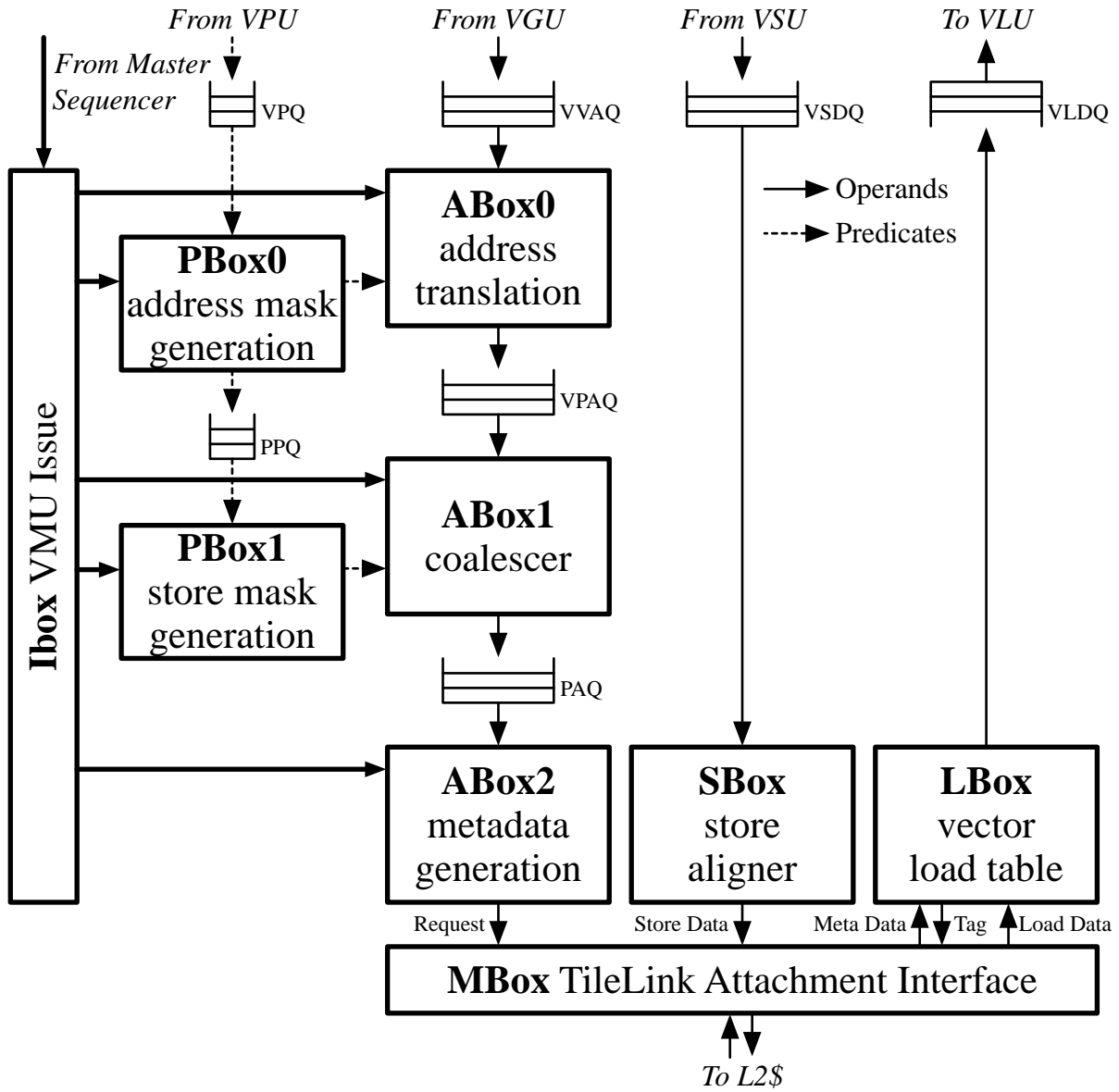**MBox** TileLink Attachment Interface

To L2$

Figure 4.5: A detailed diagram of Hwacha memory unit.

gle element per cycle and avoid the extra static cost of more address generation, translation, and memory request hardware that would most often go unused.

Another aspect of the vector memory unit that affects the entire machine is memory ordering. Hwacha's memory model is generally relaxed but still requires that loads must wait for stores to be fully issued, while loads can proceed with other loads in flight. The VMU does not issue memory requests out of order so the ordering issue with stores could be solved by a store-forwarding system. The ability to forward store data to loads is much more expensive in a vector design with potentially hundreds of outstanding store or load requests. Therefore Hwacha's VMU uses a more conservative approach and does not allow loads to begin after a store until the store is complete. Hwacha's memory system can also span multiple VMUs in the multi-lane case. This case complicates the process of waiting for store completion and a separate machine-wide unit is included to track the completion of each individual VMUs operations and release VMUs that are waiting to begin another operation.

The overall VMU design allows it to be flexible in its interface with the vector execution unit and the outer memory system. This flexibility enables the VMU to be optimized for efficiency of operation without over-constraining the rest of the design. The maximum bandwidth is achievable only for unit-stride accesses but this restriction still allows many computations to achieve high performance. Arithmetically intense applications can do even better and can achieve nearly full utilization of the functional units even with the moderate bandwidth provided by the VMU. This balance between bandwidth, flexible access patterns, and efficiency give the VMU the ability to complement the rest of the design and allow the overall system to operate at high efficiency.

The effect of technology constraints on Hwacha's register file, functional units, and memory system restricts the width of each lane and the sizes and interconnectivity of many microarchitectural components. The general microarchitecture has been designed to provide an efficient system within these constraints, but specific physical implementations have also impacted the design of Hwacha over time. The physical design restrictions and implications of Hwacha's design decisions are discussed along with several silicon implementations in the following chapter.

Chapter 5

Silicon Implementations of Hwacha

This chapter discusses the silicon implementations of the most recent version of Hwacha as discussed in previous chapters. These implementations contributed to the evolution of the microarchitecture and physical design techniques for building realizable and energy efficient designs. Each section discusses the design of each chip and the changes made to the microarchitecture of Hwacha as a result of physical design feedback.

Section 5.1 details the first tapeout to include the complete Hwacha version 4 implementation, Hurricane-1, built in ST-28. Section 5.2 describes the second chip in this series, Hurricane-2, which explored the use of small voltage and frequency domains with a multi-lane Hwacha implementation. Section 5.3 uses a new technology, TSMC 16, and a new tool flow, Hammer [92], to build a multi-core Hwacha implementation called Eagle. Finally section 5.4 describes an even larger instance of Hwacha, EagleX, showing the productivity of the new tool flow. Table 5 summarizes the key parameters of each of these chips. All chips presented in this chapter were part of collaborative group projects.

5.1 Hurricane-1

Hurricane-1 is a dual-core $7.98mm^2$ SoC design in the ST-28 fully depleted silicon-on-insulator (FD-SOI) process, and was the first complete implementation of Hwacha. It was taped-out in March of 2016, and received back fully assembled by October of 2016. The focus on Hurricane-1 was an exploration of fine-grained dynamic-voltage-and-frequency(DVFS) scaling augmented with on-chip voltage converters. The on-chip DC-DC voltage converters

Chip	Area	Main Cores	Hwacha Lanes	Peak Frequency	Peak Energy-Efficiency
Hurricane-1 [97]	$7.98mm^2$	2	2	475MHz	19.6 DP-GFLOPS/W
Hurricane-2 [75]	$16.77mm^2$	1	2	260MHz	36.5 HP-GFLOPS/W
Eagle [76]	$24.01mm^2$	8	8	1.44GHz	209.5 HP-GFLOPS/W
EagleX	$56.25mm^2$	20	20	Unmeasured	Unmeasured

allow for a tight feedback control feedback loop as the voltage regulation requires no off-chip components which would and significant communication latency. Other attempts to fully remove the dependency on off-chip components, such as FIVR[17] from Intel, have also been made to reduce the latency of voltage mode transitions. From a vector architecture perspective this was also the first chip to include a complete implementation of Hwacha version 4, and the first multi-Hwacha chip. This implementation thus provided vital feedback on the microarchitecture of Hwacha.
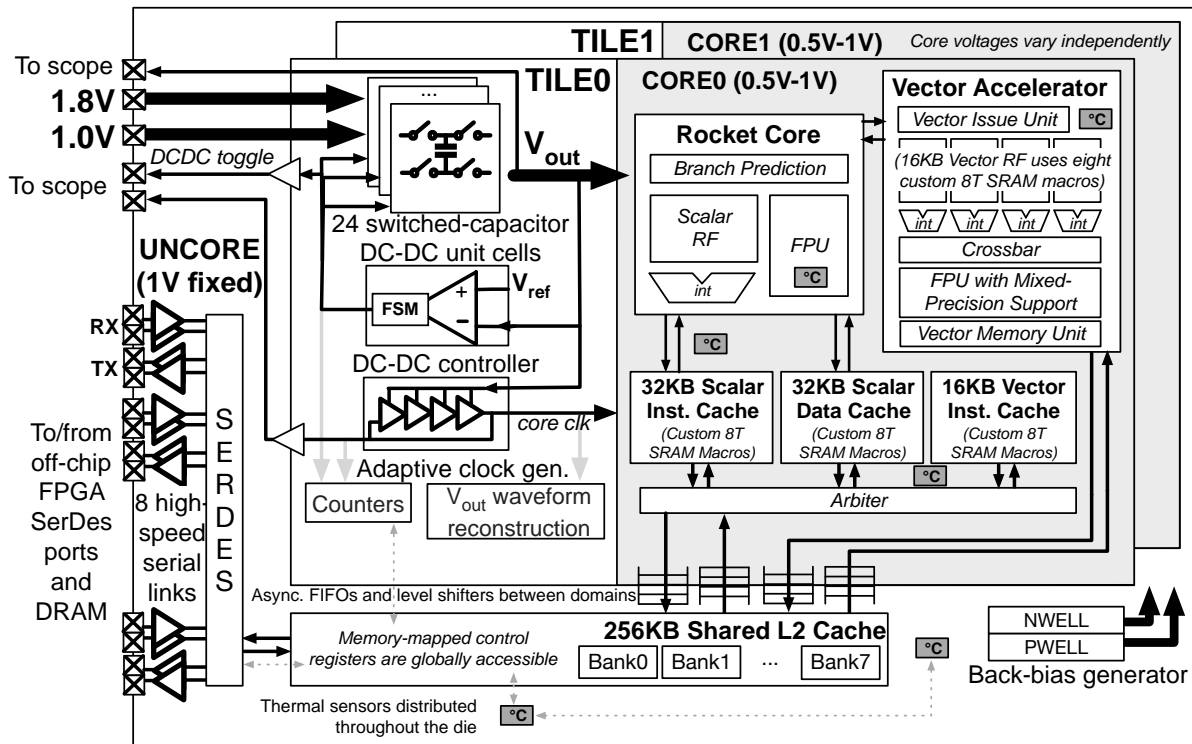

Figure 5.1: An overview of the Hurricane-1 SoC and its components [97].

Figure 5.1 shows a block diagram of the entire Hurricane-1 SoC. Each tile is homogeneous and so only the details of the first tile are shown. Shown on the left side of the figure, each tile includes a set of DC-DC unit cells to regulate its voltage, as well as an adaptive clock generator to adjust to the variable voltage from the DC-DC voltage converter. The DC-DC unit cells have 4 different modes of operation that allow them to provide $0.5V$, $0.67V$, $0.9V$, or a pass-through voltage. The regulation of this supply allows for significant ripple to avoid waste during switching events and improve the overall efficiency of the converter. The adaptive clock generator tracks this rippling voltage with a replica circuit to produce an optimal clock frequency for each tile, allowing a cycle-to-cycle variation of frequency. There is also a set of performance counters in the tile to track the operation of the DC-DC converter and the adaptive clock generator from either of the two tiles. These counters

enable power management software to introspect on current operation and adjust the power strategy dynamically. The main component of the tiles are the cores, which include a scalar Rocket core and a single-lane Hwacha implementation. Hurricane-1's cores implement the RV64IMAFDXhwacha RISC-V user-level ISA version 2.0 and the 1.9 version of the privileged spec with machine, supervisor, and user mode supported. Hurricane-1 is the first fabricated instance of Hwacha version 4, which added predication, mixed-precision, and a more complete general-purpose data-parallel ISA, compared with Hwacha version 3. The microarchitecture in this version increased the number of peak FLOPS available per cycle, and increased the ability to exploit instruction-level parallelism. Each of these cores has a set of private caches including a 32KiB scalar instruction cache, a 32KiB scalar data cache, and a 16KiB vector instruction cache. These three private caches share a single port to the globally shared 256KiB L2 cache, which Hwacha directly connects to, obviating the need for a private vector data cache. The memory system can be backed by either eight 4 Gbps serial links, or a slow parallel interface that also doubles as the semi-hosting or tethering interface for the chip. Because Hurricane-1 lacks many features of a complete computer, the semi-hosting environment allows it to use a host computer to proxy its access to things like disks, UARTs, and other real-world devices. In addition, there are temperature sensors, labeled as ^oC in the figure, scattered around the SoC to provide another DVFS control input.

Hurricane-1's principal design goal was to test using small voltage domains with the ability to rapidly change their voltage and frequency. Dynamic voltage and frequency scaling allows the system to take advantage of changes in application operating modes. For example where applications are waiting on devices or where there is not enough parallelism to utilize all processors. To investigate the potential benefit of smaller domains each core was given its own voltage and frequency domain with accompanying low-latency on-chip switched capacitor DC-DC converters. This shrinks the size of the domains from previous implementations from the tile and memory system to just a single tile. These small domains allow for many more overall power states of the system, allowing it to more precisely adapt to the requirements of the application. In addition, the on-chip converters greatly reduce the latency of voltage changes. Decreasing the latency of a voltage change allows for the power states to more closely follow the optimal operating point of the current application. By combining these rapid changes with an adaptive clock generator the overall system can approach the optimum voltage-frequency point as quickly as possible.

Hurricane-1 was produced with a physical design flow based on a Synopsys reference methodology hand-modified for the project. The flow uses Synopsys Design Compiler (DC) and IC Compiler (ICC), with several hierarchical modules for the distinct components like the tiles, serial links, and the L2 cache. The RTL design of Hurricane-1 is written in Chisel2 [9], which is a hardware construction language embedded in Scala that was developed at UC Berkeley in the 2010s. Designs in Chisel are able to leverage the multi-paradigm capabilities of Scala to produce hardware designs that can be more easily parameterized. Chisel is not directly consumable by commercial CAD tools currently and so must be processed to produce Verilog suitable for fabrication. After the Verilog RTL is produced by Chisel a few more post-processing steps are needed to produce a fully realizable design. The first

of these post-processing steps is mapping the RTL synchronous memories to vendor and technology-specific memory macros. In Hurricane-1's flow this is handled with a python script that uses the size, shape, and type of memories used in Chisel to map to a fixed set of available memory macros. The script is able to tile macros in either dimension to allow for any-sized RTL memory to be mapped with some potential sub-optimality if the RTL memory has very different dimensions from the available macros. The final processing step before the Verilog is ready for physical design is to add the IO cells and pad frame to the design. When Hurricane-1 was designed Chisel was missing some functionality that would make these macro and top-level integration tasks easier, but the process of post-processing the design gave compelling use cases for updating Chisel to fix this deficiency.

At this point the Verilog is ready for physical design but there are many constraints that still need to be manually created for the design to work well through the flow. The majority of these constraints are in the floorplan and timing constraints for the edge of the different blocks, which are both hand written for the particular design. With the Verilog and constraints ready, the flow begins by running synthesis on each of the blocks. The sequence of steps following that are a hybrid of bottom-up and top-down approaches. After block-level synthesis, there is a top-level synthesis stage followed by floorplanning. An annotated diephoto with an overlaid floorplan can be seen in Figure 5.3. After floorplanning the physical placement information is used to guide physical synthesis on the blocks and then place-and-route completes the block-level flow. Finally, using the completed blocks, physical-synthesis is run at the top-level followed by the top-level place-and-route stage. The generated GDS is then passed to Mentor Calibre for design rule checks (DRC) and layout versus schematic (LVS) checks. The design, flow, and technology require that many hand fixes were performed to ensure the final Hurricane-1 design was DRC and LVS clean. No other physical verification is done, but short sequences of instructions were run on the RTL and gate-level simulations to verify a minimum level of functionality.

This style of flow had been used for several chips prior without the block-level separation, but the new design put strain on the flow and several physical design issues occurred during tapeout. One of the issues with many voltage domains is determining how to partition on-chip resources for each domain. The power grid in Hurricane-1 was designed globally for simplicity of implementation but required that top-level metal layer routing resources be shared between domains. In previous smaller designs this had not caused issues but the increase in size and number of domains caused significant power delivery issues that were not addressed during development and led to a reduced voltage-frequency operating range as seen in Figure 5.4. Another issue that hadn't been encountered before was the latency of a block-level change appearing in the final GDS was increased by requiring a new top-level run even if the changes were isolated to local logic in the block.

In addition to these general flow issues, there were several physical design issues that occurred due to the new vector architecture of Hwacha and the increased performance requirements from including an L2 cache and a second core. The first issue encountered was related to the predication support added by the new version of Hwacha. This feature is implemented as a separate set of registers that rather than being packed into dense SRAM
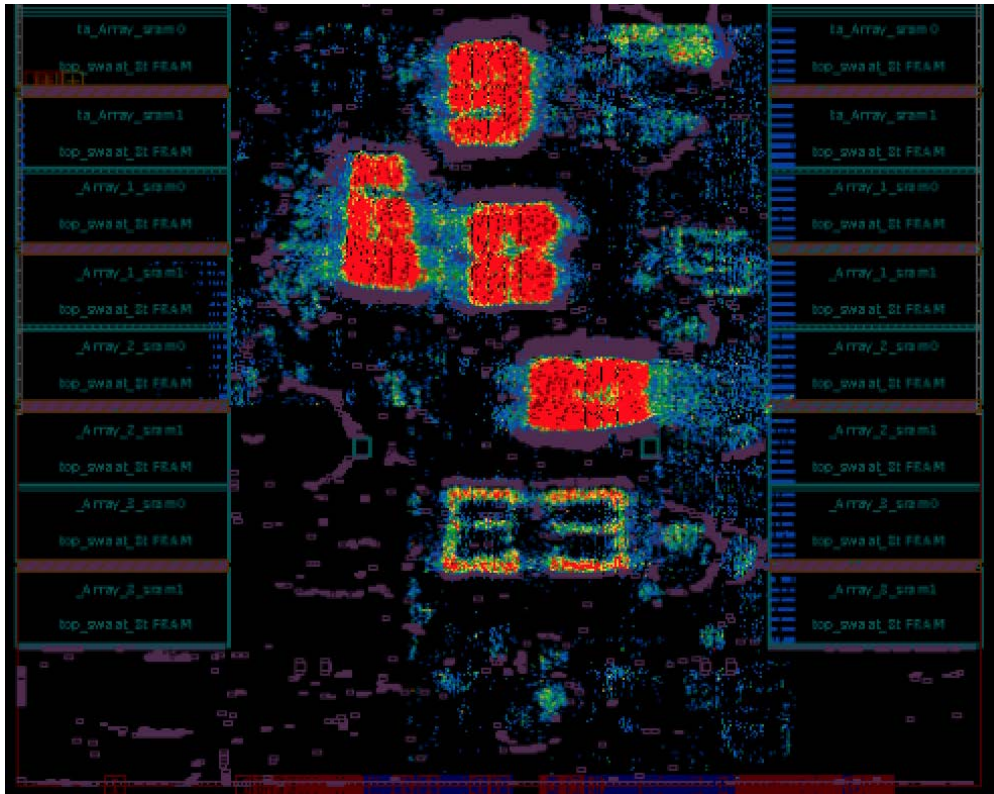
Figure 5.2: Example of Hwacha's predicate register file congestion in Hurricane-1.

are simply implemented as an SCM memory. The number of bits needed in the predicate register file is much smaller and its access patterns are more diverse so the regularity and density of SRAM did not suit it well. In addition, the many uses of these predicate registers to gate off different aspects of the microarchitecture leads to a high degree of connectivity and interconnect surrounding the predicate register file. Initially this caused a high degree of congestion around each of the Hwacha register file banks. Figure 5.2 shows the portion of the tile that contains Hwacha's register file banks colored based on routing congestion. All of the red areas are too congested to successfully route and will cause LVS shorts later in the flow, and each of the four predicate register file banks are entirely consumed by congestion. This physical design feedback was addressed both in the microarchitecture by using the operand latch structure for the predicate register file as well as the vector register file, and in the physical design flow by running a special congestion-reduction optimization pass during place and route.

Another more significant issue was congestion and placement density in the crossbar between the tiles and the L2 cache. Since this was the first time a chip had included this L2 cache and the number of connections out of the tiles was doubled this was a significantly higher throughput crossbar than those previously built. In addition, the power-routing
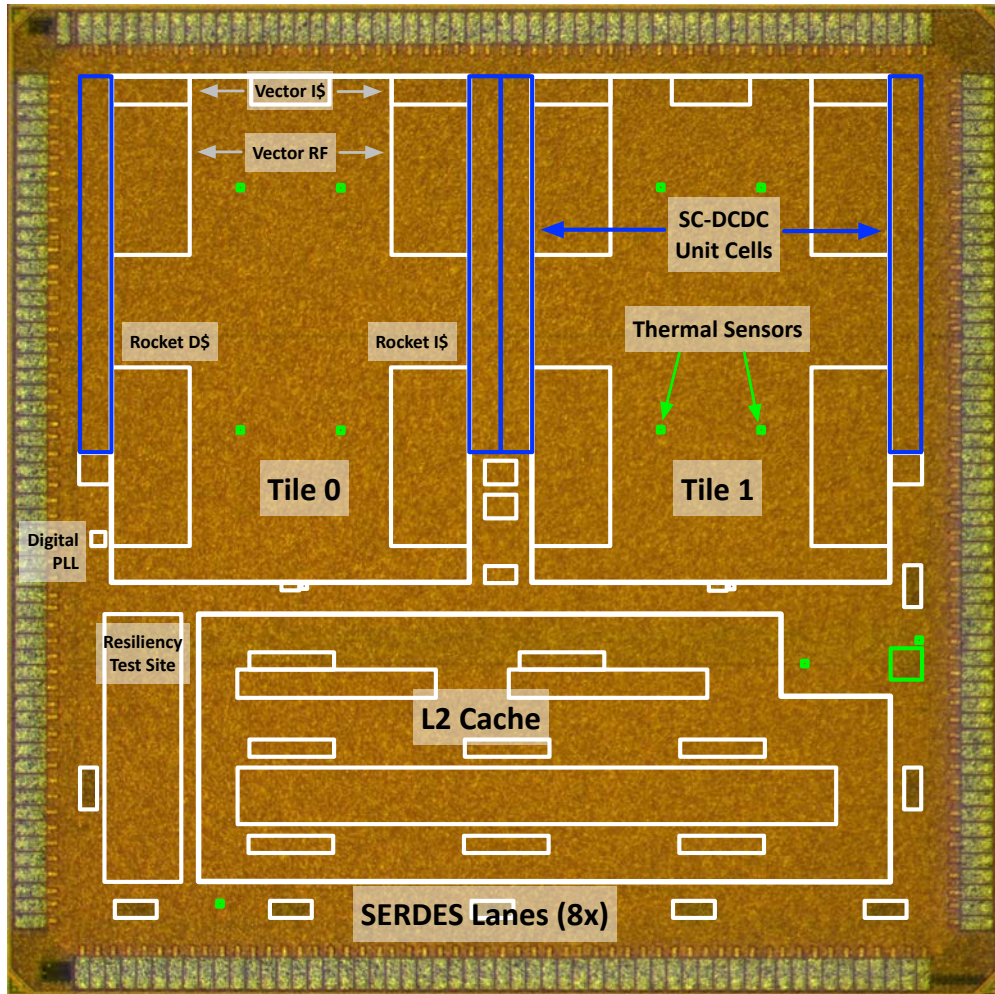
Figure 5.3: An annotated diephoto of the Hurricane-1 SoC.

strategy used in this technology and in these designs used more routing tracks than would be typical for a digital design with fewer or larger voltage domains. Power straps for each voltage domain were available throughout the entire outer memory system domain which reduced the number of available routing tracks. The ST28 technology metal stackup available for this tapeout had only 10 metal layers which is relatively few compared to modern technologies that often have metal layers often numbering in the mid-teens [14][81]. The allotted area for the crossbar was also relatively small occupying only $0.3mm^2$ between the tiles and the outer memory system. The number of Tilelink channels per tile was three: one shared between both of the instruction caches, one for Rocket's data cache, and one for Hwacha's vector memory unit. With the initial number of L2 cache banks there were 8 input channels, one for each bank. This lead to a lot of congestion at the target clock frequency of 800MHz. Reducing the number of banks to 4 reduced the amount of logic in the crossbar by half and
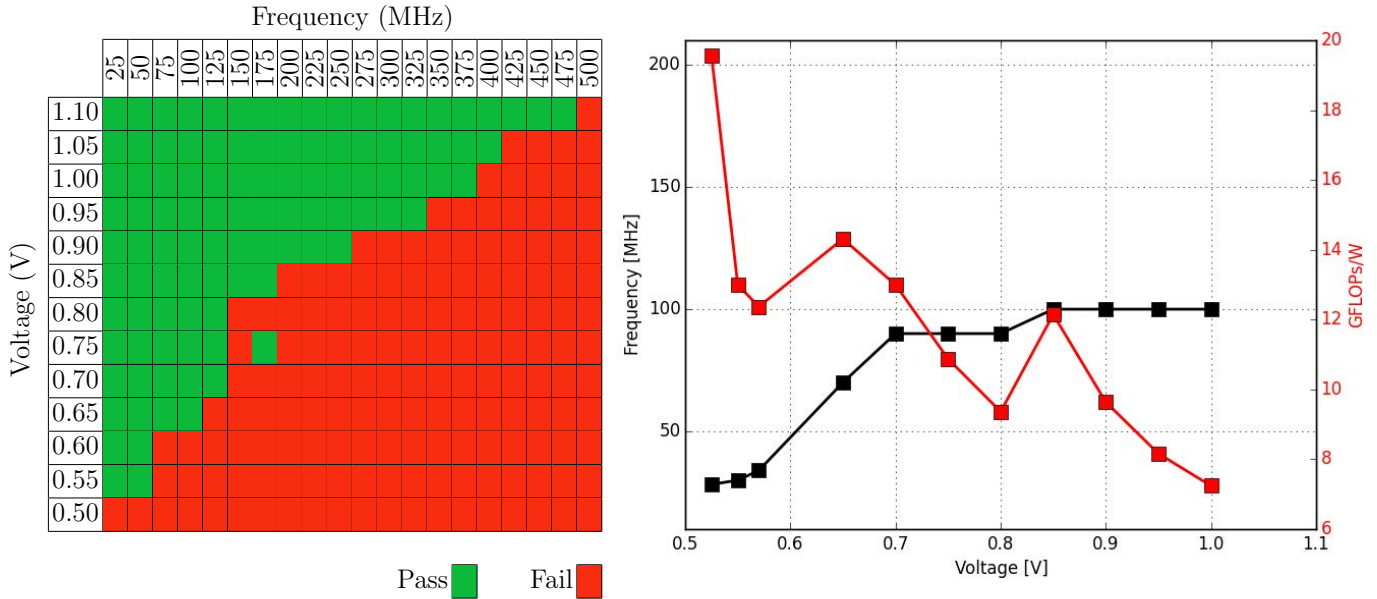
Figure 5.4: Hurricane-1 frequency and efficiency at different operating voltages.

it was then able to fit in the allocated space. Unfortunately, at this point in time the L2 cache was unable to be configured to reduce the number of channels without reducing its bandwidth and capacity. This caused the final design to have reduced memory bandwidth and diminished the effective peak performance expected from Hwacha, from ~93% to ~35%. Future modifications to the L2 cache would allow separation of the number of banks from the number of input channels as well as from the capacity.

Hurricane-1 was packaged with wire-bonding and then placed on a small daughterboard which would connect to an FPGA for testing. Figure 5.4 shows the maximum operating frequency for different voltages at which Hurricane-1 can operate. In the left half of the figure, each box represent a specific voltage and frequency pair with the green boxes corresponding to one core being able to boot and run a small program. The right half of the figure shows the energy efficiency of a double-precision matrix multiplication kernel running out of the L2 cache over a select set of operating points. The power-grid issues discussed above limit the frequency over the chip as a whole to below the sign-off frequency of 600 MHz. The non-monotonic behavior around 175 MHz is attributed to a small supply resonance observed on-die. In addition, Hwacha's high power draw exacerbates these issues and results in the leveling off of frequency in the energy-efficiency graph. Hurricane-1 achieves a peak energy efficiency of 19.6 GFLOPS/W at 525 mV and 28.3 MHz.

At lower frequencies the chip functions as expected and is capable of running complex workloads including booting Linux and running applications under operating-system support, using both the slow parallel interface and a high-speed serial link. Hurricane-1 is also able to use the second tile as a power-management unit and adapt the first tiles voltage level, based on microarchitectural counters. Unfortunately, the adaptive clock generation does not

function as well as on previous test chips and so couldn't be enabled in tandem with the on-chip DC-DC converters.

For the first silicon implementation of the new Hwacha design, Hurricane-1 was relatively successful. The additional integration of analog circuit components focused much of the physical design effort on building a system that would support those unique blocks rather than the more standard processor elements. This led to compromise in the design that reduced the performance of the Hwacha implementation. However, it provided a lot of valuable physical design feedback to the microarchitecture, and while unable to achieve extremely high energy efficiency it still provides a useful basis for future implementations to compare against.

5.2   Hurricane 2

Hurricane-2 is a single-core $16.77mm^2$ SoC design with a dual-lane Hwacha implementation in the ST-28 FD-SOI process. It was taped-out in March of 2017, and received back fully assembled in August of 2018. Hurricane-2 focused on improving on several implementation challenges encountered in previous Berkeley test chips. These challenges include: providing a high-speed off-chip memory interface, scaling up the size of a Hwacha instance to multiple lanes, and adding a separate system or power-management unit (PMU). The high-speed memory interface will be addressed with two options, a set of high-speed serial links and a more traditional DDR interface. The other challenges will be addressed simply by including a multi-lane Hwacha instance and a small core serving as a PMU. In addition to these goals of improving system realism, Hurricane-2 again uses the on-chip switched capacitor DC-DC converters and adaptive clock generation. To extend the fine-grained voltage-frequency domain idea even further, the voltage and frequency domains are even smaller than in Hurricane-1. The applications core is now made of two voltage-frequency domains, one for Hwacha and one for the scalar control core Rocket. This would allow the system to adapt its energy usage for applications with varying use of the vector accelerator.

Figure 5.5 shows a block diagram of the entire system. The single application core consists of a Rocket 5-stage single-issue in-order core that implements the RV64G ISA version 2.1 and version 1.9 of the privileged ISA. Rocket has a 16KiB instruction cache, a 16KiB data cache, branch predictors, page-table walker, and is capable of computing a single- or double-precision fused multiply-add (FMA) every cycle. The instance of Hwacha attached to Rocket in Hurricane-2 has two vector execution lanes. Each lane has four banks SRAM composing a 16KiB vector register file (VRF), a 128-bit vector memory port to the L2 cache, four double-precision, eight single-precision, and 16 half-precision FMA units. In total this means the Hwacha instance is capable of 16 double-precision, 32 single-precision, or 64 half-precision floating-point operations per cycle. These two lanes do not operate in lock step but do share the same voltage and frequency. Hwacha is designed to be relatively decoupled from Rocket using decoupled interfaces to connect to both Rocket and the outer memory system, so the modifications needed to handle the truly asynchronous domain crossing were

simple procedural replacements of standard queues with asynchronous queues. The only other changes to the tile compared to Hurricane-1 are additional microarchitectural counters tracking the occupancy of various queues in Hwacha, and cache accesses in Rocket.
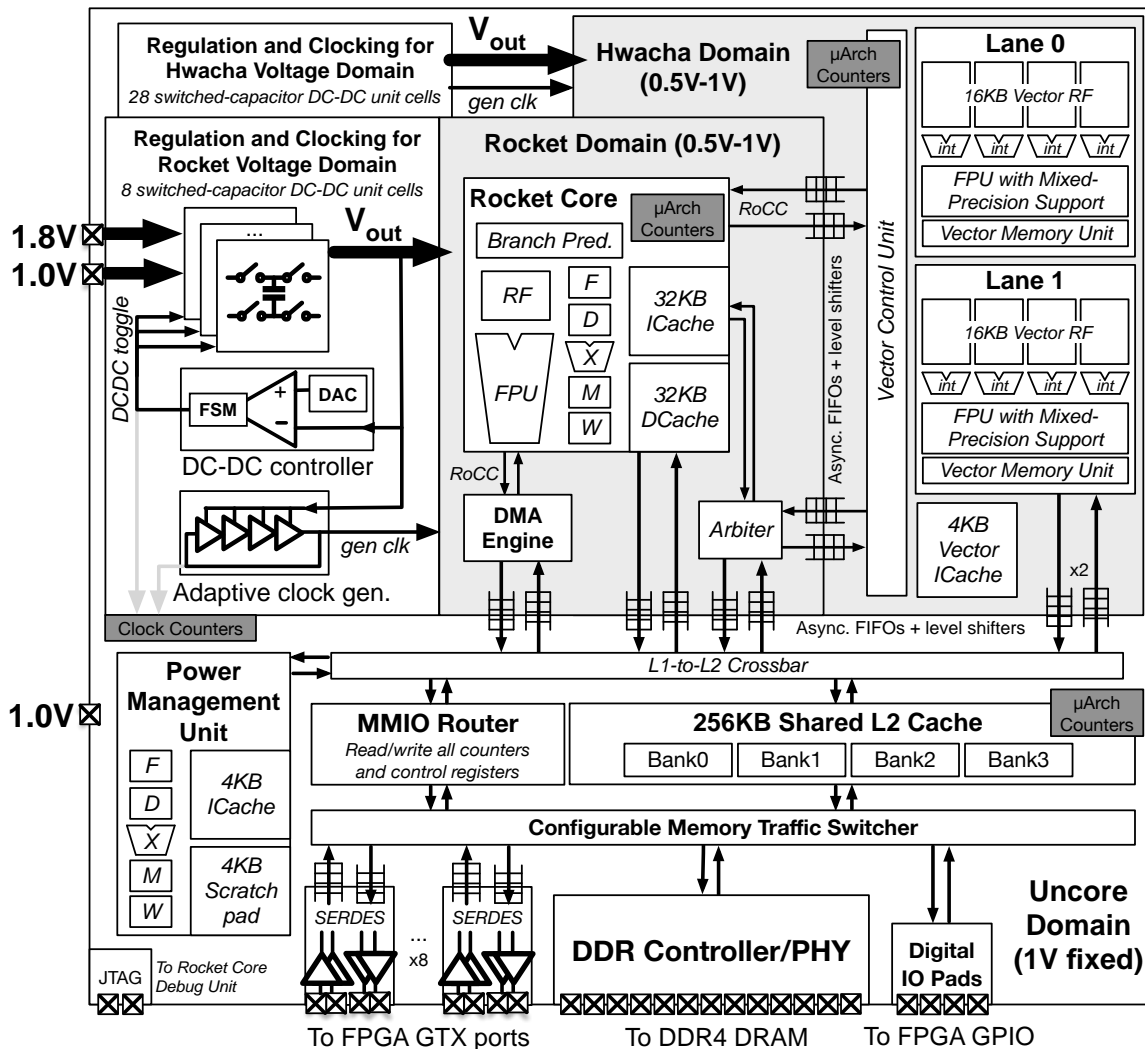

Figure 5.5: An overview of the Hurricane-2 SoC and its components.

The lower portion of the figure, the uncore, includes a lot of the changes compared with Hurricane-1. On the left side, the PMU in Hurricane-2 is a reduced functionality and scaled-down Rocket core. The PMU only has a 4KiB instruction cache, a 4KiB data scratch pad, and no floating-point support. Both the application and PMU cores share the same outer memory system which consists of a globally shared 256KiB 4-bank, 8-way L2 cache. Hurricane-2 has a unified memory-mapped IO router to communicate with on-chip control and status registers, as opposed to the ad-hoc implementation in Hurricane-1. This made the

addition of microarchitectural counters across the system easier as well as making control of the two off-chip memory interfaces more straightforward. The L2 cache can make its backing requests through one of three distinct interfaces. The highest performance option would be to use the DDR4 PHY, which was donated IP from a commercial vendor. The next highest performance would be a set of eight 5Gbps custom SERDES, with an improved bitrate over Hurricane-1 and several control bug-fixes. Finally the most reliable but slowest off-chip interface is a low-speed 4-bit parallel interface, which also operates as the tethered semi-hosting link. The off-chip interface used by the L2 cache is runtime configurable and only a single interface is designed to be active at once since they all share the same physical memory space. The only other off-chip interfaces were a few miscellaneous test inputs and outputs for the analog components, and a JTAG interface designed to connect to the debug unit on the main application core. The DC-DC on-chip converters and the adaptive clock generators are unchanged compared with Hurricane-1, except for the addition of microarchitectural counters.

One of the experiments intended to be tried with Hurricane-2 was more detailed power management algorithms based on an increased number of microarchitectural counters. The L2 cache has counters for the number of hits and misses to help a power management algorithm determine the memory-boundedness of the current program phase. Hwacha also includes counters to keep track of the number of entries filled in its internal queues, and its scheduler. Three different power-management algorithms were implemented and are discussed and compared in Figure 5.8

Hurricane-2 was built with a very similar physical design flow compared to Hurricane-1. The largest difference was the conversion from Chisel2 to Chisel3 as the primary RTL design language. Chisel3 was built to enable transformations to be performed on the RTL by using a specified intermediate representation, FIRRTL [48]. The three main features enabled by FIRRTL are an integrated SRAM-macro mapping, clock-domain enumeration, and separation of test harness from the design. The FIRRTL integrated SRAM-macro mapping procedure, gave the flexibility to have a cost function for the mappings which require multiple macros per Chisel memory, as well as enabling the connection of custom foundry ports to constants or status and control registers. Clock-domain enumeration in FIRRTL enabled the scripted generation of clock-domain-crossing separation, and extra verification of a design with many distinct clock domains. Finally, separating the test harness from the design enabled easier verification as well as smoother integration of design changes into the physical design flow.

There were several physical design challenges during the implementation of Hurricane-2. Figure 5.6 shows an annotated die photo of the chip, with an overlaid floorplan. In order to accommodate the fixed size and shape of the DDR4 PHY many non-standard decisions were made with respect to the floorplan and physical design of the chip. As seen in the floorplan the DDR4 PHY nearly doubles the size of the chip and its large height forces the remaining logic to have a very high aspect ratio, stretching the design vertically. The reduced length of the available chip perimeter for non-DDR usage meant that the bank of high-speed serial links needed to be split into two separate banks causing routing issues
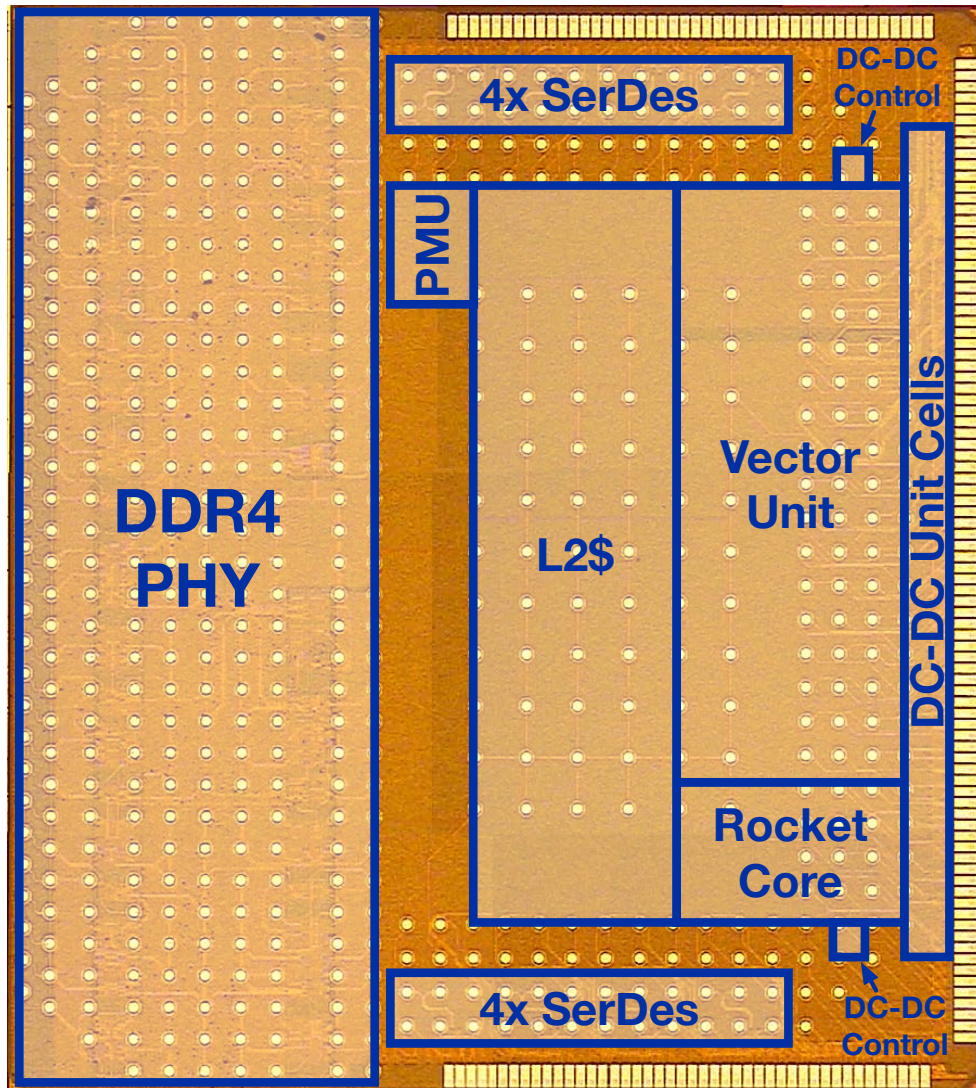
Figure 5.6: An annotated diephoto of the Hurricane-2 SoC.

in the uncore. In addition, Hurricane-2 marks the transition point for Berkeley test chips moving from wire bonded to flip-chip packaging, and as such Hurricane-2 attempts to reduce the risk involved with a new packaging technology by enabling either method to be used on a chip by chip basis. All parts of the chip except the DDR4 PHY are able to function in a wire-bonded package as seen by the large wirebond pads on the right, top and bottom sides of the chip. In addition, these pads are also connected to bumps to allow for a flip-chip packaging methodology to connect all portions of the chip.

As this was the first chip to use bumps the power distribution from the bumps to the highest metal layers was relatively limited. Inspecting the bumps above the L2 cache in the

figure, which are all power and ground bumps, you can see that only very thin lines connections exist between them. As no vias can be placed under the bumps the only connections to the top-layer metal power grid are on these very thin wires, which means there are very few. This power distribution issue was compounded by design rules related to a change in bump density from the DDR4 PHY to the rest of the chip. This design rule prohibited a complete and regular grid across the entire chip, which leads to sections of the chip that are quite far from power or ground bumps. In addition, the number of DC-DC unit cells allocated for the application core was not changed from Hurricane-1, so the same 24 cells now are required to power approximately twice as much area and compute. This lead to a suspected serious static voltage droop to the chip causing a much lower measured frequency than the sign off frequency, as seen in Figure 5.8.

Another physical design issue was implementing the crossbar between the L2 cache and the application core. This issue was similar to the problem in Hurricane-1 but the aspect ratio of the chip also impacted the amount of space available for the crossbar. The number of outputs from the application core is smaller than the aggregate number in Hurricane-1 because we eliminate the instruction cache and data cache port of the second core. However, the place-and-route tool was unable to realize this crossbar and we were again forced to reduce the number of banks in the L2 from 8 to 4. The L2 cache still did not support orthogonal configuration of the bandwidth and number of banks so the taped-out configuration had half as many L2 cache banks, bandwidth, and size compared to the ideal configuration. This coupling is especially visible in Figure 5.7 where the four L2 banks are clearly only occupying the top half of the area in the uncore. In addition, the requirement to update the hand-coded floorplan slowed the process of adapting RTL changes to the physical design flow. In this case there are potentially floorplan changes or CAD tool parameters that could have relieved the congestion issues, especially given that the uncore is relatively sparsely populated. This general rigidity of the physical design flow in the face of difficult-to-realize designs eventually lead to the redesign of the flow discussed in Section 5.3.

Due to the change in packaging strategy and the integration of large complex third-party IP, Hurricane-2 took a very long time to begin and eventually complete testing. Unlike on Hurricane-1, the custom high-speed SERDES were unable to be used as backing memory. Bit-error rates could be collected using the builtin pseudorandom binary sequence (PRBS) generators for some of the lanes. Unfortunately, after measuring these bit errors across all chips and all lanes, no single pair of transmitters and receivers worked. Since there is no ability to configure the system to use one link for sending data and one link for receiving data this means that the links are unable to be used as backing memory. The root cause of the poor error rates and low yield appears to be a physical design marginality issue since several of the links are able to work well but as a whole many of them fail frequently.

With the high-speed serial links unable to be used as backing memory the DDR PHY became the only chance for a high-performance memory system. Unfortunately, despite a lot of effort debugging the DDR it was unable to be used as backing memory.

With no functional high-bandwidth off-chip interfaces the remaining testing of efficiency and DVFS algorithms were all done with workloads that fit in the on-chip caches. The
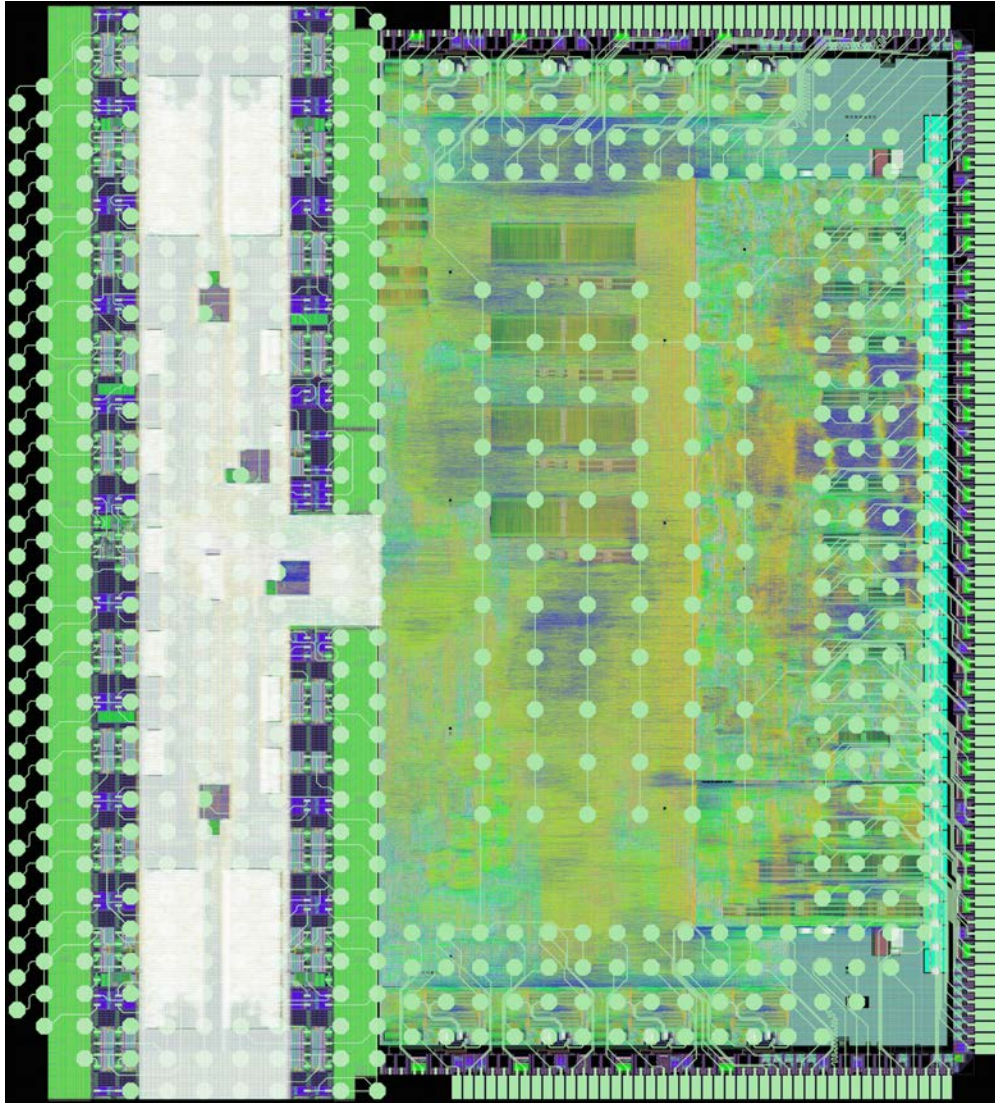
Figure 5.7: A detailed GDS plot of the Hurricane-2 SoC.

right subfigure of Figure 5.8 shows Hurricane-2's operating frequency at various voltages. As noted above the maximum frequency is well below signoff of 600 MHz due to the poor power connections. The poor power network also limits the minimum operating voltage, which is 200 mV above Hurricane-1's, reducing the system's peak energy efficiency. The most energy-efficient operating point is at 780 mV and 115 MHz, where Hwacha achieves 22.3 double-precision GFLOPS/W and 36.5 half-precision GFLOPS/W. Hwacha is able to perform four times as many half-precision operations as double-precision operations per cycle but the energy-efficiency is limited by the size of the on-chip cache. In order to keep the higher number of operations in flight, Hwacha requires a longer vector length which for matrix
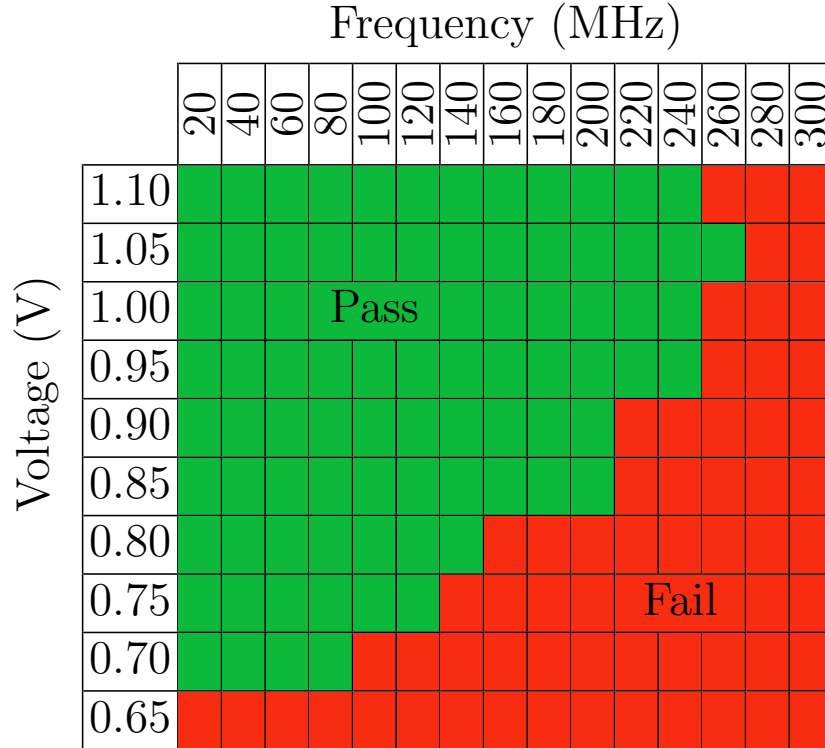
Figure 5.8: A shmoo plot showing valid Hurricane-2 voltage-frequency operating point.

multiplication increases the working-set size by more than the reduction in bytes needed per operation. This causes a bottleneck for the half-precision operations which reduces the utilization of the functional units and therefore the energy-efficiency.

Figure 5.9 shows a comparison between four different adaptive voltage scaling (AVS) algorithms applied to three synthetic benchmarks executed and measured on Hurricane-2. The baseline algorithm (none) runs the application core at maximum voltage and frequency for the entire duration. The simple algorithm replicates the voltage-frequency management of [54] by increasing voltage and frequency during periods of high activity as seen by increases in the DC-DC toggle rate. The last two algorithms are driven by the microarchitectural performance counters that were added to the system. They monitor the miss rates of the L1 data caches (AVS L1) and L2 cache (AVS L2) respectively, and decrease voltage and frequency when the core is in a memory-bound program phase.

The benchmarks run on the application core and repeatedly alternate between computing a median filter and performing a generic matrix multiply (GEMM) of 24-, 64-, or 128-element square double-precision matrices. This alternation approximates a simple image processing application with simple preprocessing followed by machine-learning inference that will be accelerated with a data-parallel processing unit. The 24-element dataset fits in L1 cache, so no cache misses occur, obviating the adaptive algorithms. The 64-element dataset fits in L2 cache, but not the L1 cache. The adaptive algorithms are able to identify the memory-bound
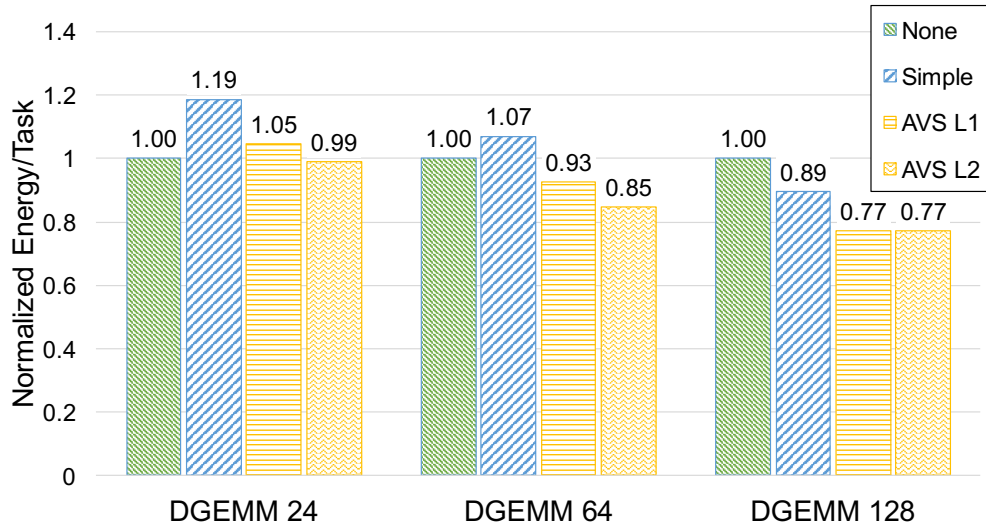
Figure 5.9: A comparison of DVFS algorithms on synthetic GEMM benchmarks on Hurricane-2.

regions, but the L1 monitor has false positives for phases that miss in the L1 but hit in the L2, and so reduces operating voltage prematurely in the 64-element case wasting energy. Finally, in the 128-element benchmark, all L1 misses become L2 misses, so both adaptive algorithms successfully slow down the core during memory-bound phases, saving energy in the core. These measurements demonstrate that fine-grained power management based on monitoring microarchitectural counters, when they are available, outperforms management solutions based on monitoring power consumption due to its faster, deterministic, and direct response. Specifically, the example in Figure 5.9 demonstrates that the addition and fine-grained monitoring of just two key microarchitectural performance counters can provide up to 14% additional energy savings compared to traditional DVFS algorithms. This experiment is benefited by the long latency and low bandwidth of the only usable off-chip memory interface on Hurricane-2. A higher-performance memory system would shrink the amount of time it would be worthwhile to operate the core in a low-power state. On the other hand, applications that have idle portions or need to read from a slower off-chip device, like a disk or sensor, could have even longer portions of time that would benefit from low-power states.

Overall, Hurricane-2 showed the necessity of a physical design flow that can adapt to changes in the RTL without causing serious quality-of-results issues. Because Hwacha is highly reliant on cache size or bandwidth in proportion to its functional-unit bandwidth, the entire system must be optimized as a whole to ensure high-performance and thus high energy-efficiency. In addition, having RTL designs that can adapt to different system level requirements in a orthogonal fashion can make both of the above issues more tractable to solve. Future designs attempt to address both of these by upgrading the physical design flow and updating the RTL designs to both use generators.

5.3   Eagle

Eagle is an 8-core $24.01mm^2$ SoC with a single Hwacha lane per core implemented in a TSMC 16nm FinFET process. It was taped-out in June of 2018, and received back fully assembled by December of 2019. Eagle was the first chip physically designed using the Hammer[92] agile physical design generator. The Eagle SoC was designed to be more realistic and more similar to mobile processors like those currently in cell phones and other edge devices. Such devices often include, large on-chip caches, many cores, accelerators, high-speed off-chip interfaces, and many connections to off-chip peripherals. This goal drove the size of the design up architecturally and physically, with the final design being over 125 million gates and $24.9mm^2$. The Hwacha instance was reverted to a single-lane implementation relying on the additional cores to increase the system's aggregate throughput. With so many vector units on chip a single L2 cache could no longer support all of them without massive physical design issues around the crossbar, which had previously caused issues even with two vector units. Anticipating this issue Eagle was designed with clusters of tiles that shared an L2 and a larger L3 which backed each of the cluster L2s. Eagle also includes a set of 28 Gbps serial links for off-chip communication, which were designed in BAG[25], rather than hand crafted as the links on Hurricane-1 and -2. There are 5 instances of an experimental PLL on Eagle, designed to clock each cluster and the uncluster separately. Finally, the uncluster contains a set of peripherals common to test chips and SoCs, including a JTAG debugger, UART, SPI, I2C, programmable GPIOs, an interrupt controller, a scratchpad, and a small system-management core.

Figure 5.10 contains a detailed block diagram of the entire SoC. The chip is broadly split into two halves, the top half of the figure represents the clusters and their contents, and the bottom half of the figure contains everything else, labeled uncluster. The upper right of the figure shows the internals of a single cluster, which is replicated four times on Eagle. Each cluster is a separate voltage and frequency domain designed to be driven by off-chip supplies, unlike the on-chip DC-DC converters of the Hurricane series. The clusters have two identical tiles, which each have a scalar Rocket core, a single-lane Hwacha instance, and L1 caches. These two tiles communicate via synchronous rational crossings with a fixed ratio of two tile cycles per L2 cache cycle. The divider for this clock is included in the cluster such that only a single clock, supplied by the PLL and clock multiplexer structure shown in gray next to the cluster, needs to be input to the block. The L2 cache is shared by both tiles and is inclusive of their L1 caches. It has four banks, and a total capacity of 256KiB, with each of these banks being capable of making independent requests to the next level of the memory system, the L3 cache. The communication between the L2 cache and the L3 employs asynchronous queues with level shifters as there is no guarantee that either the voltage of the frequency between the cluster and the uncluster will match.

To the left of the cluster in the figure is an expanded version of a single tile. Each tile contains a Rocket in-order, 5-stage processor, labeled scalar core, implementing the RV64GC ISA capable of up to two floating-point operations per cycle. Rocket has a separate 16KiB L1 instruction cache and a 16KiB L1 data cache. The Rocket custom coprocessor interface
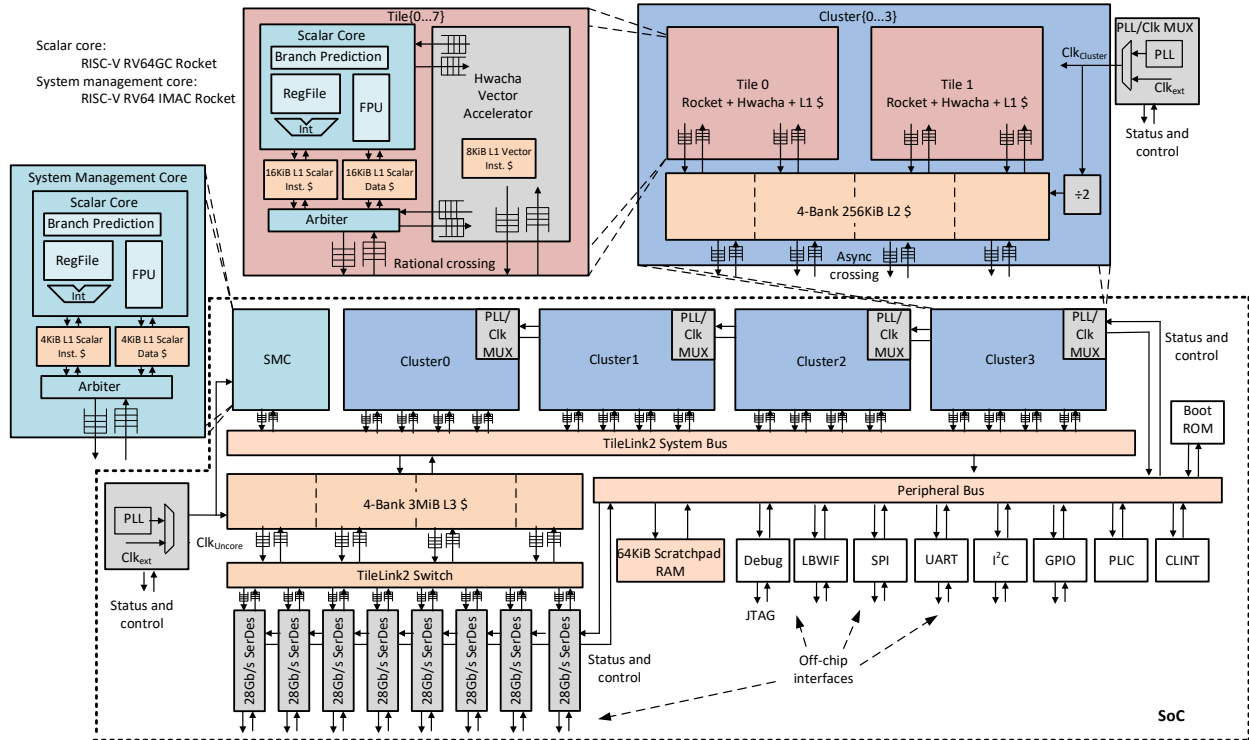
Figure 5.10: An overview of the Eagle SoC and its components.

(RoCC) connects to the Hwacha accelerator. This Hwacha instance has an 8KiB L1 vector instruction cache. Hwacha's instruction cache, and both of Rocket's caches share one Tilelink interface to the L2 cache, while Hwacha's vector memory unit has a dedicated port. All of the tiles interfaces to the L2 occur through rational clock crossings that operate at a fixed ratio of two tile cycles to one L2 cycle.

The rest of the figure constitutes the uncluster, and everything except the gray analog blocks operate in a single clock and voltage domain. The system-management core, on the far left, is a ninth Rocket core which is configured to be much smaller, only supporting RV64IMAC with 4KiB instruction and data caches. Its primary purpose is to interact with the other input-output devices via their memory-mapped register interfaces. This includes setting up the PLLs, SerDes, and the off-chip interfaces. The largest occupant of the uncluster by size is the inclusive 4MiB 4-bank L3 cache. The L3 can make requests for off-chip memory via a Tilelink switch that enables either the eight SerDes or the low bandwidth digital interface to be used. In addition there are two smaller memories in the uncluster: a small boot ROM that contains the zero-stage bootloader that is executed on reset, and a 64KiB scratchpad RAM that can be used for a first-stage bootloader.

The final components of the uncluster are the off-chip interfaces that support the mobile-like SoC design. The SPI interface can be used to load programs or bootloaders from a microSD card. The UART interface is used as the standard way to interface with the chip

via a console. The I2C interface was designed to control off-chip peripherals, specifically a set of programmable DC-DC converters on-board that can drive each cluster's voltage. Eagle also contains a set of general-purpose IOs (GPIOs) that could be used for any purpose, but were mostly intended to potentially aid in demos of the system interacting with the real world. All of these peripherals together with the analog devices on chip, and multiple cores make Eagle a reasonable analog to common mobile or edge SoCs.

In addition to mimicking mobile SoCs, Eagle was also designed to advance the concept of agile hardware design with generators. Generators are a design principle used to architect complex highly configurable systems. In a generator-based system, the designer builds a tool that generates an instance of the system rather than the designer directly building the instance. This tool in this case is a program that when executed with a given set of parameters produces the instance that corresponds to those parameters. Hurricane-1 and -2 were roughly built in the generator style, but only had generators for some components of the RTL. In contrast, Eagle has nearly all of the RTL built with generators. In addition, Eagle takes the important step of building the physical design flow with a generator as well. As noted above, the lack of responsiveness of the physical design flow to changes in the design had been a pain point in the previous test chips.

Using a generator-based design methodology has many benefits for building hardware. The highly parameterized nature of the design allows the same tool to be used for many different designs and adapt to the current instance. Because the generator is adaptable, it can also be used in many different situations. The reuse of the generator means that development time spent improving the generator will be useful many more times in the future. This allows the generator to be finely tuned for performance and stability over the many design it is used for, without fearing that this effort will be lost as soon as the next design begins. In addition to these benefits, the parameterization of the design forces the developer to think about more design points and corner cases increasing the robustness of the design. Being able to customize the parameters of a generator to a specific use case can also improve overall system efficiency by building a more specialized design point. Finally, reusing the generator decreases the amount of time spent in future design iterations saving time and effort.

Figure 5.11 shows the entire Eagle chip generator decomposed into its individual generators which are arranged by types. The RTL generators are in the upper left, with the RTL transformations directly below them. The physical design collateral to the right feeds into the physical design generator along with the output of the RTL generators and the analog generators, in the upper right of the figure. Finally the figure also includes the non-generator portions of the physical design flow in the bottom right.

Eagle's RTL generator is composed of many distinct generators stitched together to form the entire system. The primary generator is Rocket-Chip, which includes the scalar Rocket core, Tilelink interconnect network, interrupt infrastructure, and generic SoC topology [7]. Eagle also obviously includes the Hwacha generator, to complete the core design. The cache hierarchy is created with the SiFive composable cache, the only closed source component of the chip, which is arranged by the custom chip RTL into the on-chip topology with
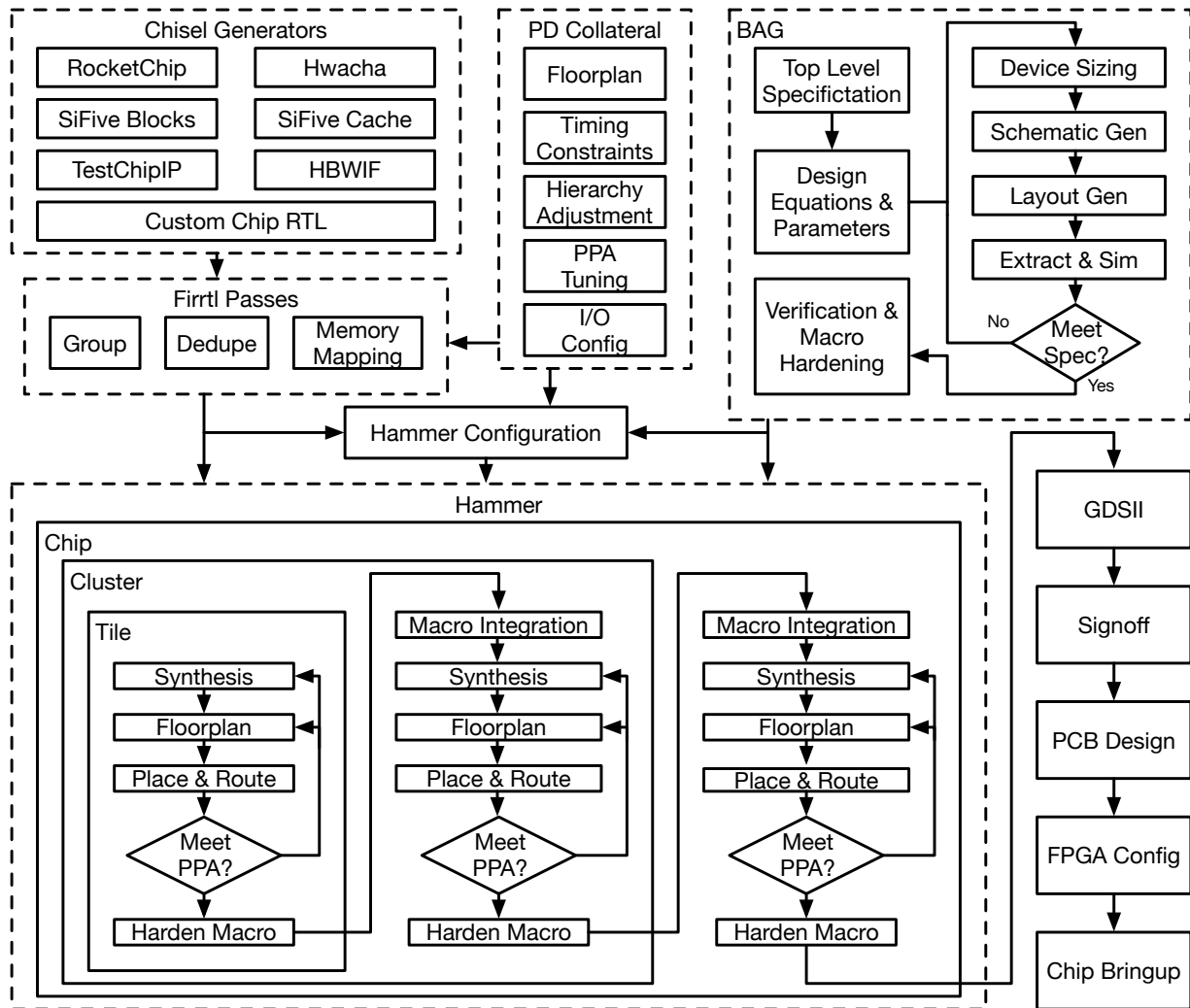
Figure 5.11: The composition of Eagle's Generators. Dotted line boundaries correspond to classes of generators.

pairs of cores connected to an L2 and an L3 that backs all L2s. The clocking structure and utilities and the slow digital backup interface are taken from the TestChipIP generator. The RTL components of the high-speed serial links come from the high-bandwidth interface (HBWIF) generator. The remaining off-chip peripherals, UART, I2C, SPI, GPIO, come from the SiFive blocks generator. The custom chip RTL generator includes connecting these component generators with Tilelink interconnect, exposing memory-mapped register to control the system, and connecting the top-level inputs and outputs to the individual blocks.

As in Hurricane-2, Eagle uses Chisel3 which does not directly produce Verilog but instead

generates FIRRTL. Eagle uses a few additional FIRRTL transformations when compared to Hurricane-2. The first set of transformations are used to manipulate the design from a logical hierarchy, as it was written, to the physical hierarchy, as it will be implemented. This transformation is actually a series of separate transformations, including grouping components into new modules, inlining module contents into the containing module, and deduplicating repeated instances to a single module definition. When performed in specific sequences these transformations can affect arbitrary re-orderings of the hierarchy to match a desired physical hierarchy. In addition, Eagle upgraded the memory-mapping infrastructure to directly support foundry memory compilers. This allows the user to dynamically generate any possible memories based on the optimal requested memory to available memory mapping. Both of these transformations make it easier for the RTL generators to interface with the physical design flow.

In addition to the digital generators and transformation, Eagle also includes analog generators built with the Berkeley Analog Generator (BAG) [25]. The analog generators construct an analog circuit of primitives in a schematic template. The designer then writes a programmatic layout generator which utilizes BAG APIs to provide technology portability. This parameterized design is then tuned to meet the specific design requirements and specifications. Additional changes may then be made to the layout generator to solve DRC or LVS issues, or update the architecture to improve device performance. The generator produces some of the collateral for integration, including GDS and LEF files. Currently BAG does not support library generation or behavioral model generation and for Eagle these were crafted by hand outside of the generator context.

Finally the physical design generator represents the newest, unproven, methodological change in Eagle's implementation flow. Eagle was built with a new physical design methodology and tool, Hammer [92]. Highly Agile Masks Made Effortlessly from RTL (Hammer), is a physical design generator designed to enable extensible and retargetable VLSI flows. In addition to this new methodology, Eagle also switched from Synopsys tools to Cadence tools while also moving technologies from ST28 to TSMC16, and building the largest SoC from Berkeley to date. These challenges coupled with intense time pressure caused rapid changes and development in Hammer throughout the tapeout process.

Hammer's core design principle is separating the three primary concerns of physical design from each other: project, tool, and technology. Hammer has APIs for each type of physical design tool and for technologies. Developers then implement these APIs in plug-ins for these tools or technologies. This layer of abstraction enables different tools or technologies to be used by the same project interchangeably. The project-level physical design concerns are isolated by supplying a set of parameters and values for design collateral in addition to programmatic use of the tool APIs. Hammer uses this collateral and the API calls to generate a set of scripts and other collateral to directly run the tools. In addition, to directly running the tools Hammer can generate build system collateral such as Makefiles to allow the user to interact with the tools without Hammer after the initial invocation. Customization of the physical design flow can happen via changes to the parameters for any changes already supported at the API level in Hammer, or by the insertion, modification, or deletion of

portions of the flow. Each step of the flow can analyze the inputs, parameters, and other collateral to make intelligent decisions about what to include or not include in the resultant tool invocations. These changes to the steps themselves can be programmatic allowing for complete control over the generation of the physical design flow. Eagle's physical design generator still required many project-specific flow changes due to the immature nature of Hammer.
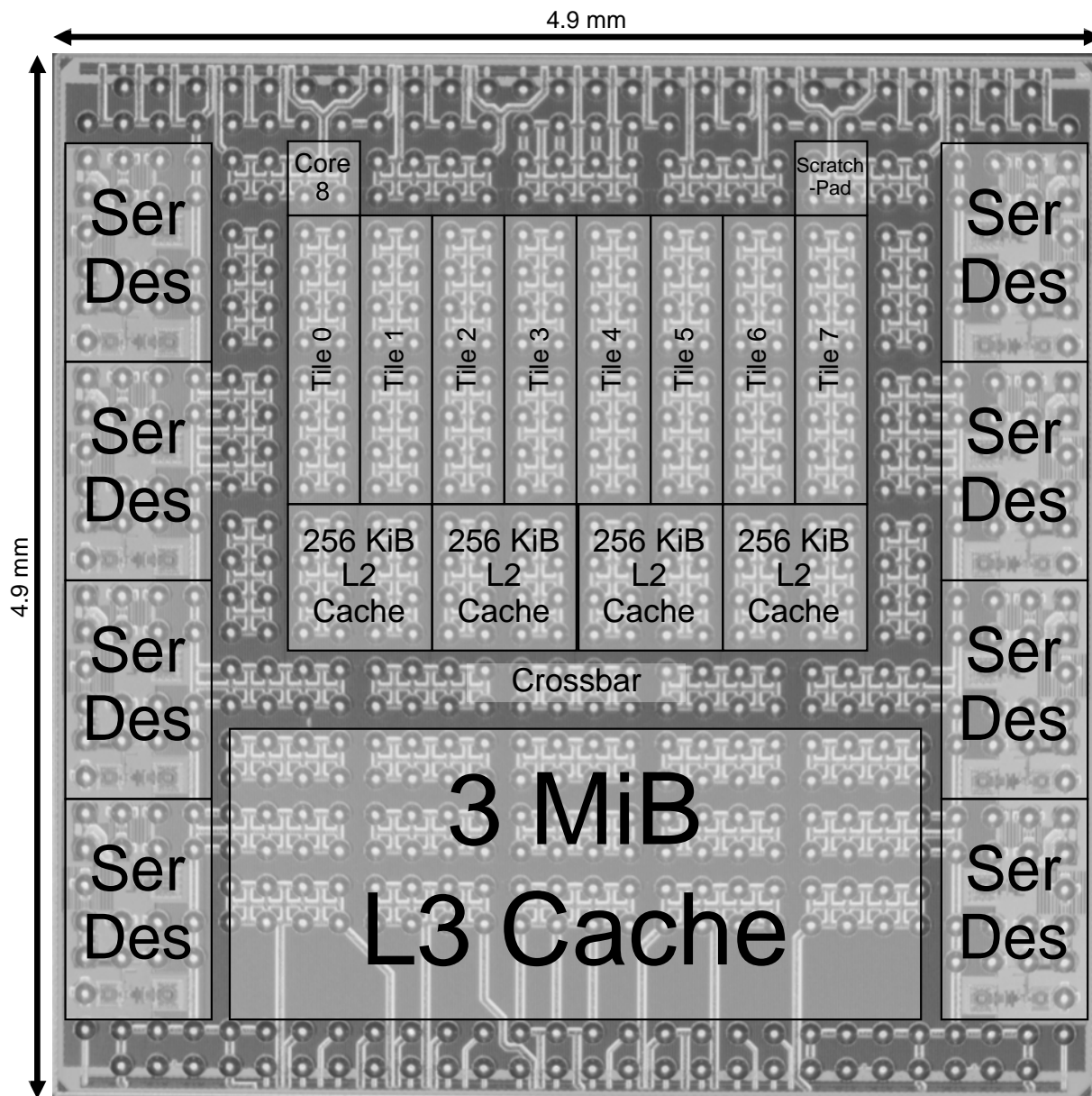

Figure 5.12: An annotated diephoto of the Eagle SoC.

In addition to using Hammer for the first time the size of Eagle meant that it needed to be designed hierarchically. Hierarchical physical design has several common variants, abutment, top-down, and bottom-up. These variants can be combined with different levels of the hierarchy using different strategies, for example abutment at the top-level with each subhierarchy using top-down. Top-down hierarchical design is the default hierarchical design flow implemented in Cadence Innovus, and starts with a planning synthesis run that determines the interfaces and approximates timing on the interfaces of the lower-level modules. After the planning synthesis run, the internal modules are placed and routed to create hardened blocks which then are used to place and route the top-level design. Bottom-up hierarchical design skips the planning synthesis run and simply places and routes the lower level modules using only the designer's foreknowledge to approximate interface timing. Both of these variants require a full top-level run to integrate the sub-modules which can be a runtime bottle neck in large chips, for example in Eagle this took about half of the total tool runtime. Hierarchical design by abutment combines blocks at the same level of hierarchy by simply placing them next to each other, eliminating the need for a final top-level integration run. Each module must agree on exactly where all of its IOs live and the timing for the cross module connections must be budgeted accurately on each side of the interface. Bottom-up methodology is easier to get up and running and is what was used for Eagle and what was eventually supported by Hammer, although there are plans to include the other variants of hierarchical design as options. The bottom-up methodology in Eagle did make it difficult to close timing precisely and be certain of the worst-case paths.

Given all of these changes there were surprisingly few physical design issues on Eagle. The large square chip size enables many different top-level floorplans, and figure 5.12 shows the chosen floorplan. In order to have the serial links off-chip connections be high performance and easily routed they need to connect to bumps at the edge of the chip, forcing the macros to be placed directly next to the sides of the chip. The desire to package them by directly flipping it onto the board with no intervening package or other structures required a relatively large gap between bumps on the chip. This combined with the fixed length of the chip edge restricted the number of serial link macros that could fit on a side to four. This in turn, meant that two sides would be required for the eight serial links used to match the bandwidth of a DDR memory interface. In order to save on analog designer effort, the serial links were put on opposite sides to enable a mirrored version to be functional, because a 90 degree rotation would not be possible in a FinFET technology. Ideally a generator like BAG would make rotations easier but that is not currently the case. This forces the application cores and memory system to reside in the middle of the chip between the pair of serial link banks.

Two primary issues arise with this top-level floorplan. The first issue with the top-level floorplan is the very narrow aspect ratio for each tile. Because each cluster has its own power domain a set of bumps will need to be dedicated to each cluster's power rail. Given the packaging strategy the best way to connect out from the chip to the board was to use sets of eight bumps in a four by two pattern. These groups could be either vertical or horizontal but this pattern constrained the clusters to be about 4 bumps wide. With two tiles per cluster, this limits the width of each tile to 2 bumps wide, and the height is then set to have
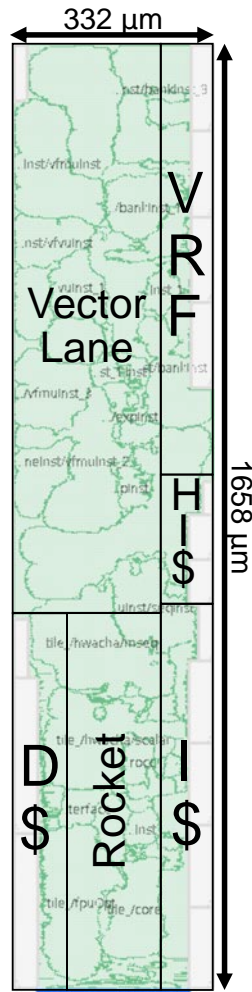
Figure 5.13: Eagle annotated tile floorplan.

a reasonable target utilization of around 70%. After all the constraints are taken care of, each tile is five times as tall as it is wide. To succeed with this aspect ratio the tile needed careful floorplanning, detailed in Figure 5.13. This included macro placement and soft guides for the Hwacha vector register file and execution unit. The standard clock tree synthesis algorithm also failed to achieve reasonable clock skew on this design with this aspect ratio. Instead a tool-assisted H-tree with custom trunk locations based on the aspect ratio was used which significantly reduced the skew and allowed the design to achieve a greater than 1 GHz sign-off frequency at the slow-slow corner.

The second issue with the top-level floorplan is that the crossbar between the L2 and L3 caches is once again confined to a small area. The area is bound on the bottom by the L3 cache which is adjacent to the bottom side of the chip, and on the top by the clusters which are adjacent to the top of the chip. There is some wasted space above the clusters due to
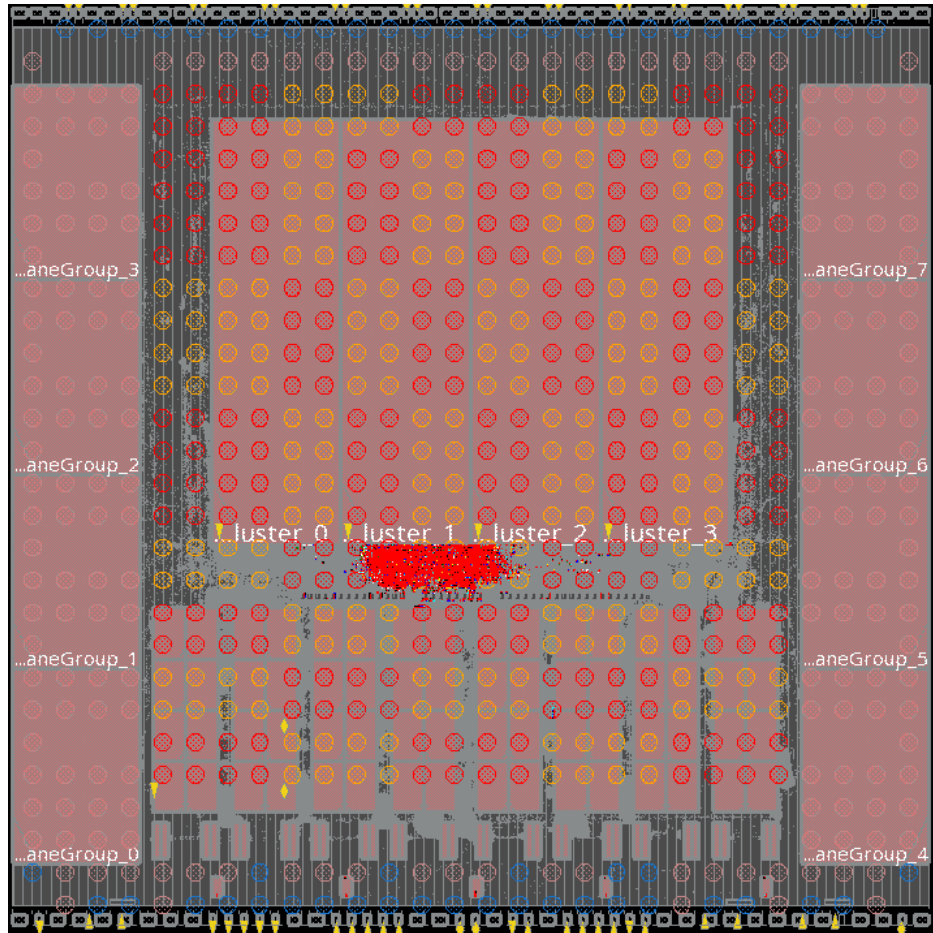
Figure 5.14: An example of the congestion experienced in the initial eight-bank L3 Eagle design. The red and white are regions of over-congestion that will result in shorts and other LVS failures.

the bump requirements mentioned above. Each set of four by two bumps on the interior of the chip are allocated to power or ground and so moving the clusters up to the top of the chip would either put some portions of them far away from power and ground or would need to reassign off-chip signaling bumps to power and ground. Even with these limitations the initial design included eight L3 cache banks with each one having a dedicated serial link. This doesn't provide full bandwidth for the incoming requests from each cluster but can support half of that required bandwidth or full bandwidth to two of the clusters. This initial design resulted in excessive congestion in the crossbar as seen in Figure 5.14. After a few attempts at a purely physical-design-based solution the number of L3 cache banks was reduced by half, enabling only a single cluster to operate at full bandwidth out of the L3 at a time. In addition, the portion of serial link bandwidth used becomes even lower falling from nearly a third to about an eighth. Unfortunately, this reduction in crossbar bandwidth

was not enough and the L2-to-L3 crossbar still proved physically unrealisable, and was thus clocked slower than anticipated, dropping from 500MHz to 250MHz. The ability to explore different top-level floorplans quickly would have been a useful technique to find alternative solutions to this congestion issue but currently Hammer floorplans are still manually written.

The Eagle SoC took a particularly long to get fully assembled and ready for testing due to inexperience and difficulty with the packaging strategy. In order to directly attach the die to a board both the board and the chip needed to be changed to meet the constraints. The board needed advanced processing techniques that brought the minimum pitch down to allow a reasonable number of bumps on each side of the chip to directly be routed out. The chip then met this constraint by reducing the density of its bumps. In addition to these pitch changes, the internal bumps needed to be ganged together to allow them to connect to the board-level vias. Initially, discussions with the board producer and the die-attachment company lead to an attachment process with solder-mask being applied around the die-attachment process. The set of boards produced in this fashion had a roughly 25% failure rate for each bump that was individually testable. Across a set of 78 bumps that needed to be connected this ensured that all boards produced this way had at least one failure. As a result a second run of boards was needed without the solder mask and with some modification to the die-attach process. These second set of boards almost all worked well enough to begin bring-up and testing.

Several components of the bring-up platform changed between Hurricane-2 and Eagle. In order for the high-speed serial links to operate at full bit-rate the FPGA needed to be upgraded. As a result of this FPGA upgrade, there was no longer a hardened CPU on the FPGA board that was able to manage the hosting for the tethered mode of Eagle. In order to replace this hardened CPU, a soft CPU on the FPGA was implemented. This soft CPU core also needs to interface with other FPGA board components and memory system. Because these components and other aspects of the FPGA hosting platform can change depending on the chip or functionality being tested, the decision was made to use the same generator-based design used for the SoC. A diagram of this platform is shown in Figure 5.15. Porting this new bring-up platform to the new FPGA took a fair amount of development for Eagle but would be shared between most future test chips.

With the bring-up platform determined and the chips packaged, Eagle was finally able to be tested. Some initial testing showed that the chip was mostly functional aside from what seemed like a straightforward bug in the memory system. This bug turned out to be more problematic as some address bits seem to be scrambled while requests move across the L2-to-L3 crossbar. A couple of address bits essentially become unusable perforating the address space. After this was discovered a significant amount of time was spent constructing linker scripts and binary rewriting tools to enable relatively simple C programs to be run bare-metal on the chip. With this setup, a small set of benchmarks, mostly focused on general matrix multiplication, was used as to collect experimental results from the chip.

Initially a sweep of valid operating points was done to find the lower and upper bound of voltage-frequency pairs. Figure 5.16 shows this sweep in the black line on the top set of axes. The resulting silicon achieved 862.5MHz at the setup corner voltage, 1.06GHz at
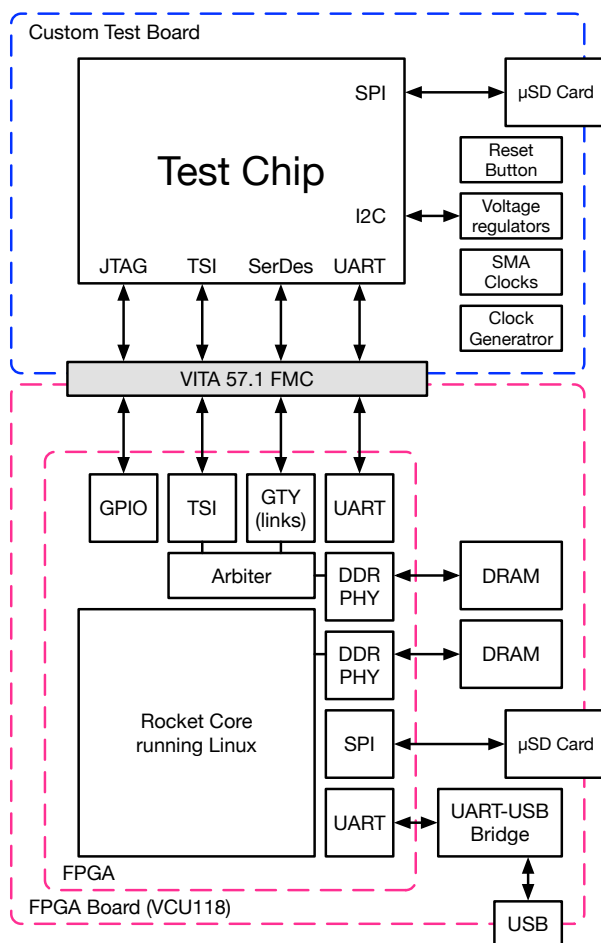
Figure 5.15: A block diagram of Eagle's bring-up platform. The FPGA is at the bottom surrounded by red and uses its fixed peripherals to aid in chip bring-up. A few small on-board components are next to the test chip outlined in blue.

nominal voltage, and 1.245GHz at the hold corner voltage, which was very close to the signoff frequency of slightly more than 1 GHz.

Next the custom-constructed GEMMs of the three supported Hwacha precisions were run on both cores of one cluster. This allows the collection of current, voltage, and frequency of the specific clock and voltage domain, which then can be used to calculate power. The program records how many floating-point operations it is doing per second on the UART and this data is collected along side the electrical data to obtain the plot in the bottom half of Figure 5.16 As mentioned above in this design, the Hwacha data bandwidth is limited to 50% of theoretical peak due to the L2 cache frequency division. A wider Hwacha data L2 port would correct this, approximately doubling the energy efficiency over these results for long vectors. At the most efficient operating point, 339 MHz at 0.55 V, the chip runs double-
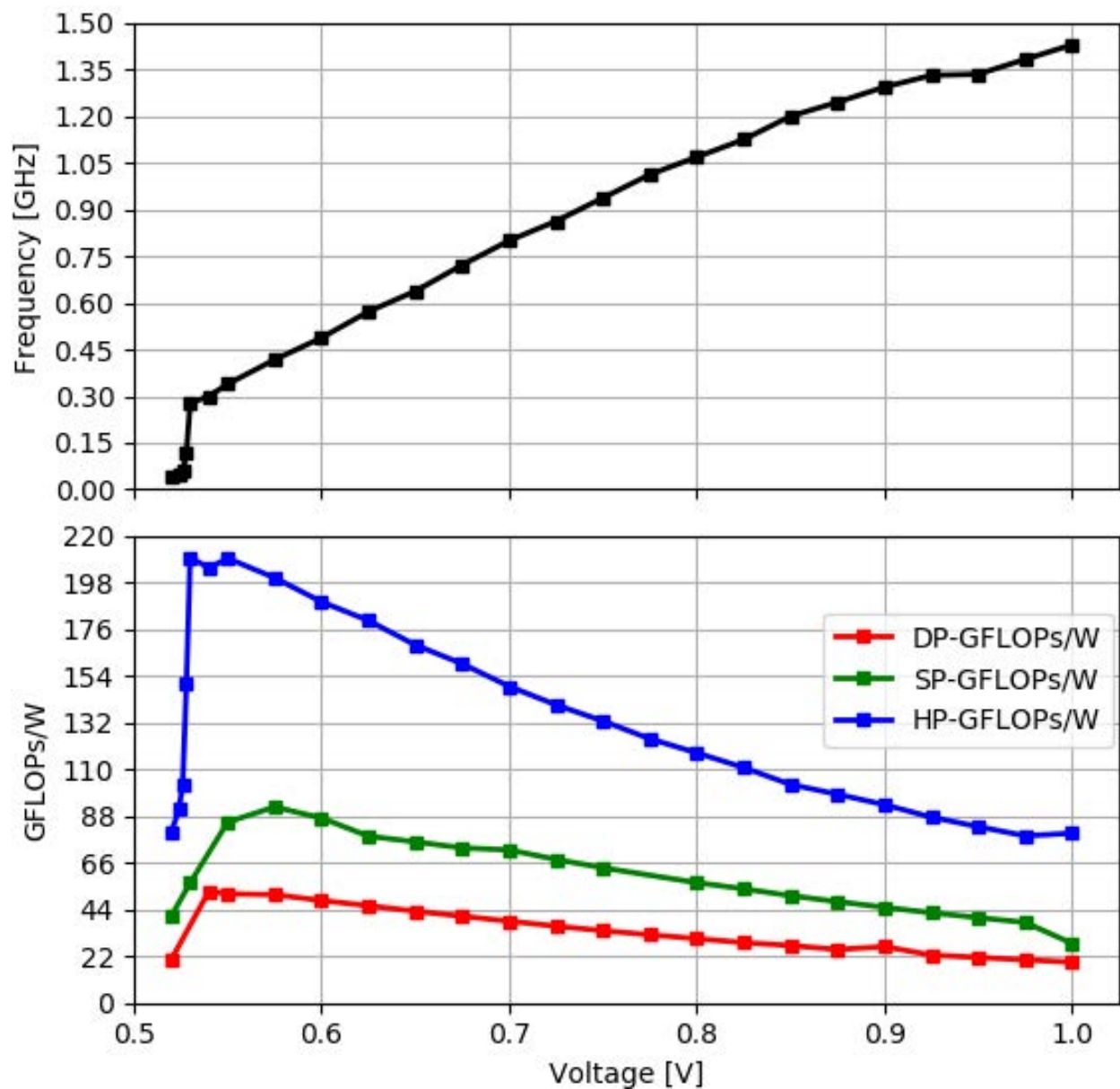
Figure 5.16: A single Eagle cluster's energy efficiency while running double-, single-, and half-precision general matrix multiplcation.

precision matrix multiplication (DGEMM) at 56.5 GFLOPS/W, single-precision (SGEMM) at 92.3 GFLOPS/W, and half-precision (HGEMM) at 209.5 GFLOPS/W, which correspond to 34.4%, 34.5%, and 33.0% of theoretical peak Hwacha performance, respectively. At the time it was published, Eagle was the highest reported energy-efficiency for a RISC-V chip with programmable precision for the precisions reported.

The Eagle test chip showed the benefits of having a generator based physical design

approach. It also showed that the Hwacha vector design is capable of achieving best-in-class performance, when given appropriate physical design effort. In addition, the updated RTL generators that provided more configurability to the uncluster and caches enabled the design to scale more easily through out the tape out process. The scaling of the design required understanding and practice with hierarchical design techniques, which were then captured in the Hammer implementation. Finally, the use of a new set of tools and a new technology, also captured in Hammer, expanded the size complexity of designs that could be implemented with a small team.

5.4   EagleX

EagleX is a $56.25mm^2$ 22-core SoC with 20 uniform application cores each with a single-lane Hwacha instance implemented in a TSMC 16nm FinFET process. It was taped out in June of 2019, and received back fully packaged and assembled by July 2020. EagleX more than doubles the size of the Eagle design and focuses on improving physical design generator support, including additional accelerators, and improving overall system performance. It includes nearly all of the features of Eagle, large on-chip caches, many cores, high-speed off-chip interfaces, and connections to off-chip peripherals. Thus it maintains its realism while pushing the limits of the size and complexity that Hammer is capable of handling. In addition, several other chips were also being taped-out at the same time with just a few additional graduate students working on those other designs. In order to succeed, Hammer would have to be upgraded to enable the reuse that its design purports to allow. The application cores were maintained at the same specifications as Eagle, while the L3 was increased in size to utilize the additional die area and provide more bandwidth and capacity for each cluster. EagleX still includes a set of eight 28 Gbps serial links and the standard off-chip interfaces from Eagle: JTAG, UART, SPI, I2C, and GPIOs. However, rather than having only a single accelerator EagleX also integrates a systolic array matrix-multiplication accelerator, Gemmini [35].

Figure 5.17 contains a detailed block diagram of all the components on EagleX. The upper left of the figure shows the details of a tile and cluster which are identical in parameters to those of Eagle. The clusters are no longer on their own voltage domain, there is instead a single voltage domain for all of the clusters. They still retain their individual clock domains with the L2 running at half the frequency of the tiles and communicating with a rational synchronous crossing to the tiles and an asynchronous crossing to the uncluster. Each cluster clock multiplexor is capable of selecting from any of the available clock sources on chip.

To the right of the cluster detail is an expanded view of the systolic array accelerator, Gemmini, and its associated core. Like Hwacha it uses the RoCC interface to communicate with Rocket as a scalar control processor. It also directly communicates with the outer level cache, which in the case of the systolic array core is the L3 cache. Gemmini is also built as a generator, so has many parameters and possible instances. This Gemmini instance has a 256KiB scratchpad for double buffering inputs and outputs, an accumulator for element-
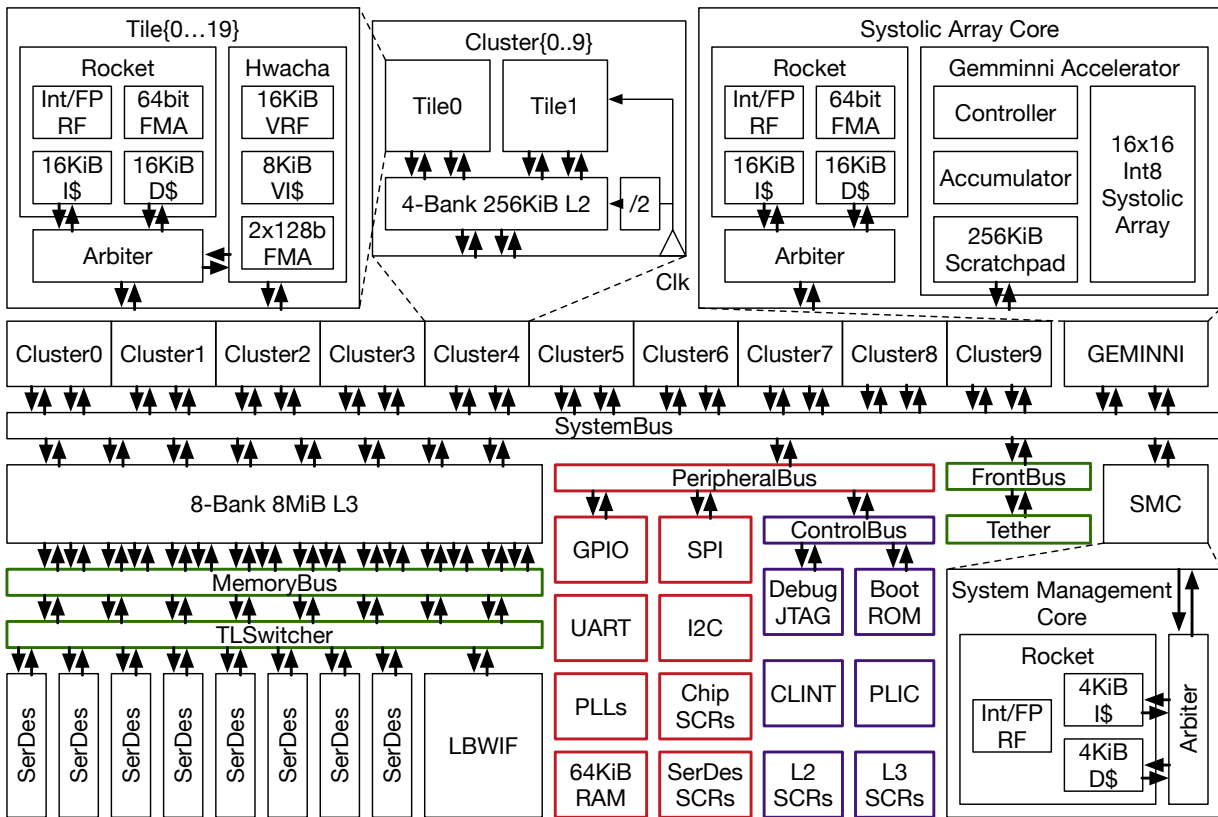
Figure 5.17: EagleX blockdiagram.

wise computation, and a 16-by-16 grid of 8-bit integer multiply-adds that are connected in a systolic manner for the core computation. This is a very early version of Gemmini that is only optimized for GEMM computations but newer versions add many features and optimizations to make it suitable for general deep neural networks. Gemmini is included to allow for an energy-efficiency comparison between the Hwacha general-purpose data-parallel accelerator and an optimized fixed-function accelerator.

Below all of the clusters and cores is the Tilelink-2 system bus, which connects the application cores with the rest of the system. Everything below and including the system bus is consider part of the uncluster. On the left-hand side of the figure is the memory subsystem. The system bus connects requests to off-chip memory first to the eight bank 8MiB L3 cache, and then through the memory bus and to the Tilelink-2 switcher. This switcher operates as it did in Eagle by allowing a memory mapped register to control whether the backside transaction of the L3 are handled by the high-speed serial links or by the low-speed low-bandwidth backup digital interface. The slow link has a divider to allow it to run arbitrarily slower than the rest of the uncluster but in practice can usually run at around 100MHz or 400Mbps, with a lower actual data-rate due to the simple serialization scheme.

Immediately to the right of the memory system are all of the peripherals, both on- and off-chip. The red shaded portions under the peripheral bus contain the off-chip peripherals and some status control registers (SCRs). These peripheral bus clients have slightly higher bandwidth and lower latency than the device in purple that are attached to the control bus. The control bus has devices that are less frequently accessed during normal operation of the chip. Finally, the last bus in the uncluster is the front bus which has the opposite relation with the system bus than the other buses, in that the front bus drives the system bus. The only device attached to the front bus is the test chip tether used as a backup bring-up interface. Additionally the uncluster contains one more core which acts as a system management core (SMC). This core has no floating-point support and minimum-sized caches, with only a single Tilelink port to the system bus.
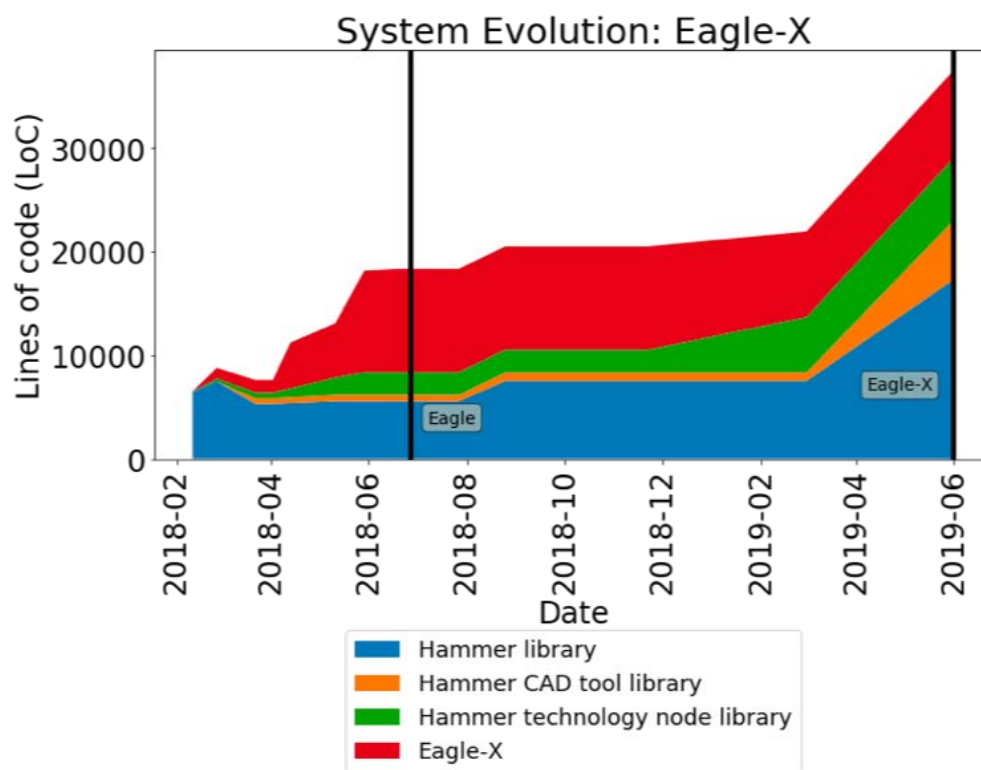
Figure 5.18: The change over time of various portions of the Eagle/EagleX Hammer code base.

The physical design of EagleX also used the Hammer physical design generator and Cadence tools. However, EagleX was taped out alongside several other test chips necessitating code sharing between the projects to make them feasible with the given resources. Each additional test chip added only two or three graduate students to work on their individual chips. Hammer was always designed to enable this type of reuse but it was unnecessary during the Eagle tapeout as there was only one project using Hammer.

Figure 5.18 shows the evolution of the physical design code used in Eagle and EagleX. The three primary concerns of physical design that Hammer separates from each other are the tool, technology and design concerns. In this graph the lines of code associated with each of these concerns as well as general library code are separated out and counted. The blue area at the bottom is the general library code which includes API definitions and other code to connect different parts of the system. The orange area represents the code in the tool plugins used in Eagle and EagleX, namely the Cadence and Mentor tool plugins. The green area represents the code in the TSMC16 FinFet technology plugin. And finally the red area represents the project-specific code written in the Eagle and EagleX physical design generator itself. This code is the only code that can not be reused by other designs. Not all of the code in the other areas will be reused by other projects, but projects that share tools or technologies will share those sections.

The graph shows the evolution of these line counts from the inception of the Eagle project in early 2018 through the first tape out of Eagle, the left vertical black bar, and all the way until the tape out of EagleX, the far right vertical black bar. As mentioned above, the initial development of Eagle was heavily focused on getting a finished design taped out and so used the project-specific modifications heavily. This usage model was intentionally built into the Hammer framework to allow for new users to easily adapt their current methodology to a Hammer-based generator without a lot of work adding features they need in a reusable manner. However, after the Eagle tape out the desire to have new projects leverage its physical design generator pushed much of the project-specific code in the reusable portions of Hammer. You can see this change over the year between Eagle and EagleX where the portion of code in the generator that is project-specific shrinks from more than half to less than a quarter. This shift to a general-purpose implementation does result in more overall code as the new implementation needs to account for varying use cases. The migration from project-specific implementation to reusable library code is one of the primary benefits of Hammer's physical design generator methodology.

The increase in size of EagleX and the desire to avoid the packaging issues of EagleX causes the top-level floorplan to deviate significantly creating many physical design changes. Figure 5.19 shows a micrograph of the EagleX test chip with an overlaid floorplan. The bottom half of the figure is similar to Eagle, with the serial links, L3 cache and peripherals plus off-chip IO placed tightly against the edges of the die. The top half deviates considerably however. Rather than have each cluster on its own independent voltage domain, all clusters share a single voltage. This relaxes the constraints on the power and ground bumps under the cluster. In addition, because of the increased perimeter of the chip less of it is used for digital I/Os and so the bumps on the north side of the chip are simply power and ground for the cluster, removing the need for the gap at the top that was included in Eagle. These changes made the physical design of EagleX somewhat easier by reducing some of the difficult to implement constraints from Eagle, but it was not without its own problems.

The first physical design issue, a seemingly perennial one with Hwacha test chips, was the L2 to L3 crossbar. On the Eagle test chip the issues with this crossbar reduced the uncluster frequency and L3 bandwidth and with more than twice as many cores in EagleX the new
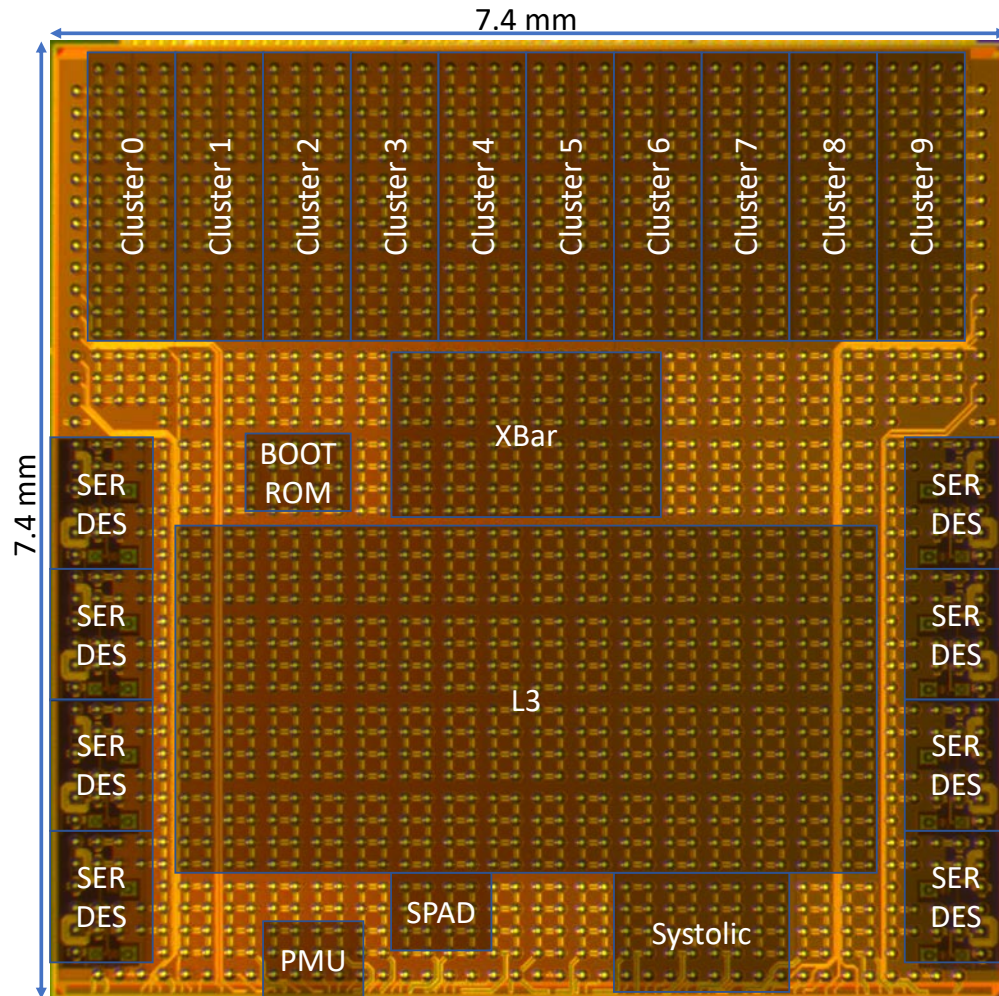
Figure 5.19: An annotated diephoto of EagleX SoC.

crossbar had an even higher required bandwidth. Several experiments were undertaken to attempt to reduce the impact of this crossbar on overall system performance. Early in the design phase, a ring interconnect was discussed and eventually implemented. Unfortunately, the physical design burden implementation of the ring interconnect was not completed due to time constraints and instead effort was focused on implementing the existing crossbar more effectively.

Several physical design solutions were explored to improve the clock rate and bandwidth of the crossbar. The first solution was to have this large module be a separate place and routed block which became more practical once Hammer fully supported arbitrary hierarchy module composition. Unfortunately, the additional pin location and timing constraints made this approach very difficult. If Hammer were to support a better pin-timing interface this process could have been made easier and potentially tractable but at the time of imple-

mentation it would have required a lot of manual effort for an experiment that might not succeed. The next technique was to update the floorplan with low utilization regions in and around the crossbar area. This would force the tool to keep most logic out of that region saving the area and routing resources for the buffers that would be needed to implement the crossbar. This technique did help and was used in the final implementation of EagleX. A more exotic idea that was discussed was to give the tool a lower-level placement of portions of the crossbar. This has helped with other regular structures like register files in the past but with a generated crossbar would require a lot more effort and wasn't possible in the given time. The last and simplest solution of increasing the area allocated for the crossbar was also used and combined with the other techniques allowed for the crossbar to hit its target bandwidth.

The integration of a new accelerator, Gemmini, had a few physical design issues that are common with large data-parallel accelerators. The first issue was developing a good floorplan for Gemmini. This is a common problem, but Gemmini, like other data-parallel accelerators, includes lots of memories for storing temporary results, making the problem more difficult. These additional memories make the floorplanning task more complex and time consuming, and a good solution required several iterations. Floorplanning in Hammer is currently handled manually despite the clear need for a better interface and automation, hopefully this can be addressed in future development. In addition to floorplanning, data-parallel accelerators like Gemmini often include a higher density of logic that can cause significant power draw. In EagleX this required changes to the power-strap layout for the Gemmini tile. Hammer performed static IR drop analysis and determined that there would have been significant droop with the power-strap settings used throughout the rest of the chip. Therefore, the percentage of routing resources devoted to power straps in Gemmini was increased. This change is less harmful to signal routing due to the systolic-array nature of Gemmini that focuses on local interconnect rather than global interconnect, which would be more negatively affected by the decrease in routing resources. In Hammer the change in power straps is affected using an API, and so only a single line of the configuration code for the physical design generator needed to be changed.

To avoid the packaging issues encountered with Eagle, EagleX was packaged in a dedicated ceramic package. The package, along with the reduced number of voltage domains, enabled the daughter board for EagleX to be much cheaper and partially offset the cost of the package. The board for EagleX was similar to Eagle, but removed the DC-DC voltage converters, and added SMAs to directly connect to two of the clock inputs, rather than having all three connected over the FMC connector. The FMC interface for EagleX was designed to be backwards-compatible with the Eagle FPGA motherboard setup and so either chip can be connected to the FPGA without reconfiguring the FPGA. The older Eagle FPGA image will not enable all features such as the ability to drive clocks zero and one, but otherwise will be functional. Having this FMC compatibility isn't too onerous and so has been adopted on most test chips developed after Eagle.

Fully populated boards were received in August of 2020. The boards were checked for any shorts immediately, but bring-up did not begin until September. Using the same Eagle

FPGA motherboard image, EagleX's JTAG connection was used to verify the accessibility of all cores on the first attempt. A week of work was necessary to update the FPGA side Tilelink deserialization, which then enabled simple bare-metal programs to be run, including Hwacha programs. Finally, after a few days of debugging Linux boot and adjusting the device tree (DTS) processing in the Berkeley boot loader (BBL), SMP linux was booted on all 20 uniform application cores on EagleX.

EagleX has not undergone extensive testing yet and so there is no detailed plot of its voltage and frequency operating points but preliminary testing indicates that it is capable of a similar operating range to Eagle.

EagleX showed the value of physical design generators being able to reuse code across projects. This allows physical design solutions to be shared across repeated uses of the same design generator continually improving quality of results. By incorporating two distinct data-parallel accelerators into one system EagleX showed the similarity between these accelerators in their physical design needs. There are still improvements to be made from the physical design side specifically for data parallel accelerators like Hwacha which require careful floorplanning. A more automated floorplanning process or one that is easier to iterate with would make the design of efficient versions of these accelerators easier. Having discussed the current design and implementation of the Hwacha vector fetch architecture, the remaining chapters will explore the design, implementation and results from a set of extensions to this architecture.

Chapter 6

Design of a Two-dimensional Extension to Hwacha

Given the extensive use of machine learning, and specifically deep learning, focusing on optimizing this use case can provide consistent benefits across many applications. As discussed in Chapter 3, deep learning makes extensive use of dense multi-dimensional computation. Higher-dimensional operations can be implemented with lower-dimensional ones and so a focus on two-dimensional extensions can still provide significant improvements on deep-learning workloads. This chapter describes a set of features that can be added to the baseline Hwacha architecture to improve energy-efficiency of two-dimensional dense linear algebra workloads.

The specific extension discussed in the rest of this chapter is focused on using multiple vector registers as sub-matrices. This approach towards the two-dimensional compute problem leverages as much of the existing microarchitecture as possible. The details of the microarchitectural changes will be expanded on in the next chapter. There are several individual features that are incrementally added to create the entire extension, with each having a devoted section below.

Section 6.1 describes the first extension, a computational extension for matrix-vector multiplication. Next, section 6.2 explains another extension for matrix memory operations. Finally section 6.3, describes some of the additional operations and extensions that could be added to further improve the energy efficiency on deep learning workloads.

6.1 Vector-Transpose Matrix Multiply

The first group of instructions added and the strongest catalyst to start this extension was a linear algebra operation that works as a step of a GEMM's inner loop, called vector-transpose matrix multiply (VTMM). This instruction takes three inputs and produces one output and implements the linear algebra operation, $Z = A^T * B + C$. A is a single vector register with length encoded in a parameter called *depth* in the VTMM instruction. B and C are each a sequence of *depth* vector registers each with length vl, the standard machine-wide vector

length.

Figure 6.1 shows the encoding for this instruction. Starting from the least-significant bits, the lower twelve bits are an opcode denoting this as the VTMM instruction, here labeled VFVMMADD for vector floating-point vector-matrix multiply-add. The next fields in the instruction are the register identifiers, first the 4-bit predicate register followed by the four 8-bit vector register identifiers, with the 32nd bit amongst the registers specifying whether or not to negate the predicate. After this there are two fields present in every floating-point instruction, the rounding mode, rm, and input and output precisions, fmt. The next field is the new depth field representing the inner dimension of the vector-matrix multiplication. Finally, the upper four bits, in the Hwacha worker-thread encodings, are dedicated as determining which of the register operands are scalars and which are vectors. In this case, all four of these bits are variable and can be set to either scalar or vector.

63	62	61	60	59 57	56 53	52 50	49	48 41	40 33	32	31 24	23 16	15 12	11 0
s/v	s/v	s/v	s/v	depth	fmt	rm	—	rs3	rs2	n	rs1	rd	p	opcode
1	1	1	1	3	4	3	1	8	8	1	8	8	4	12
d	s1	s2	s3	depth	fmt	rm	1	src3	src2	n	src1	dest	pred	VFVMMADD

Figure 6.1: The encoding of the VTMM worker-thread instruction.

The VTMM instruction essentially operates as a replacement for a group of FMAs commonly found in the inner loop of a GEMM kernel. The traditional mapping of this kernel to Hwacha uses multiple scalar values of a and rows of B and C stored in separate vector registers, as seen in Figure 6.2. The new mapping is able to replace large groups of these repetitive ordered FMAs, up to eight at the maximum $depth$ of eight, with a single VTMM instruction. In this example $depth$ is only four so each of the replacement instruction replaces the four of the FMA instructions. The first register, the destination or $vv0$, as well as the third and fourth operands are treated as the matrix operands and are a sequence of $depth$, four in this case, vl-long vectors, while the second operator is the vector that will be transposed before the multiplication. Because of this transposition, only the first $depth$ elements of $vv8$ through $vv11$ will be accessed during execution.

This instruction does not reorganize the vector register file and simply treats sets of registers as a larger unit. This means that normal memory operations are able to move the data into vector register $vv0$ through $vv3$ to represent the sub-matrix destination in the example above. This also keeps the programming model relatively simple for the user who only needs to track the $depth$ of these instructions and ensure all of the registers that will be touched have been loaded before issuing the VTMM instruction.

As these VTMM instructions now read from many more vector registers the new inner loop of the GEMM kernel will now be dominated by the memory operations required to

```
vfmadd vv0, vs8, vv4, vv0
vfmadd vv1, vs8, vv5, vv1
vfmadd vv2, vs8, vv6, vv2
vfmadd vv3, vs8, vv7, vv3
vfmadd vv0, vs9, vv4, vv0
vfmadd vv1, vs9, vv5, vv1
vfmadd vv2, vs9, vv6, vv2                    vfvmmadd vv0, vv8, vv4, vv0, 4
vfmadd vv3, vs9, vv7, vv3                    vfvmmadd vv0, vv9, vv4, vv0, 4
vfmadd vv0, vs10, vv4, vv0                   vfvmmadd vv0, vv10, vv4, vv0, 4
vfmadd vv1, vs10, vv5, vv1                   vfvmmadd vv0, vv11, vv4, vv0, 4
vfmadd vv2, vs10, vv6, vv2
vfmadd vv3, vs10, vv7, vv3
vfmadd vv0, vs11, vv4, vv0
vfmadd vv1, vs11, vv5, vv1
vfmadd vv2, vs11, vv6, vv2
vfmadd vv3, vs11, vv7, vv3
```

Figure 6.2: An example of the denser instruction stream made possible by the VTMM instruction.

populate these registers. This leads to the next set of instructions added for moving these sub-matrices into sequences of vector registers.

6.2   Sub-matrix Memory Operations

The next set of instructions improves the ability of the architecture to express loads and stores with regular address patterns that often occur when working with matrices. The first group of instructions are loads and stores that move a sequence of unit-stride vectors from memory into consecutive vector registers. These instructions also provide a stride, in bytes, between each vector register that enables the instruction to encode a sub-matrix for row-major matrices. There are no alignment constraints on the architectural vector register numbers other than being consecutive.

63	62	61	60	59 57	56	55 48	47 45	44	43 42	41	40 33	32	31 24	23 16	15 12	11 0
s/v	s/v	s/v	d?	—	sg?	—	seglen	s/u	size	st?	as2	n	as1	vd	p	opcode
1	1	1	1	3	1	8	3	1	2	1	8	1	8	8	4	12
1	0	0	0	0	1	0	seglen	s/u	size	1	stride	n	base	dest	pred	VLSMx

Figure 6.3: The encoding of the sub-matrix load worker-thread instruction.

```
vld vv0, va0
vld vv1, va1
vld vv2, va2
vld vv3, va3
vld vv4, va4                    vlsmd vv0, va0, va1, 8
vld vv5, va5
vld vv6, va6
vld vv7, va7
```

Figure 6.4: An example of the denser instruction stream made possible by the VLSM instruction.

Figure 6.3 shows the encoding for this instruction. Again, starting from the least significant bits, the lower twelve bits are an opcode marking this as a vector load sub-matrix, VLSM, operation. The register specifiers are similar to the VTMM instruction but with only two inputs. However, there are some subtle register differences for loads and stores. The destination or source for the data is still a normal vector register, but the first and second inputs, the base and stride in this case, are in address registers. This enables the VMU to begin generating addresses without waiting for the VXU or scalar unit to provide these inputs. Next there is a dedicated bit, the 41st, for all memory operations to determine whether the second input is used as a stride. The next two fields are also shared across memory operations, and represent the *size* of the memory access, and whether the data will be sign-extended into the register or not. The number of registers in the sequence follows this in the *seglen* field, up to a maximum of eight. The *sg?* determines whether the memory operation uses the segment field and in this case is hard-coded to true. The *d?* determines whether the memory operation acts on vectors of length *depth* or the standard *vl*, in this case of VLSM, this is hard-coded to false. Finally, the upper 3 bits dictate the scalar or vector nature of the three operands and are hard-coded for a vector destination, and two scalar inputs.

Figure 6.4 shows the reduction in instruction count enabled by the sub-matrix memory operations. In this example a single *vlsmd* instruction with eight segments replaces a series of eight unit-strided loads. The *d* at the end of each of the assembly mnemonics encodes the *size* field for how wide the memory access will be in this case a double-word access of 64 bits. At the end of either sequence the values in *vv*0 through *vv*7 will be identical assuming the difference between the address registers on the left side is uniform and is written to *va*1 on the right side. This compression not only saves instruction cache footprint and instruction issue bandwidth but can reduce the amount of microarchitectural resources needed to track this long series of instructions. The details of the microarchitectural effects will be discussed more in the next chapter.

These instructions help construct the sub-matrices needed for B and C in the VTMM instructions. However, the shorter A vector still has to load an entire *vl*-length vector when if it is only used for the VTMM instructions a shorter vector would be enough. To reduce

this waste of memory bandwidth, another set of memory operations is added that limits the vector length of the loads and stores. These instructions have the same semantics as the corresponding loads and stores with the exception that they only operate with an effective vector length of *depth*.

63	62	61	60	59 57	56	55 48	47 45	44	43 42	41	40 33	32	31 24	23 16	15 12	11 0
s/v	s/v	s/v	d?	depth	sg?	—	—	s/u	size	st?	—	n	as1	vd	p	opcode
1	1	1	1	3	1	8	3	1	2	1	8	1	8	8	4	12
1	0	0	1	depth	0	0	0	s/u	size	0	0	n	base	dest	pred	VLDx

Figure 6.5: The encoding of the depth-only load worker-thread instruction.

Figure 6.5 shows the encoding for these depth-only loads. The bottom 32 bits of the depth-only loads match that of the sub-matrix load, describing the destination or source, the predicate and the base address register. However, the next two fields describing the stride are omitted and set to zero. The *size* and sign-extension field are still present, but the segment fields are filled with zeros. The *depth* field is used to determine the temporary vector length of these short loads and stores, which means the d? field above it is hard-coded to true. Finally, the upper three bits again indicate that this instruction has one vector input and one scalar input.

Unlike with the previous new instructions, there is no direct comparison with previous instructions because a temporary vector length change is not possible in the worker-thread at all. However, in a typical kernel, vector lengths in Hwacha are often 32 or more elements and so enabling these short vector loads can reduce bandwidth requirements significantly. Not only does this allow the execution to proceed to useful work faster, but it also saves energy and reduces cache pollution.

Finally, another group of instructions combines both of the previous features into depth-only sub-matrix loads and stores. This allows the user to control both dimensions of the sub-matrix independently of the current vector length within the limits of the encoding.

63	62	61	60	59 57	56	55 48	47 45	44	43 42	41	40 33	32	31 24	23 16	15 12	11 0
s/v	s/v	s/v	d?	depth	sg?	—	seglen	s/u	size	st?	as2	n	as1	vd	p	opcode
1	1	1	1	3	1	8	3	1	2	1	8	1	8	8	4	12
1	0	0	1	depth	1	0	seglen	s/u	size	1	stride	n	base	dest	pred	VLDSMx

Figure 6.6: The encoding of the depth-only sub-matrix load worker-thread instruction.

Figure 6.6 shows the encoding for the depth-only sub-matrix memory operations. The encoding is very similar to the other operations but uses both the segment and depth fields so it also has those indicator fields hard-coded to true. These operations allow the loading of multiple vectors to be used as the transposed vector in the VTMM instructions, once again saving memory and instruction bandwidth.

This entire set of load and store operations combine to enable the easy manipulation of sub-matrices without moving extra data from the memory system or requiring many memory instructions to reach peak bandwidth. The use of the depth field as a separate vector length restricts the minimum vector length of the architecture to be at least as long as the depth parameter, which is 8. In many implementations this will not be a problem but does potentially incur a large area requirement on smaller implementations. In the case where this minimum would be violated, for example when all 256 vector registers are enabled, the machine can either label this configuration illegal or have depth instructions that have a longer depth than the current vector length be illegal. A more complete solution that behaves similar to setting the vector length with *vsetvl* would be difficult to implement, but would allow for the configuration to gracefully adapt to different configurations. If this solution were implemented however it could also potentially enable the configuration of some registers as *depth* registers and reuse their physical storage to lengthen the general vector length.

6.3   Future Extensions for Deep Learning

The above instructions represent a useful proof of concept for extending a general-purpose vector architecture with domain-specific instructions in the dense linear algebra domain. However, there are many other reasonable extensions that could further improve the performance of the sub-domain of deep learning. A few of those ideas are presented below building on top of the previous two-dimensional extensions.

Convolution

Once the architecture can represent shorter vectors it is natural to look at other uses of short vectors in combination with long vectors. One frequent case of this is a one-dimensional convolution where the kernel size of the convolution is often a small odd number. This limit would allow the same depth parameter that has a maximum of eight to still be used for these convolutions. A one-dimensional convolution operation could easily fit in to the standard arithmetic operation encoding which does not use the location assigned for the depth parameters above. The operation then performs the natural convolution operator with the long vector being convolved with the shorter vector which has a kernel size equal to depth.

Another concern with these operations is whether and how to pad the convolution. Producing a shorter vector of, most likely, a third vector length seems impractical while always

producing a padded result allows the user to post-process the output if they want to remove the padding. Given the large amount of encoding space available in the Hwacha ISA a simple solution would be to add a third input that is the value to be padded. This allows both floating-point and integer convolutions to use sensible padding values without needing special cases, and in addition allows the freedom of having the padding value be dynamically calculated. The alternative that saves encoding space would be to have a few options available at the opcode level. For example, there could be convolution operations with zero padding or one padding in both integer and floating-point flavors.

In addition to using the depth-based arithmetic operations, convolutions can also benefit from the sub-matrix memory operations. The sub-matrix loads can be used to load several kernels at once, or even a sub-matrix to be convolved with. Because the loads leave the group of vector registers as if they had been loaded with normal instructions, the user can easily reduce the number of memory instructions in a loop with multiple convolutions of different kernels. Using the sub-matrix memory operations also leads towards potentially implementing two-dimensional convolution operations. These operations would be difficult to encode with the third input operand for the padding if they supported rectangular kernels. If the design only supports square kernels as is common in deep learning then the natural encoding would fit with a single depth parameter. The implementation of a multi-dimensional convolution would be more significantly more complicated than the one-dimensional case, but that will be discussed in the next chapter.

Pooling

Another common operation in deep learning workloads is the pooling or sub-sampling between layers. Pooling or sub-sampling reduces the size of the tensor input and combines results from multiple inputs. This operations is similar to convolution with fixed parameters for the kernel size and weights.

The fixed parameters reduce encoding cost and allow the design to encode many different versions of the operation. The depth parameter would be used to determine the size of the field to search in the input. With a single input, and a single output, much of the encoding space can be used to represent the different functions, such as max-pool, averaging, or median.

Both pooling and convolutions involve reductions which are currently lacking in the Hwacha ISA. However, like the VTMM instructions this reduction has a fixed local pattern rather than the global reductions traditionally described by that word. This should make the cost to implement these instructions smaller than a true global reduction, but these details will be discussed in the next chapter.

Precision

Finally, extending the architecture to support smaller bit-width arithmetic types would be a large benefit for deep-learning applications which often run inference or retraining and quan-

tization and small precisions. The options for reduced-precision data types was previously a very small and straightforward design space consisting of half-precision IEEE floating point and small integers. However, the recent proliferation of deep-learning accelerators has lead to many newly proposed data types. Both NVIDIA and Google have produced chips that use new floating-point formats(TF32 [55] and BFloat16 [94] respectively) designed to improve performance on deep-learning tasks. In addition, many designs in academia and industry continue to explore sub-byte integer precision operations, and even binary neural networks.

Including these new formats as additional available data types has several issues in the Hwacha architecture many of which would extend to any user-programmable vector architecture. First, and most obviously, having more data types available will, in a non-polymorphic instruction set, require additional encoding space scaling with the number of operations and the number of data types. The architecture will also have to consider how these specialized types will be transferred into and out of the vector unit especially if they are not an integer number of bytes or if the scalar processor cannot natively handle them. And there is also a implementation complexity and area cost to support multiple data types, although in more specialized use cases the number of larger data types supported can be reduced to focus on particular applications.

Having described the architectural implications of these extensions focused on dense linear algebra the next chapter will discuss the implementation complexities. In addition, the implementation of the deep-learning extensions will be discussed at a high-level to understand the challenges associated with their complete implementation.

Chapter 7

RTL and Physical Implementation of Two-Dimensional Extension to Hwacha

There are many ways to implement a given architectural specification for a group of instructions or extensions. This chapter will cover one implementation style used to add the dense linear algebra extensions from the previous chapter to the most recent Hwacha microarchitecture. Since these extensions do not rearrange the register file for two-dimensional data, the general implementation strategy is to treat the operations as repeated one-dimensional operations.

Section 7.1 describes the general control structures added to the microarchitecture to handle these repeated sub-operations. Next, section 7.2 explains some additional details that are used to implement the VTMM instructions. Then, section 7.3 describes the straightforward application of sub-operations to the vector memory unit to implement the sub-matrix and depth-only memory operations. Section 7.4 then describes how some of these new microarchitectural features could be used to implement the future deep learning extensions described in the previous chapter. Finally, section 7.5 discusses the results of taking this implementation through the physical design process and obtaining area, energy, and performance data.

7.1 Two-Dimensional Control Structures

There are two unifying aspects of these new instructions. First, most of the new instructions share the use of multiple registers as a single input or output from an instruction. And second, they often include a secondary vector length, *depth*, encoded directly in the instruction.

To keep track of these additional operand registers and the second dimension, instructions that have this property will be tagged during decode as they are entered into the sequencers. The new types of register dependencies come in three varieties, either *depth* registers of length *vl*, *vseg* registers of length *depth*, or *vseg* registers of length *vl*. These additional register

operands impose additional constraints on the microarchitectural dependency tracking which is split between the main sequencer and the lane sequencer. The main sequencer tracks the read-after-write, write-after-read, and write-after-write dependencies between architectural vector registers, while the lane sequencer tracks the vector length and structural dependencies between instructions. For the two-dimensional instructions, the main sequencer tracks the additional registers during the setup of the architectural register dependencies. This is done with additional comparators for each of the instructions currently in the sequencer, which can be smaller than the full register specifier bit-width due to the restricted nature of the *depth* and *vseg* parameters.

These comparators initialize a set of registers that hold a bit for each pair of instructions and each type of dependency, and register source and destination. For example, there is a register signifying whether the instruction in sequencer-slot five's second register input has a read-after-write dependency on the instruction in sequencer-slot two's destination register. These comparisons are only made for instructions active in the main sequencer at the time of issue and do not need to be updated as more instructions are issued because issue occurs in program order. In addition, to save hardware resources the individual registers of the sub-matrix inputs and outputs are not tracked by themselves, rather they are only being tracked at the instruction granularity.

The granularity of the interlocks based on these dependencies were slowly expanded during the implementation. Initially, only one two-dimensional instruction could be active at a time and it needed to wait to begin execution until it was the eldest instruction. This was then upgraded to relax the eldest instruction restriction by ensuring all registers were available. And finally the above scheme fully tracking each sub-matrix instruction's inputs and outputs enables several of these instructions to be active at once without data dependency issues.

The changes to the lane sequencer are more pervasive because the second dimension progression tracking scheme, called nano-operations, affect the design at multiple stages not just at issue time. Hwacha's baseline dependency tracking allows for different portions of a vector instruction, corresponding to the spatial datapath width for that instruction, called *slices* to proceed in parallel. These slices still use the main sequencer's architectural register data-dependency tracking to ensure that each element of a vector register is processed in program order. The lane sequencer is responsible for tracking dynamic dependencies and so also takes control of tracking the new nano-operation progress for sub-matrix instructions. This is a counter that advances as each vector length completes for the sub-matrix operations. Importantly, this does not use the machine-wide vector length as some of the sub-matrix instructions use a local vector length based on *depth*. In addition the registers that will be operated on with these sub-matrix instructions depend on the nano-operation count to determine which vector register will be used. Finally, there are additional datapath elements, described in Section 7.2, that have their structural hazards tracked by the lane sequencer.

With these changes the dense linear algebra extensions can efficiently execute while overlapping execution as much as possible and using many fewer microarchitectural resources. The next section will discuss the microarchitecture and datapath used for the local reductions

during VTMM instruction execution.

7.2 Latch and Reuse for VTMM

In addition to dependency tracking, the microarchitecture needs to enable several new datapath elements for the two-dimension extension. There are two primary datapath enhancements that enable most of the extensions operations. The first of these is a reuse datapath that causes the result of an operation to be fed back in as an input for the next matching element from the subsequent sub-operation. This allows for a local reduction between matching element numbers with different sub-operation indexes, as the sequence of outputs is continually given as an input until the entire set of sub-operations is completed. During the VTMM instruction the reuse operand is set as the third input which is the sub-matrix that will be added to the multiplication result. This means that as the first and second inputs change during sub-operations each vector, or row, of the sub-matrix will be summed together into the same element number output.

This local reduction is the key to efficiently implementing the sub-matrix operations on the existing microarchitecture. Because each strip of the vector is processed as a single pass through the register file banks, consecutive strips based on different sub-operations, when issued consecutively, can avoid re-reading the reused output from the SRAM banks. In addition, when the microarchitecture detects this chaining of sub-operations it can also eliminate all but the final write to SRAM. This reduction of SRAM reads and writes by the reuse operand and destination operand, respectively, can eliminate up to $\frac{7}{8}$ths of the reads and write on these operands when *depth* is set to its maximum value of eight.

The second datapath enhancement, another operand modification called *latch*, has the ability to keep an operand constant across elements of a sub-matrix operation while changing with the sub-operation index. Essentially, this enables the transposition of a vector register into a single column. This change in operand behavior is limited to *depth*-length vector register inputs, and has two primary consequences in the microarchitecture. First, these operands will be held in special per-bank registers to avoid reading their constant values out of the SRAM banks. This change is purely an energy-efficiency optimization and not required for correctness. The second change, necessary for correctness, is that the values of these latch registers are replicated across the entire physical datapath width while being fed into the operand crossbar. This replication is important because as a single slice of a single sub-operation index is processed each register bank will produce many elements of the non-latched operands. According to the specification of latch operands each of these elements should receive the same value of the latch operand, and so we must replicate the data not just temporally to the different elements in banks but also spatially across the entire set of elements read from a single row of the physical register file. This replication is somewhat complicated by the configurable precision of vector registers and needs to correctly replicate the latched operand based on how many of its element type will fit into each row of the SRAM based register file.

In the VTMM instruction latching enables the first input to act as the transposed vector, and is another key to implement the sub-matrix operations. One of the reasons the limit of the *depth* parameter is set to eight is that it ensures the widest data type will fit exactly *depth*-times into a single row of the register file banks. With this constraint we can ensure that a single additional bank operand latch is sufficient to hold an entire latch operand. Because a single operand latch, tracked separately by the lane sequencer, holds the same values for the entire execution of the sub-matrix operation, the latch operand will reduce its SRAM read accesses to just $\frac{1}{depth*vl/ElementsPerSlice}$ reads. Both of these operand specializations combine to greatly reduce the number of SRAM activations for the VTMM operations.

In order to demonstrate that these changes are not purely specialized, the operand modifications are generalized in the microarchitecture. Each of these modifications can be set up at decode time for each instruction allowing one of its operands be a latched operand and one to be a reused operand. Each instruction can set its latch and reuse values independently but only for one operand each. This would allow other local reductions like partial dot products or other more exotic functions that make use of these same underlying datapath and control changes.

7.3   Sub-matrix load micro operations

In addition to arithmetic operations the two-dimensional extension also adds sub-matrix and short-vector loads and stores. These new operations require very few additions to the microarchitecture, due to the fact that the VMU is already decoupled from the VXU. The VMU is already designed to track independent vector lengths because it could be processing a separate vector-fetch block which could have a different vector length. This enables the changes for short-vector loads to simply reuse this mechanism at issue time.

For the sub-matrix loads, an additional state machine is added in the issue unit of the VMU that reissues the different segments of sub-matrix memory operations. In addition to reissuing each segment, the state machine also must update the base address supplied to rest of the VMU, which requires an additional virtual-address-sized adder. This adder becomes the primary additional cost of the sub-matrix memory operations but does not need to include a multiplier as the segments and their stride are calculated and issued in-order, so repeated addition is sufficient. In addition to the state machine interacting with many of the units in the VMU, the VXU also needs to be aware of the state machine to keep the dependency tracking in sync across segments.

One of the primary benefits of these operations in the microarchitecture is the reduction of sequencer slots needed to represent a large amount of data movement. Each vector memory instruction uses three sequencer slots to represent its address checking, predicate checking, and actual memory operation. With the normal sequencer size of eight slots only two memory instructions can proceed in parallel. This is particularly limiting once the VTMM instructions are in place and can consume more than eight full length vector registers at once. The sub-matrix loads and stores thus allow much more overlap between

the compute and memory portions of dense linear algebra kernels, than with the VTMM instructions alone.

7.4   Future Extensions for Machine Learning

As discussed in the previous chapter the dense linear algebra instructions can lead to further extensions for deep learning. The overlap between these domains is significant and this results in meaningful overlap in the microarchitectural implementation as well.

One of the simpler extensions of the microarchitectural changes made for VTMM is to enable other nearly repeated calculations. Using the short-vector latch support for the filter of a convolution allows for a one-dimensional convolution to be simply implemented as a new decode table entry with using the fused-multiply-add unit in the standard single-dimension way but with the first operand being labeled as a latch operand. Because this is a reuse of existing datapath and control the incremental cost of a convolution instruction is very minor, only slightly expanding the decoding cost of instructions.

A pooling instruction is another local reduction, similar to VTMM. Depending on the parameters of the pooling kernel and amount of sub-sampling this local reduction may be able to reuse some of the datapath of the VTMM instructions along the *depth* direction. However, because part of the local reduction is occurring along the vector-length dimension, if the elements that need combining here are within the same slice then new datapath elements and potentially buffering registers will be needed to complete the operation. This additional datapath could be nearly as expensive as the initial VTMM hardware making the benefits less clear than the convolution operations. It would be worthwhile to see if pooling causes a significant bottleneck, and whether using VTMM's *depth*-wise local reductions as a portion of the pooling kernel can achieve a significant speedup without additional hardware for the vector-length reduction.

Finally, the data-width reductions discussed last chapter would have a significant impact on the peak energy efficiency of the design as small bit-width operations consume quadratically less energy. In addition, there could be compounding improvements if the physical register width is maintained and the maximum bit-width operations supported is reduced. In this case if the microarchitecture only supports 32-bit wide operations then the size of *depth* could be expanded to a maximum of 16 improving the reduction of SRAM accesses from the sub-matrix operations. This configurability isn't built into the current Hwacha microarchitecture but would be a very interesting direction to explore as nearly all deep learning work is done on single-precision numbers or narrower bit-width data-types. These extensions could have a significant further improvement on Hwacha's energy efficiency while executing machine learning workloads, but require more detailed design and analysis to determine the exact benefits.

7.5 Physical Implementation Results

Despite the analysis of the microarchitecture showing a large reduction in SRAM accesses it is important to fully evaluate the design through a VLSI flow to determine the overall system improvement. As with the latest silicon implementations of Hwacha, Hammer will be used to map the extended and standard microarchitecture to the same process using the Cadence tool plugins. This experiment was done on two technologies, TSMC 16nm FinFET and Intel 22nm. Only the 22nm results are reported, but the 16nm experiment showed similar results, building confidence in the implementation's ability to provide benefits across technologies.

Previously Hammer had been designed to make the tapeout process easier and more reusable and so the Hammer flow needed for energy-efficiency results did not exist. A few additional features were needed in order to make the collection of results easier. First Hammer, needed to support dynamic power analysis based on a specific application. This support was added by implementing a plugin for the Cadence power-analysis tool Voltus, enabling both static power analysis and dynamic power analysis based on compressed activity factors and full waveforms.

These last two dynamic power analysis inputs also required additional Hammer development. The waveform generation facility was already implemented at a basic level while the activity factor generation had to be implemented from scratch. Both of these inputs require simulation of the design running a given application. As a result the Synopsys VCS simulator plugin was extended to include both of these functions, as API options for the user when running a simulation. In addition, in order to restrict the waveform and activity generation to interesting parts of longer running applications, support for adding starting and stopping triggers for waveforms and activity factors was added as an option to Hammer simulations. With these features, and the support for multiple simulations to be run at once, a series of dynamic power reports could be generated automatically for a set of applications and hardware designs.

The final portion of results analysis that was missing from Hammer was the ability to parse these reports and produce summaries or more readable results. This functionality was added as an independent set of scripts, which consumes the power analysis reports, the simulation results, and a place and route report to determine power usage, cycles to completion, and clock frequency, respectively. With these figures it is possible to determine the total energy consumed by the design and the number of operations completed. When reported as a ratio this figure is energy-efficiency on a per application basis for the given hardware configuration. This energy efficiency figure is the metric that the sub-matrix operations are trying to improve by reducing the SRAM activations, so graphing this quantity for the same benchmarks with and without the sub-matrix extension will determine the utility of the extension.

Before reporting this goal metric, the next two subsections will describe some two of the core components of energy efficiency, power and performance.

Power

The sub-matrix extension does not focus on significant performance improvements and so is unlikely to increase the power consumption the way a traditional processor extension might, by dramatically increasing the amount of work that can be done per unit time. Instead, the increase in power from the sub-matrix extension is encapsulated in the additional control logic and datapath elements that are needed for its operation. This means that the increase in power will be strongly correlated with the area of these new components, as the control logic will always be operating and the datapath elements will continue to draw static power even when not in use.

Most of the increase in area and gates from the addition of the sub-matrix extension occurs in two areas, additional buffering and latching needed to hold temporary results, and the control logic needed to track the usage of these buffers and new dependencies added with new access patterns. It is difficult to count these areas of increase separately but the overall gate count of the Hwacha instance increased by around 10% with the sub-matrix extension. This is a considerable increase considering the handful of bank-wide registers that were added and the control logic changes. A more detailed analysis and additional microarchitecture would probably be able to reduce this area increase, and therefore improve the benefits of the sub-matrix extension.

Performance

The sub-matrix extension is able to improve performance on most of the benchmarks where it is applicable. There are two regions of operation where the sub-matrix extension can improve performance, depending on the size of the application's dataset. The first region occurs at small vector lengths where the baseline microarchitecture is unable to saturate the memory bandwidth or functional unit's throughput. The second region is entered once the size of the computation enables peak performance on the baseline microarchitecture. This second region has a loose boundary with 90% of peak performance occurring around a vector length of 128 for double-precision operations, 80% at 64, and 68% at 32, but only 33% at 16 elements.

The primary benefit to performance comes from the ability of the sub-matrix instructions to effectively extend the vector length for narrow matricies, which only affects the first region. This effective increase in vector length enables the sub-matrix operations to fully utilize the memory bandwidth and functional unit throughput despite the small dimensions of the dataset. Figure 7.1 shows a sweep across different matrix sizes and different precisions, showing the difference in performance between the baseline implementation and the sub-matrix extensions. Each pair of bars represents the time to completion for a particular precision and matrix-dimension GEMM, normalized to the baselines time to completion for that benchmark. The benchmarks most clearly in the first performance region are the small-matrix-size half-precision GEMMs which show around a 40% reduction in runtime. The baseline implementation on these benchmarks are well under 50% utilization and so the
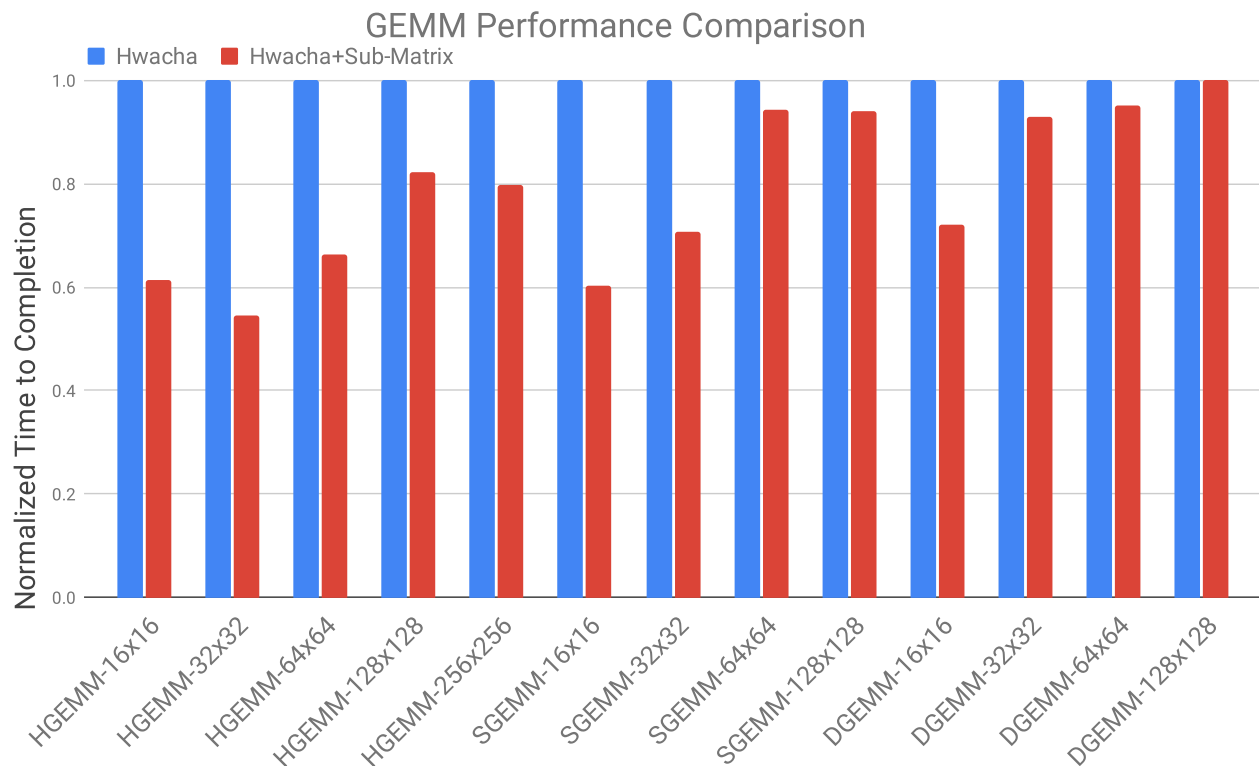
Figure 7.1: A comparison of the performance for various precision and matrix size GEMMs.

sub-matrix operation extends the vector length and is able to keep the microarchitecture busy, greatly improving performance. The second region can be seen in the double- and single-precision 128-by-128 and 64-by-64 matrix-dimension benchmarks which have less than a 10% runtime reduction, and in the case of the 128-by-128 double-precision version less than a 1% reduction. In this region the baseline architecture is already able to keep the microarchitecture almost fully utilized, so the extra vector length from sub-matrix operations has diminishing returns.

There is also a small increase in the microarchitecture's ability to keep the long-latency functional units occupied, irrespective of the vector length. This occurs at the end of a sequence of sub-operations which would normally coincide with the end of a normal single-dimension instruction, which sometimes leads to bubbles in the baseline implementation. This leads to the innermost loop of a GEMM having a few percentage points increase in utilization of the functional units, even when the machine is otherwise fully utilized.

Another important factor in performance for these kernels is overlapping the memory operations and arithmetic operations. With the limited amount of in-flight instructions enabled by 8-entry sequencers, and the desire to maintain in-order issue and completion, the

long occupancy of two-dimensional operations can make code-scheduling difficult. A single sub-matrix load can be responsible for several thousands of memory requests occupying the sequencer for potentially more than ten thousand cycles. The impact of this occupancy is somewhat ameliorated by the microarchitecture's ability to track the dependencies of the individual rows loaded by the sub-matrix load. However careful code scheduling is still necessary to achieve good performance on workloads with many sub-matrix operations. For example reducing the amount of time a kernel spends purely loading data can require a separate instruction sequence for the first iteration, which uses lower *depth* loads to enable compute instructions to enter the sequencer earlier. In addition, traditional double buffering improves performance more than expected by allowing these long occupancy sub-matrix memory operations to nearly completely overlap with the compute.

The comparison to the baseline above compares hand-tuned versions of each benchmark against each other. It shows the benefits of sub-matrix operations in reducing the runtime of dense linear-algebra computation. However, this analysis excludes a key benefit of the extension which is the energy-use reduction based on the reduced SRAM activations.
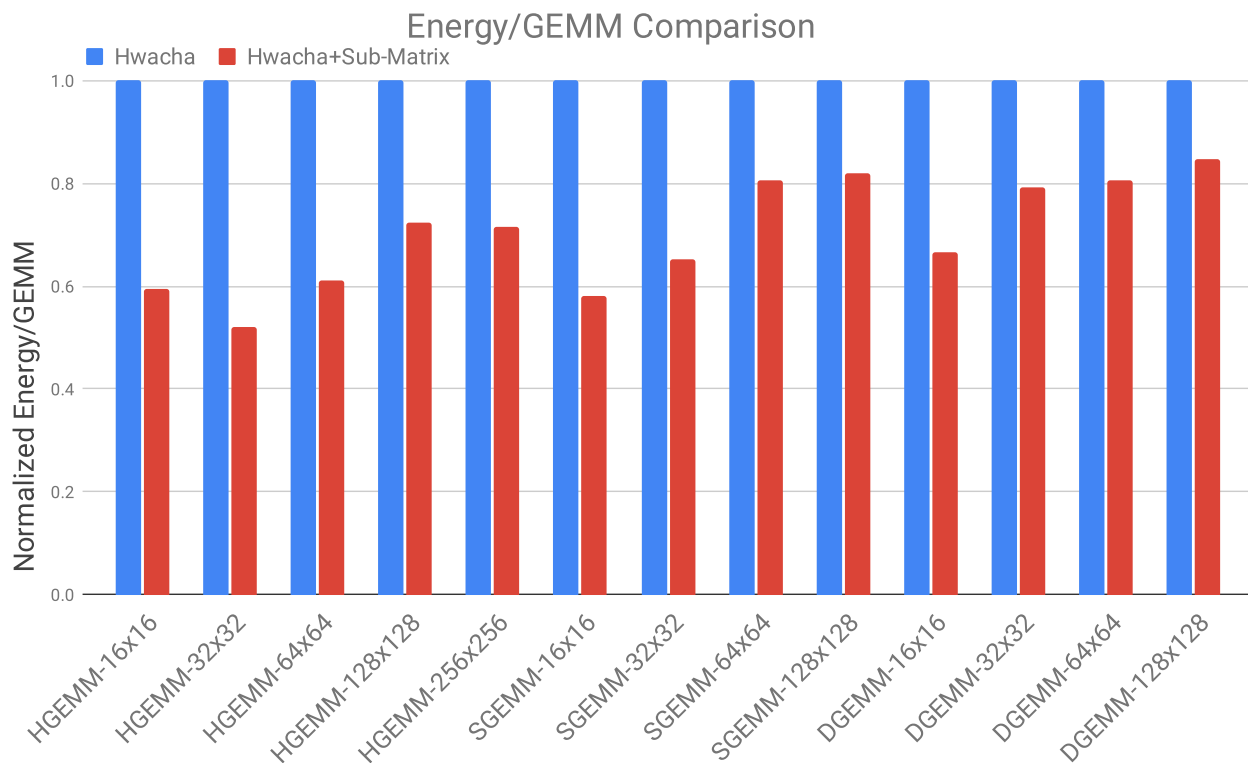

Figure 7.2: A comparison of the energy efficiency for various precision and matrix size GEMMs.

Energy

In order to determine the reduction in energy-use the power consumed by each implementation during the benchmarks needs to be measured. Fortunately, Hammer can use the standard power analysis tool, Voltus, to perform this measurement for each benchmark on both implementations. The energy-use of the sub-matrix extension is reduced for two reasons. The first reason is the reduction in SRAM accesses, and the second reason is a reduction in run-time that will reduce the time component of energy. Both of these reductions are counteracted by the increase in static power of the additional gates and storage elements needed to implement the sub-matrix extension.

This again brings into focus the two regions of performance, long-vector regions and short-vector regions. Figure 7.2 again shows a comparison of the energy-use between the baseline configuration and the sub-matrix extension normalized for each benchmark. In the short-vector region on the far left there are improvements between 40% and nearly 50%. Comparing these data points to the performance figures, there is a very slight additional benefit in energy-use from the SRAM access reduction. However in the long vector region on the far right, the difference is much more impactful. The 128-by-128 DGEMM has less than a 1% performance improvement but has more than a 15% energy reduction.
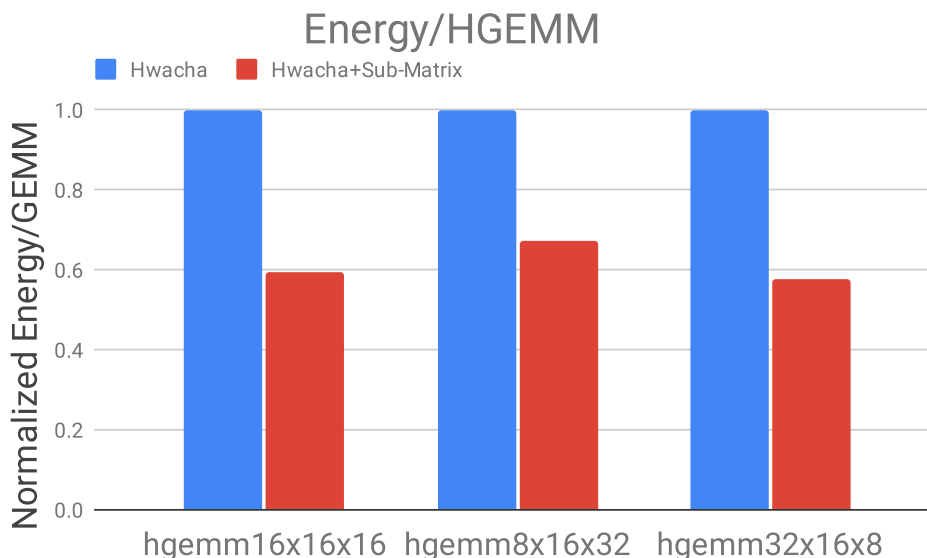

Figure 7.3: A comparison of the energy efficiency for various shapes of half-precision GEMMs.

Because the natural dimension of GEMMs mapped to Hwacha is along the rows, when this dimension is reduced, the vector length is reduced and utilization of the functional units and memory bandwidth quickly decreases. The sub-matrix extension should be able counteract this and to see this effect in more detail a comparison of different shapes of matricies is performed. Figure 7.3 shows three different HGEMMs with the same number of operations in each and the same inner dimension but with distinct outer dimensions. The first bar on

the left is the square-matrix version which is the same as the previous figure showing a little over 40% energy-use reduction. The middle benchmark has a short and squat B matrix with 32 columns and 16 rows, while the right benchmark has a B matrix with only 8 columns and 16 rows. The size of matrix A is adjusted to ensure the same number of operations occur in each GEMM. Comparing these two cases shows that the benefit of the sub-matrix extension is improved as the vector-length of the machine is reduced independent of the total amount of operations done. This matches the intuition that the relative improvement of the additional sub-matrix vector-length will be more profound at short vector-lengths.

The performance benefit on short vector benchmarks and energy improvement on long vector benchmarks make the sub-matrix extension useful for all dense linear algebra workloads. In addition, machine learning and thus linear algebra workloads occupy a significant portion of computing resources in modern applications. Therefore it would be beneficial to include this sub-matrix extension in all future versions of the Hwacha architecture.

Chapter 8

Conclusion

The intersection of specialization and programmability are critical to maintaining the desired efficiency improvements and flexibility modern applications demand in a post-Dennard scaling and post-Moore's law world. Extending existing architectures that already have many features that are desired in a fully specialized design enables a middle-ground of efficiency that can improve on the baseline architecture without the large investment of an entirely separate design. Continuing to explore this methodology, adapting it to new application spaces, or new baseline architectures could lead to further improvements in efficiency or performance. This thesis has shown that one such extension of a specific existing architecture can produce concrete and significant benefits to system efficiency.

8.1 Thesis Summary of Contributions

The primary contributions of this thesis are:

- A detailed review of data-parallel architectural paradigms, including code examples and discussion of design trade-offs in the programmability-efficiency space (Chapter 2).

- A survey of past and present multi-dimensional architectures, exploring the evolution from the natural parallelism of early computer systems into the explicit parallelism of multi-media extensions and the modern deep-learning accelerators (Chapter 3.

- An exploration of modern technology constraints and their affect on data-parallel architecture development and implementation (Chapter 4).

- A thorough discussion of four test chips implementing various instances of temporal vector-fetch architectures highlighting the challenges associated with building many chips and describing a novel physical design methodology used for two of these chips. Also presented are the best-in-class results, at the time of publishing, for a programmable data-parallel extension to RISC-V (Chapter 5).

- An extension to the Hwacha vector-fetch architecture specialized for two-dimensional workloads (Chapter 6). An analysis of the implementation complexities and architectural constraints the extension imposes (Chapter 7). And finally, a discussion of the results of mapping this implementation to two modern technologies, showing significant improvements in the overall system's energy efficiency (Section 7.5).

8.2 Future Work

There are many future directions for the work discussed in this thesis. Some of this future work is alluded to in chapters 6 and 7. A consolidation of that work and other future research ideas is presented here:

- Further extensions in the two-dimensional application space specifically targeted towards deep learning could offer further energy-efficiency benefits. These ideas are discussed in sections 6.3 and 7.4.

- The novel physical design methodology used for two of the test chips could still be improved significantly. Performing more controlled experiments between this methodology and more traditional ones would help give the idea more than anecdotal support. In addition, leveraging design knowledge from the RTL without re-encoding it as lower level physical design constraints could further elevate the methodology.

- Exploring other popular computational domains could be productive in producing energy-efficiency improvements for those domains. Also it could produce another positive example of the principle of this thesis that extensions to temporal vector-fetch architectures can be performant and efficient.

- If multiple domains can be supported independently on one baseline design then exploring the composition of such extensions would be extremely interesting. Determining whether the specialization's when combined are more than the sum of their parts or whether conflicts reduce overall system efficiency would be a valuable line of study.

- Finally, extensions to other baseline architectures could be explored to determine whether temporal vector-fetch architectures are unique in their appropriateness for these kinds of extensions or whether other general data-parallel architectures could support the same types of extensions.

Bibliography

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. doi:10.1109/MICRO.2016.7783725.

[2] Dario Amodei and Danny Hernandez. AI and compute, 2019. URL: https://openai.com/blog/ai-and-compute/.

[3] Apple. iPhone 11 pro and iPhone 11 pro max: the most powerful and advanced smartphones, 2019. URL: https://www.apple.com/newsroom/2019/09/iphone-11-pro-and-iphone-11-pro-max-the-most-powerful-and-advanced-smartphones/.

[4] S. Arora, D. Bouvier, and C. Weaver. AMD next generation 7nm Ryzen^{tm} 4000 APU r̈enoir:̈ In *2020 IEEE Hot Chips 32 Symposium (HCS)*, 2020.

[5] K. Asanović, J. Beck, B. Irissou, B. E. D. Kingsbury, and J. Wawrzynek. T0: A single-chip vector microprocessor with reconfigurable pipelines. In *ESSCIRC '96: Proceedings of the 22nd European Solid-State Circuits Conference*, pages 344–347, 1996.

[6] Krste Asanović. *Vector Microprocessors*. PhD thesis, EECS Department, University of California, Berkeley, 1998. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/6404.html.

[7] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. Technical Report UCB/EECS-2016-17.

[8] C. Auth, A. Aliyarukunju, M. Asoro, D. Bergstrom, V. Bhagwat, J. Birdsall, N. Bisnik, M. Buehler, V. Chikarmane, G. Ding, Q. Fu, H. Gomez, W. Han, D. Hanken, M. Haran, M. Hattendorf, R. Heussner, H. Hiramatsu, B. Ho, S. Jaloviar, I. Jin, S. Joshi, S. Kirby, S. Kosaraju, H. Kothari, G. Leatherman, K. Lee, J. Leib, A. Madhavan, K. Marla, H. Meyer, T. Mule, C. Parker, S. Parthasarathy, C. Pelto, L. Pipes, I. Post, M. Prince, A. Rahman, S. Rajamani, A. Saha, J. D. Santos, M. Sharma,

V. Sharma, J. Shin, P. Sinha, P. Smith, M. Sprinkle, A. S. Amour, C. Staus, R. Suri, D. Towner, A. Tripathi, A. Tura, C. Ward, and A. Yeoh. A 10nm high performance and low-power CMOS technology featuring 3rd generation FinFET transistors, self-aligned quad patterning, contact over active gate and cobalt local interconnects. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 29.1.1–29.1.4, 2017. doi:10.1109/IEDM.2017.8268472.

[9] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012. doi:10.1145/2228360.2228584.

[10] Woorham Bae. Supply-scalable high-speed I/O interfaces. *Electronics*, 9(8), 2020. URL: https://www.mdpi.com/2079-9292/9/8/1315, doi:10.3390/electronics9081315.

[11] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann. An always-on 3.8J/86processor with all memory on chip in 28nm CMOS. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 222–224, 2018. doi:10.1109/ISSCC.2018.8310264.

[12] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, 1968. doi:10.1109/TC.1968.229158.

[13] Kenneth E. Batcher. Architecture of a massively parallel processor. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, ISCA '80, page 168173, New York, NY, USA, 1980. Association for Computing Machinery. URL: https://doi.org/10.1145/800053.801922, doi:10.1145/800053.801922.

[14] C. Berry, J. Warnock, J. Isakson, J. Badar, B. Bell, F. Malgioglio, G. Mayer, D. Hamid, J. Surprise, D. Wolpert, O. Geva, B. Huott, L. Sigal, S. Carey, R. Rizzolo, R. Nigaglioni, M. Cichanowski, D. Chidambarrao, C. Jacobi, A. Saporito, A. O'neill, R. Sonnelitter, C. Zoellin, M. Wood, and J. Neves. IBM z14: 14nm microprocessor for the next-generation mainframe. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 36–38, 2018.

[15] T. Blank. The MasPar MP-1 architecture. In *Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*, pages 20–24, 1990. doi:10.1109/CMPCON.1990.63648.

[16] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986. doi:10.1147/sj.251.0051.

[17] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill. FIVR fully integrated voltage regulators on 4th generation intel core SoCs. In *APEC*, pages 432–439, 2014.

[18] A. Ceyhan, M. Jung, S. Panth, S. K. Lim, and A. Naeemi. Evaluating chip-level impact of Cu/low- κ performance degradation on circuit performance at future technology nodes. *IEEE Transactions on Electron Devices*, 62(3):940–946, 2015. doi:10.1109/TED.2015.2394407.

[19] J. Chang, Y. Chen, G. Chan, H. Cheng, P. Wang, Y. Lin, H. Fujiwara, R. Lee, H. Liao, P. Wang, G. Yeap, and Q. Li. A 5nm 135Mb SRAM in EUV and high-mobility-channel FinFET technology with metal coupling and charge-sharing write-assist circuitry schemes for high-density and low-VMIN applications. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 238–240, 2020. doi:10.1109/ISSCC19947.2020.9062967.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.

[21] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), October 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com. URL: https://hal.inria.fr/inria-00112631.

[22] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016. doi:10.1109/ISCA.2016.40.

[23] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017. doi:10.1109/JSSC.2016.2616357.

[24] Wesley A. Clark. The lincoln TX-2 computer development. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), page 143145, New York, NY, USA, 1957. Association for Computing Machinery. URL: https://doi.org/10.1145/1455567.1455592, doi:10.1145/1455567.1455592.

[25] J. Crossley, A. Puggelli, H. . P. Le, B. Yang, R. Nancollas, K. Jung, L. Kong, N. Narevsky, Y. Lu, N. Sutardja, E. J. An, A. L. Sangiovanni-Vincentelli, and E. Alon.

BAG: A designer-oriented integrated framework for the development of AMS circuit generators. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 74–81, 2013. doi:10.1109/ICCAD.2013.6691100.

[26] S. Dasgupta, T. Singh, A. Jain, S. Naffziger, D. John, C. Bisht, and P. Jayaraman. Radeon RX 5700 series: The AMD 7nm energy-efficient high-performance GPUs. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 150–152, 2020. doi:10.1109/ISSCC19947.2020.9062947.

[27] Elena Demikohovsky. Vectorization of control flow with new masked vector intrinsics. In *LLVM Developer's Meeting 2015*, 2015.

[28] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):117, March 1990. URL: https://doi.org/10.1145/77626.79170, doi:10.1145/77626.79170.

[29] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728, doi:https://doi.org/10.1002/cpe.728.

[30] Marat Dukhan. The indirect convolution algorithm, 2019. arXiv:1907.02129.

[31] Wael Mahmoud Elsharkasy. *Low Power Reliable Design using Pulsed Latch Circuits*. PhD thesis, University of California Irvine, 2017.

[32] K. Fischer, M. Agostinelli, C. Allen, D. Bahr, M. Bost, P. Charvat, V. Chikarmane, Q. Fu, C. Ganpule, M. Haran, M. Heckscher, H. Hiramatsu, E. Hwang, P. Jain, I. Jin, R. Kasim, S. Kosaraju, K. S. Lee, H. Liu, R. McFadden, S. Nigam, R. Patel, C. Pelto, P. Plekhanov, M. Prince, C. Puls, S. Rajamani, D. Rao, P. Reese, A. Rosenbaum, S. Sivakumar, B. Song, M. Uncuer, S. Williams, M. Yang, P. Yashar, and S. Natarajan. Low-k interconnect stack with multi-layer air gap and tri-metal-insulator-metal capacitors for 14nm high volume manufacturing. In *2015 IEEE International Interconnect Technology Conference and 2015 IEEE Materials for Advanced Metallization Conference (IITC/MAM)*, pages 5–8, 2015. doi:10.1109/IITC-MAM.2015.7325600.

[33] G. Fredeman, D. W. Plass, A. Mathews, J. Viraraghavan, K. Reyer, T. J. Knips, T. Miller, E. L. Gerhard, D. Kannambadi, C. Paone, D. Lee, D. J. Rainey, M. Sperling, M. Whalen, S. Burns, R. R. Tummuru, H. Ho, A. Cestero, N. Arnold, B. A. Khan, T. Kirihata, and S. S. Iyer. A 14 nm 1.1 Mb embedded DRAM macro with 1 ns access. *IEEE Journal of Solid-State Circuits*, 51(1):230–239, 2016. doi:10.1109/JSSC.2015.2456873.

[34] Andrei Frumusanu. The 2020 mac mini unleashed: Putting apple silicon M1 to the test, 2020. URL: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested.

[35] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolić, Ion Stoica, and Krste Asanović. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures, 2019. arXiv:1911.09925.

[36] Xinfei Guo, Vaibhav Verma, Patricia Gonzalez-Guerrero, Sergiu Mosanu, and Mircea R. Stan. Back to the future: Digital circuit design in the FinFET era. *Journal of Low Power Electronics*, 13(3):338–355, 2017. URL: https://www.ingentaconnect.com/content/asp/jolpe/2017/00000013/00000003/art00006, doi:doi:10.1166/jolpe.2017.1489.

[37] F. Hamzaoglu, U. Arslan, N. Bisnik, S. Ghosh, M. B. Lal, N. Lindert, M. Meterelliyoz, R. B. Osborne, J. Park, S. Tomishima, Y. Wang, and K. Zhang. A 1Gb 2GHz embedded DRAM in 22nm tri-gate CMOS technology. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 230–231, 2014. doi:10.1109/ISSCC.2014.6757412.

[38] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[39] William Daniel Hillis. *The connection machine*. PhD thesis, Massachusetts Institute of Technology, 1988.

[40] S. Hsu, A. Agarwal, M. Anders, S. Mathew, H. Kaul, F. Sheikh, and R. Krishnamurthy. A 280mV-to-1.1V 256b reconfigurable SIMD vector permutation engine with 2-dimensional shuffle in 22nm CMOS. In *2012 IEEE International Solid-State Circuits Conference*, pages 178–180, 2012. doi:10.1109/ISSCC.2012.6176966.

[41] S. Hsu, A. Agarwal, M. Anders, S. Mathew, R. Krishnamurthy, and S. Borkar. An 8.8GHz 198mW 16x64b 1R/1W variationtolerant register file in 65nm CMOS. In *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*, pages 1785–1797, 2006. doi:10.1109/ISSCC.2006.1696235.

[42] F. Hsueh, H. Chiu, C. Shen, J. Shieh, Y. Tang, C. Yang, H. Chen, W. Huang, B. Chen, K. Chen, G. Huang, W. Chen, K. Hsu, S. R. Srinivasa, N. Jao, A. Lee, H. Lee, V. Narayanan, K. Wang, M. Chang, and W. Yeh. TSV-free FinFET-based monolithic 3D+-IC with computing-in-memory SRAM cell for intelligent IoT devices. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 12.6.1–12.6.4, 2017. doi:10.1109/IEDM.2017.8268380.

[43] Graham Hunter. Supporting SIMD instruction sets with variable vector lengths, 2018. URL: http://lists.llvm.org/pipermail/llvm-dev/2018-June/123780.html.

[44] Intel. Accelerating innovation through a standard chiplet interface: The advanced interface bus (AIB). URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/accelerating-innovation-through-aib-whitepaper.pdf.

[45] Intel. Intel 64 and IA-32 architectures software developer's manual combined volumes 2A, 2B, 2C, and 2D: Instruction set reference, a-z, 2020. URL: https://software.intel.com/content/dam/develop/public/us/en/documents/325383-sdm-vol-2abcd.pdf.

[46] Stanford Humman-Centered Artificial Intelligence. Artificial intelligence index 2019 annual report, 2019. URL: https://hai.stanford.edu/research/ai-index-2019.

[47] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers, 2020. arXiv:2007.00072.

[48] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216, Nov 2017. doi:10.1109/ICCAD.2017.8203780.

[49] P. Jain, U. Arslan, M. Sekhar, B. C. Lin, L. Wei, T. Sahu, J. Alzate-vinasco, A. Vangapaty, M. Meterelliyoz, N. Strutt, A. B. Chen, P. Hentges, P. A. Quintero, C. Connor, O. Golonzka, K. Fischer, and F. Hamzaoglu. A 3.6Mb 10.1Mb/mm2 embedded non-volatile ReRAM macro in 22nm FinFET technology with adaptive forming/set/reset schemes yielding down to 0.5V with sensing time of 5ns at 0.7V. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 212–214, 2019. doi:10.1109/ISSCC.2019.8662393.

[50] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7):6778, June 2020. URL: https://doi.org/10.1145/3360307, doi:10.1145/3360307.

[51] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt,

Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 112, New York, NY, USA, 2017. Association for Computing Machinery. URL: https://doi.org/10.1145/3079856.3080246, doi:10.1145/3079856.3080246.

[52] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim. HBM (high bandwidth memory) DRAM technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017. doi:10.1109/IMW.2017.7939084.

[53] E. Karl, Y. Wang, Y. Ng, Z. Guo, F. Hamzaoglu, U. Bhattacharya, K. Zhang, K. Mistry, and M. Bohr. A 4.6GHz 162Mb SRAM design in 22nm tri-gate CMOS technology with integrated active VMIN-enhancing assist circuitry. In *2012 IEEE International Solid-State Circuits Conference*, pages 230–232, 2012. doi:10.1109/ISSCC.2012.6176988.

[54] B. Keller, M. Cochet, B. Zimmer, Y. Lee, M. Blagojevic, J. Kwak, A. Puggelli, S. Bailey, P. Chiu, P. Dabbelt, C. Schmidt, E. Alon, K. Asanović, and B. Nikolić. Submicrosecond adaptive voltage scaling in a 28nm FD-SOI processor SoC. In *ESSCIRC*, pages 269–272, 2016.

[55] P. Kharya. TensorFloat-32 in the A100 GPU accerlerates AI training, HPC up to 20x, May 2020. URL: https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

[56] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, page 52, USA, 2004. IEEE Computer Society.

[57] D. J. Kuck. ILLIAC IV software and application programming. *IEEE Transactions on Computers*, C-17(8):758–770, 1968. doi:10.1109/TC.1968.229159.

[58] V. V. Kulkarni, W. Y. Lim, B. Zhao, D. L. Yan, Y. Wang, J. Zhou, and M. A. Arasu. A 5.1Gb/s 60.3fJ/bit/mm PVT tolerant NoC transceiver. In *2016 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 141–144, 2016. doi:10.1109/ASSCC.2016.7844155.

[59] A. Kumar, S. Kottapalli, I. M. Steiner, B. Valentine, I. Hirsh, G. Vedaraman, L. P. Looi, M. Arafa, A. Rudoff, S. Mandava, B. Fahim, and S. A. Vora. Future Intel Xeon scalable processor. In *2018 IEEE Hot Chips 30 Symposium (HCS)*, 2018.

[60] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982. doi: 10.1109/MC.1982.1653825.

[61] M. LaPedus. Big trouble at 3nm, 2018. URL: https://semiengineering.com/big-trouble-at-3nm/.

[62] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308323, September 1979. URL: https://doi.org/10.1145/355841.355847, doi:10.1145/355841.355847.

[63] Y. LeCun. Deep learning hardware: Past, present, and future. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 12–19, 2019. doi: 10.1109/ISSCC.2019.8662396.

[64] LLVM. User guide for NVPTX back-end, 2020. URL: https://llvm.org/docs/NVPTXUsage.html.

[65] R. Mahajan, R. Sankman, N. Patel, D. Kim, K. Aygun, Z. Qian, Y. Mekonnen, I. Salama, S. Sharan, D. Iyengar, and D. Mallik. Embedded multi-die interconnect bridge (EMIB) – a high density, high bandwidth packaging interconnect. In *2016 IEEE 66th Electronic Components and Technology Conference (ECTC)*, pages 557–565, 2016. doi:10.1109/ECTC.2016.201.

[66] P. Meinerzhagen, C. Roth, and A. Burg. Towards generic low-power area-efficient standard cell based memory architectures. In *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, pages 129–132, 2010. doi:10.1109/MWSCAS.2010.5548579.

[67] E. Mensink, D. Schinkel, E. A. M. Klumperink, E. van Tuijl, and B. Nauta. Power efficient gigabit communication over capacitively driven RC-limited on-chip interconnects. *IEEE Journal of Solid-State Circuits*, 45(2):447–457, 2010. doi:10.1109/JSSC.2009.2036761.

[68] NVIDIA. NVIDIA A100 tensor core gpu architecture, 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[69] V. Pano, I. Tekin, I. Yilmaz, Y. Liu, K. R. Dandekar, and B. Taskin. TSV antennas for multi-band wireless communication. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(1):100–113, 2020. doi:10.1109/JETCAS.2020.2974236.

[70] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996. doi:10.1109/40.526924.

[71] 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*, ISCA '13, page 2435, New York, NY, USA, 2013. Association for Computing Machinery. URL: https://doi.org/10.1145/2485922.2485925, doi:10.1145/2485922.2485925.

[72] S. F. Reddaway. DAPa distributed array processor. In *Proceedings of the 1st Annual Symposium on Computer Architecture*, ISCA '73, page 6165, New York, NY, USA, 1973. Association for Computing Machinery. URL: https://doi.org/10.1145/800123.803971, doi:10.1145/800123.803971.

[73] Karl Rupp. CPU, GPU and MIC hardware characteristics over time, 2013. URL: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/.

[74] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):6372, January 1978. URL: https://doi.org/10.1145/359327.359336, doi:10.1145/359327.359336.

[75] Colin Schmidt, Alon Amid, John Wright, Ben Keller, Howard Mao, Keertana Settaluri, Jarno Salomaa, Jerry Zhao, Albert Ou, Krste Asanović, and Borivoje Nikolić. Programmable fine-grained power management and system analysis of RISC-V vector processors in 28-nm FD-SOI. *IEEE Solid-State Circuits Letters*, 3:210–213, 2020. doi:10.1109/LSSC.2020.3010295.

[76] Colin Schmidt, John Wright, Zhongkai Wang, Eric Chang, Albert Ou, Woorham Bae, Sean Huang, Anita Flynn, Brian Richards, Krste Asanović, Elad Alon, and Borivoje Nikolić. An eight-core 1.44GHz RISC-V vector machine in 16nm FinFET. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pages 58–60, 2021. doi:10.1109/ISSCC42613.2021.9365789.

[77] A. Seznec and R. Espasa. Conflict-free accesses to strided vectors on a banked cache. *IEEE Transactions on Computers*, 54(7):913–916, 2005. doi:10.1109/TC.2005.110.

[78] Yakun Sophia Shai. *Design and Modeling of Specialized Architectures*. PhD thesis, Harvard University, 2016.

[79] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer,

C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1427, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3352460.3358302, doi:10.1145/3352460.3358302.

[80] A. Shokrollahi, D. Carnelli, J. Fox, K. Hofstra, B. Holden, A. Hormati, P. Hunt, M. Johnston, J. Keay, S. Pesenti, R. Simpson, D. Stauffer, A. Stewart, G. Surace, A. Tajalli, O. T. Amiri, A. Tschank, R. Ulrich, C. Walter, F. Licciardello, Y. Mogentale, and A. Singh. A pin-efficient 20.83Gb/s/wire 0.94pJ/bit forwarded clock CNRZ-5-coded SerDes up to 12mm for MCM packages in 28nm CMOS. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 182–183, 2016. doi:10.1109/ISSCC.2016.7417967.

[81] T. Singh, S. Rangarajan, D. John, R. Schreiber, S. Oliver, R. Seahra, and A. Schaefer. Zen 2: The AMD 7nm energy-efficient high-performance x86-64 microprocessor core. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 42–44, 2020.

[82] K. Sohn, W. Yun, R. Oh, C. Oh, S. Seo, M. Park, D. Shin, W. Jung, S. Shin, J. Ryu, H. Yu, J. Jung, K. Nam, S. Choi, J. Lee, U. Kang, Y. Sohn, J. Choi, C. Kim, S. Jang, and G. Jin. A 1.2V 20nm 307GB/s HBM DRAM with at-speed wafer-level I/O test scheme and adaptive refresh considering temperature distribution. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 316–317, 2016. doi:10.1109/ISSCC.2016.7418034.

[83] P. Steenkiste. Network-based multicomputers: a practical supercomputer architecture. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):861–875, 1996. doi:10.1109/71.532117.

[84] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, March 2012.

[85] Chen Sun, Mark Wade, Yunsup Lee, Jason Orcutt, Luca Alloatti, Michael Georgas, Andrew Waterman, Jeffrey Shainline, Rimas Avizienis, Sen Lin, Benjamin Moss, R. Kumar, Fabio Pavanello, Amir Atabaki, Henry Cook, Albert Ou, Jonathan Leu, Yu-Hsin Chen, Krste Asanović, and Vladimir Stojanovic. Single-chip microprocessor that communicates directly using light. *Nature*, 528:534–538, 12 2015. doi:10.1038/nature16454.

[86] Hikaru Takayashiki, Masayuki Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. A skewed multi-banked cache for many-core vector processors. *Supercomputing Fron-*

tiers and Innovations, 6(3), 2019. URL: https://superfri.org/superfri/article/view/288.

[87] K. Tan, P. Chiang, Y. Wang, H. Zhao, A. Roldan, H. Zhao, N. Narang, S. W. Lim, D. Carey, S. L. Chaitanya Ambatipudi, P. Upadhyaya, Y. Frans, and K. Chang. A 112-GB/S PAM4 transmitter in 16nm FinFET. In *2018 IEEE Symposium on VLSI Circuits*, pages 45–46, 2018. doi:10.1109/VLSIC.2018.8502273.

[88] M. Tremblay, B. Joy, and K. Shin. A three dimensional register file for superscalar processors. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 1, pages 191–201 vol.1, 1995. doi:10.1109/HICSS.1995.375394.

[89] W. J. Turner, J. W. Poulton, J. M. Wilson, X. Chen, S. G. Tell, M. Fojtik, T. H. Greer, B. Zimmer, S. Song, N. Nedovic, S. S. Kudva, S. R. Sudhakaran, R. Bashirullah, W. Zhao, W. J. Dally, and C. T. Gray. Ground-referenced signaling for intra-chip and short-reach chip-to-chip interconnects. In *2018 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, 2018. doi:10.1109/CICC.2018.8357077.

[90] A. A. Vyas, C. Zhou, and C. Y. Yang. On-chip interconnect conductor materials for end-of-roadmap technology nodes. *IEEE Transactions on Nanotechnology*, 17(1):4–10, 2018. doi:10.1109/TNANO.2016.2635583.

[91] M. Wade, E. Anderson, S. Ardalan, P. Bhargava, S. Buchbinder, M. L. Davenport, J. Fini, H. Lu, C. Li, R. Meade, C. Ramamurthy, M. Rust, F. Sedgwick, V. Stojanovic, D. Van Orden, C. Zhang, C. Sun, S. Y. Shumarayev, C. O'Keeffe, T. T. Hoang, D. Kehlet, R. V. Mahajan, M. T. Guzy, A. Chan, and T. Tran. TeraPHY: A chiplet technology for low-power, high-bandwidth in-package optical I/O. *IEEE Micro*, 40(2):63–71, 2020. doi:10.1109/MM.2020.2976067.

[92] E. Wang, C. Schmidt, A. Izraelevitz, J. Wright, B. Nikolić, E. Alon, and J. Bachrach. A methodology for reusable physical design. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 243–249, 2020. doi:10.1109/ISQED48828.2020.9136999.

[93] N. C. Wang, S. Sinha, B. Cline, C. D. English, G. Yeric, and E. Pop. Replacing copper interconnects with graphene at a 7-nm node. In *2017 IEEE International Interconnect Technology Conference (IITC)*, pages 1–3, 2017. doi:10.1109/IITC-AMC.2017.7968949.

[94] S. Wang and P. Kanwar. BFloat16: The secret to high performance on cloud TPUs, 2019. URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[95] L. Wei, J. G. Alzate, U. Arslan, J. Brockman, N. Das, K. Fischer, T. Ghani, O. Golonzka, P. Hentges, R. Jahan, P. Jain, B. Lin, M. Meterelliyoz, J. ODonnell, C. Puls, P. Quintero, T. Sahu, M. Sekhar, A. Vangapaty, C. Wiegand, and F. Hamza-oglu. A 7Mb STT-MRAM in 22FFL FinFET technology with 4ns read sensing time at 0.9V using write-verify-write scheme and offset-cancellation sensing technique. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 214–216, 2019. doi:10.1109/ISSCC.2019.8662444.

[96] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):6576, April 2009. URL: https://doi.org/10.1145/1498765.1498785, doi:10.1145/1498765.1498785.

[97] John Charles Wright, Colin Schmidt, Ben Keller, Daniel Palmer Dabbelt, Jaehwa Kwak, Vighnesh Iyer, Nandish Mehta, Pi-Feng Chiu, Stevo Bailey, Krste Asanović, and Borivoje Nikolić. A dual-core RISC-V vector processor with on-chip fine-grain power management in 28-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2721–2725, 2020. doi:10.1109/TVLSI.2020.3030243.

[98] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):2024, March 1995. URL: https://doi.org/10.1145/216585.216588, doi:10.1145/216585.216588.

[99] Y. Yamada and S. Momose. Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA. In *2018 IEEE Hot Chips 30 Symposium (HCS)*, 2018.

[100] Joseph Yiu. Introduction to the arm Cortex-M55 processor, 2020. URL: https://pages.arm.com/rs/312-SAX-488/images/Arm-Custom-Instructions-for-Armv8-M.pdf.

[101] V. Zyuban and P. Kogge. The energy complexity of register files. In *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*, pages 305–310, 1998. doi:10.1145/280756.280943.