Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators

by

Christopher Francis Batten

B.S. in Electrical Engineering, University of Virginia, May 1999
M.Phil. in Engineering, University of Cambridge, August 2000

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2010

Author . . . . . .
Department of Electrical Engineering and Computer Science
January 29, 2010

Certified by . . . . . .
Krste Asanović
Associate Professor, University of California, Berkeley
Thesis Supervisor

Accepted by . . . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students
Electrical Engineering and Computer Science

Simplified Vector-Thread Architectures for Flexible and Efficient Data-Parallel Accelerators

by

Christopher Francis Batten

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2010, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis explores a new approach to building data-parallel accelerators that is based on simplifying the instruction set, microarchitecture, and programming methodology for a vector-thread architecture. The thesis begins by categorizing regular and irregular data-level parallelism (DLP), before presenting several architectural design patterns for data-parallel accelerators including the multiple-instruction multiple-data (MIMD) pattern, the vector single-instruction multiple-data (vector-SIMD) pattern, the single-instruction multiple-thread (SIMT) pattern, and the vector-thread (VT) pattern. Our recently proposed VT pattern includes many control threads that each manage their own array of microthreads. The control thread uses vector memory instructions to efficiently move data and vector fetch instructions to broadcast scalar instructions to all microthreads. These vector mechanisms are complemented by the ability for each microthread to direct its own control flow.

In this thesis, I introduce various techniques for building simplified instances of the VT pattern. I propose unifying the VT control-thread and microthread scalar instruction sets to simplify the microarchitecture and programming methodology. I propose a new single-lane VT microarchitecture based on minimal changes to the vector-SIMD pattern. Single-lane cores are simpler to implement than multi-lane cores and can achieve similar energy efficiency. This new microarchitecture uses control processor embedding to mitigate the area overhead of single-lane cores, and uses vector fragments to more efficiently handle both regular and irregular DLP as compared to previous VT architectures. I also propose an explicitly data-parallel VT programming methodology that is based on a slightly modified scalar compiler. This methodology is easier to use than assembly programming, yet simpler to implement than an automatically vectorizing compiler.

To evaluate these ideas, we have begun implementing the Maven data-parallel accelerator. This thesis compares a simplified Maven VT core to MIMD, vector-SIMD, and SIMT cores. We have implemented these cores with an ASIC methodology, and I use the resulting gate-level models to evaluate the area, performance, and energy of several compiled microbenchmarks. This work is the first detailed quantitative comparison of the VT pattern to other patterns. My results suggest that future data-parallel accelerators based on simplified VT architectures should be able to combine the energy efficiency of vector-SIMD accelerators with the flexibility of MIMD accelerators.

Thesis Supervisor: Krste Asanović
Title: Associate Professor, University of California, Berkeley

Acknowledgements

I would like to first thank my research advisor, Krste Asanović, who has been a true mentor, passionate teacher, inspirational leader, valued colleague, and professional role model throughout my time at MIT and U.C. Berkeley. This thesis would not have been possible without Krste's constant stream of ideas, encyclopedic knowledge of technical minutia, and patient yet unwavering support. I would also like to thank the rest of my thesis committee, Christopher Terman and Arvind, for their helpful feedback as I worked to crystallize the key themes of my research.

Thanks to the other members of the Scale team at MIT for helping to create the vector-thread architectural design pattern. Particular thanks to Ronny Krashinsky, who led the Scale team, and taught me more than he will probably ever know. Working with Ronny on the Scale project was, without doubt, the highlight of my time in graduate school. Thanks to Mark Hampton for having the courage to build a compiler for a brand new architecture. Thanks to the many others who made both large and small contributions to the Scale project including Steve Gerding, Jared Casper, Albert Ma, Asif Khan, Jaime Quinonez, Brian Pharris, Jeff Cohen, and Ken Barr.

Thanks to the other members of the Maven team at U.C. Berkeley for accepting me like a real Berkeley student and helping to rethink vector threading. Particular thanks to Yunsup Lee for his tremendous help in implementing the Maven architecture. Thanks to both Yunsup and Rimas Avizienis for working incredibly hard on the Maven RTL and helping to generate such detailed results. Thanks to the rest of the Maven team including Chris Celio, Alex Bishara, and Richard Xia. This thesis would still be an idea on a piece of green graph paper without all of their hard work, creativity, and dedication. Thanks also to Hidetaka Aoki for so many wonderful discussions about vector architectures. Section 1.4 discusses in more detail how the members of the Maven team contributed to this thesis.

Thanks to the members of the nanophotonic systems team at MIT and U.C. Berkeley for allowing me to explore a whole new area of research that had nothing to do with vector processors. Thanks to Vladimir Stajanović for his willingness to answer even the simplest questions about nanophotonic technology, and for being a great professional role model. Thanks to Ajay Joshi, Scott Beamer, and Yong-Jin Kwon for working with me to figure out what to do with this interesting new technology.

Thanks to my fellow graduate students at both MIT and Berkeley for making every extended intellectual debate, every late-night hacking session, and every conference trip such a wonderful experience. Particular thanks to Dave Wentzlaff, Ken Barr, Ronny Krashinsky, Edwin Olson, Mike Zhang, Jessica Tseng, Albert Ma, Mark Hampton, Seongmoo Heo, Steve Gerding, Jae Lee, Nirav Dave, Michael Pellauer, Michal Karczmarek, Bill Thies, Michael Taylor, Niko Loening, and David Liben-Nowell. Thanks to Rose Liu and Heidi Pan for supporting me as we journeyed from one coast to the other. Thanks to Mary McDavitt for being an amazing help throughout my time at MIT and even while I was in California.

Thanks to my parents, Arthur and Ann Marie, for always supporting me from my very first experiment with *Paramecium* to my very last experiment with vector threading. Thanks to my brother, Mark, for helping me to realize that life is about working hard but also playing hard. Thanks to my wife, Laura, for her unending patience, support, and love through all my ups and downs. Finally, thanks to my daughter, Fiona, for helping to put everything into perspective.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Serious technology issues are breaking down the traditional abstractions in computer architecture. Power and energy efficiency are now first-order design constraints, and the road map for standard CMOS technology has never been more challenging. At the same time, emerging data-parallel applications are growing in popularity but also in complexity, with each application requiring a mix of both regular and irregular data-level parallelism. In response to these technology and application demands, computer architects are turning to multicore and manycore processors augmented with data-parallel accelerators, where tens to hundreds of both general-purpose and data-parallel cores are integrated in a single chip.

In this thesis, I argue for a new approach to building data-parallel accelerators that is based on our recently proposed vector-thread architectural design pattern. The central theme of my thesis is that simplifying the vector-thread pattern's instruction set, microarchitecture, and programming methodology enables large arrays of vector-thread cores to be used in data-parallel accelerators. These simplified vector-thread accelerators combine the energy efficiency of single-instruction multiple-data (SIMD) accelerators with the flexibility of multiple-instruction multiple-data (MIMD) accelerators.

This chapter begins by examining the current transition to multicore and manycore processors and the emergence of data-parallel accelerators, before outlining the key contributions of the thesis and how they advance the state of the art in vector-thread architectures.

1.1 Transition to Multicore & Manycore General-Purpose Processors

Figure 1.1 shows multiple trends for a variety of high-performance desktop and server general-purpose processors from several manufactures over the past 35 years. We focus first on four metrics: the number of transistors per die, the single-thread performance as measured by the SPECint benchmarks, the processor's clock frequency, and the processor's typical power consumption. We can clearly see the effect of Moore's Law with the number of transistors per die doubling about

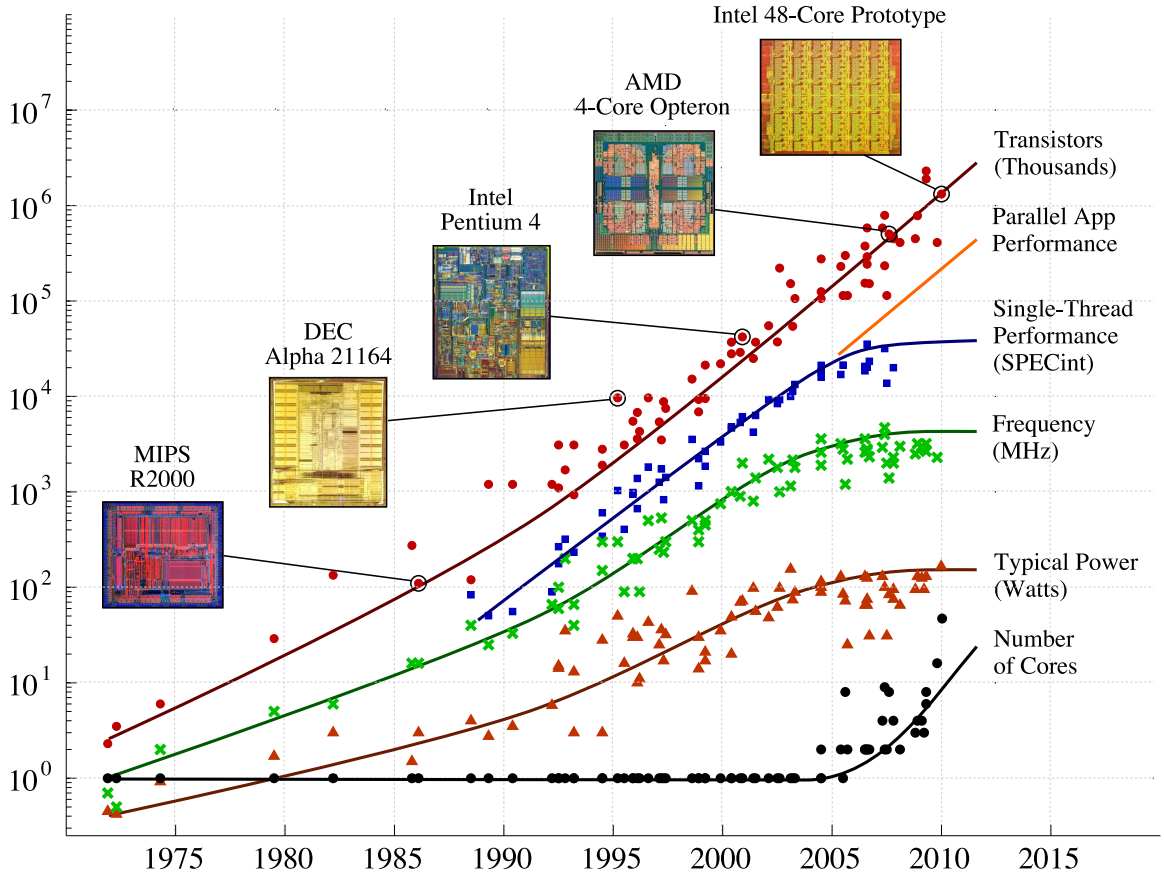


Figure 1.1: Trends in Transistors, Performance, and Power for General-Purpose Processors – Various metrics are shown for a selection of processors usually found in high-performance desktop and server systems from 1975 to 2010. For over 30 years, engineers used increased clock frequencies and power hungry architectural techniques to turn the wealth of available transistors into single-thread performance. Unfortunately, power constraints are forcing engineers to integrate multiple cores onto a single die in an attempt to continue performance scaling, albeit only for parallel applications. (Data gathered from publicly available data-sheets, press-releases, and SPECint benchmark results. Some data gathered by M. Horowitz, F. Labonte, O. Shacham, K. Olukoton, and L. Hammond of Stanford University. Single-thread performance is reported as the most recent SPECint results normalized to the performance of an Intel 80286 processor. SPECint results for many recent processors include auto-parallelization making it difficult to estimate single-thread performance. Conversion factors for different SPECint benchmark suites are developed by analyzing processors that have SPECint results for more than one suite.)

every two years. Computer architects exploited this wealth of transistors to improve single-thread performance in two ways. Architects used faster transistors and deeper pipelines to improve processor clock frequency, and they also used techniques such as caches, out-of-order execution, and superscalar issue to further improve performance.

For example, the MIPS R2000 processor released in 1985 required 110 K transistors, ran at 16 MHz, had no on-chip caches, and used a very simple five-stage pipeline [Kan87]. The DEC Alpha 21164 was released just ten years later and required 10 M transistors, ran at 300 MHz, included on-chip instruction and data caches, and used a seven-stage pipeline with four-way superscalar issue, all of which served to improve single-thread performance by almost two orders of magnitude over processors available in the previous decade [ERPR95]. The Intel Pentium 4, which was released five years later in 2000, was an aggressive attempt to exploit as much single-thread performance as possible resulting in another order-of-magnitude performance improvement. The Pentium 4 required 42 M transistors, ran at 2+ GHz, included large on-chip caches, and used a 20-stage pipeline with superscalar issue and a 126-entry reorder buffer for deep out-of-order execution [HSU+01]. These are all examples of the technology and architectural innovation that allowed single-thread performance to track Moore's Law for over thirty years. It is important to note that this performance improvement was basically transparent to software; application programmers simply waited two years and their programs would naturally run faster with relatively little work.

As illustrated in Figure 1.1, the field of computer architecture underwent a radical shift around 2005 when general-purpose single-threaded processors became significantly limited by power consumption. Typical processor power consumption has leveled off at 100–125 W, since higher power consumption requires much more expensive packaging and cooling solutions. Since power is directly proportional to frequency, processor clock frequencies have also been recently limited to a few gigahertz. Although single-threaded techniques such as even deeper pipelines and wider superscalar issue can still marginally improve performance, they do so at significantly higher power consumption. Transistor densities continue to improve, but computer architects are finding it difficult to turn these transistors into single-thread performance. Due to these factors, the industry has begun to de-emphasize single-thread performance and focus on integrating multiple cores onto a single die [OH05, ABC+06]. We have already started to see the number of cores approximately double every two years, and, if programmers can parallelize their applications, then whole application performance should once again start to track Moore's Law.

For example, the AMD 4-core Opteron processor introduced in 2007 required 463 M transistors, ran at 2+ GHz, and included four out-of-order superscalar cores in a single die [DSC+07]. Almost every general-purpose processor manufacturer is now releasing multicore processors including Sun's 8-core Niagara 2 processor [NHW+07] and 16-core Rock processor [CCE+09], Intel's 8-core Xeon processor [RTM+09], Fujitsu's 8-core SPARC64 VIIIfx processor [Mar09], and IBM's 8-core POWER7 processor [Kal09]. The current transition to multiple cores on a chip is often referred to

as the *multicore era*, and some predict we will soon enter the *manycore era* with hundreds of very simple cores integrated onto a single chip [HBK06]. Examples include the Tilera TILE64 processor with 64 cores per chip [BEA+08] and the Intel terascale prototypes with 48–80 cores per chip [VHR+07, int09].

Power consumption constrains more than just the high-performance processors in Figure 1.1. Power as well as energy have become primary design constraints across the entire spectrum of computing, from the largest data center to the smallest embedded device. Consequently, arrays of processing cores have also been adopted in laptops, handhelds, network routers, video processors, and other embedded systems. Some multicore processors for mobile clients are simply scaled down versions of high-performance processors, but at the embedded end of the spectrum, multicore-like processors have been available for many years often with fixed-function features suitable for the target application domain. Examples include ARM's 4-core Cortex-A9 MPCore processor [arm09], Raza Microelectronics' 8-core XLR network processor [Kre05], Cavium Networks' 16-core Octeon network processor [YBC+06], Cisco's Metro network router chip with 188 Tensilica Extensa cores [Eat05], and PicoChip's 430-core PC101 signal-processing array [DPT03]. The precise distinction between multicore/manycore and general-purpose/embedded is obviously gray, but there is a clear trend across the entire industry towards continued integration of more processing cores onto a single die.

Figure 1.2 illustrates in more detail *why* the move to multicore and manycore processors can help improve performance under a given power constraint. The figure qualitatively shows the performance (tasks per second) on the x-axis, the energy-efficiency (energy per task) on the y-axis, and power consumption as hyperbolic iso-power curves. Any given system will have a power constraint that maps to one of the iso-power curves and limits potential architectural design choices. For example, a processor in a high-performance workstation might have a 125 W power constraint, while a processor in a mobile client might be limited to just 1 W. Point A in the figure shows where a hypothetical simple RISC core, similar to the MIPS R2000, might lie in the energy-performance space. The rest of the space can be divided into four quadrants. Alternative architectures which lie in the upper left quadrant are probably unattractive, since they are both slower and less energy-efficient than the simple RISC core. Alternative architectures which lie in the lower left quadrant are lower performance but might also require much less energy per task. While such architectures are suitable when we desire the lowest possible energy (e.g., processors for wireless sensor networks), in this thesis I focus on architectures that balance performance and energy-efficiency. The best architectures are in the lower right quadrant with improved performance and energy-efficiency, but building such architectures can be challenging for general-purpose processors which need to handle a wide variety of application types. Instead, most general-purpose architectural approaches lie in the upper right quadrant with improved performance at the cost of increased energy per task.

For example, an architecture similar to the DEC Alpha 21164 might lie at point B if implemented

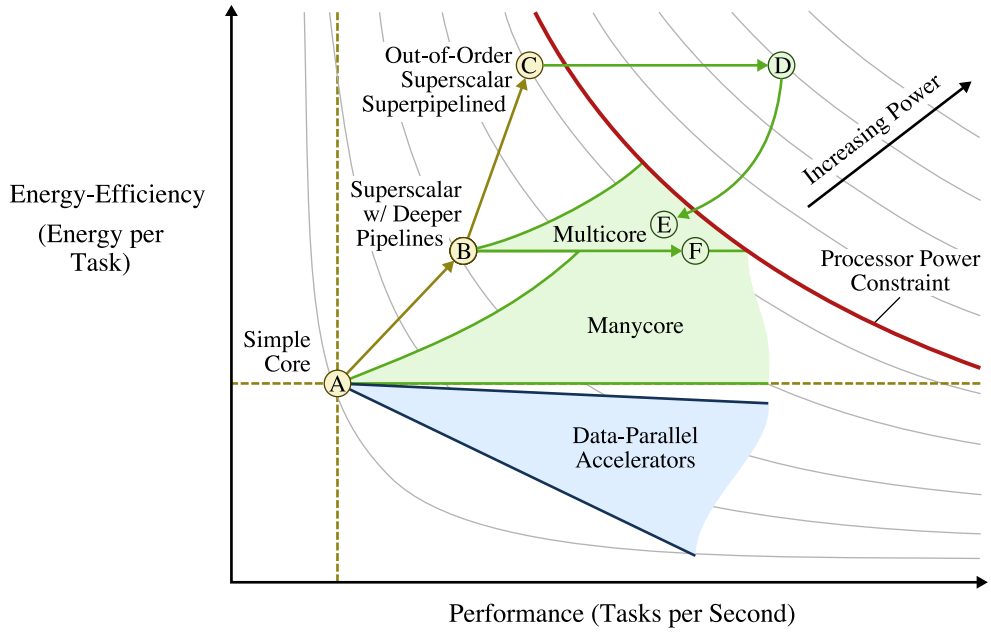


Figure 1.2: Qualitative Comparison of Multicore, Manycore, and Data-Parallel Accelerators – Out-of-order execution, superscalar issue, and superpipelining increase processor performance but at significant cost in terms of energy per task (A → B → C). Multicore processors improve performance at less energy per task by using multiple complex cores and scaling the voltage to improve energy-efficiency (C → D → E) or by using multiple simpler and more energy-efficient cores (C → B → F). Unfortunately, parallelization and energy overheads can reduce the benefit of these approaches. Data-parallel accelerators exploit the special properties of data-parallel applications to improve performance while also reducing the energy per task.

in the same technology as the simple RISC core at point A. It would have higher single-thread performance but also require more energy per task due to additional pipeline registers, branch prediction logic, unnecessary work through misspeculation, and parallel dependency checking logic. An architecture similar to the Intel Pentium 4 implemented in the same technology might lie at point C and would again improve single-thread performance at even higher energy per task due to more expensive pipelining, higher misspeculation costs, and highly associative out-of-order issue and reordering logic. Architectural techniques for single-thread performance might continue to marginally improve performance, but they do so with high energy overhead and thus increased power consumption. Although there has been extensive work analyzing and attempting to mitigate the energy overhead of complex single-thread architectures, multicore processors take a different approach by relying on multiple cores instead of continuing to build an ever more complicated single-core processor.

Figure 1.2 illustrates two techniques for improving performance at the same power consumption as the complex core at point C: multiple complex cores with voltage scaling (C → D → E) or multiple simple and more energy-efficient cores (C → B → F). The first technique instantiates multiple complicated cores as shown by the transition from point C to point D. For completely parallel

applications, the performance should improve linearly, since we can execute multiple tasks in parallel, and ideally the energy to complete each task should remain constant. Instead of spending a certain amount of energy to complete each task serially, we are simply spending the same amount of energy in parallel to finish executing the tasks sooner. Of course, as Figure 1.2 illustrates, higher performance at the same energy per task results in higher power consumption which means point D exceeds our processor power constraint. Since energy is quadratically proportional to voltage, we can use frequency and voltage scaling to turn some of the performance improvement into reduced energy per task as shown by the transition from point D to point E. Using static or dynamic voltage scaling to improve the energy-efficiency of parallel processing engines is a well understood technique which has been used in a variety of contexts [GC97, DM06, dLJ06, LK09]. A second technique, which achieves a similar result, reverts to simpler cores that are both lower energy and lower performance, and then compensates for the lower performance by integrating multiple cores onto a single chip. This is shown in Figure 1.2 as the transition from point C to point B to point F. Comparing point C to point E and F we see that multicore processors can improve performance at less energy per task under a similar power constraint. Note that we can combine voltage scaling and simpler core designs for even greater energy-efficiency. Manycore processors are based on a similar premise as multicore processors, except they integrate even more cores onto a single die and thus require greater voltage scaling or very simple core designs to still meet the desired power constraint.

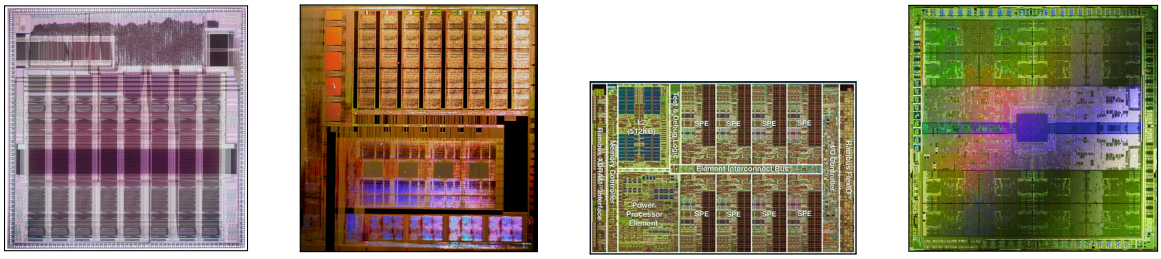
The multicore approach fundamentally depends on resolving two key challenges: the *parallel programming challenge* and the *parallel energy-efficiency challenge*. The first challenge, parallel programming, has been well-known for decades in the high-performance scientific community, but now that parallel programming is moving into the mainstream it takes on added importance. Applications that are difficult to parallelize will result in poor performance on multicore processors. Even applications which are simple to parallelize might still be difficult to analyze and thus tune for greater performance. The second challenge, parallel energy-efficiency, is simply that it is difficult to make each individual core significantly more energy-efficient than a single monolithic core. Voltage scaling without drastic loss in performance is becoming increasingly difficult in deep sub-micron technologies due to process variation and subthreshold leakage at low voltages [GGH97, FS00, CC06]. This means instantiating many complex cores is unlikely to result in both an energy-efficient and high-performance solution. While simpler cores can be more energy-efficient, they can only become so simple before the loss in performance starts to outweigh reduced power consumption. Finally, even with energy-efficient cores, there will inevitably be some energy overhead due to multicore management, cache coherence, and/or direct communication costs. Overall, the parallel programming challenge and the parallel energy-efficiency challenge mean that points above and to the left of the ideal multicore architectures represented by points E and F more accurately reflect real-world designs. In the next section, we discuss tackling these challenges through the use of accelerators specialized for efficiently executing data-parallel applications.

1.2 Emergence of Programmable Data-Parallel Accelerators

Data-parallel applications have long been the dominant form of parallel computing, due to the ubiquity of data-parallel kernels in demanding computational workloads. A data-parallel application is one in which the most straightforward parallelization approach is to simply partition the input dataset and apply a similar operation across all partitions. Conventional wisdom holds that highly data-parallel applications are limited to the domain of high-performance scientific computing. Examples include molecular dynamics, climate modeling, computational fluid dynamics, and generic large-scale linear-algebra routines. However, an honest look at the current mainstream application landscape reveals a wealth of data-parallelism in such applications as graphics rendering, computer vision, medical imaging, video and audio processing, speech and natural language understanding, game physics simulation, encryption, compression, and network processing. Although all of these data-level parallel (DLP) applications contain some number of similar tasks which operate on independent pieces of data, there can still be a significant amount of diversity. On one end of the spectrum is *regular data-level parallelism*, characterized by well-structured data-access patterns and few data-dependent conditionals meaning that all tasks are very similar. In the middle of the spectrum is *irregular data-level parallelism*, characterized by less structured data-access patterns and some number of (possibly nested) data-dependent conditionals meaning that the tasks are less similar. Eventually an application might become so irregular that the tasks are performing completely different operations from each other with highly unstructured data-access patterns and significant dependencies between tasks. These applications essentially lie on the opposite end of the spectrum from regular DLP and are better characterized as *task-level parallel* (TLP).

DLP applications can, of course, be mapped to the general-purpose multicore processors discussed in the previous section. In fact, the distinguishing characteristics of DLP applications (many similar independent tasks) make them relatively straightforward to map to such architectures and thus help mitigate the parallel programming challenge. Unfortunately, general-purpose multicore processors have no special mechanisms for efficiently executing DLP applications. Such processors execute DLP and TLP applications in the same way, meaning that we can expect both an increase in performance *and* increase in energy per task compared to a single simple core regardless of the type of parallelism. Given the large amount of data-level parallelism in modern workloads and the energy overheads involved in multicore architectures, it's not difficult to see that we will soon be once again seriously limited by system power constraints.

The desire to exploit DLP to improve both performance and energy-efficiency has given rise to *data-parallel accelerators* which contain an array of programmable data-parallel cores. In this thesis, I loosely define a data-parallel accelerator to be a coprocessor which is meant to augment a general-purpose multicore processor, and which contains dedicated mechanisms well suited to executing DLP applications. We should expect data-parallel accelerators to lie in the lower right quadrant of Figure 1.2 with higher performance and lower energy per task as compared to a single



(a) UC Berkeley T0 Vector Processor

(b) Stanford Imagine Stream Processor

(c) IBM Cell General Accelerator

(d) NVIDIA Fermi Graphics Processor

Figure 1.3: Examples of Data-Parallel Accelerators – Demands for higher performance and/or lower energy per task on data-parallel applications has lead to the emergence of data-parallel accelerators from uni-processor vector and stream processors, to more general SIMD accelerators, to general-purpose graphic processors with hundreds of data-parallel engines on a single chip.

general-purpose core. This is especially true for regular DLP applications, but probably less so for irregular DLP applications. Even so, an accelerator which can flexibly execute a wider array of DLP applications will be more attractive than one which can only execute regular DLP applications. Ideally a system will combine both general-purpose cores and data-parallel cores so that the best energy-performance trade-off can be achieved for any given application [WL08].

Figure 1.3 illustrates a few representative data-parallel accelerators. The Berkeley Spert-II system included the T0 vector processor with an 8-lane vector unit and was used as an accelerator for neural network, multimedia, and digital signal processing [WAK+96, Asa98]. The Stanford Imagine stream processor was a research prototype which included an 8-lane vector-like processing unit and was evaluated for a variety of data-parallel kernels [RDK+98]. These early examples contain a single tightly coupled data-parallel accelerator, while the more recent IBM Cell processor contains an array of eight simple subword-SIMD processors controlled by a single general-purpose processor integrated on the same chip [GHF+06]. Perhaps the most wide-spread example of data-parallel accelerators are the graphics processors found in most mainstream workstations and mobile clients. NVIDIA's CUDA framework [NBGS08] and AMD/ATI's Close-to-the-Metal initiative [ati06] enable the many vector-like cores on a graphics processor to be used as a programmable data-parallel accelerator. For example, the recently announced Fermi graphics processor from NVIDIA includes 32 data-parallel cores each with a flexible 16-lane vector-like engine suitable for graphics as well as more general data-parallel applications [nvi09]. To help us reason about these different approaches and motivate the rest of my work, Chapter 2 will introduce a set of *architectural design patterns* which capture the key characteristics of various data-parallel accelerators used both in industry and research.

1.3 Leveraging Vector-Threading in Data-Parallel Accelerators

In this thesis, I propose a new approach to building data-parallel accelerators that is based on the *vector-thread architectural design pattern*. Vector-threading (VT) includes a *control thread* that manages a vector of *microthreads*. The control thread uses *vector memory instructions* to efficiently move blocks of data between memory and each microthread's registers, and *vector fetch instructions* to broadcast scalar instructions to all microthreads. These vector mechanisms are complemented by the ability for each microthread to direct its own control flow when necessary. This logical view is implemented by mapping the microthreads both spatially and temporally to a set of vector lanes contained within a vector-thread unit (VTU). A seamless intermixing of vector and threaded mechanisms allows VT to potentially combine the energy efficiency of SIMD accelerators with the flexibility of MIMD accelerators.

The Scale VT processor is our first implementation of these ideas specifically targeted for use in embedded devices. Scale uses a generic RISC instruction set architecture for the control thread and a microthread instruction set specialized for VT. The Scale microarchitecture includes a simple RISC control processor and a complicated (but efficient) four-lane VTU. The Scale programming methodology requires either a combination of compiled code for the control processor and assembly programming for the microthreads or a preliminary vectorizing compiler specifically written for Scale. For more information, see our previous publications on the Scale architecture [KBH+04a, KBH+04b], memory system [BKGA04], VLSI implementation [KBA08], and compiler [HA08].

Based on our experiences designing, implementing, and evaluating Scale, I have identified three primary directions for improvement to simplify both the hardware and software aspects of the VT architectural design pattern. These improvements include a unified VT instruction set architecture, a single-lane VT microarchitecture based on the vector-SIMD pattern, and an explicitly data-parallel VT programming methodology, and they collectively form the key contribution of this thesis. To evaluate these ideas we have begun implementing the Maven data-parallel accelerator, which includes a malleable array of tens to hundreds of vector-thread engines. These directions for improvement are briefly outlined below and discussed in more detail in Chapter 3.

- **Unified VT Instruction Set Architecture –** I propose unifying the VT control-thread and microthread scalar instruction sets such that both types of thread execute a very similar generic RISC instruction set. This simplifies the microarchitecture and the programming methodology, as compared to the specialized microthread instruction set used in Scale. Chapter 4 will discuss the Maven instruction set in general as well as the specific challenges encountered with a unified instruction set.
- **Single-Lane VT Microarchitecture Based on the Vector-SIMD Pattern –** I propose a new approach to implementing VT instruction sets that closely follows a simple traditional vector-SIMD microarchitecture as much as possible. This is in contrast to the more complex mi-

croarchitecture used in Scale. I also propose using single-lane VTUs as opposed to Scale's multi-lane VTU or the the multi-lane vector units common in other data-parallel accelerators. A vector-SIMD-based single-lane VT microarchitecture is simpler to implement and can have competitive energy-efficiency as compared to more complicated multi-lane VTUs. Chapter 5 will describe the basic Maven microarchitecture before introducing techniques that reduce the area overhead of single-lane VTUs and improve efficiency when executing irregular DLP.

- **Explicitly Data-Parallel VT Programming Methodology –** I propose an explicitly data-parallel programming methodology suitable for VT data-parallel accelerators that combines a slightly modified C++ scalar compiler with a carefully written support library. The result is a clean programming model that is considerably easier than assembly programming, yet simpler to implement as compared to an automatic vectorizing compiler, such as the Scale compiler. Chapter 6 will explain the required compiler changes and the support library's implementation, as well as show several examples to illustrate the Maven programming methodology.

The unifying theme across these directions for improvement is a desire to simplify all aspects of the VT architectural design pattern, yet still preserve the primary advantages. To evaluate our ideas, we have implemented Maven as well as various other data-parallel cores using a semi-custom ASIC methodology in a TSMC 65 nm process. Chapter 7 will use the resulting placed-and-routed gate-level models to evaluate the area, performance, and energy of several compiled microbenchmarks. This work is the first detailed quantitative evaluation of the VT design pattern compared to other design patterns for data-parallel accelerators.

1.4 Collaboration, Previous Publications, and Funding

As with all large systems research projects, this thesis describes work which was performed as part of a group effort. Although I was personally involved at all levels, from gate-level design to compiler optimizations, the ideas and results described within this thesis simply would not have been possible without collaborating with my colleagues and, of course, the integral guidance of our research adviser, Krste Asanović.

The initial VT architectural design pattern was developed by Krste Asanović, Ronny Krashinsky, and myself while at the Massachusetts Institute of Technology from 2002 through 2008, and we designed and fabricated the Scale VT processor to help evaluate the VT concept. Ronny Krashinsky was the lead architect for Scale, and I worked with him on the Scale architecture and the overall direction of the project. I was responsible for the Scale memory system which included developing new techniques for integrating vector processors into cache hierarchies, designing and implementing a microarchitectural simulator for the non-blocking, highly-associative cache used in our early studies, and finally implementing the RTL for the memory system used in the fabricated Scale prototype. Much of the work in this thesis evolved from my experiences working on the Scale project.

The Maven VT core was developed by myself and a group of students during my time as a visiting student at the University of California, Berkeley from 2007 through 2009. I was the lead architect for the Maven project and helped direct the development and evaluation of the architecture, microarchitecture, RTL implementation, compiler, and applications. I was directly responsible for the Maven instruction set architecture, the Maven C++ compiler and programming support libraries, the implementation of several benchmarks, the development of the Maven assembly test suite, and the detailed paper design of the microarchitecture for the Maven single-lane core. Hidetaka Aoki made significant contributions to some of our very early ideas about single-lane VTUs. Yunsup Lee played a very large role in the Maven project including working on the initial C++ compiler port to a modified MIPS back-end, developing the Maven functional simulator, and setting up the CAD tool-flow. Most importantly, Yunsup took the lead in writing the RTL for the Maven VTU. Rimas Avizienis wrote the Maven proxy kernel and implemented the RTL for both the Maven control processor and the multithreaded MIMD processor used in our evaluation. Chris Celio wrote the RTL for the Maven vector memory unit. The benchmarks used to evaluate Maven were written by myself, Chris Celio, Alex Bishara, and Richard Xia. Alex took the lead in implementing the graphics rendering application for Maven (discussed in Section 7.7, and also worked on augmenting the functional simulator with a simple performance model.

The majority of the work in this thesis is previously unpublished. Some of the initial vision for the Maven processor was published in a position paper co-authored by myself and others entitled "The Case for Malleable Stream Architectures" from the *Workshop on Streaming Systems, 2008* [BAA08].

This work was funded in part by DARPA PAC/C Award F30602-00-20562, NSF CAREER Award CCR-0093354, the Cambridge-MIT Institute, an NSF Graduate Research Fellowship, and donations from Infineon Corporation and Intel Corporation.

Chapter 2

Architectural Design Patterns for Data-Parallel Accelerators

This chapter will introduce a framework for reasoning about the diversity in both data-parallel applications and data-parallel accelerators. The chapter begins by illustrating in more detail the differences between regular and irregular data-level parallelism (Section 2.1), and then discusses the overall structure of data-parallel accelerators found in both industry and academia (Section 2.2). Although the memory system and inter-core network play a large role in such accelerators, what really distinguishes a data-parallel accelerator from its general-purpose counterpart is its highly-optimized data-parallel execution cores. Sections 2.3–2.7 present five *architectural design patterns* for these data-parallel cores: multiple-instruction multiple-data (MIMD), vector single-instruction multiple-data (vector-SIMD), subword single-instruction multiple-data (subword-SIMD), single-instruction multiple-thread (SIMT), and vector-thread (VT). An architectural design pattern captures the key features of the instruction set, microarchitecture, and programming methodology for a specific style of architecture. The design patterns are abstract enough to enable many different implementations, but detailed enough to capture the distinguishing characteristics. Each pattern includes example mappings of two simple yet representative loops reflecting regular and irregular data-level parallelism, and also discusses an accelerator which exemplifies the pattern. The chapter concludes with a comparison of the patterns (Section 2.8), and a discussion of how the five patterns can help categorize various accelerators in industry and academia (Section 2.9).

2.1 Regular and Irregular Data-Level Parallelism

Different types of data-level parallelism (DLP) can be categorized in two dimensions: the regularity with which data memory is accessed and the regularity with the control flow changes. *Regular data-level parallelism* has well structured data accesses where the addresses can be compactly encoded and are known well in advance of when the data is ready. Regular data-level parallelism also

(a)	**Regular Data Access & Regular Control Flow** Unit-stride loads (`A[i]`,`B[i]`) and store (`C[i]`)	`for ( i = 0; i < n; i++ )` `  C[i] = A[i] + B[i];`
(b)	**Regular Data Access & Regular Control Flow** Strided load (`A[2*i]`) and store (`C[2*i]`)	`for ( i = 0; i < n; i++ )` `  C[2*i] = A[2*i] + A[2*i+1];`
(c)	**Regular Data Access & Regular Control Flow** Shared load (`x`)	`for ( i = 0; i < n; i++ )` `  C[i] = x * A[i] + B[i];`
(d)	**Irregular Data Access & Regular Control Flow** Indexed load (`D[A[i]]`) and store (`E[C[i]]`)	`for ( i = 0; i < n; i++ )` `  E[C[i]] = D[A[i]] + B[i];`

(e)	**Regular Data Access & Irregular Control Flow**
	Unit-stride loads/store with data-dependent if-then-else conditional
		`for ( i = 0; i < n; i++ )`
		`  x = ( A[i] > 0 ) ? y : z;`
		`  C[i] = x * A[i] + B[i];`

(f)	**Irregular Data Access & Irregular Control Flow**
	Conditional load (`B[i]`), store (`C[i]`), and computation (`x * A[i] + B[i]`)
		`for ( i = 0; i < n; i++ )`
		`  if ( A[i] > 0 )`
		`  C[i] = x * A[i] + B[i];`

(g)	**Transforming Irregular Control Flow**
	Previous example split into irregular compress loop and then shorter loop with regular control flow and both regular and irregular data access
		`for ( m = 0, i = 0; i < n; i++ )`
		`  if ( A[i] > 0 )`
		`  D[m] = A[i]; E[m] = B[i];`
		`  I[m++] = i;`
		`for ( i = 0; i < m; i++ )`
		`  C[I[i]] = x * D[i] + E[i];`

(h)	**Irregular Data Access & Irregular Control Flow**
	Inner loop with data-dependent number of iterations (`B[i]`) and pointer dereference (`*A[i]`)
		`for ( i = 0; i < n; i++ )`
		`  C[i] = 0;`
		`  for ( j = 0; j < B[i]; j++ )`
		`  C[i] += (*A[i])[j];`

(i)	**Irregular Data Access & Irregular Control Flow**
	Inner loop with complex data-dependent exit condition, nested control flow, and conditional loads/stores
		`for ( i = 0; i < n; i++ )`
		`  C[i] = false; j = 0;`
		`  while ( !C[i] & (j < m) )`
		`  if ( A[i] == B[j++] )`
		`  C[i] = true;`

(j)	**Inter-Task Dependency Mixed with DLP** Cross-iteration dependency (`C[i] = C[i-1]`)	`for ( C[0] = 0, i = 1; i < n; i++ )` `  C[i] = C[i-1] + A[i] * B[i];`
(k)	**Task-Level Parallelism Mixed with DLP** Call through function pointer deference (`*A[i]`)	`for ( i = 0; i < n; i++ )` `  (*A[i])( B[i] );`

Table 2.1: Different Types of Data-Level Parallelism – Examples expressed in a C-like pseudocode and are ordered from regular data-level parallelism at the top to irregular data-level parallelism at the bottom.

has well structured control flow where the control decisions are either known statically or well in advance of when the control flow actually occurs. *Irregular data-level parallelism* might have less structured data accesses where the addresses are more dynamic and difficult to predict, and might also have less structured control flow with data-dependent control decisions. Irregular DLP might also include a small number of inter-task dependencies that force a portion of each task to wait for previous tasks to finish. Eventually a DLP kernel might become so irregular that it is better categorized as exhibiting task-level parallelism. Table 2.1 contains many simple loops which illustrate the spectrum from regular to irregular DLP.

Table 2.1a is a simple vector-vector addition loop where elements of two arrays are summed and written to a third array. Although the number of loop iterations (i.e., the size of the arrays) is often not known until run-time, it does not depend on the array data and still results in a very regular execution. These kind of data accesses are called *unit-stride accesses* meaning that the array elements are accessed consecutively. This loop, as well as all the loops in Table 2.1, can be mapped to a data-parallel accelerator by considering each loop iteration as a separate data-parallel task. This loop represents the most basic example of regular DLP: elements are accessed in a very structured way and all of the iterations perform identical operations. Although simple, many data-parallel applications contain a great deal of code which is primarily unit-stride accesses with regular control flow so we should expect data-parallel accelerators to do very well on these kind of loops.

Table 2.1b–c illustrates loops which are also well structured but have slightly more complicated data accesses. In Table 2.1b, each iteration accesses every other element in the `A` and `C` arrays. These kind of accesses are called *strided accesses* and in this example the stride is two. Although still well structured, because strided accesses read or write non-consecutive elements they can result in execution inefficiencies especially for power-of-two or very large strides. Notice that each loop iteration loads two consecutive elements of the array `A`. These make up a special class of strided accesses called *segment accesses*, and they are common when working with arrays of structures or objects [BKGA04]. For example, the loop iteration might be accessing the real and imaginary parts of a complex number stored in an array of two-element structures. Table 2.1c is a common loop in linear algebra where we multiply each element in the array `A` by the scalar value `x` and add the result to the corresponding element in the array `B`. Unlike the previous examples, this loop requires a scalar value to be distributed across all iterations. These kind of accesses are sometimes called *shared accesses*. Although not as simple as unit-stride, data-parallel accelerators should still be able to take advantage of strided, segment, and shared accesses for improved performance and efficiency.

Table 2.1d illustrates a loop with irregular data accesses but with regular control flow. Elements from one array are used to index into a different array. For example, while the array `A` is accessed with unit stride, the order in which the array `D` is accessed is not known until run-time and is data dependent. These kind of accesses are often called *indexed accesses* although they are also called gather accesses (for loads) and scatter accesses (for stores). Many data-parallel applications have a

mix of both regular and irregular data accesses, so effective data-parallel accelerators should be able handle many different ways of accessing data. It is important to note, that even if the accelerator does no better than a general-purpose multicore processor with respect to irregular data accesses, keeping the data loaded on the accelerator enables more efficient execution of the regular portions of the application.

Table 2.1e–f illustrates loops with irregular control flow. In Table 2.1e, an `if` statement is used to choose which scalar constant should be multiplied by each element of the array `A`. Although the conditional is data-dependent so that each task might use a different constant, the execution of each task will still be very similar. These type of small conditionals lend themselves to some form of conditional execution such as predication or a conditional move instruction to avoid branch overheads. Table 2.1f illustrates a more complicated conditional where the actual amount of work done by each task might vary. Some tasks will need to complete the multiplication and addition, while other tasks will skip this work. Notice that the data-dependent control flow also creates irregular data accesses to the `B` and `C` arrays. While the conditional store to `C` is essential for correctness, it might be acceptable to load all elements of `B` even though only a subset will actually be used. In general-purpose code, speculatively loading an element might cause a protection fault, but this is less of a concern for data-parallel accelerators which often have very coarse-grain protection facilities. However, using conditional execution for the entire loop is less attractive, since it can result in significant wasted work. Data-parallel accelerators which can handle irregular control flow are able to execute a wider variety of data-parallel applications, and as with irregular data access, even if this does not significantly improve the irregular portions of an application it can enable much more efficient execution of the regular portions.

Table 2.1g illustrates a technique for transforming irregular DLP into regular DLP. The first irregular loop compresses only those elements for which the condition is true into three temporary arrays. The second loop is then completely regular and iterates over just those elements for which the condition is true. This kind of transformation is often called *compression*, and the dual transformation where elements are moved from consecutive indices to sparse indices is called *expansion*. A data-parallel accelerator which has no mechanisms for irregular control flow might need to execute the first loop on the general-purpose processor before being able to execute the second loop on the data-parallel accelerator. Accelerators for which irregular control flow is possible but inefficient might also use this type of transformation, and might even provide special vector compress and expand instructions to facilitate these transformations. There is a fundamental tension between the time and energy required to perform the transformation and the efficiency of handling irregular control flow. Data-parallel accelerators which can efficiently execute irregular control flow can avoid these types of transformations and execute the control flow directly.

Table 2.1h–i illustrates more complicated irregular control flow with nested loops. These loops can be mapped to a data-parallel accelerator in one of two ways: *outer loop parallelization* where

each task works on a specific iteration of the outer loop, or *inner loop parallelization* where each task works on a specific iteration of the inner loop. In Table 2.1h, the outer loop contains an inner loop with a data-dependent number of iterations. Each element of the array `A` is a pointer to a separate subarray. The number of elements in each of these subarrays is stored in the array `B`. So the inner loop sums all elements in a subarray, and the outer loop iterates over all subarrays. In Table 2.1i, each iteration of the outer loop searches the array `B` for a different value (`A[i]`) and writes whether or not this value was found to the output array `C`. In both examples, all tasks are performing similar operations (accumulating subarray elements or searching for a value), but the irregular control flow means that different tasks can have very different amounts of work to do and/or be executing similar operations but at different points in time. Both examples also contain irregular data accesses. This kind of irregular DLP can be challenging for some data-parallel accelerators, but note that both examples still include a small number of regular data accesses and, depending on the data-dependent conditionals, there might actually be quite a bit of control flow regularity making it worthwhile to map these loops to a data-parallel accelerator.

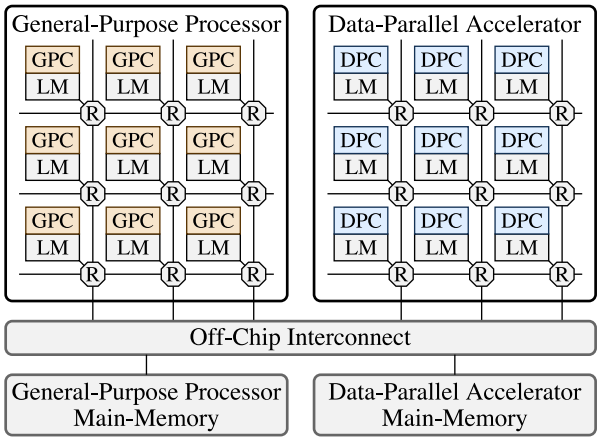
The loops in Table 2.1j–k are at the opposite end of the spectrum from regular data-level parallelism. The loop in Table 2.1j contains a cross-iteration dependency, since each iteration uses the result of the previous iteration. Even though this results in inter-task dependencies, there is still regular DLP available in this example. We can use regular unit-stride accesses to read the `A` and `B` arrays, and all tasks can complete the multiplication in parallel. In Table 2.1k, each iteration of the loop calls an arbitrary function through a function pointer. Although all of the tasks could be doing completely different operations, we can still use a regular data access to load the array `B`. It might be more reasonable to categorize such loops as task-level parallel or pipeline parallel where it is easier to reason about the different work each task is performing as opposed to focusing on partitioning the input dataset. (See [MSM05] for an overview of data-level parallelism, task-level parallelism, pipeline parallelism, as well as other parallel programming patterns.)

Clearly there is quite a variety in the types of DLP, and there have been several studies which demonstrate that full DLP applications contain a mix of regular and irregular DLP [SKMB03, KBH+04a, RSOK06, MJCP08]. Accelerators which can handle a wider variety of DLP are more attractive than those which are restricted to just regular DLP for many reasons. First, it is possible to improve performance and energy-efficiency even on irregular DLP. Second, even if the performance and energy-efficiency on irregular DLP is similar to a general-purpose processor, by keeping the work on the accelerator we make it easier to exploit regular DLP inter-mingled with irregular DLP. Finally, a consistent way of mapping both regular and irregular DLP simplifies the programming methodology.

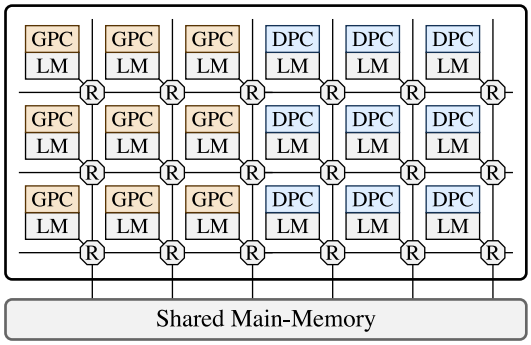
2.2 Overall Structure of Data-Parallel Accelerators

Data-parallel accelerators augment a general-purpose processor and are specialized for executing the types of data-parallelism introduced in the previous section. The general-purpose processor distributes data-parallel work to the accelerator, executes unstructured code not suitable for the accelerator, and manages system-level functions. Ideally, data-parallel code should have improved performance and energy-efficiency when executing on the accelerator as compared to the same code running on the general-purpose processor. Figure 2.1 shows various approaches for combining a general-purpose processor with a data-parallel accelerator.

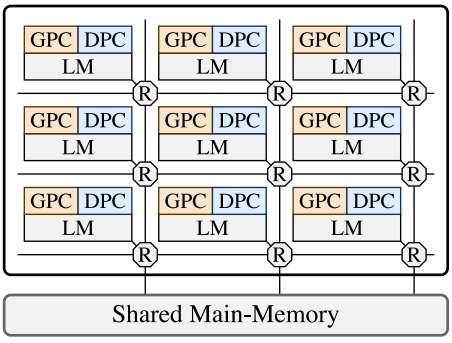
Figure 2.1a shows a two-chip solution where the accelerator is a discrete component connected to the general-purpose processor through an off-chip interconnect. In this organization, both the general-purpose processor and the accelerator can share a common pool of main-memory or possibly include dedicated memory for each chip. Both the Berkeley T0 vector processor and the Stanford Imagine stream processor shown in Figures 1.3a and 1.3b are discrete accelerators that serve as coprocessors for a general-purpose processor. These examples include a single data-parallel core, but more recent accelerators use a tiled approach to integrate tens of cores on a single chip. In



(a) Discrete Accelerator



(b) Partially Integrated Accelerator



(c) Fully Integrated Accelerator

Figure 2.1: General-Purpose Processor Augmented with a Data-Parallel Accelerator – (a) discrete accelerators might have their own main-memory and communicate with the general-purpose processor via an off-chip interconnect, (b) partially integrated accelerators share main-memory and communicate with the general-purpose processor via an on-chip interconnect, (c) fully integrated accelerators tightly couple a general-purpose and data-parallel core into a single tile (GPC = general-purpose core, DPC = data-parallel core, LM = local memory, R = on-chip router)

Figure 2.1a a tile is comprised of one or more cores, some amount of local memory, and an on-chip network router which are then instantiated across both the general-purpose processor and the accelerator in a very regular manner. The NVIDIA Fermi architecture shown in Figure 1.3d includes 32 cores tiled across the discrete accelerator.

Unfortunately, discrete accelerators suffer from large startup overheads due to the latency and bandwidth constraints of moving data between the general-purpose processor and the accelerator. The overheads inherent in discrete accelerators have motivated a trend towards integrated accelerators that can more efficiently exploit finer-grain DLP. Figure 2.1b shows an integration strategy where the data-parallel cores are still kept largely separate from the general-purpose cores. An early example, shown in Figure 1.3c, is the IBM Cell processor, which integrated a general-purpose processor with eight data-parallel cores on the same die [GHF+06]. Figure 2.1c shows an even tighter integration strategy where a general-purpose and data-parallel core are combined into a single tile. Industry leaders have indicated their interest in this approach, and it seems likely that fully integrated accelerators will be available sometime this decade [HBK06, amd08].

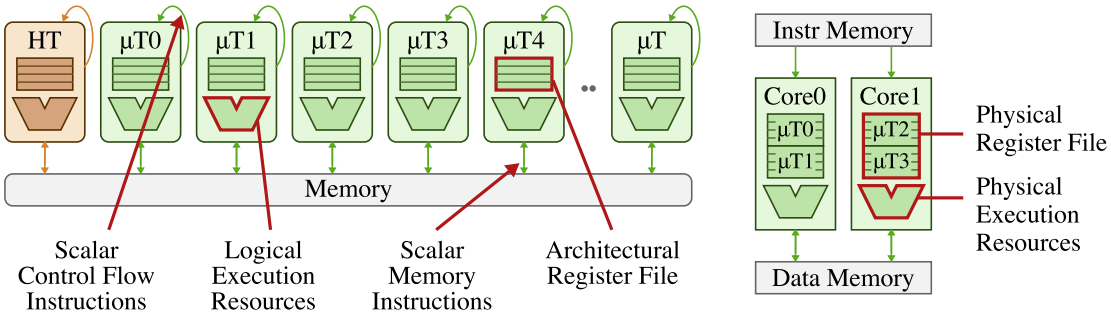
It is important to distinguish the data-parallel accelerators described so far from previous large-scale multi-node data-parallel machines. Some of these machines are similar in spirit to the patterns presented later in this chapter including multithreaded machines such as the Denelcor HEP machine and Cray MTA machine, and vector machines such as those offered by Cray, CDC, Convex, NEC, Hitachi, and Fujitsu. (See [URv03] for a survey of multithreaded machines and [EVS98, Oya99, HP07, Appendix F.10] for a survey of vector machines.) These machines are expensive supercomputers primarily intended for high-performance scientific computing. They are designed with less emphasis on reducing the energy per task, and more focus on achieving the absolute highest performance possible. Programming such machines can often be cumbersome, especially for irregular DLP on vector supercomputers. Data-parallel accelerators, on the other hand, must be low cost and thus are implemented either on a single-chip or tightly integrated with a general-purpose processor. They must be easy to program and flexibly handle a diverse range of DLP applications with high energy-efficiency. We can learn much from multi-node data-parallel machines, and they inspire some of the architectural design patterns described in this chapter. Data-parallel accelerators, however, offer a new set of opportunities and challenges.

Although the communication mechanism between the general-purpose processor and accelerator, the intra-accelerator network, and the accelerator memory system are all critical aspects of an effective implementation, in this thesis I focus on the actual data-parallel cores. These specialized cores are what distinguish accelerators from general-purpose processors and are the key to the accelerator's performance and efficiency advantages. Just as there are many types of data-level parallelism, there are many types of data-parallel cores; each with its own strengths and weaknesses. To help understand this design space, the rest of this chapter presents five architectural patterns for the design of data-parallel cores.

2.3 MIMD Architectural Design Pattern

The multiple-instruction multiple-data (MIMD) pattern is perhaps the simplest approach to building a data-parallel accelerator. A large number of scalar cores are replicated across a single chip. Programmers can map each data-parallel task to a separate core, but without any dedicated DLP mechanisms, it is difficult to gain an energy-efficiency advantage when executing DLP applications. These scalar cores can be extended to support per-core multithreading which helps improve performance by hiding various control flow, functional unit, and memory latencies.

Figure 2.2 shows the programmer's logical view and an example implementation for the multithreaded MIMD pattern. I assume that all of the design patterns in this chapter include a *host thread* as part of the programmer's logical view. The host thread runs on the general-purpose processor and is responsible for application startup, configuration, interaction with the operating system, and managing the data-parallel accelerator. I refer to the threads which run on the data-parallel accelerator as *microthreads*, since they are lighter weight than the threads which run on the general-purpose processor. Microthreads are under complete control of the user-level application and may have significant limitations on what kind of instructions they can execute. Although a host thread may be virtualized by the operating system, microthreads are never virtualized meaning there is a one-to-one correspondence between microthreads and hardware thread contexts in the data-parallel accelerator. In this pattern, the host thread communicates with the microthreads through shared memory. The primary advantage of the MIMD pattern is the flexible programming model, and since every core can execute a fully independent task, there should be little difficulty in mapping both regular and irregular DLP applications. This simplifies the parallel programming challenge compared to the other design patterns, but the primary disadvantage is that this pattern does little to address the parallel energy-efficiency challenge.



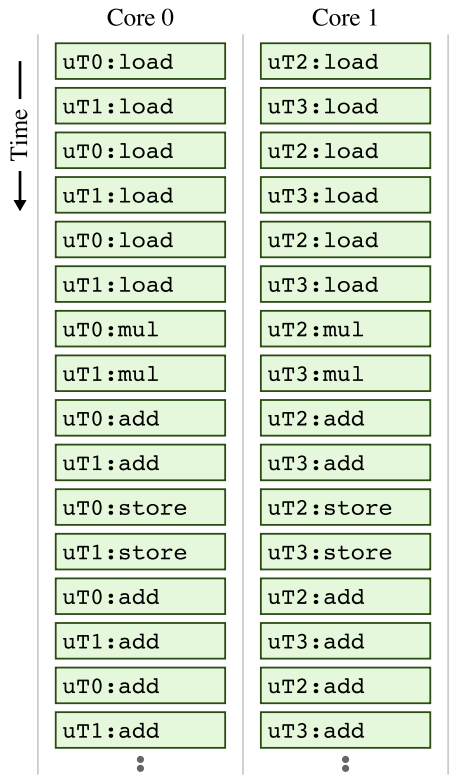
(a) Programmer's Logical View

(b) Implementation of 2 Cores Each With 2-Microthreads

Figure 2.2: MIMD Architectural Design Pattern – A general-purpose host thread uses shared memory to manage an array of microthreads. Each scalar microthread works on a different partition of the input dataset and is responsible for its own data access and control flow. Microthreads correspond to physical hardware contexts which are distributed across cores in the implementation. (HT = host thread, µT = microthread)

```
1  div    m, n, nthreads
2  mul    t, m, tidx
3  add    a_ptr, t
4  add    b_ptr, t
5  add    c_ptr, t
6
7  sub    t, nthreads, 1
8  br.neq t, tidx, notlast
9  rem    m, n, nthreads
10 notlast:
11
12 load   x, x_ptr
13
14 loop:
15 load   a, a_ptr
16 load   b, b_ptr
17 mul    t, x, a
18 add    c, t, b
19 store  c, c_ptr
20
21 add    a_ptr, 1
22 add    b_ptr, 1
23 add    c_ptr, 1
24
25 sub    m, 1
26 br.neq m, 0, loop
```

(a) Pseudo-Assembly

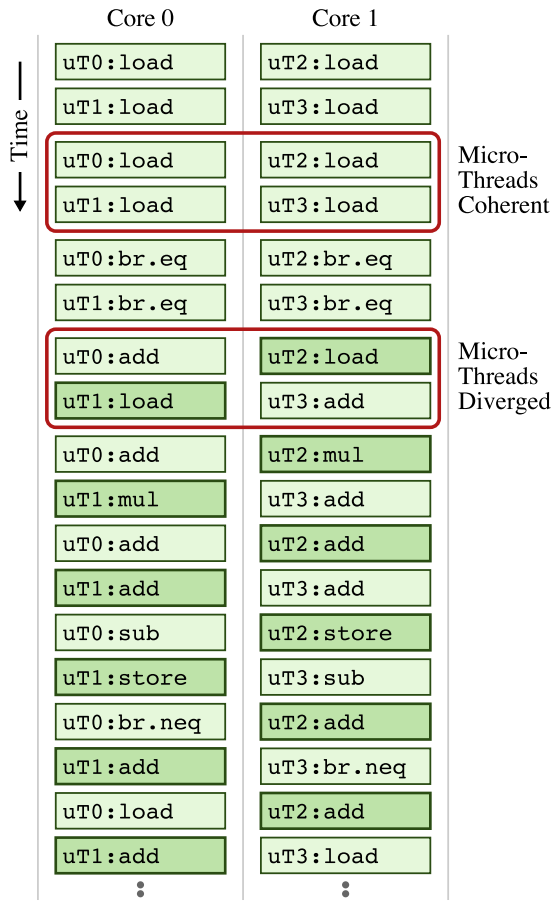


(b) Execution Diagram for 2-Core, 4-Microthread Implementation

Figure 2.3: Mapping Regular DLP to the MIMD Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1c. Execution diagram starts with load at line 12. Since there are no explicit mechanisms for rgular DLP exposed as part of a MIMD instruction set, DLP codes are always encoded as a set of scalar operations. For regular DLP, these operations execute in lock step but with little improvement in energy efficiency. (Assume `*_ptr` and `n` are inputs. `nthreads` = total number of microthreads, `tidx` = current microthread's index, `ut`*i* = microthread *i*. Pseudo-assembly line (1–2) determine how many elements each microthread should process, (3–5) set array pointers for this microthread, (7–9) last microthread only processes leftover elements, (12) shared scalar load, (15–16) scalar loads, (17–18) scalar arithmetic, (19) unit-stride vector store, (21–23) increment array pointers, (25–26) decrement loop counter and branch if not done.)

```
1  div    m, n, nthreads
2  mul    t, m, tidx
3  add    a_ptr, t
4  add    b_ptr, t
5  add    c_ptr, t
6
7  sub    t, nthreads, 1
8  br.neq t, tidx, notlast
9  rem    m, n, nthreads
10 notlast:
11
12 load   x, x_ptr
13
14 loop:
15 load   a, a_ptr
16 br.eq  a, 0, done
17
18 load   b, b_ptr
19 mul    t, x, a
20 add    c, t, b
21 store  c, c_ptr
22
23 done:
24 add    a_ptr, 1
25 add    b_ptr, 1
26 add    c_ptr, 1
27
28 sub    m, 1
29 br.neq m, 0, loop
```

(a) Pseudo-Assembly



(b) Execution Diagram for 2-Core, 4-Microthread Implementation

Figure 2.4: Mapping Irregular DLP to the MIMD Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1f. Execution diagram starts with load at line 12. Irregular DLP maps naturally since each microthread can use scalar branches for complex control flow. The microthreads are coherent (executing in lock-step) before the scalar branch but then diverge after the branch. (Assume `*_ptr` and `n` are inputs. `nthreads` = total number of microthreads, `tidx` = current microthread's index, `ut`i = microthread i. Pseudo-assembly line (1–2) determine how many elements each microthread should process, (3–5) set array pointers for this microthread, (7–9) last microthread only processes leftover elements, (12) shared scalar load, (15–16) check data-dependent conditional and skip work if possible, (18–21) conditionally executed scalar ops, (24–26) increment array pointers, (28–29) decrement loop counter and branch if not done.)

Figure 2.3a demonstrates how we might map the regular DLP loop shown in Table 2.1c to the MIMD pattern. The first ten lines of the pseudo-assembly code divide the work among the microthreads such that each thread works on a different consecutive partition of the input and output arrays. Although more efficient partitioning schemes are possible, we must always allocate the work at a fine-grain in software. Also notice that in the MIMD pattern all microthreads redundantly load the shared scalar value `x` (line 12). This might seem trivial, but the lack of a specialized mechanism to handle shared loads and possibly also shared computation can adversely impact many regular DLP codes. Similarly there are no specialized mechanisms to take advantage of the regular data accesses. Figure 2.3b shows an example execution diagram for a 2-core, 4-microthread implementation with two-way multithreading. The scalar instructions from each microthread are interleaved in a fixed pattern. Since threads will never bypass values between each other, a fixed thread interleaving can improve energy-efficiency by eliminating the need for some interlocking and bypassing control logic. Unfortunately, a fixed interleaving can also cause poor performance on scalar code or when one or more threads are stalled. A more dynamic scheduling scheme can achieve higher performance by issuing any thread on any cycle, but then we must once again include the full interlocking and bypassing control logic to support back-to-back instructions from the same thread. For this and all execution diagrams in this chapter, we limit the number of function unit resources for simplification. In this example, each core can only execute one instruction at a time, but a realistic implementation would almost certainly include support for executing multiple instructions at once either through dynamic superscalar issue or possibly static VLIW scheduling; both of which can exploit instruction-level parallelism to enable increased performance at the cost of some energy overhead.

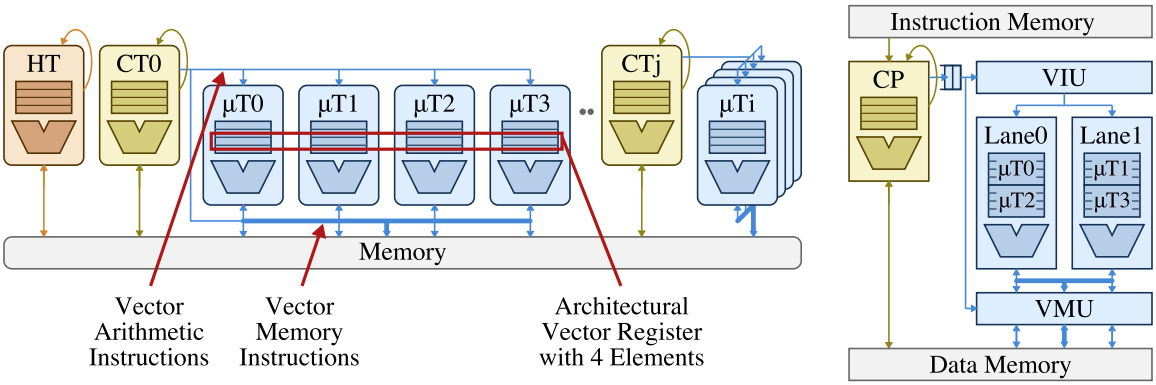
Figure 2.4a demonstrates how we might map the irregular DLP loop shown in Table 2.1f to the MIMD pattern. It is very natural to map the data-dependent conditional to a scalar branch which simply skips over the unnecessary work when possible. It is also straight-forward to implement conditional loads and stores of the `B` and `C` arrays by simply placing them after the branch. Figure 2.4b shows how the microthreads are *coherent* (execute in lock-step) before the branch and then *diverge* after the data-dependent conditional with μT1 and μT2 quickly moving on to the next iteration. After a few iterations the microthreads will most likely be completely diverged.

The recently proposed 1000-core Illinois Rigel accelerator is a good example of the MIMD pattern with a single thread per scalar core [KJJ+09]. The Rigel architects specifically rationalize the decision to use standard scalar RISC cores with no dedicated hardware mechanisms for regular DLP to enable more efficient execution of irregular DLP [KJJ+09]. Sun's 8-core Niagara processors exemplify the spirit of the multithreaded MIMD pattern with 4–8 threads per core for a total of 32–64 threads per chip [KAO05, NHW+07]. The Niagara processors are good examples of the multithreading pattern, although they are not specifically data-parallel accelerators. Niagara threads are heavier-weight than microthreads, and Niagara is meant be a stand-alone processor as

opposed to a true coprocessor. Even so, the Niagara processors are often used to execute both regular and irregular DLP codes, and their multithreading enables good performance on these kinds of codes [Wil08]. Niagara processors dynamically schedule active threads, giving priority to the least-recently scheduled thread. Threads can become inactive while waiting for data from cache misses, branches to resolve, or long-latency functional units such as multiplies and divides. These MIMD accelerators can be programmed using general-purpose parallel programming frameworks such as OpenMP [ope08b] and Intel's Thread Building Blocks [Rei07], or in the case of the Rigel accelerator, a custom task-based framework is also available [KJL+09].

2.4 Vector-SIMD Architectural Design Pattern

In the vector single-instruction multiple-data (vector-SIMD) pattern a *control thread* uses vector memory instructions to move data between main memory and vector registers, and vector arithmetic instructions to operate on vectors of elements at once. As shown in Figure 2.5a, one way to think of this pattern is as if each control thread manages an array of microthreads which execute in lockstep; each microthread is responsible for one element of the vector. In this context, microthreads are sometimes referred to as virtual processors [ZB91]. The number of microthreads per control thread, or synonymously the number of elements per vector register, is also called the hardware vector



(a) Programmer's Logical View

(b) Implementation of 2-Lane, 4-Microthread Core

Figure 2.5: Vector-SIMD Architectural Design Pattern – A general-purpose host thread uses shared memory to manage a large number of control threads. Each control thread is in turn responsible for managing a small array of microthreads using a combination of vector memory and arithmetic commands. The input dataset is partitioned across the control threads at a coarse granularity and then partitioned again across the microthreads at a fine granularity. A control thread and its array of microthreads map to a single core in the implementation; the control thread executes on a control processor, and the microthreads are striped both spatially and temporally across some number of vector lanes. Each control thread's array of microthreads execute in lockstep. The vector issue unit, vector lanes, and vector memory unit are often referred to as a vector unit. (HT = host thread, CT = control thread, μT = microthread, CP = control processor, VIU = vector issue unit, VMU = vector memory unit.)

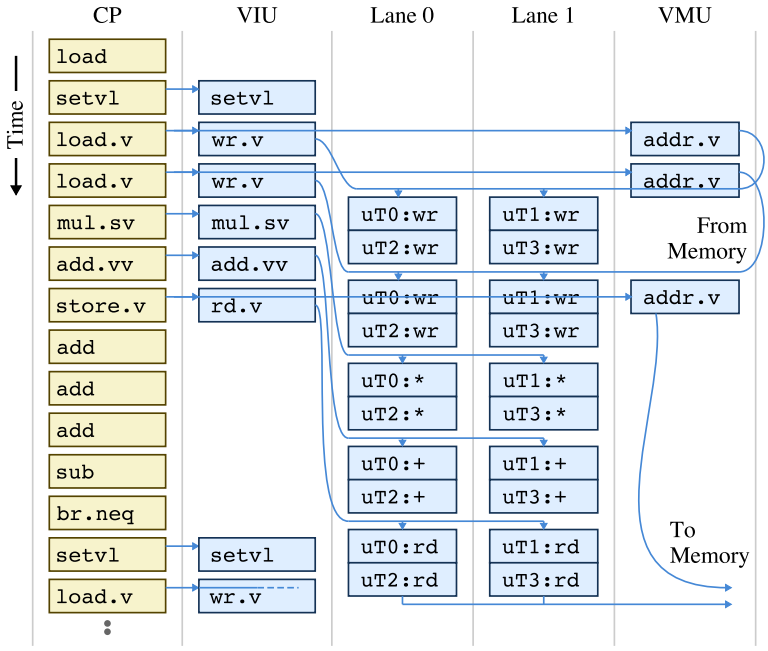
length (e.g., four in Figure 2.5a). As in the MIMD pattern, a host thread runs on the general purpose processor and manages the data-parallel accelerator. Unlike the MIMD pattern, a host thread in the vector-SIMD pattern only interacts with the control threads through shared memory and does not directly manage the microthreads. This explicit two-level hierarchy is an important part of the vector-SIMD pattern. Even though the host thread and control threads must still allocate work at a coarse-grain amongst themselves via software, this configuration overhead is amortized by the hardware vector length. The control thread in turn distributes work to the microthreads with vector instructions enabling very efficient execution of fine-grain DLP.

Figure 2.5b shows a prototypical implementation for a single vector-SIMD core. The control thread is mapped to a *control processor* (CP) and the microthreads are striped across an array of *vector lanes*. In this example, there are two lanes and the vector length is four meaning that two microthreads are mapped to each lane. The *vector memory unit* (VMU) handles executing vector memory instructions which move data between the data memory and the vector register file in large blocks. The *vector issue unit* (VIU) handles the dependency checking and eventual dispatch of vector arithmetic instructions. Together the VIU, VMU, and the vector lanes are called a *vector unit*. Vector units can have varying numbers of lanes. At one extreme are purely spatial implementations where the number of lanes equals the hardware vector length, and at the other extreme are purely temporal implementations with a single lane. Multi-lane implementations improve throughput and amortize area overheads at the cost of increased design complexity, control broadcast energy in the VIU, and memory crossbar energy in the VMU. An important part of the vector-SIMD pattern is represented by the queue in Figure 2.5b, which enables the control processor to be decoupled from the vector unit [EV96]. This decoupling allows the control thread to run-ahead and better tolerate various latencies, but also means the control thread must use vector memory fence instructions to properly order memory dependencies between the control thread and the microthreads. In addition to decoupling, vector-SIMD control processors can leverage more sophisticated architectural techniques such as multithreading [EV97], out-of-order execution [EVS97], and superscalar issue [QCEV99]. Such control processors accelerate vector execution but also enable higher performance on general-purpose codes. Eventually, vector-SIMD control processor might become so general-purpose that the resulting data-parallel accelerator should really be regarded as a unified accelerator in the spirit of Figure 2.1c. (See [HP07, Appendix F] for a more detailed introduction to vector instruction sets and microarchitecture.)

Regular DLP maps very naturally to the vector-SIMD pattern. Figure 2.6a shows the vector-SIMD pseudo-assembly corresponding to the regular loop in Table 2.1c. Since this thesis focuses on a single core, I do not show any of the instructions required for coarse-grain data partitioning across cores. Unit-stride vector memory instructions (lines 5–6,9) efficiently move consecutive blocks of data in and out of vector registers. It is also common to provide special vector instructions for handling strided, indexed, and sometimes segment accesses. Vector-vector arithmetic instruc-

```
1  load    x, x_ptr
2
3  loop:
4   setvl   vlen, n
5   load.v  VA, a_ptr
6   load.v  VB, b_ptr
7   mul.sv  VT, x, VA
8   add.vv  VC, VT, VB
9   store.v VC, c_ptr
10
11  add     a_ptr, vlen
12  add     b_ptr, vlen
13  add     c_ptr, vlen
14
15  sub     n, vlen
16  br.neq  n, 0, loop
```

(a) Pseudo-Assembly



(b) Execution Diagram for 2-Lane, 4-Microthread Core

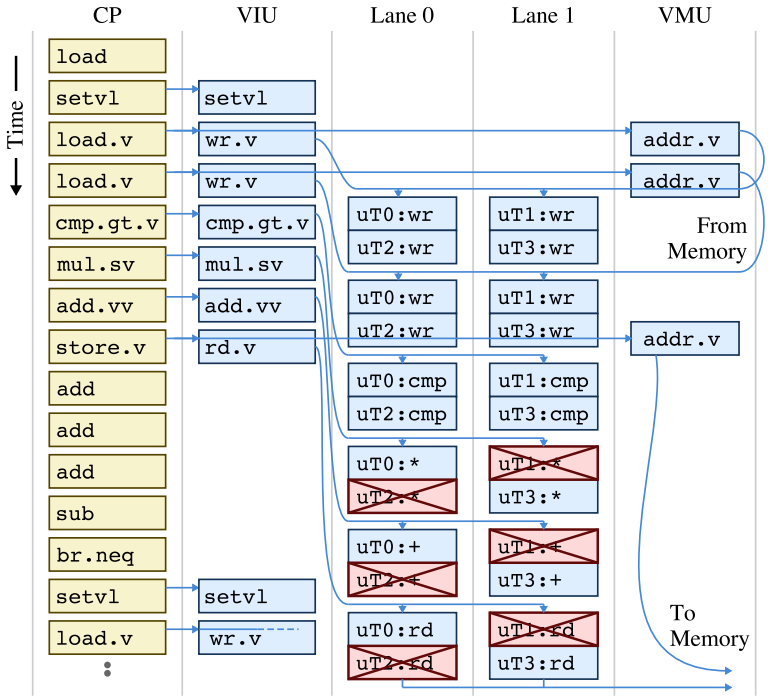
Figure 2.6: Mapping Regular DLP to the Vector-SIMD Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1c. Regular DLP maps naturally using vector memory and arithmetic commands. Each iteration of the stripmine loop works on `vlen` elements at once. (Assume `*_ptr` and `n` are inputs. `V`*i* = vector register *i*, `*.v` = vector command, `*.vv` = vector-vector op, `*.sv` = scalar-vector op, `el`*i* = element *i*. Pseudo-assembly line (1) shared scalar load, (4) set active hardware vector length, (5–6) unit-stride vector loads, (7–8) vector arithmetic, (9) unit-stride vector store, (11–13) increment array pointers, (15–16) decrement loop counter and branch if not done.)

tions (line 8) efficiently encode regular arithmetic operations across the full vector of elements, and a combination of a scalar load and a scalar-vector instruction (lines 1,7) can easily handle shared accesses. In the vector-SIMD pattern the hardware vector length is not fixed by the instruction set but is instead stored in a special control register. The `setvl` instruction takes the *application vector length* (`n`) as an input and writes the minimum of the application vector length and the hardware vector length to the given destination register `vlen` (line 4). As a side-effect, the `setvl` instruction sets the *active vector length* which essentially specifies how many of the microthreads are active and should participate in a vector instruction. Software can use the `setvl` instruction to process the vectorized loop in blocks equal to the hardware vector length without knowing what the actual hardware vector length is at compile time. The `setvl` instruction will naturally handle the final iteration when the application vector length is not evenly divisible by the hardware vector length; `setvl` simply sets the active vector length to be equal to the final remaining elements. This technique is called *stripmining* and enables a single binary to handle varying application vector lengths while still running on many different implementations with varying hardware vector lengths.

Figure 2.6b shows the execution diagram corresponding to the regular DLP pseudo-assembly

```
1  load     x, x_ptr
2
3  loop:
4   setvl    vlen, n
5   load.v   VA, a_ptr
6   load.v   VB, b_ptr
7   cmp.gt.v VF, VA, 0
8
9   mul.sv   VT, x, VA,  VF
10  add.vv   VC, VT, VB, VF
11  store.v  VC, c_ptr,  VF
12
13  add      a_ptr, vlen
14  add      b_ptr, vlen
15  add      c_ptr, vlen
16
17  sub      n, vlen
18  br.neq   n, 0, loop
```

(a) Pseudo-Assembly



(b) Execution Diagram for 2-Lane, 4-Element Implementation

Figure 2.7: Mapping Irregular DLP to the Vector-SIMD Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1f. Irregular DLP maps less naturally requiring the use of vector flags to conditionally execute operations for just a subset of the elements. Managing complex nested control flow can be particularly challenging. (Assume `*_ptr` and `n` are inputs. `V`i = vector register i, `VF` = vector flag register, `*.v` = vector command, `*.vv` = vector-vector op, `*.sv` = scalar-vector op, `el`i = element i. Pseudo-assembly line (1) shared scalar load, (4) set active hardware vector length, (5–6) unit-stride vector loads, (7) set vector flag register `VF` based on comparing `VA` to zero, (9–10) conditional vector arithmetic under flag, (11) conditional vector store under flag, (13–15) increment array pointers, (17–18) decrement loop counter and branch if not done.)

for the two-lane, four-microthread implementation pictured in Figure 2.5b. The vector memory commands are broken into two parts: the address portion goes to the VMU which will issue the request to memory while the register write/read portion goes to the VIU. For vector loads, the register writeback waits until the data returns from memory and then controls writing the vector register file two elements per cycle over two cycles. Notice that the VIU/VMU are decoupled from the vector lanes to allow the implementation to overlap processing multiple vector loads. The vector arithmetic operations are also processed two elements per cycle over two cycles. The temporal mapping of microthreads to the same lane is an important aspect of the vector-SIMD pattern. We can easily imagine using a larger vector register file to support longer vector lengths that would keep the vector unit busy for tens of cycles. The fact that one vector command can keep the vector unit busy for many cycles decreases instruction issue bandwidth pressure. So as in the MIMD pattern we can exploit instruction-level parallelism by adding support for executing multiple instructions per

microthread per cycle, but unlike the MIMD pattern it may not be necessary to increase the issue bandwidth, since one vector instruction occupies a vector functional unit for many cycles. Almost all vector-SIMD accelerators will take advantage of multiple functional units and also support bypassing (also called *vector chaining*) between these units. A final point to note is how the control processor decoupling and multi-cycle vector execution enables the control thread to continue executing while the vector unit is still processing older vector instructions. This decoupling means the control thread can quickly work through the loop overhead instructions (lines 11–16) so that it can start issuing the next iteration of the stripmine loop as soon as possible.

Figure 2.6 helps illustrate three ways the vector-SIMD pattern can improve energy-efficiency: (1) the microthreads simply do not execute some operations which must be executed for each element in the MIMD pattern (e.g., the shared load on line 1, the pointer arithmetic on lines 11–13, and the loop control overhead on lines 15–16); these instructions are instead executed by the control thread either once outside the stripmine loop or once for every `vlen` elements inside the stripmine loop; (2) for those arithmetic operations that the microthreads do need to execute (e.g., the actual multiplication and addition associated with lines 7–8), the CP and VIU can amortize various overheads such as instruction fetch, decode, and dependency checking over `vlen` elements; and (3) for those memory accesses which the microthreads still need to execute (e.g., the unit-stride accesses on lines 5–6,9) the VMU can efficiently move the data in large blocks. (See [LSFJ06] for a first-order quantitative study of the energy-efficiency of vector-SIMD compared to a standard general-purpose processor.)

Mapping irregular DLP to the vector-SIMD pattern can be significantly more challenging than mapping regular DLP. Figure 2.7a shows the vector-SIMD pseudo-assembly corresponding to the irregular loop in Table 2.1f. The key difference from the regular DLP example in Figure 2.6 is the use of a vector flag to conditionally execute the vector multiply, addition, and store instructions (lines 9–11). Although this might seem simple, leveraging this kind of conditional execution for more complicated irregular DLP with nested conditionals (e.g., Table 2.1h–k) can quickly require many independent flag registers and complicated flag arithmetic [SFS00]. Figure 2.7b shows the execution diagram for this irregular DLP loop. In this example, microthreads which are *inactive* because the corresponding flag is false still consume execution resources. This is called a *vlen-time* implementation, since the execution time of a vector instruction is proportional to the vector length regardless of how many microthreads are inactive. A more complicated *density-time* implementation skips inactive elements meaning that the execution time is proportional to the number of active elements [SFS00]. Density-time can be quite difficult to implement in a multi-lane vector unit, since the lanes are no longer strictly executing in lock-step. This is one motivation for our interest in single-lane vector units.

Note that even though this is an irregular DLP loop, we can still potentially improve energy-efficiency by offloading instructions onto the control thread, amortizing control overheads, and

using vector memory accesses. These savings must be balanced by the wasted energy for processing elements which are inactive. There are however opportunities for optimization. It is possible to bypass flag writes so that inactive elements can attempt to use clock or data gating to minimize energy. Once the entire flag register is written it may be possible to discard vector instructions for which no microthreads are active, but the implementation still must fetch, decode, and process these instructions. A density-time implementation will of course eliminate any work for inactive microthreads at the cost of increased control complexity.

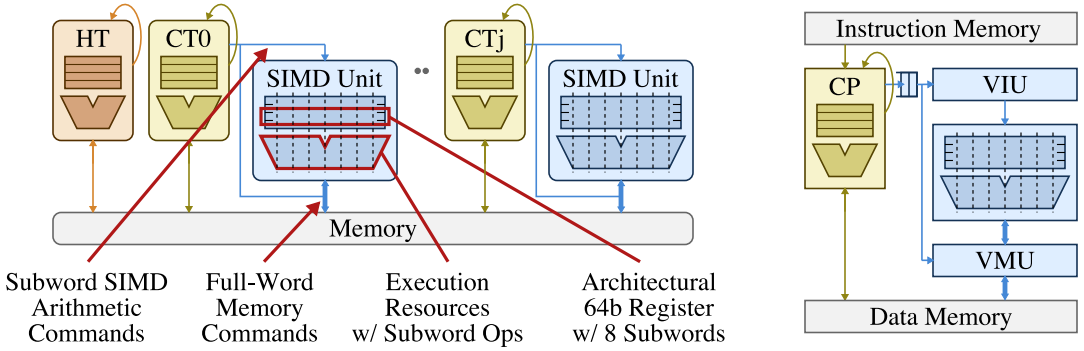
The Berkeley Spert-II system uses the T0 vector-SIMD processor to accelerate neural network, multimedia, and digital signal processing applications [WAK+96, Asa98]. The T0 instruction set contains 16 vector registers; unit-stride, strided, and indexed vector memory instructions; and a variety of integer and fixed-point vector arithmetic instructions. Conditional vector move instructions enable data-dependent conditional control flow. The T0 microarchitecture uses a simple RISC control processor and eight vector lanes, and it supports a vector length of 32. Three independent vector functional units (two for vector arithmetic operations and one for vector memory operations) enable execution of up to 24 operations per cycle. The Spert-II system includes a Sun Sparc processor for general-purpose computation and the discrete T0 data-parallel accelerator on an expansion card. The Spert-II system uses hand-coded assembly to vectorize critical kernels, but vectorizing compilers are also possible [DL95].

2.5 Subword-SIMD Architectural Design Pattern

The subword single-instruction multiple-data (subword-SIMD) architectural pattern is closely related to the vector-SIMD pattern but also captures some important differences. Figure 2.8 illustrates the programmer's logical view and an example implementation for this pattern. As with the vector-SIMD pattern, a host thread manages a collection of control threads which map to control processors each with its own vector-like unit. However, the defining characteristic of the subword-SIMD pattern is that the "vector-like unit" is really a standard full-word scalar datapath with standard scalar registers often corresponding to a double-precision floating-point unit. The pattern leverages these existing scalar datapaths in the SIMD unit to execute multiple narrow-width operations in a single cycle. For example, in Figure 2.8 each register in the SIMD unit is 64-bit wide but we can treat this register as containing 8×8-bit, 4×16-bit, 2×32-bit, or 1×64-bit operands. The execution resources are similarly partitioned enabling special SIMD instructions to execute vector-like arithmetic operations. Some subword-SIMD variants support bitwidths larger than the widest scalar datatype, in which case the datapath can only be fully utilized with subword-SIMD instructions. Other variants unify the control thread and SIMD unit such that the same datapath is used for both control, scalar arithmetic, and subword-SIMD instructions. Unified subword-SIMD saves area but eliminates any chance of control-thread decoupling.

In subword-SIMD, the number of elements which can be processed in a cycle is a function of how many elements can be packed into the word width. Since this number can vary depending on element size, the subword-SIMD datapath must be tightly integrated. Data movement into and out of the SIMD register file must be in full-word blocks, and there are often alignment constraints or performance implications for unaligned loads and stores. Software is often responsible for shuffling data elements via special permute operations so that elements are in the correct positions. These factors lead to a large amount of cross-element communication which can limit subword-SIMD widths to around 128–256 bits and also complicate temporal as opposed to purely spatial implementations. As illustrated in Figure 2.8 the full-word SIMD width is exposed to software requiring code to be rewritten for differently sized SIMD units. Ultimately, a programmer must think about the unit as a single wide datapath as opposed to a vector of elements, and this can make mapping irregular DLP applications to subword-SIMD units challenging. One advantage of subword SIMD is that the full datapath can be sometimes be leveraged for non-DLP code which uses full-width operands (e.g., double-precision operations on a 64-bit subword-SIMD datapath).

There are several characteristics of the subword-SIMD pattern which helps distinguish this pattern from the vector-SIMD pattern. Subword SIMD has short vector lengths which are exposed to software as wide fixed-width datapaths, while vector-SIMD has longer vector lengths exposed to software as a true vector of elements. In vector-SIMD, the vector length is exposed in such a way that the same binary can run on many different implementations with varying hardware resources



(a) Programmer's Logical View

(b) Implementation of Subword-SIMD Core

Figure 2.8: Subword-SIMD Architectural Design Pattern – A general-purpose host thread uses shared memory to manage a large number of control threads. Each control thread has its own SIMD unit which can execute full-word memory commands and subword-SIMD arithmetic commands to process multiple narrow width operations at once. The input dataset is partitioned across the control threads at a coarse granularity and then partitioned again across subword elements at a very fine granularity. A control thread and its SIMD unit map to a single core in the implementation. In the subword-SIMD pattern, the hardware vector length is fixed and part of the instruction set meaning there is little abstraction between the logical view and the actual implementation. (HT = host thread, CT = control thread, CP = control processor, VIU = vector issue unit, VMU = vector memory unit.)

and/or amounts of temporal versus spatial microthreading. Additionally, vector-SIMD has more flexible data-movement operations which alleviates the need for software data shuffling. In this thesis, I focus less on the subword-SIMD pattern, because the vector-SIMD pattern is better suited to applications with large amounts of data-parallelism as opposed to a more general-purpose workload with smaller amounts of data-parallelism.

The IBM Cell processor includes a general-purpose processor implementing the standard Power instruction set as well as an array of eight data-parallel cores interconnected by a on-chip ring network [GHF+06]. Each data-parallel core includes a unified 128-bit subword-SIMD datapath which can execute scalar operations as well as 16×8-bit, 8×16-bit, 4×32-bit, or 2×64-bit operations. The data-parallel cores can only access memory with aligned 128-bit operations, so special permute operations are required for unaligned accesses. Unaligned or scalar stores require multiple instructions to read, merge, and then write a full 128-bit value. Subword-SIMD conditional moves are provided for data-dependent conditionals, although each data-parallel core's control thread also has standard scalar branches. The Cell processor is a good example of the subword-SIMD pattern for dedicated data-parallel accelerators, but almost all general-purpose processors have also included some form of subword-SIMD for over a decade. Examples include Intel's MMX and SSE extensions for the IA-32 architecture, AMD's 3DNow! extensions for the IA-32 architecture, MIPS' MIPS-3D extensions for the MIPS32 and MIPS64 architectures, HP's MAX extensions for the PA-RISC architecture, and Sun's VIS extensions for the Sparc architecture (see [SS00] for a survey of subword-SIMD extensions in general-purpose processors). These extensions help general-purpose processors better execute data-parallel applications, and are a first step towards the full integration shown in Figure 2.1c. In terms of programming methodology, many modern compilers include intrinsics for accessing subword-SIMD operations, and some compilers include optimization passes that can automatically vectorize regular DLP.

2.6 SIMT Architectural Design Pattern

The single-instruction multiple-thread (SIMT) pattern is a hybrid pattern with a programmer's logical view similar to the MIMD pattern but an implementation similar to the vector-SIMD pattern. As shown in Figure 2.9, the SIMT pattern supports a large number of microthreads which can each execute scalar arithmetic and control instructions. There are no control threads, so the host thread is responsible for directly managing the microthreads (usually through specialized hardware mechanisms). A *microthread block* is mapped to a SIMT core which contains vector lanes similar to those found in the vector-SIMD pattern. However, since there is no control thread, the VIU is responsible for amortizing overheads and executing the microthread's scalar instructions in lockstep when they are coherent. The VIU also manages the case when the microthreads execute a scalar branch possibly causing them to diverge. Micro-threads can sometimes reconverge through static hints in the scalar instruction stream or dynamic hardware mechanisms. SIMT only has scalar loads

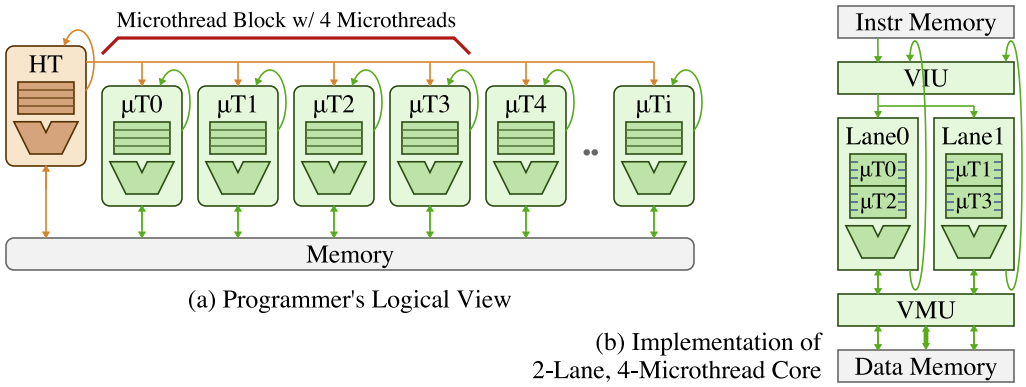


Figure 2.9: SIMT Architectural Design Pattern – A general-purpose host thread manages a large number of microthreads using dedicated hardware schedulers. The input dataset is partitioned across the microthreads at a fine granularity, and each microthread executes scalar instructions. A block of microthreads is mapped to a single core which is similar in spirit to a vector unit. A SIMT vector issue unit can dynamically transform the same scalar instruction executed across a core's microthreads into vector-like arithmetic commands, and a SIMT vector memory unit can dynamically transform regular data accesses across a core's microthreads into vector-like memory commands. (HT = host thread, µT = microthread, VIU = vector issue unit, VMU = vector memory unit.)

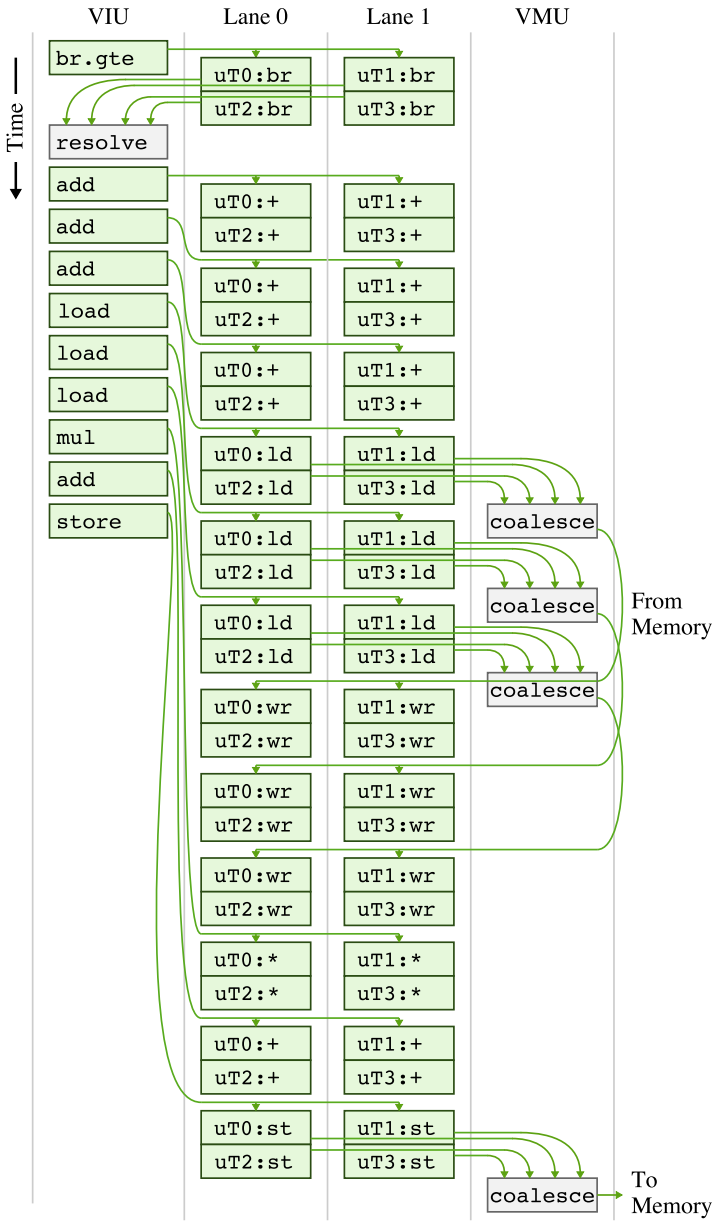
and stores, but the VMU can include a memory coalescing unit which dynamically detects when these scalar accesses can be turned into vector-like memory operations. The SIMT pattern usually exposes the concept of a microthread block to the programmer: barriers are sometimes provided for intra-block synchronization, and application performance depends heavily on the coherence and coalescing opportunities within a microthread block.

Regular DLP maps to the SIMT pattern in a similar way as in the MIMD pattern except that each microthread is usually only responsible for a single element as opposed to a range of elements (see Figure 2.10a). Since there are no control threads and thus nothing analogous to the vector-SIMD pattern's `setvl` instruction, a combination of dedicated hardware and software is required to manage the stripmining. The host thread tells the hardware how many microthread blocks are required for the computation and the hardware manages the case when the number of requested microthread blocks is greater than what is available in the actual hardware. In the common case where the application vector length is not statically guaranteed to be evenly divisible by the microthread block size, each microthread must use a scalar branch to verify that the computation for the corresponding element is actually necessary (line 1). Notice that as with the MIMD pattern, each microthread is responsible for its own address calculation to compute the location of its element in the input and output arrays (lines 3–5).

Figure 2.10b shows the execution diagram corresponding to the regular DLP pseudo-assembly for the two-lane, four-element implementation pictured in Figure 2.9b. Scalar branch management corresponding to the branch at line 1 will be discussed later in this section. Without a control thread, all four microthreads redundantly perform address calculations (lines 3–4) and the actual

```
1  br.gte tidx, n, done
2
3  add    a_ptr, tidx
4  add    b_ptr, tidx
5  add    c_ptr, tidx
6
7  load   x, x_ptr
8
9  load   a, a_ptr
10 load   b, b_ptr
11 mul    t, x, a
12 add    c, t, b
13 store  c, c_ptr
14 done:
```

(a) Pseudo-Assembly

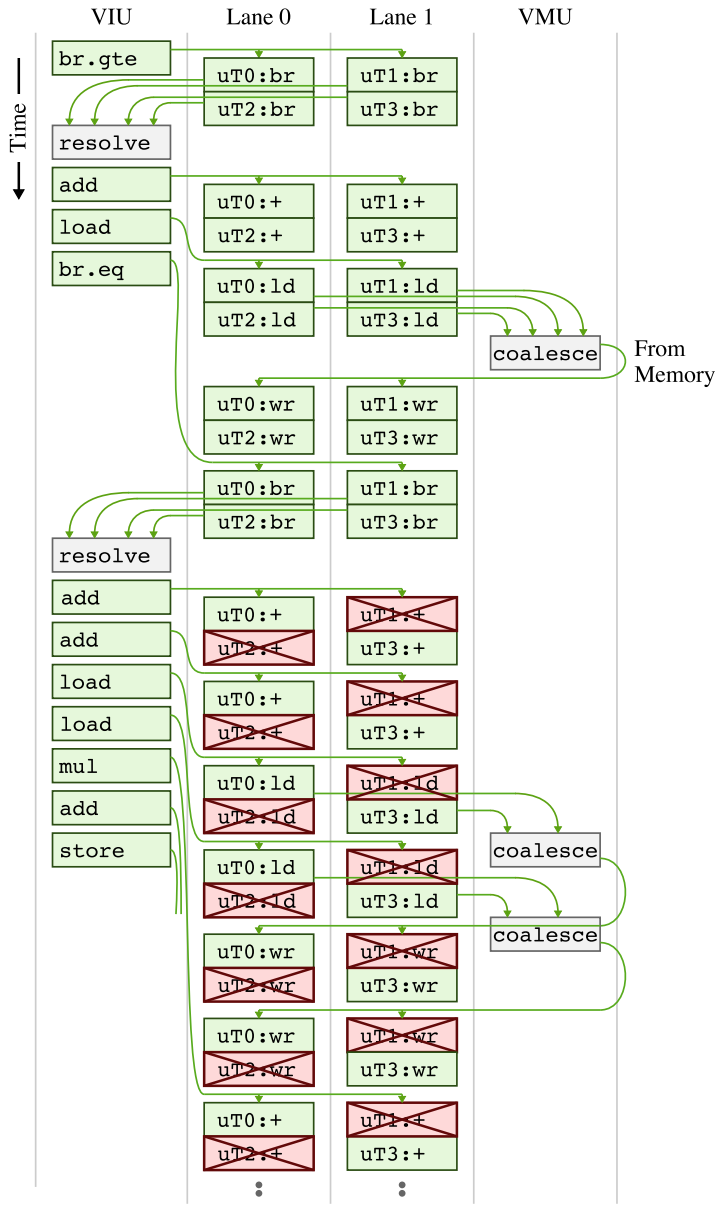


(b) Execution Diagram for 2-Lane, 4-Microthread Implementation

Figure 2.10: Mapping Regular DLP to the SIMT Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1c. Since there are no explicit mechansims for regular DLP exposed as part of a SIMT instruction set, DLP codes are always encoded as a set of scalar operations for each microthread. The implementation attempts to execute coherent scalar arithmetic ops with vector-like efficiencies and turn unit-stride scalar accesses into vector-like memory ops with dynamic coalescing. (Assume `*_ptr` and `n` are inputs. `tidx` = current microthread's index, `uti` = microthread *i*. Pseudo-assembly line (1) branch to handle situations where application vector length `n` is not evenly divisible by the hardware vector length, (3–5) set array pointers for this microthread, (7) shared scalar load, (9–10) scalar loads, (11–12) scalar arithmetic, (13) scalar store.)

```
1  br.gte tidx, n, done
2
3  add    a_ptr, tidx
4  load   a, a_ptr
5  br.eq  a, 0, done
6
7  add    b_ptr, tidx
8  add    c_ptr, tidx
9
10 load   x, x_ptr
11 load   b, b_ptr
12 mul    t, x, a
13 add    c, t, b
14 store  c, c_ptr
15 done:
```

(a) Pseudo-Assembly



(b) Execution Diagram for 2-Lane, 4-Microthread Implementation

Figure 2.11: Mapping Irregular DLP to the SIMT Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1f. Irregular DLP maps naturally since each SIMT microthread already exclusively uses a scalar instruction set and thus can leverage scalar branches which are dynamically turned into vector-like flags. (Assume `*_ptr` and `n` are inputs. `tidx` = current microthread's index, `uti` = microthread *i*. Pseudo-assembly line (1) branch to handle situations where application vector length `n` is not evenly divisible by the hardware vector length, (3–5) check data-dependent conditional and skip work if possible, (7–8) set remaining array pointers for this microthread, (10) shared scalar load, (11) scalar loads, (12–13) scalar arithmetic, (14) scalar store.)

scalar load instruction (lines 9–10) even though these are unit-stride accesses. SIMT instruction sets may, however, provide special addressing modes to help streamline this process. The VMU dynamically checks all four addresses, and if they are consecutive, then the VMU will transform these accesses into a single vector-like memory operation. Also notice that since there is no control thread to amortize the shared load at line 7, all four microthreads must redundantly load `x`. The VMU may be able to dynamically coalesce this into one scalar load which is then broadcast to all four microthreads. Since the microthreads are coherent when they execute the scalar multiply and addition instructions (lines 11–12), the VIU should be able to execute them with vector-like efficiencies. The VMU attempts to coalesce well-structured stores (line 13) as well as loads.

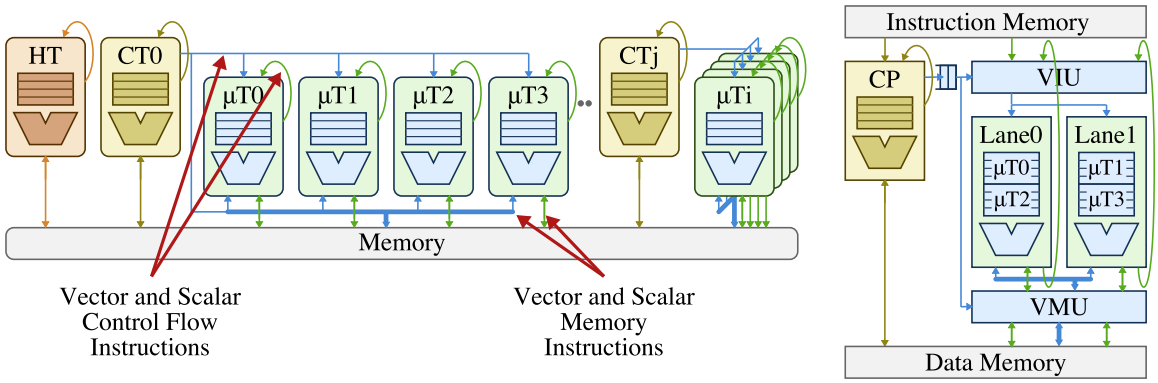
Figure 2.10 illustrates some of the issues that can prevent the SIMT pattern from achieving vector-like energy-efficiencies on regular DLP. The microthreads must redundantly execute instructions that would otherwise be amortized onto the control thread (lines 1–7). Regular data accesses are encoded as multiple scalar accesses which then must be dynamically transformed (at some energy overhead) into vector-like memory operations. In addition, the lack of a control thread prevents access-execute decoupling which can efficiently tolerate memory latencies. Even so, the ability to achieve vector-like efficiencies on coherent arithmetic microthread instructions helps improve SIMT energy-efficiency compared to the MIMD pattern.

Of course, the SIMT pattern's strength is on irregular DLP as shown in Figure 2.11. Instead of using vector flags which are cumbersome for complicated control flow, SIMT enables encoding data-dependent conditionals as standard scalar branches (lines 1,5). Figure 2.11b shows how the SIMT pattern handles these scalar branches in a vector-like implementation. After issuing the scalar branch corresponding to line 5, the VIU waits for the microthread block to calculate the branch resolution based on each microthread's scalar data. Essentially, the VIU then turns these branch resolution bits into a dynamically generated vector flag register which is used to mask off inactive elements on either side of the branch. Various SIMT implementations handle the details of microthread divergence differently, but the basic idea is the same. In contrast to vector-SIMD (where the control processor is decoupled from the vector unit making it difficult to access the vector flag registers), SIMT can avoid fetching instructions when the vector flag bits are all zero. So if the entire microthread block takes the branch at line 5, then the VIU can completely skip the instructions at lines 7–14 and start the microthread block executing at the branch target. Conditional memory accesses are naturally encoded by simply placing them after a branch (lines 10–11,14). SIMT instruction sets will likely include conditional execution (e.g., predication or conditional move instructions) to help mitigate branch overhead on short conditional regions of code. Notice that even though the microthreads have diverged there are still opportunities for partial vector-like energy-efficiencies. In this example, the VIU can still amortize by the number of active elements the instruction fetch, decode, and dependency checking for the scalar multiply and addition instructions at lines 12–13. Just as in the vector-SIMD pattern, both vlen-time and density-time implementations are possible.

The SIMT pattern has been recently developed as a way for more general data-parallel programs to take advantage of the large amount of compute resources available in modern graphics processors. For example, the NVIDIA Fermi graphics processor includes 32 SIMT cores each with 16 lanes suitable for graphics as well as more general data-parallel applications [nvi09]. Each microthread block contains 32 microthreads, but there are many more than two microthreads per lane because Fermi (and indeed all graphics processors) have heavily multithreaded VIUs. This means that many microthread blocks are mapped to the same data-parallel core and the VIU is responsible for scheduling these blocks onto the execution resources. A block is always scheduled as a unit, and there are no vector-like efficiencies across blocks time-multiplexed onto the same core. This is analogous to mapping many control threads to the same control processor each with their own independent array of microthreads mapped to the same set of vector lanes. Multithreading the VIU allows Fermi to better hide the branch resolution latency illustrated in Figure 2.11b and also hide memory latencies compensating for the lack of control-thread decoupling. Fermi microthreads have a standard scalar instruction set which includes scalar loads and stores, integer arithmetic operations, double-precision floating point arithmetic operations, and atomic memory operations for efficient inter-microthread synchronization and communication. Some execution resources (e.g., load/store units and special fixed function arithmetic units) are shared among two cores. Various SIMT frameworks such as Microsoft's DirectX Compute [Mic09], NVIDIA's CUDA [NBGS08], Stanford's Brook [BFH+04], and OpenCL [ope08a] allow programmers to write high-level code for the host thread and to specify the scalar code for each microthread as a specially annotated function. A combination of off-line compilation, just-in-time optimization, and hardware actually executes the data-parallel program. Unfortunately, much of the low-level software, instruction set, and microarchitecture for graphics processors are not publicly disclosed, but the SIMT pattern described in this section captures the general features of these accelerators based on publicly available information.

2.7 VT Architectural Design Pattern

The vector-thread (VT) pattern is also a hybrid pattern but takes a very different approach as compared to the SIMT pattern introduced in the previous section. Figure 2.12a illustrates the programmer's logical view for the VT pattern. Like the vector-SIMD pattern, the host thread manages a collection of control threads through shared memory and each control thread in turn manages an array of microthreads. Similar to the vector-SIMD pattern, this allows various overheads to be amortized onto the control thread, and control threads can also execute vector memory commands to efficiently handle regular data accesses. Unlike the vector-SIMD pattern, the control thread does not execute vector arithmetic instructions but instead uses a *vector fetch instruction* to indicate the start of a scalar instruction stream which should be executed by the microthreads. The microthreads execute coherently, but as in the SIMT pattern, they can also diverge after executing scalar branches.



(a) Programmer's Logical View

(b) Implementation of
2-Lane, 4-Microthread Core

Figure 2.12: VT Architectural Design Pattern – A general-purpose host thread uses shared memory to manage a large number of control threads. Each control thread is in turn responsible for managing a small array of microthreads using a combination of vector memory commands and vector fetch commands. The input dataset is partitioned across the control threads at a coarse granularity and then partitioned again across the microthreads at a fine granularity. A control thread and its array of microthreads map to a single core which is similar in spirit to a vector-SIMD core. A control thread's array of microthreads begins executing in lockstep with each vector fetch but can diverge when microthreads execute data-dependent control flow. (HT = host thread, CT = control thread, µT = microthread, CP = control processor, VIU = vector issue unit, VMU = vector memory unit.)

Figure 2.12b shows a typical implementation for a two-lane VT core with four microthreads. The control thread is mapped to a decoupled control processor and the microthreads are mapped both spatially and temporally to the vector lanes. Vector memory instructions use the VMU to execute in a very similar fashion as with vector-SIMD and thus achieve similar efficiencies. The VIU has its own port to the instruction memory which allows it to process vector fetch instructions. For vector-fetched instructions which are coherent across the microthreads, the implementation should achieve similar efficiencies as a vector-SIMD implementation. As with the MIMD and SIMT-patterns, VT allows scalar branches to encode complex data-dependent control flow which can simplify the programming methodology. Compared to a vector-SIMD implementation, VT requires additional control complexity in the VIU to handle the microthreads scalar instruction set but should have similar efficiency on regular DLP and simplifies mapping irregular DLP.

Regular DLP maps very naturally to the VT pattern. Figure 2.13a shows the VT pseudo-assembly corresponding to the regular DLP loop in Table 2.1c. Stripmining (line 5), loop control (line 11–16), and regular data accesses (lines 6–7,9) are handled just as in the vector-SIMD pattern. Instead of vector arithmetic instructions, we use a vector fetch instruction (line 8) with one argument which indicates the instruction address at which all microthreads should immediately start executing (e.g., the instruction at the `ut_code` label). All microthreads execute these scalar instructions (lines 20–21) until they reach a stop instruction (line 22). An important part of the VT pattern

is the interaction between vector registers as accessed by the control thread, and scalar registers as accessed by each microthread. In this example, the unit-stride vector load at line 6 writes the vector register `VA` with `vlen` elements. Each microthread's scalar register `a` implicitly refers to that microthread's element of the vector register (e.g., µT0's scalar register `a` implicitly refers to the first element of the vector register `VA` and µT(`vlen`-1)'s scalar register `a` implicitly refers to the last element of the vector register `VA`.). In other words, the vector register `VA` as seen by the control thread and the scalar register `a` as seen by the microthreads are two views of the same register. The microthreads cannot access the control thread's scalar registers, since this would significantly complicate control processor decoupling. Shared accesses are thus communicated with a scalar load by the control thread (line 1) and then a scalar-vector move instruction (line 2) which copies the given scalar register value into each element of the given vector register. This scalar value is then available for all microthreads to reference (e.g., as microthread scalar register `z` on line 20).

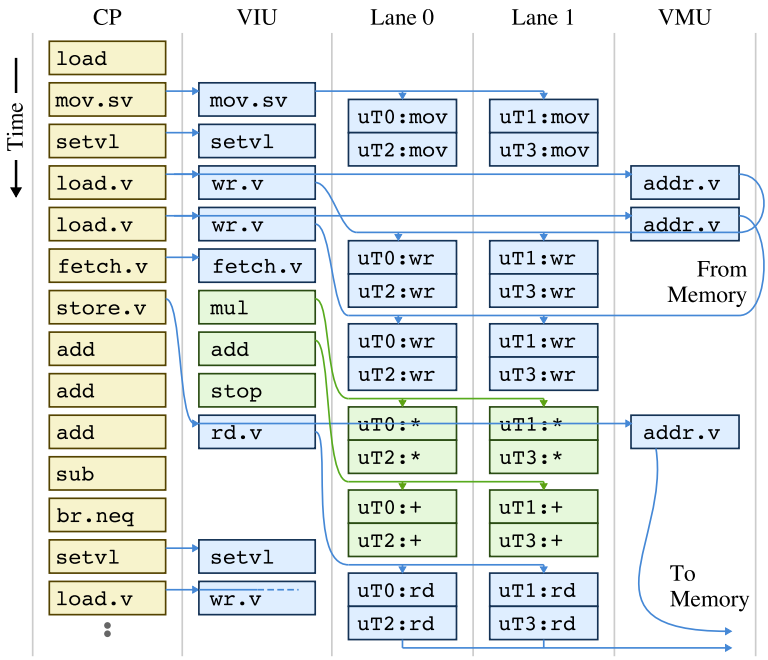
Figure 2.13b illustrates how this regular DLP loop would execute on the implementation pictured in Figure 2.12b. An explicit scalar-vector move instruction (line 2) writes the scalar value into each element of the vector register two elements per cycle over two cycles. The unit-stride vector load instructions (lines 6–7) execute exactly as in the vector-SIMD pattern. The control processor then sends the vector fetch instruction to the VIU. The VIU fetches the scalar multiply and addition instructions (lines 20–21) and issues them across the microthreads. When the VIU fetches and decodes the stop instruction (line 22) it moves on to process the next command waiting for it in the vector command queue. As in vector-SIMD, VT supports control processor decoupling, and by offloading the microthread instruction fetch to the VIU, the control processor is actually able to more quickly run-ahead to the next set of instructions.

Figure 2.13 demonstrates how VT achieves vector-like energy-efficiency on regular DLP. Control instructions are executed once by the control thread per-loop (lines 1–2) or per-iteration (lines 11–16). Because the microthreads are coherent in this example, the VIU is able to amortize instruction fetch, decode, and dependency checking for vector arithmetic instructions (lines 20–21). VT uses the exact same vector memory instructions to efficiently move blocks of data between memory and vector registers (lines 6–7,9). There are however some overheads including the extra scalar-vector move instruction (line 2), vector fetch instruction (line 8), and microthread stop instruction (line 22).

Irregular DLP also maps naturally and potentially more efficiently to the VT pattern, since one can use scalar branches to encode complicated control flow. Figure 2.14a shows the VT pseudo-assembly corresponding to the irregular loop in Table 2.1f. The primary difference from the regular DLP loop is that this example includes a scalar branch as the first vector-fetched instruction on line 20. Microthreads thus completely skip the multiplication and addition if possible. The conditional store is encoded by simply placing the store after the branch (line 24) similar to the MIMD and SIMT examples. As with the SIMT pattern, VT instruction sets will likely include conditional execution (e.g., predication or conditional move instructions) to help mitigate branch overhead on

```
1  load    x, x_ptr
2  mov.sv  VZ, x
3
4  loop:
5   setvl   vlen, n
6   load.v  VA, a_ptr
7   load.v  VB, b_ptr
8   fetch.v ut_code
9   store.v VC, c_ptr
10
11  add     a_ptr, vlen
12  add     b_ptr, vlen
13  add     c_ptr, vlen
14
15  sub     n, vlen
16  br.neq  n, 0, loop
17  ...
18
19 ut_code:
20  mul     t, z, a
21  add     c, t, b
22  stop
```

(a) Pseudo-Assembly



(b) Execution Diagram for 2-Lane, 4-Microthread Implementation

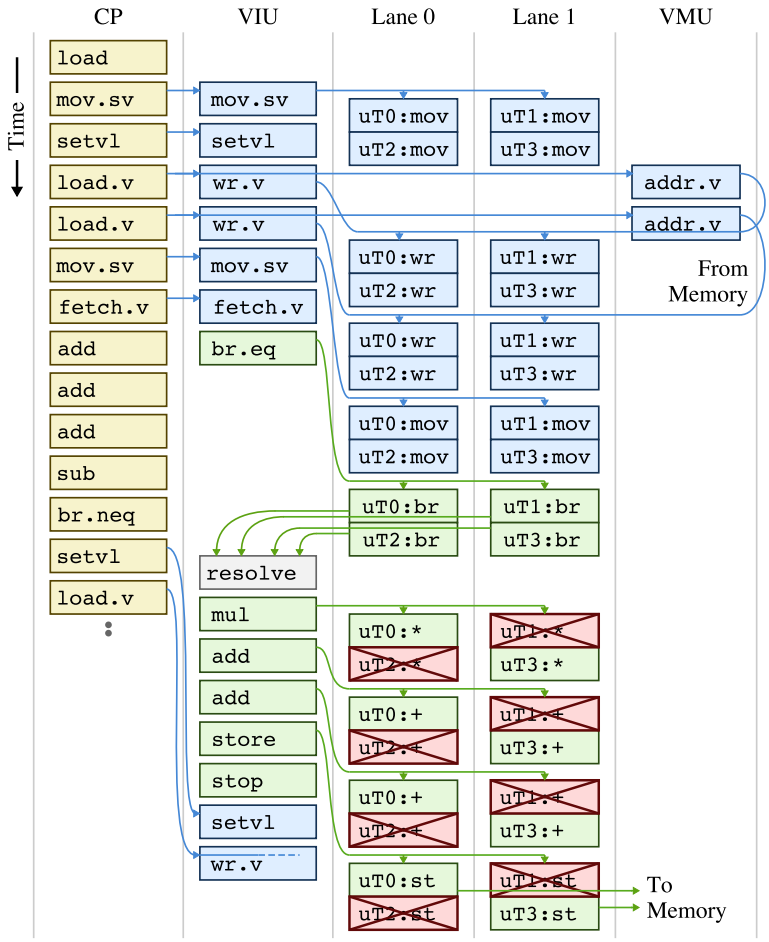
Figure 2.13: Mapping Regular DLP to the VT Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1c. Regular DLP maps naturally using vector memory and vector-fetched scalar arithmetic commands. Each iteration of the stripmine loop works on `vlen` elements at once. (Assume `*_ptr` and `n` are inputs. `V`i = vector register i, `*.v` = vector command, `*.sv` = scalar-vector op, `ut`i = microthread i. In vector-fetched microthread code, scalar register i corresponds to vector register `V`i. Pseudo-assembly line (1–2) shared load and move scalar `x` to all elements in vector register `VX`, (5) set active hardware vector length, (6–7) unit-stride vector loads, (8) vector fetch so that all microthreads start executing the scalar ops at label `ut_code`, (9) unit-stride vector store, (11–13) increment array pointers, (15–16) decrement loop counter and branch if not done, (20–21) microthread scalar ops, (22) stop executing microthreads and continue to next vector command.)

short conditional regions of code. Notice that we must explicitly pass the `c_ptr` base address so that each microthread can calculate an independent store address (line 23) when the branch is not taken. The vector-SIMD pattern uses a conditional vector store instruction to amortize some of this address calculation overhead. In this example, loading the `B` array is hoisted out of the conditional code (line 7). Depending on how often the branch is taken, it might be more efficient to transform this vector load into a vector-fetched scalar load so that we only fetch data which is actually used. The MIMD and SIMT examples use this approach, while the vector-SIMD example also encodes the load as a unit-stride vector memory instruction.

Figure 2.14b shows the execution diagram corresponding to the irregular DLP loop. Similar to the SIMT pattern, the VIU must wait until all microthreads resolve the scalar branch to determine how to proceed. If all microthreads either take or do not take the branch, then the VIU can simply

```
1  load    x, x_ptr
2  mov.sv  VZ, x
3
4  loop:
5   setvl   vlen, n
6   load.v  VA, a_ptr
7   load.v  VB, b_ptr
8   mov.sv  VD, c_ptr
9   fetch.v ut_code
10
11  add     a_ptr, vlen
12  add     b_ptr, vlen
13  add     c_ptr, vlen
14
15  sub     n, vlen
16  br.neq  n, 0, loop
17  ...
18
19 ut_code:
20  br.eq   a, 0, done
21  mul     t, z, a
22  add     c, t, b
23  add     d, tidx
24  store   c, d
25 done:
26  stop
```

(a) Pseudo-Assembly



(b) Execution Diagram for 2-Lane, 4-Microthread Implementation

Figure 2.14: Mapping Irregular DLP to the VT Pattern – Example (a) pseudo-assembly and (b) execution corresponding to Table 2.1f. Irregular DLP also maps naturally since each microthread can use standard scalar branches for data-dependent control flow. (Assume `*_ptr` and `n` are inputs. `V`*i* = vector register *i*, `*.v` = vector command, `*.sv` = scalar-vector op, `el`*i* = microthread *i*, `tidx` = current microthread's index. In vector-fetched microthread code, scalar register *i* corresponds to vector register `V`*i*. Pseudo-assembly line (1–2) shared load and move scalar `x` to all elements of vector register `VX`, (5) set active hardware vector length, (6–7) unit-stride vector loads, (8) move array `C` base pointer to all elements of vector register `VD`, (9) vector fetch so that all microthreads start executing the scalar ops at label `ut_code`, (11–13) increment array pointers, (15–16) decrement loop counter and branch if not done, (20) check data-dependent conditional and skip work if possible, (21–22) microthread scalar ops, (23–24) scalar store, (26) stop executing microthreads and continue to next vector command.)

start fetching from the appropriate address. If some microthreads take the branch while others do not, then the microthreads diverge and the VIU needs to keep track of which microthreads are executing which side of the branch. Although dynamic reconvergence detection hardware or static reconvergence hints in the microthread instruction stream are possible, they are not required for good performance on most applications. Each new vector fetch instruction naturally causes the microthreads to reconverge. As with the vector-SIMD and SIMT patterns, VT can also provide vlen-time and density-time implementations.

The VT architectural design pattern was developed as part of our early work on intermingling vector and threaded execution mechanisms. VT was proposed independently from the SIMT pattern, although we have since recognized some of the similarities between the two approaches. The Scale VT processor was our first implementation of the VT pattern [KBH+04a, BKGA04, KBA08]. Scale includes a specialized microthread instruction set with software-exposed clustered functional units, sophisticated predication, and explicit thread-fetches instead of scalar branches. Thread-fetches essentially enable variable-length branch-delay slots that help hide microthread control flow latencies. The Scale implementation includes a simple RISC control processor and a four-lane VTU. Each lane has four execution clusters resulting in a peak vector unit throughput of 16 operations per cycle. Scale's programming methodology uses either a combination of compiled code for the control thread and hand-coded assembly for the microthreads, or a preliminary version of a vectorizing compiler written specifically for Scale [HA08].

2.8 Comparison of Architectural Design Patterns

This chapter has presented five architectural patterns for the design of data-parallel cores. Table 2.2 summarizes the mechanisms provided by each pattern to exploit regular and irregular DLP. Since the MIMD pattern has no explicit regular DLP mechanisms, programmers simply use scalar memory, arithmetic, and control instructions regardless of whether the application exhibits regular or irregular DLP. The MIMD pattern can include conditional execution such as predication or conditional move instructions to help mitigate branch overheads in irregular control flow. The vector-SIMD pattern is at the opposite end of the spectrum with a significant focus on mechanisms for efficiently exploiting regular DLP. Notice that scalar memory and arithmetic instructions executed on the control thread can also be used to exploit regular DLP by amortizing shared loads and computation. The vector-SIMD pattern relies on indexed vector memory accesses, conditional loads/stores, and vector flags or vector conditional move instructions for irregular DLP. Very complicated irregular DLP might need to be executed serially on the control thread. The subword-SIMD pattern is similar in spirit to the vector-SIMD pattern but uses subword and full-word operations instead of true vector instructions. Irregular data accesses are usually encoded with a combination of full-word memory operations with subword permute instructions. The SIMT pattern lacks a control thread meaning that only microthread instructions are available for exploiting DLP. Scalar instruc-

	Regular Data Access	Regular Control Flow	Irregular Data Access	Irregular Control Flow
MIMD	µT scalar ld/st	µT scalar instr	µT scalar ld/st	µT scalar branch —— µT cexec scalar instr
Vector SIMD	vector unit-stride, strided ld/st	vector instr	vector indexed ld/st —— vector cexec ld/st	vector cexec instr
Subword SIMD	full-word ld/st	subword instr	full-word ld/st with subword permute instr	subword cexec instr
SIMT	µT scalar ld/st with HW coalescing	µT scalar instr with HW coherence	µT scalar ld/st	µT scalar branches —— µT cexec scalar instr
VT	vector unit-stride, strided ld/st	vector-fetched µT scalar instr with HW coherence	µT scalar ld/st	µT vector-fetched scalar branches —— µT cexec instr

Table 2.2: Mechanisms for Exploiting Data-Level Parallelism – Each architectural design pattern has different mechanisms for handling both regular and irregular DLP. Note that control thread instructions can be used for regular DLP to refactor shared data accesses and computation. (µT = microthread, CT = control thread, ld/st = loads/stores, instr = instruction, cexec = conditionally executed (e.g., predication or conditional move instructions), HW = hardware)

tions with hardware coherence (i.e., instructions executed with vector-like efficiencies) and memory coalescing are critical to achieving good performance and low energy with the SIMT pattern. The VT pattern uses its control thread to execute vector memory instructions and also to amortize control overhead, shared loads, and shared computation. Regular DLP is encoded as vector-fetched coherent microthread instructions and irregular DLP is encoded with vector-fetched scalar branches. As with the other patterns, both SIMT and VT can include conditionally executed microthread instructions to help mitigate branch overhead.

Based on a high-level understanding of these design patterns it is possible to abstractly categorize the complexity required for mapping regular and irregular DLP. The MIMD and SIMT patterns are the most straight-forward, since the programmer's view is simply a collection of microthreads. Regular DLP maps easily to vector-SIMD and subword-SIMD, but irregular DLP can be much more challenging. Our hope is that mapping both regular and irregular DLP should be relatively straightforward for the VT pattern. However, there are still some complexities when using the VT pattern

including refactoring scalar microthread memory operations into vector memory instructions, and partitioning code into multiple vector-fetched blocks.

We can also qualitatively predict where a core using each pattern might fall in the energy-performance space first described in Figure 1.2. Compared to a single-threaded MIMD core, a multithreaded MIMD core will result in higher performance by hiding various execution latencies, but this comes at some energy cost due to the larger physical register file. Thread scheduling might either add overhead or reduce overhead if a simple scheduler enables less pipeline and bypassing logic. The general energy-performance trends should be similar for both regular and irregular DLP. A vector-SIMD core should achieve lower energy per task on regular DLP and higher performance due to control processor decoupling and hiding execution latencies through vector length. A multi-lane vector-SIMD core will improve throughput and potentially result in additional control amortization, but comes at the cost of increased cross-lane broadcast and VIU/VMU complexity. It is important to note that even a single-lane vector-SIMD core should improve performance at reduced energy per task on regular DLP. On irregular DLP the energy-performance advantage of the vector-SIMD pattern is much less clear. Very irregular DLP loops might need to be mapped to the control thread resulting in poor performance and energy-efficiency. Irregular DLP mapped with vector flags and vector conditional move instructions may come at energy costs proportional to the vector length. Density-time implementations have their own sources of overhead. The subword-SIMD pattern can be seen as a less effective version of the vector-SIMD pattern. Subword-SIMD has short vector lengths, may not support control thread decoupling, and has potentially even less support for irregular DLP so we can expect subword-SIMD to lie between MIMD and vector-SIMD in the energy-performance space. Although the SIMT pattern can perform better than the vector-SIMD pattern on irregular DLP, SIMT requires extra hardware to dynamically exploit regular DLP in the microthread instruction stream. Without a control thread there is less opportunity for amortizing overheads. Thus we can expect the SIMT pattern to perform better than vector-SIMD on very irregular DLP, but worse on regular DLP. The VT pattern attempts to preserve some of the key features of the vector-SIMD pattern that improve efficiency on regular DLP (e.g., control thread, vector memory instructions, and vector-fetched coherent microthread instructions) while also allowing a natural way to efficiently exploit irregular DLP with vector-fetched scalar branches.

2.9 Example Data-Parallel Accelerators

The previous sections discussed one or two accelerators which best exemplified each pattern including the Illinois Rigel 1000-core accelerator [KJJ+09], the Sun Niagara multithreaded processors [KAO05, NHW+07], the Berkeley Spert-II system with the T0 vector processor [WAK+96, Asa98], the IBM Cell subword-SIMD accelerator [GHF+06], the NVIDIA Fermi graphics processor [nvi09], and the Scale VT processor [KBH+04a, KBA08]. In this section, I briefly describe other data-parallel accelerators and then illustrate which set of patterns best characterize the pro-

grammer's logical view and implementation. As we will see, most accelerators actually leverage aspects from more than one pattern.

MIT Raw & Tilera TILE64 Processors – The MIT Raw processor is a classic example of the MIMD pattern with 16 simple RISC cores connected by multiple mesh networks [TKM+03]. Although Raw lacks dedicated hardware support for exploiting DLP, much of its evaluation workload included a combination of regular and irregular DLP [TLM+04, GTA06]. The commercial successor to Raw is the Tilera TILE64 processor with 64 simple cores also connected by multiple mesh networks [BEA+08]. TILE64 adds limited DLP support in the form of unified 32-bit subword-SIMD, which allows the primary scalar integer datapath to execute 4×8-bit, 2×16-bit, or 1×32-bit operations per cycle. TILE64 is targeted towards the video and network processing application domains. Both the Raw and TILE64 processors also include a novel statically ordered on-chip network. This network enables data to be efficiently streamed between cores and is particularly well suited to spatially mapping graphs of pipeline-parallel and task-parallel kernels. When viewed with this mechanism in mind, these processors resemble the systolic-array architectural pattern. The systolic-array pattern was not discussed in this chapter, but has been used for data-parallel accelerators in the past. This pattern streams data through an array of processing elements to perform the required computation.

Cisco CRS-1 Metro and Intel IXP-2800 Network Processors – Although network processors are less general than most of the accelerators discussed in this chapter, they are beginning to employ arrays of standard programmable processors. Packet processing is to some degree a data-parallel application, since an obvious approach is to partition incoming packets such that they can be processed in parallel. However, this application is a good example of highly irregular DLP, because packets arrived skewed in time and the processing per packet can vary widely. Thus it is not surprising that network processors have leaned towards the MIMD pattern with many simple scalar or multithreaded cores. The Intel IXP-2800 network processor includes an integrated general-purpose XScale processor for executing the host-thread and 16 multithreaded engines specialized for packet processing [int04]. The Cisco CRS-1 Metro network processor leverages 188 scalar Tensilica Extensa cores, with packets being distributed one per core [Eat05].

Alpha Tarantula Vector Processor – Tarantula was a industrial research project that explored adding vector-SIMD extensions to a high-performance Alpha processor with the specific purpose of accelerating data-parallel applications [EAE+02]. In contrast to the subword-SIMD extensions common in today's general-purpose processors [SS00], Tarantula was much closer to the vector-SIMD pattern. Although it was meant to be used with one core per chip, the sophisticated general-purpose control processor means it can be seen as a first step towards the full integration shown in Figure 2.1c. Tarantula has several interesting features including tight integration with an out-of-order superscalar multithreaded control processor, an address reordering scheme to minimize

memory bank conflicts, and extensive support for masked vector operations. The Tarantula instruction set includes 32 vector registers each with 128×64-bit elements. The vector unit uses 16 vector lanes and several independent vector functional units; the physical register file includes more than 32 vector registers to support the multithreaded control processor.

Stanford Imagine & SPI Storm-1 Stream Processors – Stream processors are optimized for a subset of data-parallel applications that can be represented as streams of elements flowing between computational kernels. These processors have a very similar implementation as the vector-SIMD pattern. The primary difference is an emphasis on a software-controlled memory hierarchy that enables stream processors to carefully manage bandwidth utilization throughout the memory system. Additionally, stream processors usually have a dedicated hardware interface between the general-purpose host processor and the accelerator's control processor. Irregular DLP is handled with vector masks or conditional streams (similar to vector compress and expand operations). The Stanford Imagine processor includes a small microcontroller as the control processor and eight vector lanes, and it supports a hardware vector length of eight [RDK+98]. The commercial successor to Imagine is the SPI Storm-1 processor which increases the number of vector lanes to 16 and integrates the general-purpose host processor on-chip [KWL+08]. Both processors contain vector functional units that are statically scheduled through a VLIW encoding and include a software exposed clustered vector register file. These processors include novel vector memory instructions that efficiently stage segments in a large software-controlled stream register file (similar to a backup vector register file). Data-parallel accelerators based on stream processors can be thought of as adhering to the main features of the vector-SIMD pattern, albeit with some extra features suitable for the more limited streaming programming model.

Berkeley VIRAM Vector Processor – The Berkeley VIRAM processor is a classic instance of the vector-SIMD pattern with four 64-bit vector lanes and a hardware vector length of 32 [Koz02, KP03]. It abstracts the hardware vector length with an instruction similar to `setvl`, and it includes vector flag, compress, and expand operations for conditional execution. However, VIRAM resembles the subword-SIMD pattern in one key aspect, since it allows software to specify the element width of vector operations. The vector length can thus range from 32×64-bit, to 64×32-bit, to 128×16-bit. Other large-scale vector-SIMD machines, such as the CDC STAR-100 and the TI ASC, have included a similar feature that enabled a 64-bit lane to be split into two 32-bit lanes [HP07, Appendix F.10].

NVIDIA G80 Family Graphics Processors – NVIDIA first proposed the SIMT pattern when introducing the G80 family of graphics processors as a more general platform for data-parallel acceleration [LNOM08]. Specifically, the GeForce 8800 GPU was one of the first GPUs to unify the various specialized graphics pipelines (e.g., vertex, geometry, and pixel pipelines) into a single unified programmable shader that could also be programmed for general compute operations with

a general-purpose programming language. The GeForce 8800's VIU is highly multithreaded to help hide various execution latencies. One of the key reasons the NVIDIA G80 family was quickly adopted for use as data-parallel accelerators was the high-level explicitly data-parallel CUDA programming methodology [NBGS08].

ATI Radeon R700 Family Graphics Processor – The ATI Radeon R700 family of graphics processors closely resembles the SIMT pattern [ati09]. One extension is that each microthread executes five-way VLIW instructions. As discussed in the SIMT pattern, no control thread is exposed and microthreads can execute control instructions which are managed by the VIU. Similar to the NVIDIA GPUs, the VIU is highly multithreaded to help hide various execution latencies. ATI's programming methodology was initially very different from NVIDIA's high-level CUDA framework. ATI released the raw hardware instruction set as part of its Close-to-the-Metal initiative [ati06], and hoped third-parties would develop the software infrastructure necessary for programmers to use the ATI GPUs as data-parallel accelerators. More recently, ATI has been leveraging OpenCL [ope08a] to simplify their programming methodology.

Intel Larrabee Graphics Processor – The future Intel accelerator known as Larrabee is initially planned for use in graphics processors, but Larrabee's programmable data-parallel cores makes it suitable for accelerating a wider range of data-parallel applications [SCS+09]. Larrabee includes tens of data-parallel cores each with its own 16-lane vector unit. Like vector-SIMD, it has fixed-width elements (32 b) and more sophisticated vector memory instructions to support indexed accesses. Like subword-SIMD, it has a fixed vector length exposed as part of the instruction set and includes vector permute instructions, which require significant inter-microthread communication. Larrabee includes vector flags for conditional execution. One interesting aspect of Larrabee is that it uses a four-way multithreaded control processor, and thus each control thread has its own independent set of vector registers. Even though it is common for some of the threads to be executing the same code (i.e., exploiting DLP across both the control threads *and* the microthreads), the Larrabee architects found it advantageous to choose multithreading over longer vector lengths to enable better handling of irregular DLP.

ClearSpeed CSX600 Processor – The CSX600 data-parallel accelerator includes a specialized control processor that manages an array of 96 SIMD processing elements [cle06]. A common front-end distributes identical instructions to all of the processing elements. Instead of vector memory instructions, each processing element can independently load from or store to its own private local memory. There is, however, support for an independently controlled direct-memory access engine which can stream data in to and out of a staging area. Irregular control flow is handled with a flag stack to simplify nested conditionals. The CSX600 is an instance of the distributed-memory SIMD (DM-SIMD) pattern not discussed in this chapter. As illustrated by the CSX600, The DM-SIMD pattern includes an amortized control unit with an array of processing engines similar to to vector

lanes, but each lane has its own large local memory.

TRIPS Processor – The TRIPS processor has a specific emphasis on flexibly supporting instruction-level, data-level, and task-level parallelism [SNL+03]. Although TRIPS has been specifically advocated as a data-parallel accelerator [SKMB03], it does not conveniently fit into any of the architectural patterns introduced in this chapter. It can support various types of parallelism, but it does so with coarse-grain modes, while the patterns in this chapter avoid modes in favor of intermingling regular and irregular DLP mechanisms. TRIPS lacks a control processor for amortizing control overheads, has nothing equivalent to vector memory operations, and relies on a small caching scheme to amortize instruction fetch for regular DLP. When in DLP mode, TRIPS is unable to support arbitrary control flow as in the SIMT and VT patterns. While TRIPS can flexibly support many different types of parallelism, the patterns in this chapter focus on exploiting DLP as the primary goal.

Chapter 3

Maven: A Flexible and Efficient Data-Parallel Accelerator

Maven is a new instance of the vector-thread (VT) architectural design pattern well suited for use in future data-parallel accelerators. Maven is a *malleable array of vector-thread engines* that can scale from a few to hundreds of flexible and efficient VT cores tiled across a single chip. As shown in Figure 3.1, each Maven VT core includes a control processor, a vector-thread unit, and a small L1 instruction cache. L2 cache banks are also distributed across the accelerator, and an on-chip network serves to connect the cores to the cache banks. Although the memory system and on-chip network are important aspects of the Maven accelerator, in this thesis, I focus on the instruction set, microarchitecture, and programming methodology of a single Maven VT core. These data-parallel cores are one of the key distinguishing characteristics of data-parallel accelerators as compared to general-purpose processors. In this chapter, I introduce the three primary techniques for building simplified instances of the VT pattern: a unified VT instruction set architecture, a single-lane VT microarchitecture based on the vector-SIMD pattern, and an explicitly data-parallel VT programming methodology. These simplified VT cores should be able to combine the energy-efficiency of SIMD accelerators with the flexibility of MIMD accelerators.

3.1 Unified VT Instruction Set Architecture

In the Maven unified VT instruction set, both the control thread and the microthreads execute the same scalar instruction set. This is significantly different from our earlier work on the Scale VT processor, which uses a specialized microthread instruction set. Although an elegant feature in itself, a unified VT instruction set can also simplify the microarchitecture and programming methodology. The microarchitecture can be more compact by sharing common functional units between the control thread and the microthreads, and the microarchitecture development can also be simplified by refactoring common hardware modules such as the instruction decode logic. A

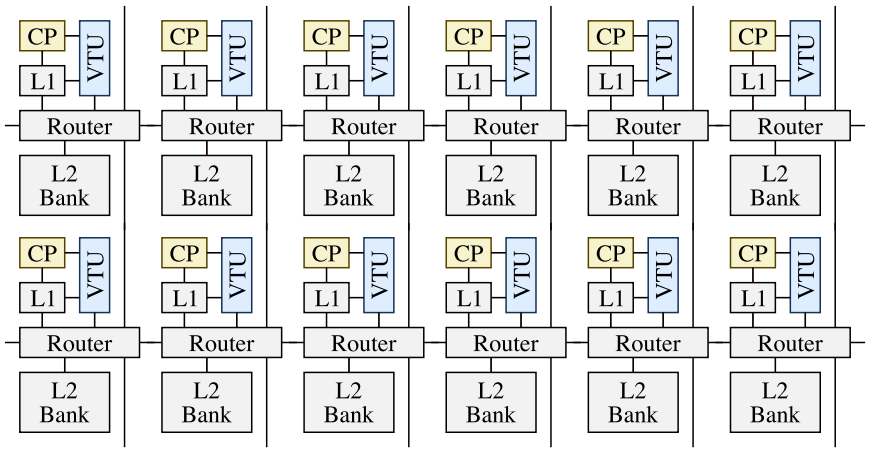


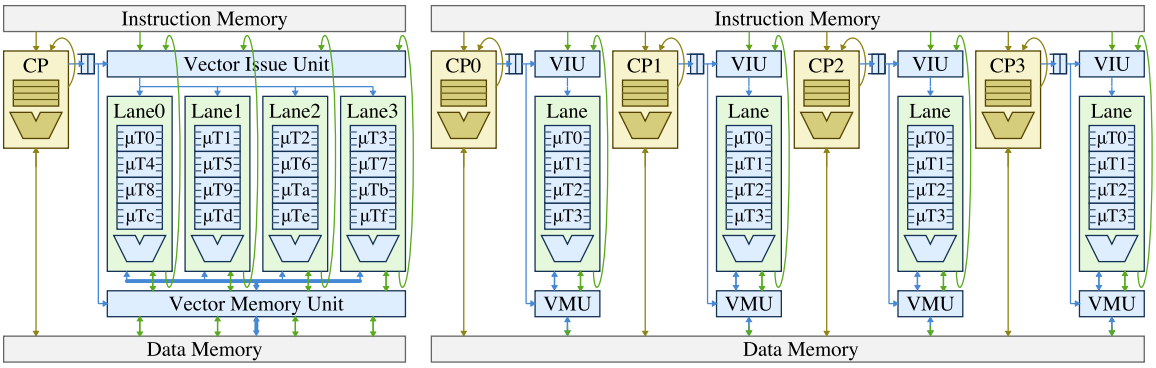
Figure 3.1: Maven Data-Parallel Accelerator – Maven is a malleable array of vector-thread engines with each Maven VT core including a control processor, a vector-thread unit, and a small L1 instruction cache. An on-chip network connects the cores to each other and to the shared L2 cache. (CP = control processor, VTU = vector-thread unit, L1 = private L1 instruction cache, L2 = shared L2 cache)

unified VT instruction set can be easier for programmers and compilers to reason about, since both types of thread execute the same kind of instructions. Similarly, a single compiler infrastructure can be used when compiling code for both types of thread, although code annotations might be useful for performance optimizations specific to one type of thread. For example, we might be able to improve performance by scheduling some instructions differently for the microthreads versus the control thread. This can also allow the same compiled code to be executed by both types of thread which leads to better code reuse and smaller binaries. Of course the instruction sets cannot be completely unified. The control thread instruction set is a proper superset of the microthread instruction set because microthreads are not able to execute vector instructions and system instructions.

There are, however, some challenges in designing a unified VT instruction set. Minimizing architectural state increases in importance, since it can significantly impact the supported number of microthreads. Even a well designed unified VT instruction set will unfortunately give up some opportunities for optimization via specialized control-thread and microthread instructions. For example, Scale provided multi-destination instructions that are only really suitable for its software-exposed clustered architecture. Scale also provided a specialized microthread control flow instruction that simplified hiding the branch resolution latency. As another example, we might choose to increase the number of control-thread scalar registers relative to the number of microthread scalar registers. This could improve control-thread performance on unstructured code with instruction-level parallelism.

3.2 Single-Lane VT Microarchitecture Based on Vector-SIMD Pattern

One of the first design decisions when applying the VT architectural design pattern to a large-scale accelerator is choosing the size of each data-parallel core. There is a spectrum of possible



(a) Single Core with 4-Lane VTU

(b) Four Cores Each with a Single-Lane VTU

Figure 3.2: Multi-Lane versus Single-Lane Vector-Thread Units – A single-core with a four-lane VTU has the same ideal vector compute and memory throughput as four cores each with a single-lane VTU. The multi-lane VTU might have better energy-efficiency and performance, since it can support longer hardware vector lengths (e.g., 16 µTs) compared to the single-lane VTUs (e.g., 4 µTs each), but single-lane VTUs can still be very competitive when executing regular DLP. The advantage of single-lane VTUs is that they are easier to implement than multi-lane VTUs especially with respect to efficiently executing irregular DLP. (CP = control processor, µT = microthread, VIU = vector issue unit, VMU = vector memory unit, VTU = vector-thread unit)

grain sizes from coarse-grain designs with few cores each containing many VT lanes and many microthreads per lane, to fine-grain designs with many cores each containing few VT lanes and few microthreads per lane. At the extremely coarse-grain end of the spectrum, we might use a single VT core with one control processor managing hundreds of lanes. Although such an accelerator would support very long hardware vector lengths, it would likely have *worse* energy-efficiency compared to multiple medium-sized VT cores. Control overheads amortized by the control processor and VIU would probably be outweighed by broadcasting control information to all lanes and by supporting a large crossbar in the VMU for moving data between memory and the lanes. At the extremely fine-grain end of the spectrum, we might use hundreds of cores each with a single lane and one microthread per lane. Of course with only one microthread per lane, there is little energy or performance benefit. For these reasons, we can narrow the design space to cores with a few lanes per VTU and a moderate number of microthreads per lane.

Figure 3.2 illustrates one possible core organization with four VT lanes and four microthreads per lane for a total hardware vector length of 16. This four-lane configuration is similar in spirit to the Scale VT processor. The VIU will broadcast control information to all four lanes, and the VMU includes a crossbar and rotation network for arbitrarily rearranging elements when executing vector loads and stores. To support efficient execution of indexed accesses (and to some extent strided accesses), the VMU should be capable of generating up to four independent memory requests per cycle, and the memory system should be able to sustain a corresponding amount of address and data bandwidth. The VIU and VMU become more complex as we increase the number of vector lanes

in a VT core. Because the VT pattern allows microthreads to diverge, the VIU will need to manage multiple instruction fetch streams across multiple lanes at the same time to keep the execution resources busy, and the VMU will need to handle arbitrary microthread loads/stores across the lanes. Since VT is explicitly focused on executing irregular DLP efficiently, we might also choose to implement some form of density-time execution to minimize overheads associated with inactive microthreads. All of these concerns suggest decoupling the VT lanes so that they no longer execute in lock step, but decoupled lanes require more control logic per lane.

Figure 3.2 illustrates a different approach where each core contains a single-lane VTU. Multi-lane VTUs use a spatio-temporal mapping of microthreads to lanes, while a single-lane VTU uses just temporal execution to achieve its performance and energy advantages. Trade-offs between multi-lane and single-lane cores in terms of flexibility, performance, energy-efficiency, design complexity, and area are briefly described below.

- **Flexibility –** Mapping highly irregular DLP or other forms of parallelism, such as task-level parallelism or pipeline parallelism, can be more difficult with multi-lane cores as opposed to single-lane cores. Because each single-lane core has its own control thread, the cores can operate independently from each other enabling completely different tasks to be easily mapped to different cores. Mapping completely different tasks to different microthreads within a multi-lane core will result in poor performance due to limited instruction bandwidth and the tight inter-lane coupling. Of course, we can map different tasks to different multi-lane cores, but then it is not clear how well we will be able to utilize the multi-lane VTUs. Single-lane cores move closer to the finer-grain end of the core-size spectrum increasing flexibility.
- **Performance –** A single-lane core will obviously have lower throughput than a multi-lane core, but as shown in Figure 3.2b, we can make up for this loss in performance by mapping an application across multiple cores each with its own single-lane VTU. We assume the multi-lane core has one address generator and address port per lane and that the number of functional units per lane is always the same, so a regular DLP application mapped to both the single four-lane core and the four single-lane cores in Figure 3.2 will have similar ideal arithmetic and memory throughput. An irregular DLP application might even have higher performance on the four single-lane cores, since the cores can run completely separately. A multi-lane core should support longer hardware vector lengths, but the marginal performance improvement decreases as we increase the vector length. For example, the memory system usually operates at a natural block size (often the L2 cache line size), so vector memory accesses longer than this block size might see less performance benefit. As another example, longer vector lengths refactor more work onto the control thread, but there is a limit to how much this can impact performance. Longer hardware vector lengths help decoupling, but the four single-lane cores have four times the control processor throughput. This can make up for the shorter vector lengths and provide

a similar amount of aggregate decoupling across all four control processors as compared to a multi-lane core with longer vector lengths. Density-time mechanisms can improve performance in both multi- and single-lane vector units at the expense of a more irregular execution. Such mechanisms are much easier to implement in single-lane vector units, since the irregular execution occurs in time as opposed to both time and space.

- **Energy-Efficiency –** Multi-lane cores amortize control overheads through spatial and temporal execution of the microthreads, while single-lane cores amortize control energy solely through temporal execution of the microthreads. The longer hardware vector lengths supported by a multi-lane core do help improve this control energy amortization, but there are several mitigating factors. First, the marginal energy-efficiency improvement decreases as we move to longer hardware vector lengths, since control energy is only one part of the total core energy. In addition, multi-lane VTUs have increased control-broadcast energy in the VIU and crossbar energy in the VMU. We can also always trade-off area for energy-efficiency by increasing the physical vector register file size in single-lane cores to enable greater amortization through temporal execution of more microthreads. This of course assumes the energy overhead of the larger register file does not outweigh the benefit of greater amortization.
- **Design Complexity –** Single-lane VTUs can be simpler to implement because there is no need for any tightly coupled inter-lane communication. Each core is designed independently, and then cores are tiled across the chip to form the full data-parallel accelerator. We may, however, want to group a few single-lane cores together physically to share local memory or on-chip network interfaces, but this does not affect the logical view where each core is a completely separate programmable entity with its own control thread. In addition, it is simpler to implement density-time execution in a single-lane VTU.
- **Area –** Probably the most obvious disadvantage of using multiple single-lane cores is the area overhead of providing a separate control processor per lane. We introduce a technique called *control processor embedding* that allows the control processor to share long-latency functional units and memory ports with the VTU. This significantly reduces the area overhead of including a control processor per vector lane. A unified instruction set helps enable control processor embedding, since both the control thread and the microthreads have identical functional unit requirements. It is important to note that an efficient multi-lane VTU, such as Scale, will mostly likely require decoupled lanes with more control logic per lane. This reduces the marginal increase in area from a multi-lane VTU to multiple single-lane VTUs.

Our interest in using single-lane VT cores for Maven is based on the assumption that a small increase in area is worth reduced design complexity, increased flexibility, equivalent (or greater) performance, and comparable energy-efficiency as compared to multi-lane VT cores.

We would like a single-lane Maven VT core to execute regular DLP with vector-like energy ef-

ficiency, so we have chosen to design the Maven VT core to be as similar as possible to a traditional vector-SIMD microarchitecture. This in contrast to the Scale VTU which had a unique microarchitecture specific to VT. Of course, Scale's unique microarchitecture enabled a variety of interesting features such as atomic instruction block interleaving, cross-microthread register-based communication, vector segment accesses, and decoupled clustered execution. By following the vector-SIMD pattern more closely, we can simplify the implementation and perform a detailed comparison of the exact mechanisms required to enable VT capabilities.

We add *vector fragments* to a vector-SIMD microarchitecture to handle vector-fetched arithmetic and control flow instructions. A fragment contains a program counter and a bit mask representing which microthreads are part of the fragment. A vector-SIMD microarchitecture always executes full vector operations that contain all microthreads, while a VT microarchitecture executes vector fragments that can contain a subset of the microthreads. Vector fragments are fundamentally different than the vector flag registers common in vector-SIMD implementations. A vector flag register simply contains a set of bits, while a vector fragment more directly represents the control flow for a subset of microthreads. Vector fragments, and the various mechanisms used to manipulate them, are the key to efficiently executing both regular and irregular DLP on a Maven VT core.

We can extend the basic vector fragment mechanism in three ways to help improve efficiency when executing irregular DLP. *Vector fragment merging* allows fragments to dynamically reconverge at run-time. *Vector fragment interleaving* helps hide the relatively long vector-fetched scalar branch latency as well as other execution latencies. *Vector fragment compression* helps eliminate the energy and performance overheads associated with inactive microthreads in a fragment. Adding support for vector fragments and these additional extensions is largely limited to modifying the VIU of a traditional vector-SIMD core. This allows such VT cores to maintain many of energy-efficiency advantages of a vector-SIMD core, while at the same time increasing the scope of programs that can be mapped to the core.

3.3 Explicitly Data-Parallel VT Programming Methodology

Although accelerators have often relied on either hand-coded assembly or compilers that can automatically extract DLP, there has been recent interest in explicitly data-parallel programming methodologies. In this approach, the programmer writes code for the host thread and explicitly specifies a data-parallel task that should be executed on all of the microthreads in parallel. As shown in Figure 3.3, there is usually a two-level hierarchy of the microthreads exposed to the programmer. This hierarchy enables more efficient implementations and is common across the patterns: in the MIMD pattern this corresponds to the set of microthreads mapped to the same core, in the SIMT pattern this corresponds to microthread blocks, and in SIMD and VT patterns this corresponds to the control threads. A key difference in the patterns is whether or not this two-level hierarchy is captured in the instruction set implicitly (as in the MIMD and SIMD patterns) or explicitly (as in

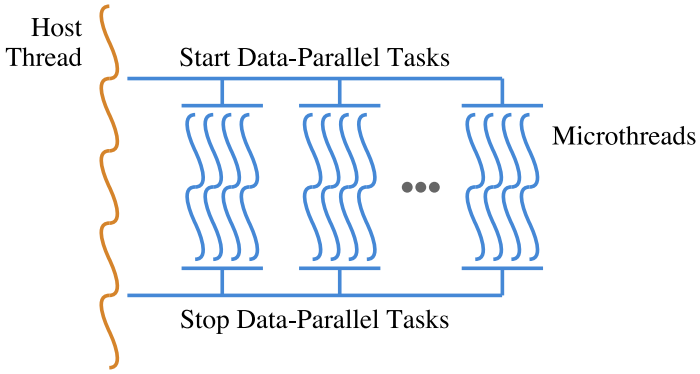


Figure 3.3: Explicitly Data-Parallel Programming Methodology – The host thread explicitly specifies data-parallel tasks and manages executing them on all the microthreads. Microthreads are implicitly or explicitly grouped into a two-level hierarchy. Start Data-Parallel Tasks

the SIMD and VT patterns). Any explicitly data-parallel programming methodology must expose the two-level hierarchy in some way for good performance, but with the SIMD and VT patterns the methodology must also provide a mechanism for generating high-quality code for the control threads.

In Maven, we leverage a slightly modified scalar compiler to generate efficient code for the control thread and microthreads, and we use a carefully written support library to capture the fine-grain interaction between the two types of thread. A unified VT instruction set is critical to leveraging the same scalar compiler for both types of thread. The result is a clean programming model that is considerably easier to implement than assembly programming, yet simpler to implement as compared to an automatic vectorizing compiler, such as the one used in Scale.

Chapter 4

Maven Instruction Set Architecture

This chapter describes Maven's unified VT instruction set and the design decisions behind various aspects of the instruction set. Section 3.1 has already motivated our interest in a unified VT instruction set. Section 4.1 gives an overview of the instruction set including the programmer's model, a brief description of all scalar and vector instructions, and two example assembly programs that will be referenced through the rest of the chapter. Section 4.2 goes into more detail about the specific challenges involved in unifying the control thread and microthread instruction sets. Sections 4.3–4.5 elaborate in more detail about certain aspects of the instruction set including the vector configuration process (Section 4.3), vector memory instructions (Section 4.4), and the register usage and calling convention used by Maven (Section 4.5). Section 4.6 discusses various instruction set extensions that can allow reasonable emulations of the MIMD, vector-SIMD, and SIMT architectural design patterns. The chapter concludes with future research directions (Section 4.7) and related work (Section 4.8).

4.1 Instruction Set Overview

As with all VT architectures, the programmer's logical view of a single Maven VT core consists of a control thread which manages an array of microthreads (see Figure 4.1). A unique feature of the Maven instruction set compared to earlier VT architectures is that both the control thread and the microthreads execute a nearly identical 32-bit scalar RISC instruction set. Maven supports a full complement of integer instructions as well as IEEE compliant single-precision floating-point instructions. The control thread includes a program counter, 32 general-purpose registers, 4–32 vector registers, and several vector control registers. Scalar register number zero and vector register number zero are always fixed to contain the value zero. Each microthread has its own programmer counter and one element of the vector registers, meaning that each microthread effectively has 4–32 scalar general-purpose registers. Since vector register number zero is fixed to contain the value zero, each microthread's scalar register number zero is also fixed to contain the value zero. The three

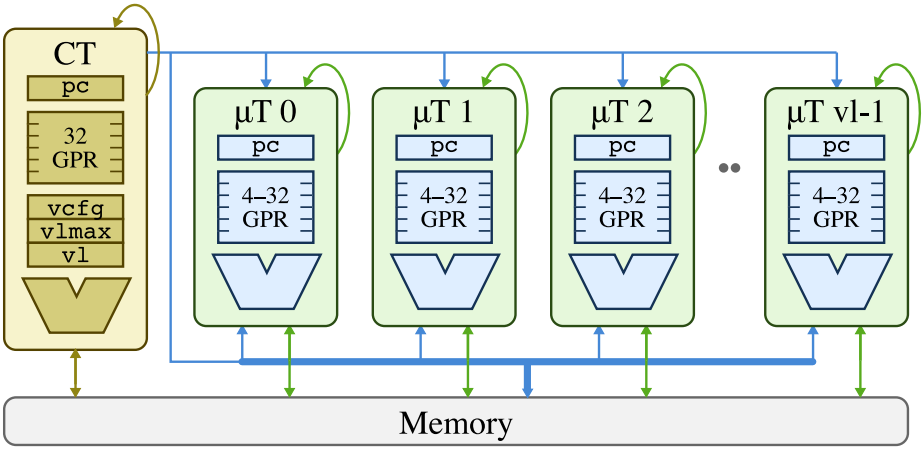


Figure 4.1: Maven Programmer's Logical View – The control thread includes 32 general-purpose registers and a standard MIPS-like instruction set with support for both integer and single-precision floating-point arithmetic. Vector configuration instructions read and write three control registers (`vl`, `vlmax`, `vcfg`) which change the number of available and active microthreads. Each microthread has its own program counter (`pc`) and can execute the same set of instructions as the control thread except for vector and system instructions. (CT = control thread, µT = microthread, GPR = general purpose register)

vector control registers (`vcfg`, `vlmax`, `vl`) change the number of available and active microthreads (see Section 4.3 for more details).

The Maven instruction set could leave the hardware vector length completely unbounded so that implementations could provide very short hardware vector lengths or very long hardware vector lengths without restriction. There are, however, some benefits to defining a minimum and maximum hardware vector length as part of the instruction set. A programmer can avoid stripmining overhead if the application vector length is statically known to be smaller than the minimum hardware vector length, and a programmer can statically reserve memory space for a hardware register if there is a maximum hardware vector length. For these reasons, Maven specifies a minimum hardware vector length of four and a maximum hardware vector length of 64.

The Maven programmer's model makes no guarantees concerning the execution interleaving of microthreads. Microthreads can be executed in any order and any subset of microthreads can be executed in parallel. Software is also not allowed to rely on a consistent microthread execution interleaving over multiple executions of the same code. Avoiding any execution guarantees enables both spatial and temporal mapping of microthreads and also allows microthread mappings to change over time. Although the Section 3.2 made a case for single-lane VTUs, future implementations of the Maven instruction set are free to also use multi-lane VTUs. An example microthread instruction interleaving for a two-lane, four-microthread VTU is shown in Figure 2.13b. The figure illustrates how µT0–1 execute in parallel, µT2–3 execute in parallel, and µT0–1 execute before µT2–3. Figure 2.13b implies that a vector fetch or other vector instruction acts as a barrier across the lanes, but this is overly restrictive and Maven does not guarantee this behavior. An implementation with de-

coupled lanes might allow each lane to execute very different parts of the program at the same time. The programmer can, of course, rely on the fact that the vector store on line 9 will correctly read the result of the vector-fetched multiply and addition on lines 20–21 for each microthread. In other words, microthreads execute in program order including the effect of vector memory operations on that microthread.

Although there are no guarantees concerning the microthread execution interleaving, the control thread is guaranteed to execute in parallel with the microthreads. For example, Figure 2.13b illustrates the control thread executing the loop control overhead in parallel with the microthreads executing the multiply and addition operations. Software is allowed to explicitly rely on this parallel execution to enable communication between the control thread and microthreads while they are running. This enables more flexible programming patterns such as a *work-queue pattern* where the control thread distributes work to the microthreads through a work queue in memory or an *early-exit pattern* where one control thread might notify other control threads through memory to abort the computation running on their microthreads. A disadvantage of guaranteeing parallel control-thread execution, is that it eliminates the possibility of a completely sequential implementation. A sequential implementation executes the control thread until a vector fetch instruction, sequentially executes each microthread until completion, and then resumes execution of the control thread. Compared to sequential implementations, parallel control-thread execution has higher performance because it enables control-thread decoupling, and it also enables more flexible programming patterns making it a more attractive option.

Table 4.1 shows the scalar instruction set used by both the control thread and microthreads. The control thread can execute all listed instructions (including the microthread specific instructions), while the microthreads can execute all listed instructions except for control-thread specific instructions. Most of the listed instructions' function and encoding are based on the MIPS32 instruction set [mip09, Swe07], with the exceptions being the floating-point instructions and those instructions marked by an asterisk. Maven unifies the MIPS32 integer and floating-point registers into a single register space with 32 general-purpose registers. All floating-point instructions can read and write any general-purpose register, and floating-point comparisons target a general-purpose register instead of a floating-point condition-code register. These modifications required small changes to the floating-point instruction encoding format. Maven also eliminates the special `hi` and `lo` registers for integer multiply and division instructions. Instead, Maven provides integer multiply, divide, and remainder instructions that read and write any general-purpose register. Maven also provides atomic memory operations instead of using the standard MIPS32 load-linked and store-conditional instructions. The `stop` and `utidx` instructions have special meaning when executed by a microthread: a vector fetch starts execution of a microthread and the `stop` instruction stops execution of a microthread; the `utidx` instruction writes the index of the executing microthread to a general purpose register. The microthread index can be used to implement indexed vector memory accesses or to en-

Scalar Integer Arithmetic Instructions	
`addu, subu, addiu`	Integer arithmetic
`and, or xor, nor, andi, ori, xori`	Logical operations
`slt, sltu, slti, sltiu`	Set less than comparisons
`mul, div*, divu*, rem*, remu*`	Long-latency integer arithmetic
`lui`	Load upper 16 bits from immediate
`sllv, srlv, srav, sll, srl, sra`	Left and right shifts
Scalar Floating-Point Arithmetic Instructions	
`add.s, sub.s, mul.s, div.s`	Floating-point arithmetic
`abs.s, neg.s, sqrt.s`	
`round.w.s, trunc.w.s, ceil.w.s, floor.w.s`	Floating-point to integer conversions
`cvt.s.w`	Integer to floating-point conversion
`c.f.s, c.un.s, c.eq.s, c.ueq.s, c.olt.s`	Floating-point comparisons
`c.ult.s, c.ole.s, c.ule.s, c.sf.s`	
`c.seq.s, c.ngl.s, c.lt.s, c.le.s, c.ngt.s`	
Scalar Memory Instructions	
`lb, lh, lw, lbu, lhu`	Load byte, halfword, word (signed/unsigned)
`sb, sh, sw`	Store byte, halfword, word
Scalar Control Instructions	
`beq, bne, bltz, bgez, blez, bgtz`	Conditional branches
`j, jr, jal, jalr`	Unconditional jumps
`movz, movn`	Conditional moves
Scalar Synchronization Instructions	
`sync`	Memory fence
`amo.add*, amo.and*, amo.or*`	Atomic memory operations
Control-Thread Specific Instructions	
`mfc0, mtc0`	Move to/from coprocessor 0 registers
`syscall, eret`	System call, return from exception
Microthread Specific Instructions	
`stop*`	Stop microthread
`utidx*`	Write microthread index to destination

Table 4.1: Maven Scalar Instructions – This is a relatively standard RISC instruction set. Non-floating-point instructions without an asterisk are similar in function and encoding to the corresponding instructions in the MIPS32 instruction set [mip09, Swe07]. Floating-point instructions use the same set of general-purpose registers as integer operations, and floating-point comparison instructions target a general-purpose register instead of a floating-point condition code register. Instructions with an asterisk are specific to Maven and include divide and remainder instructions that target a general-purpose register and atomic memory operations. The control thread can execute all listed instructions including the microthread specific instructions; the `stop` instruction is a nop and the `utidx` instruction always returns -1. Microthreads can execute all but the control-thread specific instructions.

Vector Configuration Instructions				
`vcfgivl`	r_{dst},	r_{len},	n_{vreg}	Set `vlmax` based on n_{vreg} vector regs, then execute `setvl`
`vcfgivli`	r_{dst},	n_{len},	n_{vreg}	Set `vlmax` based on n_{vreg} vector regs, then execute `setvli`
`setvl`	r_{dst},	r_{len}		Set `vl` based on app vector length r_{len}, then set r_{dst} = min(`vl`,r_{len})
`setvli`	r_{dst},	n_{len}		Set `vl` based on app vector length n_{len}, then set r_{dst} = min(`vl`,n_{len})
Vector Unit-Stride & Strided Memory Instructions				
`l{w,h,b}.v`	R_{dst},	r_{base}		Load vector reg R_{dst} from mem[r_{base}] with unit-stride
`l{hu,bu}.v`	R_{dst},	r_{base}		(unsigned and signed variants)
`l{w,h,b}st.v`	R_{dst},	r_{base},	r_{stride}	Load vector reg R_{dst} from mem[r_{base}] with stride r_{stride}
`l{hu,bu}st.v`	R_{dst},	r_{base},	r_{stride}	(unsigned and signed variants)
`s{w,h,b}.v`	R_{src},	r_{base}		Store vector reg R_{src} to mem[r_{base}] with unit-stride
`s{w,h,b}st.v`	R_{src},	r_{base},	r_{stride}	Store vector reg R_{src} to mem[r_{base}] with stride r_{stride}
Vector Memory Fence Instructions				
`sync.{l,g}`				Memory fence between executing thread's scalar loads and stores
`sync.{l,g}.v`				Memory fence between vector ld/st, μT ld/st
`sync.{l,g}.cv`				Memory fence between CT scalar ld/st, vector ld/st, μT ld/st
Vector Move Instructions				
`mov.vv`	R_{dst},	R_{src}		Move vector reg R_{src} to vector reg R_{dst}
`mov.sv`	R_{dst},	r_{src}		Move control thread reg r_{src} to all elements of vector reg R_{dst}
`mtut`	R_{dst},	r_{utidx},	r_{src}	Move control thread reg r_{src} to element r_{utidx} of vector reg R_{dst}
`mfut`	R_{src},	r_{utidx},	r_{dst}	Move element r_{utidx} of vector reg R_{src} to control thread reg r_{src}
Vector Fetch Instructions				
`vf`	label			Start microthreads executing instructions at label
`vfr`	r_{addr}			Start microthreads executing instructions at address in reg r_{addr}

Table 4.2: Maven Vector Instructions – As with all VT instruction sets, Maven includes vector configuration and memory instructions as well as vector-fetch instructions. (CT = control-thread, μT = microthread, r = scalar register specifier, R = vector register specifier, n = immediate value, {`w,h,b`} = word, halfword, and byte vector memory variants, {`hu,bu`} = unsigned halfword and unsigned byte vector memory variants, {`l,g`} = local and global synchronization variants)

able different microthreads to execute different code based on their index. To facilitate code reuse, the control thread can also execute these two instructions: the `stop` instruction does nothing and the `utidx` instruction always returns -1 enabling software to distinguish between when it is running on the control thread and when it is running on a microthread. Instructions unique to Maven are encoded with an unused primary MIPS32 opcode and a secondary opcode field to distinguish instructions.

In addition to these differences from MIPS32, Maven has no branch delay slot and excludes many instructions such as branch-likely instructions, branch-and-link instructions, test-and-trap instructions, unaligned loads and stores, merged multiply-accumulates, rotates, three-input floating-point instructions, and bit manipulation instructions such as count leading zeros and bit-field ex-

traction. Some of these instructions are deprecated by MIPS32 (e.g., branch-likely instructions), some are not very useful with our C/C++-based programming methodology (e.g., branch-and-link instructions), and a few might be useful in future versions of the Maven instruction set (e.g., merged multiply-accumulates).

Table 4.2 shows the vector instructions that can be executed by the control thread to manage the array of microthreads. The vector configuration instructions enable reading and writing vector control registers that change the number of active microthreads and are described in more detail in Section 4.3. Maven includes a standard set of vector load/store instructions and vector memory fences, which are described in more detail in Section 4.4. Although it is possible to move vector registers with a vector-fetched scalar move instruction, a dedicated `mov.vv` instruction simplifies implementing vector register allocation in the Maven programming methodology. It is also possible to require that data be moved from the control-thread scalar registers to vector registers through memory, but it can be much more efficient to provide dedicated instructions such as `mov.sv` which copies a control-thread scalar register value to every element of a vector register and `mtut`/`mfut` (move to/from microthread) which transfers data between a control-thread scalar register and a single element in a vector register. Maven provides two vector fetch instructions with different ways of specifying the target address from which microthreads should begin executing: the `vf` instruction includes a 16-bit offset relative to the `vf` instruction's program counter, and the `vfr` instruction reads the target address from a general-purpose register. The Maven vector instructions are encoded with several unused primary MIPS32 opcodes (for the vector load/store instructions and the `vf` instruction) or with a MIPS32 coprocessor 2 primary opcode and a secondary opcode field.

The assembly code for two small examples is shown in Figure 4.2. These examples correspond to the regular DLP loop in Table 2.1c and the irregular DLP loop in Table 2.1f, and both examples were compiled using the Maven programming methodology described in Chapter 6. The assembly code is similar in spirit to the pseudo-assembly shown in Figures 2.13 and 2.14, but of course the real assembly code corresponds to an actual Maven binary. These examples will be referenced throughout the rest of this chapter.

4.2 Challenges in a Unified VT Instruction Set

Minimizing architectural state in a unified VT instruction set is critical, since every microthread will usually need its own copy of this state. For example, the base MIPS32 instruction set includes special 32-bit `hi` and `lo` registers. The standard integer multiply instruction reads two 32-bit general-purpose registers and writes the full 64-bit result to the `hi` and `lo` registers. The standard integer divide instruction also reads two 32-bit general-purpose registers and writes the 32-bit quotient into the `lo` register and the 32-bit remainder into the `hi` register. These special registers enable multiply and divide instructions to write two registers without an extra write port in the general-

```
19c: vcfgivl v0, a3, 5
1a0: mov.sv  VA0, a4
1a4: blez    a3, 1e0
1a8: move    v0, zero

1ac: subu    a4, a3, v0
1b0: setvl   a4, a4
1b4: sll     v1, v0, 2
1b8: addu    a6, a1, v1
1bc: addu    a5, a2, v1
1c0: lw.v    VV0, a6
1c4: lw.v    VV1, a5
1c8: vf      160 <ut_code>
1cc: addu    v0, v0, a4
1d0: addu    v1, a0, v1
1d4: slt     a4, v0, a3
1d8: sw.v    VV0, v1
1dc: bnez    a4, 1ac

1e0: sync.l.cv
1e4: jr      ra

ut_code:
160: mul     v0, a0, v0
164: addu    v0, v0, v1
168: stop
```

(a) Regular DLP Example

```
1b0: vcfgivl v0, a3, 7
1b4: mov.sv  VA1, a4
1b8: blez    a3, 1f4
1bc: move    v0, zero

1c0: subu    a4, a3, v0
1c4: setvl   a4, a4
1c8: sll     v1, v0, 2
1cc: addu    a5, a0, v1
1d0: mov.sv  VA0, a5
1d4: addu    a5, a2, v1
1d8: addu    v1, a1, v1
1dc: lw.v    VV0, v1
1e0: lw.v    VV1, a5
1e4: vf      160 <ut_code>
1e8: addu    v0, v0, a4
1ec: slt     v1, v0, a3
1f0: bnez    v1, 1c0

1f4: sync.l.cv
1f8: jr      ra

ut_code:
160: blez    v0, 17c
164: mul     v0, a1, v0
168: utidx   a2
16c: sll     a2, a2, 2
170: addu    a2, a0, a2
174: addu    v1, v0, v1
178: sw      v1, 0(a2)
17c: stop
```

(b) Irregular DLP Example

Figure 4.2: Example Maven Assembly Code – Assembly code generated from Maven programming methodology discussed in Chapter 6 for (a) regular DLP loop corresponding to Table 2.1c and (b) irregular DLP loop corresponding to Table 2.1f. (`$i` = scalar register `i`, `$vi` = vector register `i`. Each line begins with instruction address. Assume following register initializations: `a0` = base pointer for array `C`, `a1` = base pointer for array `A`, `a2` = base pointer for array `B`, `a3` = size of arrays, `a4` = scalar value `x`. See Figures 4.1 and 4.2 for a list of all Maven instructions. Section 4.5 discusses the register usage and calling conventions.)

purpose register file. MIPS32 also includes a separate 32-entry floating-point register space and eight floating-point condition-code registers. Separate `hi`, `lo`, and floating-point registers allow the long-latency integer multiplier, integer divider, and floating-point units to run decoupled from the primary integer datapath without additional write ports or any sophisticated write-port arbitration, which would be required with a unified register file. Unfortunately, these separate registers also more than double the amount of architectural state compared to the integer registers alone. As we will see in Chapter 7, the register file is a significant portion of the overall core area so the additional architectural state translates into less microthreads for a given area constraint and thus shorter hardware vector lengths.

Maven unifies MIPS32's 32 integer registers, `hi` and `lo` registers, 32 floating-point registers, and eight floating-point condition code registers into a single 32-entry general-purpose register space. These changes require new integer multiply, divide, and remainder instructions that write a single general-purpose register. These new instructions are more limited (i.e., multiplies produce a 32-bit result instead of a 64-bit result and two instructions are required to calculate both the quotient and remainder), but they are actually a better match for our C/C++-based programming methodology. A more important issue with a unified register space is that it complicates decoupling long-latency operations, but earlier studies [Asa98] and our own experiments have indicated that the overhead is relatively small for moving from a 2-read/1-write register file to a 2-read/2-write register file. By adding a second write port we can simplify decoupled implementations, yet still support a unified register space. However, the dependency checking logic is still more complicated to efficiently utilize multiple functional units with varying latencies and a unified register space. Maven also avoids instructions that have additional implicit architectural state such as the `cvt.w.s` instruction, which relies on a floating-point control register to determine the rounding mode. Maven instead supports the standard MIPS32 conversion instructions which explicitly specify the rounding mode in the opcode (e.g., `round.w.s`).

There are some trade-offs involved in choosing to base the Maven unified scalar instruction set on a standard RISC instruction set such as MIPS32, particularly with respect to control flow instructions. MIPS32 includes a single-cycle branch-delay slot which Maven eliminates to simplify future implementations which may leverage more sophisticated issue logic (e.g., limited superscalar issue or even out-of-order issue). The control processor branch resolution latency is relatively short, so a simple branch predictor can effectively help hide this latency. Unfortunately, the branch resolution latency for vector-fetched scalar branches can be quite long, which also implies a single-cycle branch-delay slot would be less effective. Figure 2.14b shows how the VIU must wait for all microthreads to resolve the branch, and this latency will increase with more microthreads per lane. Depending on the specifics of the actual implementation, this long branch resolution latency could significantly degrade performance, especially for situations with few instructions on either side of the branch.

One standard architectural technique is to replace some branches with *conditional execution* (e.g., predication). Conditionally executed instructions are always processed but only take effect based on additional implicit or explicit architectural state. Essentially conditional execution turns control flow into data flow; more instructions need to be processed, but a branch instruction is no longer needed. There are many conditional execution mechanisms including: *full predication* which provides many explicit predication registers and any instruction can be conditionally executed; *partial predication* which provides fewer predication registers (or possibly one implicit predication register) and only a subset of the instructions can be conditionally executed; *conditional move* which provides an instruction that conditionally copies a source register to a destination register usually based on the contents of a third general-purpose register; *conditional select* which provides an instruction that copies one of its two sources to a destination register usually based on a separate predicate register; and *conditional store* which provides a special store instruction which conditionally writes to memory based on an additional predicate register. Several studies have illustrated that more extensive conditional execution mechanisms have higher implementation cost but can also result in higher performance [SFS00, MHM+95]. Replacing highly divergent branches with conditional execution can also help energy-efficiency by maintaining microthread coherence. Maven supports the standard MIPS32 conditional execution instructions: `movz` conditionally moves a value from one general-purpose register to another if a third general-purpose register is zero, and `movn` does the same if the third general-purpose register is non-zero. Even though more sophisticated conditional execution mechanisms would probably result in higher performance, Maven includes just these two conditional move instructions because they are part of the standard MIPS32 instruction set, are straight-forward to implement, and are already supported by standard MIPS32 compilers.

4.3 Vector Configuration Instructions

Maven uses a flexible vector configuration mechanism that enables the same implementation to support few microthreads each with many registers or many microthreads each with few registers. Figure 4.3 illustrates the general concept for an implementation similar to the one shown in Figure 2.12b. This implementation includes a total of 128 physical registers in the vector-thread unit (VTU). The standard Maven unified scalar instruction set requires 32 registers per microthread meaning that this implementation can support a minimum of four microthreads (see Figures 4.3a and 4.3e). If the programmer can statically determine that all 32 registers are not needed for a portion of the code, then the VTU can be configured with less registers per microthread. For example, the same implementation can support twice as many microthreads (i.e., twice the vector length) if software only requires 16 registers per microthread (see Figures 4.3b and 4.3f). Maven allows software to configure between four and 32 registers per microthread meaning that the maximum number of microthreads for this example is 32 (see Figures 4.3d and 4.3h). The available logical register numbers always start at zero and increase consecutively. For example, with four registers

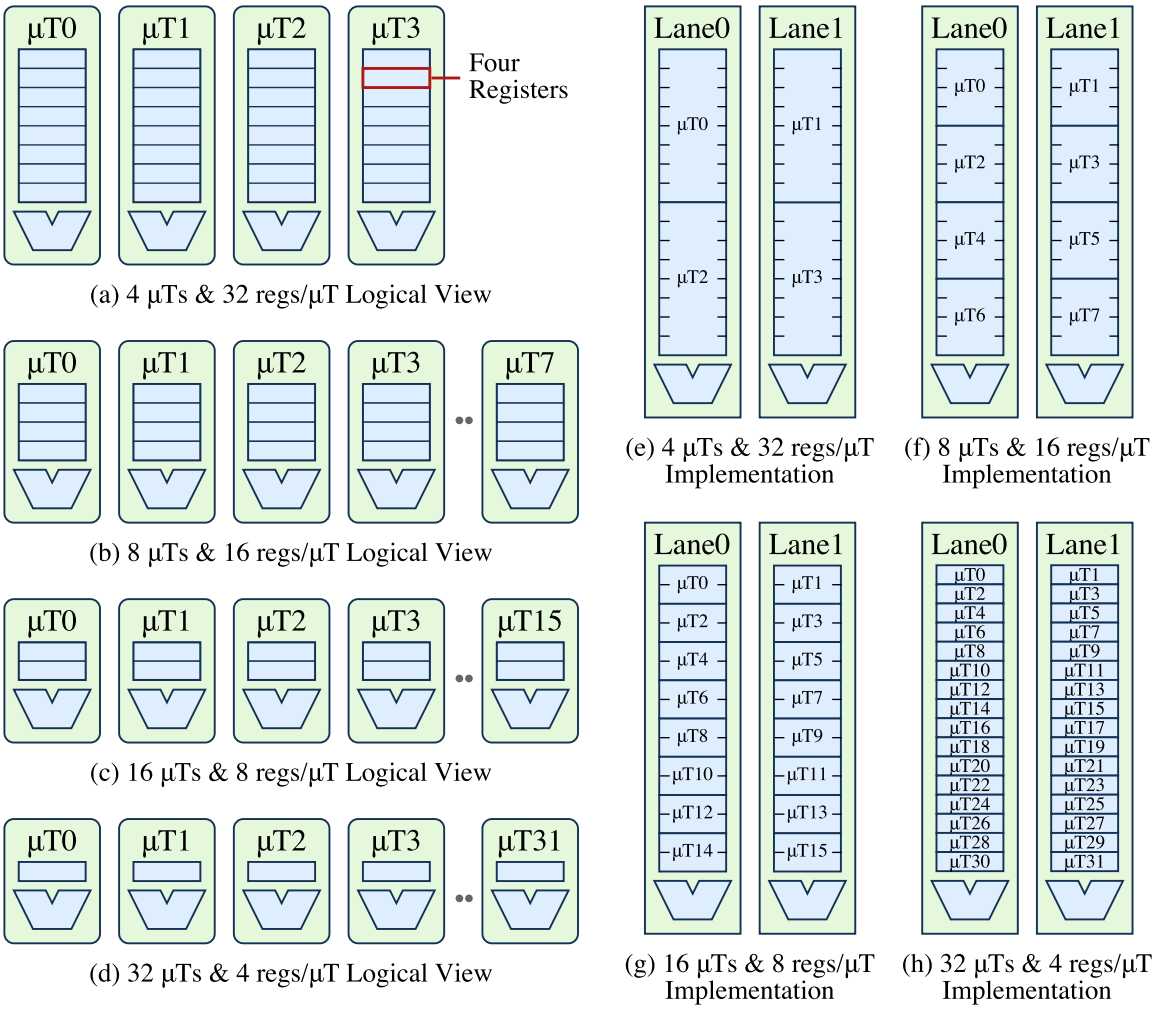


Figure 4.3: Examples of Various Vector Configurations – The same Maven implementation can support few microthreads each with many registers (a,e) or many microthreads each with few registers (d,h). Register number zero is shown as a normal register for simplification, but an actual implementation would exploit the fact that register number zero always contains the value zero such that there is no need to provide multiple copies of this register. (µT = microthread)

per microthread the logical registers numbers available for use are 0–4. This implies that register zero is always available. The results of accessing an unavailable register is undefined. Reconfiguring the VTU clobbers all vector registers, so software must save to memory any values that should be live across a reconfiguration. Reconfiguration can also require dead-time in the VTU while instructions prior to the `vcfgivl` instruction drain the pipeline. Thus reconfiguration should be seen as a coarse-grain operation done at most once per loop nest. Reconfigurable vector registers have some significant implications for the register usage and calling conventions, which are discussed in Section 4.5.

The example in Figure 4.3 does not exploit the fact that vector register number zero always

contains the value zero. An actual implementation only needs 124 physical registers to support four microthreads with each microthread having all 32 registers that are part of Maven's unified scalar instruction set. Since each microthread does not need its own copy of register number zero, the 124 physical registers can sometimes support more microthreads than pictured. For example, with four registers per microthread (one of which is register number zero) the implementation can actually support 41 microthreads ($41 = \lceil 124/(3-1) \rceil$) as opposed to the 32 microthreads shown in Figures 4.3d and 4.3h.

The example assembly code in Figure 4.2 illustrates how the Maven configuration mechanism is exposed to software through the `vcfgivl` instruction on line 1. As shown in Table 4.2, the `vcfgivl` instruction has two sources and one destination. The second source is an immediate specifying the number of registers required per microthread. The regular DLP example in Figure 4.2a requires five registers per microthread while the irregular DLP example in Figure 4.2b requires seven registers per microthread. Notice the vector register numbers and microthread scalar registers numbers do not exceed these bounds. When executing a `vcfgivl` instruction, the hardware will write the number of registers per microthread into the `vcfg` control register and also write the number of microthreads that are correspondingly available (i.e., the maximum vector length) into the `vlmax` control register. The value for `vlmax` is simply $\texttt{vlmax} = \lceil N_{physreg}/(N_{utreg}-1) \rceil$ where $N_{physreg}$ is the number of physical registers and N_{utreg} is the number of registers requested per microthread via the `vcfgivl` instruction. Implementations can take advantage of the fact that the regular DLP example requires less registers per microthread to support longer vector lengths, and this can result in higher performance and better energy efficiency.

In addition to configuring the number of available microthreads, the `vcfgivl` instruction also performs the same operation as a `setvl` instruction which was briefly described in Section 2.7. As shown in Table 4.2, the `setvl` instruction has one source (the application vector length) and one destination (the resulting number of active microthreads). As a side effect, the `setvl` and `vcfgivl` instructions also write the number of active microthreads into the `vl` control register. The value for the number of active microthreads is $\texttt{vl} = \max(\texttt{vlmax}, N_{utapp})$ where N_{utapp} is the application vector length (i.e., the total number of microthreads required by the application). As the examples in Figure 4.2 illustrate, the `setvl` instruction is used for stripmining so that the loops can process `vl` elements at a time without statically knowing anything about the physical resources available in the VTU. Integrating the functionality of `setvl` into the `vcfgivl` instruction avoids having to use two instructions for what is an very common operation, and thus should help reduce overhead especially for loops with short application vector lengths.

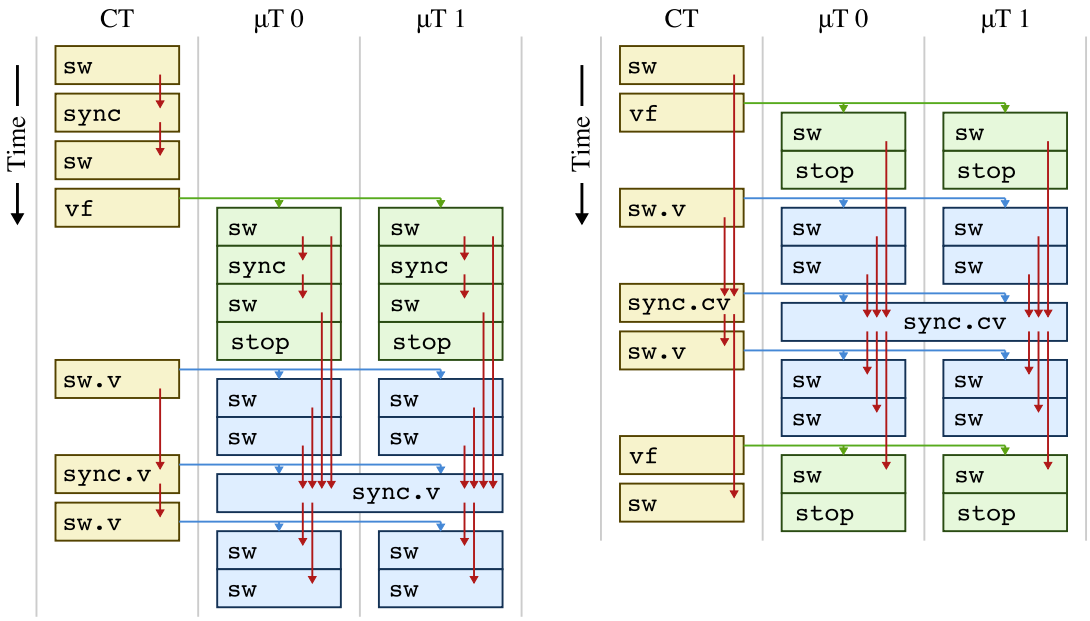
The `setvl` and `vcfgivl` instructions enable software to flexibly configure the physical VT resources in two dimensions: the number of microthreads and the number of registers per microthread. This gives a great deal of freedom in the implementation in terms of the number of lanes and physical registers while still allowing the same compiled binary to efficiently run on all variations.

4.4 Vector Memory Instructions

Maven includes vector instructions for handling unit-stride and strided accesses to vectors of words, halfwords, and bytes (see Table 4.2). These instructions are very similar to those found in other vector-SIMD instruction sets. Figure 4.2a illustrates using unit-stride vector loads and stores on lines 11–12,17. Most vector-SIMD instructions sets also include indexed vector loads/stores which use a control-thread scalar register as a base address and a vector register to indicate an offset for each accessed element. Maven achieves a similar effect with vector-fetched scalar accesses. Each microthread computes its own address and then simply uses a standard scalar load/store (e.g., `lw`/`sw`). Maven could additionally provide dedicated indexed vector loads/stores, but unlike unit-stride and strided accesses there is little regularity to exploit for performance or energy-efficiency. Regardless of their encoding, indexed accesses cannot be decoupled effectively, since the addresses are not known early enough in the pipeline, and they cannot be transferred in efficient blocks. It is also important to note that vector-fetched scalar loads/stores are more general than indexed vector accesses, since they can also implement conditional vector loads/stores as illustrated in Figure 4.2b on line 30.

As discussed in Section 4.1, there are few constraints on the execution interleaving of the control thread and microthreads, and each Maven core is meant to be used in a full data-parallel accelerator with many independent cores. Thus it is important for the Maven instruction set to provide various memory fence instructions to constrain load/store ordering and prevent unwanted race conditions. Table 4.2 shows the three types of Maven memory fences: `sync`, `sync.v`, and `sync.cv`. Each memory fence type has a local (`.l`) and global variant (`.g`). Local memory fences only enforce ordering as viewed by the control thread and microthreads on a single core, while global memory fences enforce ordering as viewed by all control threads and microthreads in the data-parallel accelerator. Notice that the `sync.g` instruction is an alias for the standard MIPS32 `sync` instruction.

Figure 4.4 illustrates the three types of memory fences with an abstract execution diagram. Unlike the execution diagrams presented earlier, this diagram does not attempt to capture the details of the implementation but instead tries to express the execution at the semantic level. A `sync` fence is the least restrictive and ensures that all scalar loads/stores before the fence are complete before any scalar loads/stores after the fence. In Figure 4.4a, a `sync` fence is used to order two consecutive stores on the control thread, and it is also used to order two consecutive stores on a microthread. A `sync` fence on one microthread does not constrain any other microthread. A `sync.v` fence ensures that all vector loads/stores *and* microthread scalar loads/stores before the fence are complete before any similar operations after the fence. In Figure 4.4a, a `sync.v` fence is used to order two consecutive vector stores, and it also ensures that the microthread stores are complete before the final vector store. These kind of fences are useful when mapping inter-microthread communication through memory. For example, to permute a vector stored in a vector register we can use vector-fetched scalar stores to place each element in the proper location in a temporary memory buffer



(a) Memory Fence Example for `sync` & `sync.v`

(b) Memory Fence Example for `sync.cv`

Figure 4.4: Memory Fence Examples – These are abstract execution diagrams showing how the control thread and two microthreads execute logically with respect to various memory fences. Memory fences introduce additional dependencies between various load and store; these dependencies are shown with red arrows. Diagrams apply equally well for both the local and global memory fence variants and if any store is replaced with a load. (CT = control thread, µT = microthread)

before reloading with a unit-stride vector load. A `sync.l.v` fence is required in between the vector-fetched scalar stores and the unit-stride vector load. Otherwise the lack of any inter-microthread execution guarantees (even with respect to vector memory instructions) might allow the unit-stride load to read an older value from memory. A `sync.v` fence does not constrain control-thread scalar loads/stores in any way. A `sync.cv` fence constrains all types of memory accesses on a core, and thus is the most restrictive type of memory fence. In Figure 4.4b, a `sync.cv` fence is used to order two control-thread scalar stores, two vector stores, and two sets of vector-fetched scalar stores. These kind of fences are useful when communicating between the control thread and the microthreads through memory. For example, using a `sync.l.cv` fence after initializing a data structure with the control thread ensures that the following vector memory accesses will read the correctly initialized data. Figure 4.2 illustrates using this kind of fence at the end of a function to ensure that the caller sees properly updated results that are stored in memory. Although there are several other types of fences that constrain various subsets of memory accesses, these three types cover the most common usage patterns.

In addition to memory fences, Maven also provides atomic memory operations for inter-thread communication (see Table 4.1). Atomic memory operations can provide more scalable performance as compared to the standard MIPS32 load-linked and store-conditional instructions. Atomic mem-

ory operations do not enforce a particular memory ordering, so a programmer is responsible for combing an atomic memory operation with a memory fence to implement certain parallel communication mechanisms (e.g., a mutex).

4.5 Calling Conventions

Both the control thread and the microthreads can have different calling conventions, but in Maven we exploit the unified instruction set to create a common calling convention for both types of thread. This calling convention only handles passing scalar values in registers. One can also imagine a calling convention for when the control thread passes vector registers as function arguments, but we do not currently support this feature.

Table 4.3 shows the control-thread and microthread scalar register usage convention, which is a modified version of the standard MIPS32 o32 convention [Swe07]. Maven includes eight instead of four argument registers, and both floating-point and integer values can be passed in any argument register (`a0-a7`) or return register (`v0-v1`). Stack passing conventions are the same as in o32 except that register allocated arguments do not need to reserve space on the stack. Since the scalar convention is identical for both the control thread and the microthreads, the same compiled function can be called in either context.

The above discussion assumes that Maven is configured with 32 registers per microthread. Although reconfigurable vector registers enable higher performance and better energy efficiency, they also complicate the calling conventions. For example, assume the control thread configures the VTU for eight vector registers and then vector-fetches code onto the microthreads. These microthreads will only have eight scalar registers available and be unable to call a function using the standard

Register Number	Assembly Name	Usage
0	`zero`	Always contains zero
1	`at`	Reserved for use by assembler
2,3	`v0,v1`	Values returned by function
4–11	`a0-a7`	First few arguments for function
12–15	`t4-t7`	Functions can use without saving
16–23	`s0-s7`	Functions must save and then restore
24,25	`t8,t9`	Functions can use without saving
26,27	`k0,k1`	Reserved for use by kernel
28	`gp`	Global pointer for small data
29	`sp`	Stack pointer
30	`fp`	Frame pointer
31	`ra`	Function return address

Table 4.3: Scalar Register Usage Convention – This convention is used both by the control-thread and by the microthreads and is similar to the standard MIPS32 o32 convention except for eight instead of four argument registers. (Adapted from [Swe07])

scalar calling convention. Critical registers, such as the stack pointer and return address register, will be invalid. Currently, Maven takes a simplistic approach where the microthread calling convention requires all 32 registers.

4.6 Extensions to Support Other Architectural Design Patterns

We would like to compare the energy-efficiency and performance of Maven to the other architectural design patterns presented in Chapter 2. This section describes how the basic Maven instruction set can be extended to emulate the MIMD, vector-SIMD, and SIMT architectural design patterns for evaluation purposes.

4.6.1 MIMD Extensions to Maven Instruction Set Architecture

When emulating the MIMD pattern, software is limited to just the scalar instructions in Table 4.1 and should avoid the vector instructions listed in Table 4.2. All communication between microthreads occurs through memory, possibly with the help of the standard MIPS32 memory fence (`sync`) and Maven specific atomic memory operations. Microthreads can take advantage of the `utidx` instruction to identify their thread index.

Most applications have a serial portion where a master thread will perform some work while the other worker threads are waiting. Usually this waiting takes the form of a tight loop that continually checks a shared variable in memory. When microthreads are running on different cores, this waiting loop will impact energy-efficiency but not performance. In the MIMD pattern, multiple microthreads can be mapped to the same core meaning that this waiting loop steals execution cycles from the master thread significantly degrading performance. To address this issue, the Maven MIMD extensions include a coprocessor 0 control register named `tidmask` that indicates which microthreads are active. Any thread can read or write this control register, and software must establish a policy to coordinate when threads are made inactive or active. An inactive thread is essentially halted at the current program counter and executes no instructions until another thread activates it by setting the appropriate bit in `tidmask`. The `stop` instruction can also be used to allow a thread to inactivate itself. Note that this differs from how the `stop` instruction is ignored by the control thread when using the basic VT pattern.

4.6.2 Vector-SIMD Extensions to Maven Instruction Set Architecture

The Maven vector-SIMD extensions are based on the Torrent instruction set used in the T0 vector processor [Asa96]. Differences include support for floating-point instead of fixed-point, and more extensive vector flag support instead of the vector conditional move instruction used in the Torrent instruction set. The Maven vector-SIMD extensions includes an explicit eight-entry vector flag register file. To simplify instruction encoding and some vector flag arithmetic, vector flag register zero is always defined to contain all ones. When emulating the vector-SIMD pattern,

Vector Indexed Memory Instructions

`l{w,h,b}x.v`	R_{dst}, r_{base}, R_{idx}	Load vector reg R_{dst} from mem[r_{base}] with offset in R_{idx}
`l{hu,bu}x.v`	R_{dst}, r_{base}, R_{idx}	(unsigned and signed variants)
`s{w,h,b}x.v`	R_{src}, r_{base}, R_{idx}	Store vector reg R_{src} to mem[r_{base}] with offset in R_{idx}

Vector Integer Arithmetic Instructions

`intop.vv`	R_{dst}, R_{src0}, R_{src1}, [F]	Vector-vector arithmetic, (R_{dst} = R_{src0} intop R_{src1})
`intop.vs`	R_{dst}, R_{src0}, r_{src1}, [F]	Vector-scalar arithmetic, (R_{dst} = R_{src0} intop r_{src1})
`intop.sv`	R_{dst}, r_{src0}, R_{src1}, [F]	Scalar-vector arithmetic, (R_{dst} = r_{src0} intop R_{src1})

`addu.{vv,vs}, subu.{vv,vs,sv}, mul.{vv,vs}`
`div.{vv,vs,sv}, rem.{vv,vs,sv}, divu.{vv,vs,sv}, remu.{vv,vs,sv}`
`and.{vv,vs}, or.{vv,vs}, xor.{vv,vs}, nor.{vv,vs}`
`sll.{vv,vs,sv}, srl.{vv,vs,sv}, sra.{vv,vs,sv}`

Vector Floating-Point Arithmetic Instructions

`fpop.s.v`	R_{dst}, R_{src}, [F]	Vector arithmetic, R_{dst} = fpop(R_{src})
`fpop.s.vv`	R_{dst}, R_{src0}, R_{src1}, [F]	Vector-vector arithmetic, R_{dst} = (R_{src0} fpop R_{src1})
`fpop.s.vs`	R_{dst}, R_{src0}, r_{src1}, [F]	Vector-scalar arithmetic, R_{dst} = (R_{src0} fpop r_{src1})
`fpop.s.sv`	R_{dst}, r_{src0}, R_{src1}, [F]	Scalar-vector arithmetic, R_{dst} = (r_{src0} fpop R_{src1})

`add.s.{vv,vs,sv}, sub.s.{vv,vs,sv}, mul.s.{vv,vs,sv}, div.s.{vv,vs,sv}`
`abs.s.v, neg.s.v, sqrt.s.v`
`round.w.s.v, trunc.w.s.v, ceil.w.s.v, floor.w.s.v, cvt.s.w.v`

Table 4.4: Maven Vector-SIMD Extensions – The vector-SIMD extensions add indexed vector load/store instructions, vector arithmetic instructions, and vector flag support. See Table 4.5 for list of instructions related to vector flag support. (r = control-thread scalar register specifier, R = vector register specifier, [F] = optional vector flag register specifier for operations under flag, {`w,h,b`} = word, halfword, and byte vector memory variants, {`hu,bu`} = unsigned halfword and unsigned byte vector memory variants, {`vv,vs,sv`} = vector-vector, vector-scalar, scalar-vector variants)

software can use all of the vector instructions in Table 4.2 except for the vector fetch instructions. Tables 4.4 and 4.5 show the additional vector instructions needed for the vector-SIMD pattern. There are vector equivalents for most of the scalar MIPS32 operations. Most of the vector-SIMD instructions are encoded with a MIPS32 coprocessor 2 primary opcode and a secondary opcode field, although some instructions require unused primary MIPS32 opcodes.

The vector-SIMD extensions include indexed vector loads and stores each of which requires two sources: a scalar base register and a vector register containing offsets for each microthread. A full complement of vector integer and arithmetic instructions are available. Some instructions include three variants: vector-vector (`.vv`) with two vector registers as sources and vector-scalar/scalar-vector (`.vs`/`.sv`) with a a vector register and a control-thread scalar register as sources. Vector-scalar and scalar-vector variants are necessary for those operations that are not commutative. For example, the `subu.vs` instruction subtracts a scalar value from each element in a vector register while the `subu.sv` subtracts each element in a vector register from a scalar value. All vector

Vector Flag Integer Compare Instructions

`seq.f.vv`	F_{dst}, R_{src0}, R_{src1}	Flag reg F_{dst} = (R_{src0} == R_{src1})
`slt.f.vv`	F_{dst}, R_{src0}, R_{src1}	Flag reg F_{dst} = (R_{src0} ¡ R_{src1})

Vector Flag Floating-Point Compare Instructions

`c.fpcmp.s.f.vv`	F_{dst}, R_{src0}, R_{src1}	Vector-vector flag compare, F_{dst} = (R_{src0} fpcmp R_{src1})
`c.fpcmp.s.f.vs`	F_{dst}, R_{src0}, r_{src1}	Vector-scalar flag compare, F_{dst} = (R_{src0} fpcmp r_{src1})
`c.fpcmp.s.f.sv`	F_{dst}, r_{src0}, R_{src1}	Scalar-vector flag compare, F_{dst} = (r_{src0} fpcmp R_{src1})

`c.f.s.{vv,vs,sv}, c.un.s.{vv,vs,sv}, c.eq.s.{vv,vs,sv}, c.ueq.s.{vv,vs,sv}`
`c.olt.s.{vv,vs,sv}, c.ult.s.{vv,vs,sv}, c.ole.s.{vv,vs,sv}, c.ule.s.{vv,vs,sv}`
`c.sf.s.{vv,vs,sv}, c.nlge.s.{vv,vs,sv}, c.seq.s.{vv,vs,sv}, c.ngl.s.{vv,vs,sv}`
`c.lt.s.{vv,vs,sv}, c.nge.s.{vv,vs,sv}, c.le.s.{vv,vs,sv}, c.ngt.s.{vv,vs,sv}`

Vector Flag Arithmetic Instructions

`fop.f`	F_{dst}, F_{src0}, F_{src1}	Flag transformation, F_{dst} = (F_{src0} fpcmp F_{src1})

`or.f, and.f, xor.f, not.f`

Vector Flag Move Instructions

`mov.f`	F_{dst}, F_{src}	Move F_{src} to F_{dst}
`mov.fv`	F_{dst}, R_{src}	Move R_{src} to F_{dst}
`mov.vf`	R_{dst}, F_{src}	Move F_{src} to R_{dst}
`popc.f`	r_{dst}, F_{src}	Write number of ones in flag reg F_{src} to CT reg r_{dst}
`findfone.f`	r_{dst}, F_{src}	Write position of first one in flag reg F_{src} to CT reg r_{dst}

Table 4.5: Maven Vector-SIMD Extensions (Flag Support) – The vector-SIMD extensions includes support for writing flag registers based on integer and floating-point comparisons, as well as other flag arithmetic operations. (CT = control thread, r = control-thread scalar register specifier, R = vector register specifier, F = vector flag register specifier, {`w,h,b`} = word, halfword, and byte vector memory variants, {`hu,bu`} = unsigned halfword and unsigned byte vector memory variants)

arithmetic operations include an optional source specifying a vector flag register. Each element of a vector is only written to the vector register if the corresponding bit of the flag register is one. The Maven vector-SIMD extensions do not include conditional loads and stores, since implementing them can be quite challenging.

Various vector integer and arithmetic comparison instructions are provided that write a specific flag register. Flag arithmetic instructions can be used for implementing more complicated data-dependent conditionals. For example, a simple `if/then/else` conditional can be mapped with a comparison operation that writes flag register F, executing the `then` instructions under flag F, complementing the flag register with a `not.f` instruction, and then executing the `else` instructions. Flag arithmetic also be used for nested `if` statements by using the `or.f` instruction to merge nested comparison operations.

4.6.3 SIMT Extensions to Maven Instruction Set Architecture

To emulate the SIMT pattern, we could simply restrict software in a similar way as for the Maven MIMD extensions. This would require a completely different implementation from any of the other patterns. We found it easier to start from the basic Maven VT instruction set and corresponding Maven implementation, and simply restrict the type of instructions allowed when emulating the SIMT pattern. For example, SIMT software should avoid using the control thread as much as possible, since an actual SIMT implementation would not include a programmable control processor. SIMT software should begin by initializing scalar variables for each microthread with the `mov.sv` instruction. Normally this would be done with dedicated hardware in an actual SIMT implementation. SIMT software should also avoid using the `setvl` instruction, and instead use a vector-fetched scalar branch to check if the current microthread is greater than the application vector length. The SIMT software can take advantage of the reconfigurable vector registers if possible, since SIMT implementations might have similar reconfiguration facilities implemented with dedicated hardware. SIMT microthreads can use all of the scalar instructions in Table 4.1 including atomic memory operations. SIMT software should use vector-fetched scalar loads/stores instead of unit-stride or strided vector memory operations. Partitioning a long vector-fetched block into multiple shorter vector-fetched blocks emulates statically annotated reconvergence points.

4.7 Future Research Directions

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

Calling Conventions with Reconfigurable Vector Registers – As discussed in in Section 4.5, reconfigurable vector registers significantly complicate the register usage and calling convention. The current policy avoids any issues by limiting reconfiguration to leaf functions, but it would be useful to enable more flexible reconfiguration policies. One option might be to provide different versions of a function, each of which assumes a different number of available vector registers. The calling convention could standardize a few common configurations to reduce duplicate code. A bigger issue is the inability for the microthreads to make function calls unless the VTU is configured with 32 registers. A possible improvement would be to modify the order in which logical registers are eliminated as the number of registers per microthread is decreased. For example, if the `gp`, `sp`, `fp`, and `ra` registers were eliminated only after 16 of the other registers were eliminated the microthreads could call functions as long as 16 or more registers were available.

Shared Vector Registers – Section 4.3 discussed how a Maven implementation can exploit the fact that register number zero is shared among all microthreads to increase vector lengths with the same amount of physical resources. This is a specific example of a more general concept known as *shared vector registers* first explored in the Scale VT processor. Currently Maven only provides

private vector registers meaning that there is a unique element for each microthread. A shared vector register has only one element and is shared among all the microthreads, and it can be used to hold constants or possibly even reduction variables. The `vcfgivl` instruction would need to be extended so that the programmer could specify the required number of private and shared registers. Adding shared registers such that any Maven instruction can access the shared register is challenging without radical re-encoding. There are no available bits to hold the private register specifier as well as additional shared register specifiers. One option would be to place the shared registers in a separate register space and provide a special instruction to move values between the private and shared registers. Another option would be to allow some of the standard 32 registers currently used by Maven to be marked as shared. The order in which registers are marked private or shared could be specified statically as part of the instruction set. For example, some of the temporary registers might be able to be marked as shared. Of course, an additional challenge would be improving the programming methodology such that private and shared registers are allocated efficiently.

Vector Unconfiguration – Section 4.3 introduced a method for reconfiguring the vector registers based on the number of required vector registers. It would also be useful to add a `vuncfg` instruction that unconfigures the vector unit. This instruction acts as a hint to the hardware that the VTU is not being used. An implementation could then implement a policy to power-gate the VTU after a certain period of time, thereby reducing static power consumption. The VTU would be turned on at the next `vcfgivl` instruction. This instruction might also include an implicit `sync.l.cv` memory fence. A standard software policy for vectorized functions might be to first configure the VTU, complete the vectorized computation, and then unconfigure the VTU. Currently, most vectorized Maven functions already use a `sync.l.cv` memory fence before returning, so the `vuncfg` instruction would just be a more powerful mechanism for ending a vectorized function. Finally, unconfiguring the VTU also tells the operating system that there is no need to save the vector architectural state on a context swap.

Vector Segment Memory Accesses – Working with arrays of structures or objects is quite common, but accessing these objects with multiple strided accesses can be inefficient. Segment memory accesses, such as those available in the Scale VT processor and stream processors [RDK+98, KWL+08], can load multiple consecutive elements into consecutive vector registers. For example, an array of pixels each with a red, blue, and green color component could be loaded with either three strided accesses or with a single segment access that loads the three color components into three consecutive registers in each microthread. Segment accesses are straight-forward to add to the instruction set, but implementing them efficiently can be difficult.

Vector Compress and Expand Instructions – As described in Section 2.1, vector compress and expand instructions can be used to transform irregular DLP to more regular DLP. Even though VT architectures can handle most irregular DLP naturally with vector-fetched scalar branches, it might

still be useful in some applications to collect just the active elements. For example, a programmer might want to break a large vector-fetched block into smaller blocks to facilitate chaining off vector loads and/or to reduce the required number of vector registers. If the large vector-fetched block contains control-flow intensive code, the programmer will need to save the control-flow state so that the microthreads can return to the proper control path in the second vector-fetched block. An alternative might be to compress active elements, and process similar elements with different vector-fetched blocks. Unfortunately, vector compress and expand instructions usually leverage a vector flag register to specify which elements to compress, but this is not possible in the Maven VT instruction set. Instead, Maven could provide a new microthread scalar instruction which marks that microthread as participating in a vector compress operation. The next vector compress instruction would only compress elements from those microthreads which have been marked and then reset this additional architectural state. It would probably be necessary for the control thread to have access to the number of microthreads participating in the compress operation for use in counters and pointer arithmetic.

Vector Reduction Instructions – Currently the Maven instruction set has no special provisions for reduction operations that can be used when implementing cross-iteration dependencies such as the one shown in Table 2.1j. These loops can still be mapped to Maven, but the cross-iteration portion either needs to be executed with vector memory instructions or by the control thread. Because the VT pattern includes vector instructions in addition to vector-fetched scalar instructions, Maven can eventually support any of the standard reduction instructions found in vector-SIMD architectures. For example, a *vector element rotate instruction* would essentially pass a register value from one microthread to the next, while a *vector extract instruction* would copy the lower portion of a source vector register to the higher portion of a destination vector register. Unfortunately, since these are vector instructions, they can not be easily interleaved with vector-fetched scalar instructions. The programming methodology must partition vector-fetched blocks into smaller blocks to enable the use of these vector reduction instructions. An alternative would be to leverage the shared vector registers for cross-microthread communication. The instruction set could provide a new *vector fetch atomic* instruction for manipulating a shared register for cross-microthread communication, similar to the atomic instruction blocks used in Scale. A vector fetch atomic instruction requires the hardware to execute the corresponding vector-fetched atomic block for each microthread in isolation with respect to the other microthreads and shared registers. A microarchitecture can implement these atomic blocks by effectively causing maximum divergence where each microthread executes the atomic region by itself. Although this serializes the computation, it is a flexible mechanism and keeps the data in vector registers for efficient execution of any surrounding regular DLP.

4.8 Related Work

This section highlights selected previous work specifically related to the unified VT instruction set presented in this chapter.

Scale VT Processor – The Maven instruction set has some similarities and important differences compared to the Scale instruction set [KBH+04a]. Both have very similar control-thread scalar instruction sets based on the MIPS32 instruction set with the primary difference being Maven's support for floating-point instructions. Both instruction sets support reconfigurable vector registers, unit-stride and strided vector memory instructions, and vector fetch instructions. Scale includes dedicated vector segment accesses to efficiently load arrays of structures. The most important difference between the two instruction sets, is that the Scale microthread instruction set is specialized for use in a VT architecture, while Maven unifies the control-thread and microthread instruction sets. Scale's microthread instruction set has software exposed clustered register spaces compared to Maven's monolithic microthread register space; Scale microthread code must be partitioned into atomic instruction blocks while Maven microthread code simply executes until it reaches a stop instruction; Scale provides thread-fetch instructions for microthread control flow, while Maven uses standard scalar branch instructions; Scale has richer support for predication with an implicit predication register per cluster and the ability to predicate any microthread instruction while Maven is limited to the standard MIPS32 conditional move instructions; Scale has hardware support for cross-microthread communication while Maven requires all such communication to be mapped through memory; Scale includes shared registers in addition to private registers which enables implementations to better exploit constants or shared values across microthreads. Overall the Maven instruction set is simpler, while the Scale instruction set is more complicated and specialized for use in a VT architecture.

Fujitsu Vector Machines Reconfigurable Vector Registers – Most vector-SIMD processors have fixed hardware vector lengths, but several Fujitsu vector machines include a reconfigurable vector register mechanism similar to Maven. For example, the Fujitsu VP-200 machine has a total of 8,192 64-bit elements that could be reconfigured from eight vector registers of 1024 elements to 256 registers of 32 elements [Oya99].

Graphics Processor Reconfigurable Number of Microthreads – Graphics processors from both NVIDIA and ATI include the ability to increase the total number of microthreads if each microthread requires fewer scalar registers [LNOM08, nvi09, ati09]. This does not change the number of microthreads per microthread block, but instead it changes the amount of VIU multithreading. Thus this reconfiguration will improve performance but will unlikely help the energy efficiency. Maven's configuration instructions change the actual number of microthreads managed by each control thread, and thus directly impacts the amount of energy amortization.

Microthreaded Pipelines – Jesshope's work on microthreaded pipelines [Jes01] has some similarities to the Maven VT instruction set. The proposal includes a `cre` instruction that is similar to Maven's `vf` instruction. The `cre` instruction causes an array of *microthread pipelines* to each execute one iteration of a loop, and a subset of the master thread's instruction set is supported by the microthreaded pipelines. An important difference, is that the microthreaded pipelines cannot execute conditional control flow, while the Maven instruction set is specifically designed to efficiently handle such control flow. Jesshope's work also lacks vector memory instructions to efficiently transfer large blocks of data between memory and the microthreaded pipelines. There are also significant differences in the implementation of the two approaches. The VT pattern attempts to achieve vector-like efficiencies for microthread instructions which are coherent, while the microthreaded pipeline approach has a much more dynamic thread execution model that always results in scalar energy efficiencies. In addition, the microthreaded pipeline technique focuses more on efficiently executing cross-iteration dependencies, while Maven focuses more on efficiently executing irregular control flow.

Unified Scalar/Vector – Jouppi et al.'s work on a unified scalar/vector processor leverages the primary scalar datapath to provide features similar to the vector-SIMD pattern with a purely temporal implementation [JBW89]. Both scalar and vector instructions operate on the same register file; vector instructions can start at any offset in the register file and simply perform the same operation for multiple cycles while incrementing the register specifier. This technique enables an elegant mapping of irregular DLP, since mixing scalar arithmetic and control instructions with vector arithmetic operations is trivial. Unfortunately, control thread decoupling is no longer possible and the energy efficiency of such an approach is unclear. Since registers are accessed irregularly, implementing multiple vector lanes or banked vector register files becomes more difficult. Dependency checking is more complicated and must occur per vector element.

Chapter 5

Maven Microarchitecture

The Maven data-parallel accelerator will include an array of tens to hundreds of cores each of which implements the Maven instruction set introduced in the previous chapter. Section 3.2 has already motivated our interest in Maven VT cores based on small changes to a single-lane vector-SIMD unit. Section 5.1 gives an overview of the single-lane Maven VT core microarchitecture and provides an example program execution that will be carried throughout the rest of the chapter. Sections 5.2–5.5 introduce four techniques that complement the basic Maven microarchitecture. Section 5.2 describes *control processor embedding* which helps mitigate the area overhead of including a separate control processor per vector lane. Section 5.3 describes *vector fragment merging* which helps microthreads automatically become coherent again after diverging. Section 5.4 describes *vector fragment interleaving* which helps hide latencies (particularly the branch resolution latency) by interleaving the execution of multiple independent vector fragments. Section 5.5 describes *vector fragment compression* which is a flexible form of density-time execution suitable for use in VT architectures. Although this thesis focuses on a single data-parallel core, Section 5.6 provides some insight into how these cores can be combined into a full data-parallel accelerator. Section 5.7 discusses various extensions that enable the Maven microarchitecture to reasonably emulate the MIMD, vector-SIMD, and SIMT architectural design patterns. The chapter concludes with future research directions (Section 5.8) and related work (Section 5.9).

5.1 Microarchitecture Overview

Figure 5.1 shows the microarchitecture of a Maven VT core which contains four primary modules: the *control processor* executes the control thread and fetches vector instructions which are then sent to the vector issue unit and vector memory unit; the *vector lane* includes the vector register file and vector functional units; the *vector issue unit* (VIU) handles issuing vector instructions and managing vector-fetched instructions, and the *vector memory unit* (VMU) handles the vector and microthread memory accesses. The vector lane, VIU, and VMU form the Maven vector-thread

unit (VTU) which closely resembles a single-lane vector-SIMD unit. The primary differences between the two implementations are in the VIU, with some smaller changes in the vector lane and VMU. This close resemblance helps ensure that Maven will achieve vector-like efficiencies on regular DLP, while the differences in the VIU allow Maven to more easily execute a wider range of irregular DLP. Each of these four modules is explained in detail below, and then an example program is used to illustrate how all four modules work together to execute both regular and irregular DLP.

Maven includes a simple five-stage single-issue in-order RISC control processor. The control processor uses a 31-entry register file with two read ports and two write ports. The first write port is for short-latency integer operations, while the second write port is for long-latency operations such as loads, integer multiplies/divides, and floating-point operations. This allows the control processor to overlap short-latency operations while long-latency operations are still in flight without needing complex dynamic write-port arbitration. An early commit point in the pipeline allows these short-latency operations to be directly written to the architectural register file. The figure illustrates how the control processor is embedded into the vector lane by sharing the lane's long-latency functional units and memory ports. Section 5.2 will discuss control processor embedding in more detail.

A Maven vector lane includes a large vector register file with 248×32-bit elements and five vector functional units: vector functional unit zero (VFU0) which can execute simple integer operations, integer multiplies, and floating-point multiplies and square roots; vector functional unit one (VFU1) which can execute simple integer operations, integer divides, and floating-point additions and divides; the vector address unit (VAU) which reads addresses from the vector register file and sends them to the VMU for microthread loads and stores; the vector store-data read unit (VSDRU) which reads store data from the vector register file and sends it to the VMU for vector and microthread stores; and the vector load-data writeback unit (VLDWU) which writes load data from the VMU into the vector register file. The long-latency operations in the VFU0/VFU1 are fully pipelined and have latencies ranging from four to 12 cycles. Each vector functional unit includes a small controller that accepts a vector operation and then manages its execution over many cycles. The controller handles fetching the appropriate sources from the vector register file, performing the operation, and writing the result back into vector register file for each element of the vector. A single vector operation can therefore occupy a vector functional unit for a number of cycles equal to the current hardware vector length. Note that the vector functional unit controllers do not perform hazard checking nor handle any stall logic; the VIU is responsible for issuing vector operations to functional units only when there are no hazards and the operation can execute to completion (with the exception of the vector memory instructions described below). To support parallel execution of all five functional units, the vector register file requires six read ports and three write ports. The register file can be configured to support vector lengths of 8–32 with 32–4 registers per microthread respectively. Note that there is actually enough state to support a vector length of 64 with only

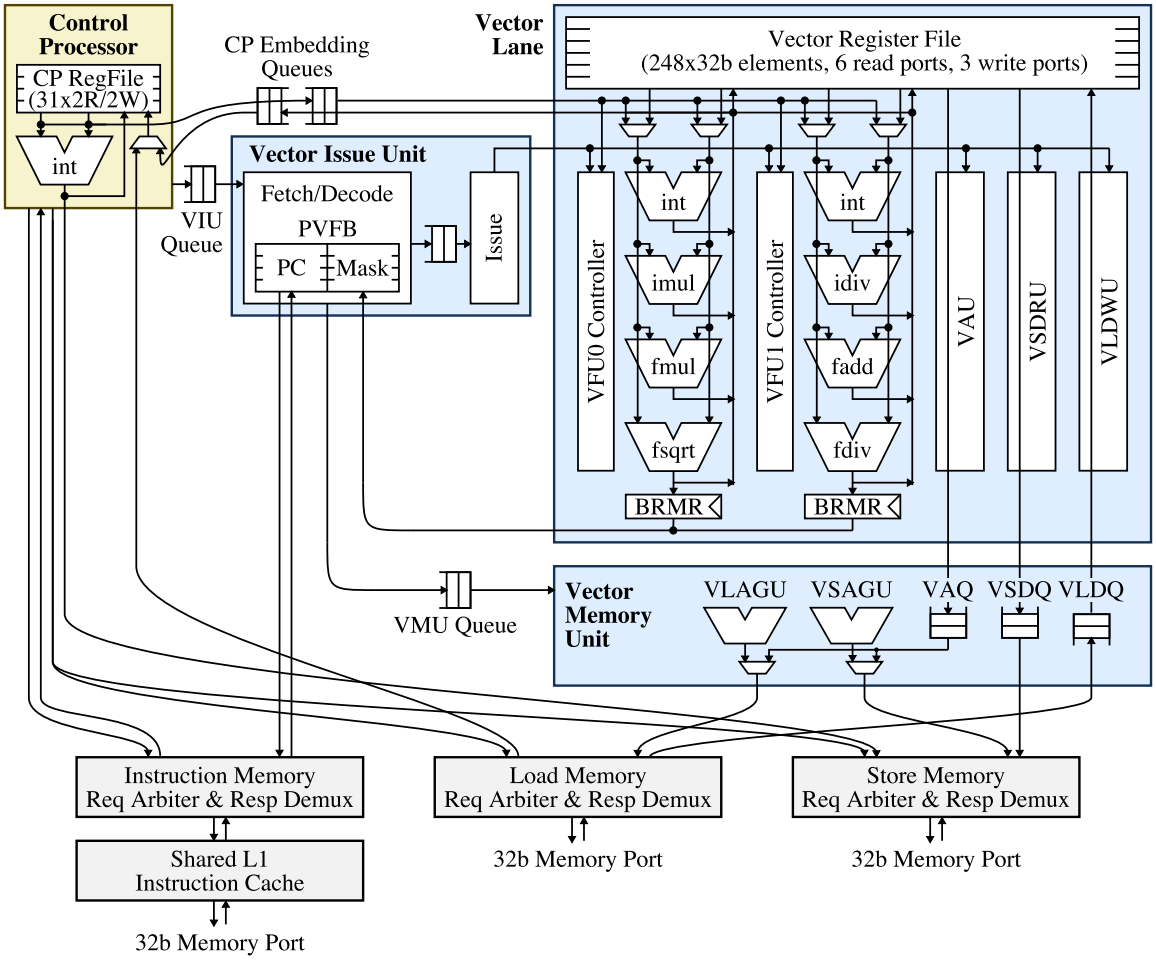


Figure 5.1: Maven VT Core Microarchitecture – This microarchitecture corresponds to the high-level implementation shown in Figure 2.12b with a single-lane VTU. The control processor is embedded meaning it shares long-latency functional units and memory ports with the VTU. Maven includes five vector functional units: VFU0 executes simple integer arithmetic, integer multiplication, and floating-point multiplication/square-root; VFU1 executes simple integer arithmetic, integer division, and floating-point addition/division; the VAU reads addresses and generates requests for microthread loads/stores; the VSDRU reads store-data from register file for vector and microthread stores; and the VLDWU writes load-data into register file for vector and microthread loads. The vector issue unit manages vector instructions and vector-fetched scalar instructions, while the vector memory unit manages vector and microthread loads/stores. (CP = control processor, VIU = vector issue unit, VMU =vector memory unit, PVFB = pending vector fragment buffer, PC = program counter, BRMR = branch resolution mask register, VFU = vector functional unit, VAU = vector address unit, VSDRU = vector store-data read unit, VLDWU = vector load-data writeback unit, VLAGU = vector load-address generation unit, VSAGU = vector store-address generation unit, VAQ = vector address queue, VSDQ = vector store-data queue, VLDQ = vector load-data queue, int = simple integer ALU, imul/idiv = integer multiplier and divider, fmul/fadd/fsqrt/fdiv = floating-point multiplier, adder, square root unit, and divider.)

four registers per microthread. We limit the vector length to 32 to reduce VIU control logic and microarchitectural state that scales with the maximum number of microthreads.

The VIU acts as the interface between the control processor and the vector lane. The control processor fetches vector instructions and pushes them into the VIU queue, and the VIU then processes them one at a time. If a vector instruction requires access to control-thread scalar data, then the control processor will push the corresponding values into the VIU queue along with the vector instruction. The VIU is divided into two decoupled stages: the *fetch/decode stage* and the *issue stage*. When the fetch/decode stage processes a simple vector instruction (e.g., a `mov.vv` or `mtut` instruction) sent from the control processor, it simply decodes the instruction and sends it along to the issue stage. When the fetch/decode stage processes a vector fetch instruction, it creates a new *vector fragment*. Vector fragments, and the various mechanisms used to manipulate them, are the key to efficiently executing both regular and irregular DLP on Maven. Each vector fragment contains a program counter and a bit mask identifying which microthreads are currently fetching from that program counter. Vector fragments are stored in the *pending vector fragment buffer* (PVFB) located in the fetch/decode stage of the VIU. When processing a vector fetch instruction the newly created vector fragment contains the target instruction address of the vector fetch instruction and a bit mask of all ones, since all microthreads start at this address. The fetch/decode stage then starts to fetch scalar instructions for this fragment and sends the corresponding *vector fragment micro-op* (vector operation plus microthread bit mask) to the issue stage. The issue stage is responsible for checking all structural and dependency hazards before issuing vector fragment micro-ops to the vector lane. Fragment micro-ops access the vector register file sequentially, one element at a time, and cannot stall once issued. The regular manner in which vector fragments execute greatly simplifies the VIU. The issue stage tracks just the first microthread of each vector fragment micro-op, which effectively amortizes hazard checking over the entire vector length. Even though the VIU issues vector fragment micro-ops to the vector lane one per cycle, it is still possible to exploit instruction-level parallelism by overlapping the execution of multiple vector instructions, since each occupies a vector functional unit for many cycles.

When the VIU processes a vector-fetched scalar branch instruction, it issues the corresponding branch micro-op to VFU0/VFU1 and then waits for all microthreads to resolve the branch. Each microthread determines whether the branch should be taken or not taken and writes the result to the appropriate bit of the branch resolution mask register (BRMR). Once all microthreads have resolved the branch, the branch resolution mask is sent back to the VIU. If the branch resolution mask is all zeros, then the VIU continues fetching scalar instructions along the fall-through path, or if the branch resolution mask is all ones, then the VIU starts fetching scalar instructions along the taken path. If the microthreads diverge, then the VIU creates a new fragment to track those microthreads which have taken the branch, removes those microthreads from the old fragment, and continues to execute the old fragment. Microthreads can continue to diverge and create new frag-

ments, but eventually the VIU will reach a stop instruction which indicates that the current fragment is finished. The VIU then chooses another vector fragment to start processing and again executes it to completion. Once all vector fragments associated with a specific vector fetch instruction are complete, the VIU is free to start working on the next vector instruction waiting in the VIU queue. Vector-fetched unconditional jump instructions are handled exclusively in the fetch/decode stage of the VIU. Vector-fetched jump register instructions require VFU0/VFU1 to read out each element of the address register and send it back to the fetch/decode stage of the VIU one address per cycle. The fetch/decode stage either merges the corresponding microthread into an existing fragment in the PVFB or creates a new fragment in the PVFB if this is the first microthread with the given target address. This allows the microthreads to remain coherent if they are all jumping to the same address, which is common when returning from a function.

The VMU manages moving data between memory and the vector register file. For vector memory instructions, the control processor first fetches and decodes the instruction before sending it to the VIU. The fetch/decode stage of the VIU then splits it into two parts: the address generation micro-op which is sent to the VMU, and either the store-data read micro-op or the load-data writeback micro-op which are sent to the issue stage. The VMU turns a full vector memory micro-op into multiple memory requests appropriate for the memory system. We currently assume the memory system naturally handles 256-bit requests, so a 32-element `lw.v` instruction (total size of 1024 bits) needs to be broken into four (or five if the vector is unaligned) 256-bit requests. Strided accesses may need to be broken into many more memory requests. The vector load-address generation unit (VLAGU) and vector store-address generation unit (VSAGU) are used by the VMU to generate the correct addresses for each 256-bit request. As shown in Figure 5.1, Maven supports issuing both a load and a store every cycle. Since read and write datapaths are always required, the performance improvement is well worth the additional address bandwidth to support issuing loads and stores in parallel. For a vector load, the memory system will push the load data into the vector load-data queue (VLDQ). To prevent a core from stalling the memory system, the VMU is responsible for managing the VLDQ such that the memory system always has a place to write the response data. Once data arrives in the VLDQ, the VIU can issue the load-data writeback micro-op to the VLDWU which will write the vector into the register file one element per cycle. For a vector store, the store-data read micro-op is issued to the VSDRU and one element per cycle is read from the vector register file and pushed into the vector store data queue (VSDQ). The VMU then takes care of issuing requests to send this data to the memory system. The issue stage must handle the store-data read micro-op and the load-data writeback micro-op carefully, since they can stall the vector lane if the VSDQ or VLDQ are not ready. The issue stage monitors the VSDQ and the VLDQ, and only issues the store-data read micro-op or load-data writeback micro-op when there is a good chance a stall will not occur (i.e., there is plenty of space in the VSDQ or plenty of data in the VLDQ). Conservatively guaranteeing these micro-ops will never stall incurs a significant performance and

buffering penalty, so Maven includes a global stall signal to halt the entire vector lane when either the VSDQ is full or the VLDQ is empty. This simplifies the VIU, VMU, and vector lane design, but requires careful consideration of the the stall signal's impact on the critical path. Microthread loads and stores are handled in a similar way as vector memory instructions, except that the addresses come from the vector lane via the VAU, and the VIU must inform the VMU of how many elements to expect. By routing all memory operations through the VIU, we create a serialization point for implementing vector memory fences. Section 5.6 will illustrate how stream buffers located close to the L2 cache banks enable reading and writing the cache memories in wide 256-bit blocks. The blocks are then streamed between these buffers and the cores at 32 bits per cycle, which naturally matches the VSDRU and VLDWU bandwidth of one element per cycle.

Figure 5.2 shows how a small example executes on the Maven microarchitecture. The pseudocode is shown in Figure 5.2a and resulting pseudo-assembly (similar in spirit to what was used in Section 2.7) is shown in Figure 5.2b. This example contains nested conditionals to help illustrate how Maven handles irregular DLP with vector fragments. We assume the hardware vector length is four, and that the four microthreads have the following data-dependent execution paths: μT0: `opY` → `opZ`; μT1: `opW` → `opX` →`opZ`; μT2: `opY` → `opZ`; μT3: `opW` → `opZ`. The unit-stride vector load at address `0x004` is split into the load-data writeback micro-op which is handled by the VIU (`wr.v`) and the address generation micro-op which is sent to the VMU (`addr.v` not shown, see Figure 2.13b for an example). Some time later the data comes back from memory, and the VIU is able to issue the load-data writeback to the VLDWU. Over four cycles, the VLDWU writes the returned data to the vector register file. The VIU is then able to process the vector fetch instruction at address `0x008`, and the corresponding update to the PVFB is shown in the figure. The VIU creates a new vector fragment (A) with the target address of the vector fetch instruction (`0x100`) and a microthread mask of all ones. The first instruction of fragment A is the branch at address `0x100`. It is important to note that this execution diagram, as well as all other execution diagrams in this thesis, assume a vector lane can execute only a single vector micro-op at a time. This assumption is purely to simplify the discussion. Any practical implementation will support multiple vector functional units (e.g., Maven has five vector functional units), and allow results to be chained (bypassed) between these functional units. For example, Maven allows the unit-stride vector load to chain its result to the first vector-fetched branch instruction such that these two vector instructions can overlap. Chaining requires more complicated control logic in the VIU but is essential for good performance.

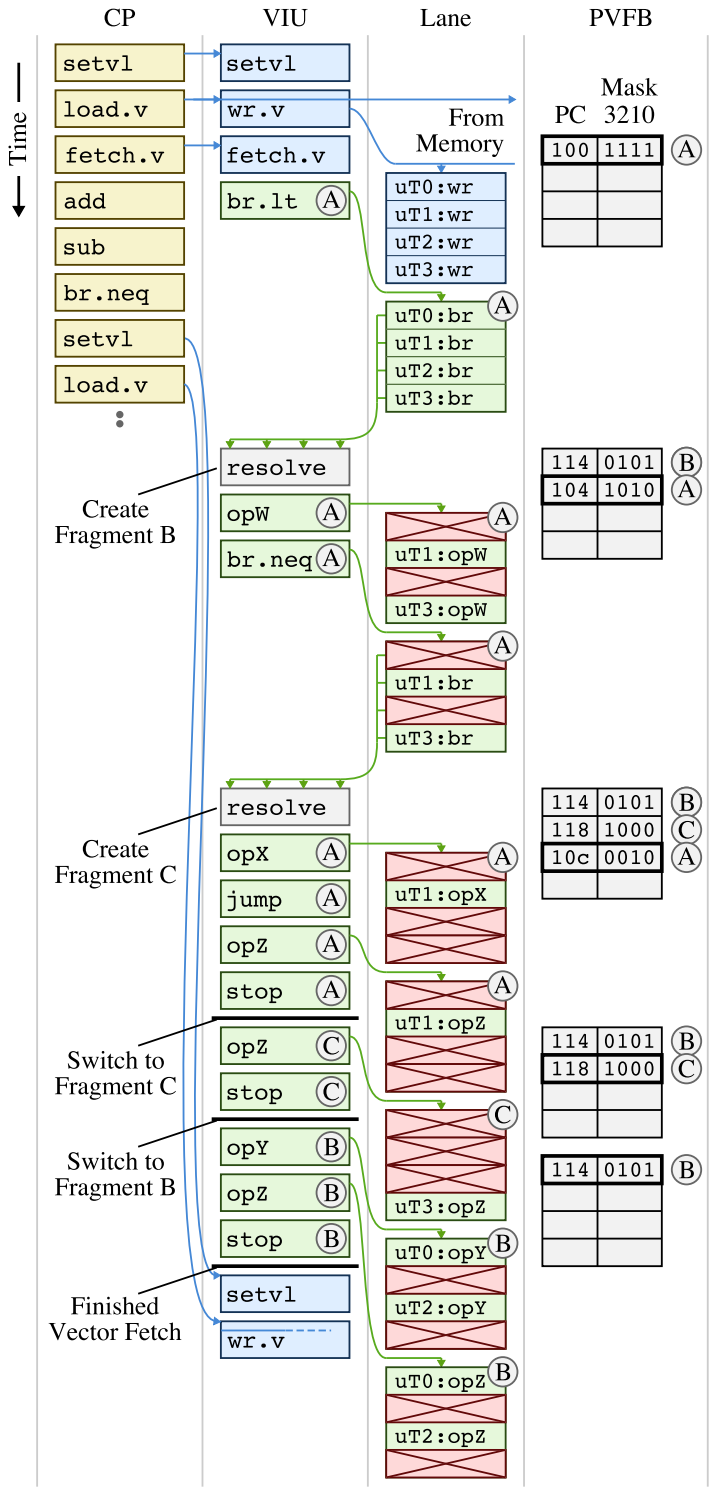
The figure shows how the VIU waits for all four microthreads to resolve the branch before determining how to proceed. In this example, the microthreads diverge with μT0 and μT2 taking the branch and μT1 and μT3 falling through. The VIU creates a new fragment (B) which contains the branch target address (`0x114`) and the mask bits for μT0 and μT2 set to one (`0101`). The VIU now proceeds to continue executing fragment A but only for μT1 and μT3. Notice how μT0 and

```
for ( i = 0; i < n; i++ )
  if ( A[i] >= 0 )
    opW;
    if ( A[i] == 0 )
      opX;
  else
    opY;
  opZ;
```

(a) Pseudocode

```
loop:
 000: setvl   vlen, n
 004: load.v  VA, a_ptr
 008: fetch.v ut_code
 00c: add     a_ptr, vlen
 000: sub     n, vlen
 014: br.neq  n, 0, loop
 ...

ut_code:
 100: br.lt   a, 0, else
 104: opW
 108: br.neq  a, 0, done
 10c: opX
 110: jump    done
else:
 114: opY
done:
 118: opZ
 11c: stop
```

(b) Pseudo-Assembly



(c) Maven Execution Diagram

Figure 5.2: Executing Irregular DLP on the Maven Microarchitecture – Example (a) pseudocode, (b) pseudo-assembly, (c) and execution for a loop with nested conditionals illustrating how the PVFB manages divergence through vector fragments. (Code syntax similar to that used in Figure 2.14. Vector memory unit not shown for simplicity. CP = control processor, VIU = vector issue unit, PVFB = pending vector fragment buffer.)

µT2 are masked off when the vector lane executes the `opW` and `br.neq` instructions. The `br.neq` instruction at address `0x108` illustrates a nested conditional and the potential for nested divergence, since the microthreads have already diverged once at the first branch. The VIU again waits for the branch to resolve before creating a new vector fragment (C) which records the fact that µT3 has taken the branch. Vector fragment A now only has a single microthread (µT1). The VIU continues to execute fragment A until it reaches a stop instruction. Notice how the VIU quickly redirects the vector-fetched instruction stream when it encounters unconditional control flow (i.e., the jump at address `0x110`).

Now that fragment A has completed, the VIU will switch and start executing one of the two remaining fragments (B,C). Exactly which fragment to execute next is a hardware policy decision, but based on the microarchitecture as currently described, there is no real difference. In this example, the VIU uses a stack-based approach and switches to fragment C, before it finally switches to fragment B. It is perfectly valid for these later fragments to continue to diverge. Once all fragments are complete and the PVFB is empty, the VIU can move onto the next vector instruction. In this example, the control processor has already run-ahead and pushed the beginning of the next iteration of the stripmine loop into the VIU queue.

Based on this example, we can make several interesting observations. Note that the PVFB does not need to track any per-path history, but instead simply tracks the leading control-flow edge across all microthreads. This means the size of the PVFB is bounded to the maximum number of microthreads (i.e., 32). An equivalent observation is that the mask bit for each microthread will be set in one and only one entry of the PVFB. As with all VT architectures, Maven is able to achieve vector-like efficiencies on regular DLP, but also notice that Maven is able to achieve partial vector-like efficiencies when executing vector fragments caused by irregular DLP. For example, the VIU can still amortize instruction fetch, decode, and dependency checking by a factor of two when executing instruction `opW` of fragment A. Unfortunately, this example also illustrates a missed opportunity with respect to instruction `opZ`. Although this instruction could be executed as part of just one fragment, it is instead executed as part of three different fragments. Section 5.3 will describe *vector fragment merging* which attempts to capture some of this reconvergence dynamically. It is important to note, though, that the microthreads always reconverge when the VIU moves onto the next vector instruction (e.g., for the next iteration of the stripmine loop). According to the Maven execution semantics any microthread interleaving is permissible, which means that executing fragments in any order is also permissible. Section 5.4 will describe *vector fragment interleaving* which hides various execution latencies, such as the branch resolution latency, by switching to a new fragment before the current fragment is completed. In this example, there are 17 wasted execution slots due to inactive microthreads, and the amount of wasted time will continue to rise with even greater divergence. Section 5.5 will describe *vector fragment compression* which skips inactive microthreads at the cost of increased control-logic and register-indexing complexity.

5.2 Control Processor Embedding

Control processor (CP) embedding allows the CP to share certain microarchitectural units with the VTU, and it is an important technique to mitigating the area overhead of single-lane VTUs. Figure 5.1 illustrates the two ways in which the Maven CP is embedded: shared long-latency functional units and shared memory ports.

An embedded CP still includes its own scalar register file and relatively small short-latency integer datapath. Instead of including its own long-latency functional units (e.g., integer multiplier/divider and floating-point units), the CP reads the source operands from its scalar register file and pushes them along with the appropriate command information into the CP embedding request queue (shown at the top of Figure 5.1). This request queue is directly connected to VFU0 and VFU1. The vector functional unit controllers include a simple arbiter which executes the CP operation when the vector functional unit would be otherwise unused. Since these CP operations steal unused cycles and do not read or write the vector register file, the VIU has no need to track or manage the CP operations. Although simple, this approach can starve the CP, limiting decoupling. A more sophisticated option is to send CP operations through the VIU. The results of the long-latency operations are eventually written back into the CP embedding response queue. The CP is decoupled allowing it to continue to execute independent instructions until it reaches an instruction which relies on the result of the long-latency operation. The CP then stalls for the corresponding result to appear in the CP embedding response queue.

Although sharing the long-latency functional units helps reduce the size of the CP, we would still need the following six memory ports for a Maven VT core with a single vector lane: CP instruction port, CP load port, CP store port, VIU vector-fetched instruction port, VMU load port, and VMU store port. Although it is possible to merge the CP load and store port, Maven instead shares the three CP ports with the three VTU ports (shown at the bottom of Figure 5.1). Three two-input arbiters use round-robin arbitration to choose between the two types of requests. In addition, both the CP and the VTU share a small L1 instruction cache.

Note that a unified VT instruction set helps simplify CP embedding. Since both types of threads execute the same long-latency instructions, they can share the same long-latency functional units without modification. CP embedding is equally possible in vector-SIMD architectures, and it is particularly well suited to single-lane implementations. Embedding a control processor in a multi-lane vector unit or VTU requires one lane to be engineered differently than the rest which might complicate the design. CP embedding helps reduce CP area, and the only significant disadvantage is that the CP will experience poor performance on compute or memory-intensive tasks executing in parallel with the microthreads. However, this is rare, since the CP executes the more control oriented portions, while the microthreads execute the more compute and memory-intensive portions.

5.3 Vector Fragment Merging

Once microthreads diverge there is no way for them to reconverge until the next vector instruction, e.g., a vector memory instruction or vector fetch instruction. Separately executing multiple fragments that could be executed as a merged single fragment will result in lower performance and energy efficiency. Vector fragment merging is a hardware mechanism that can cause microthreads to reconverge dynamically at run-time. This dynamic hardware approach complements the Maven programming methodology described in Chapter 6, which allows programmers to statically reconverge microthreads by splitting a large vector-fetched block into multiple smaller vector-fetched blocks.

Vector fragment merging only requires modifications to the fetch/decode stage in the VIU. The general approach is to merge fragments with the same program counter into a single fragment, with the corresponding mask values combined using a logic "or" operation. The challenges are determining when to merge fragments and choosing the fragment execution order so as to maximize merging. One option is to associatively search the PVFB on every cycle to see if any other fragments have the same program counter as the currently executing fragment. Unfortunately, this frequent associative search can be expensive in terms of energy. A cheaper alternative is for the VIU to only search the PVFB when executing control-flow instructions. This can capture some common cases such as multiple branches with the same target address often used to implement more complicated conditionals or the backward branches used to implement loops. It also allows a compiler to encourage reconvergence at known good points in the program by inserting extra unconditional jump instructions that target a common label as hints for the hardware to attempt merging. Vector fragment merging is equally applicable to both multi-lane and single-lane VTUs.

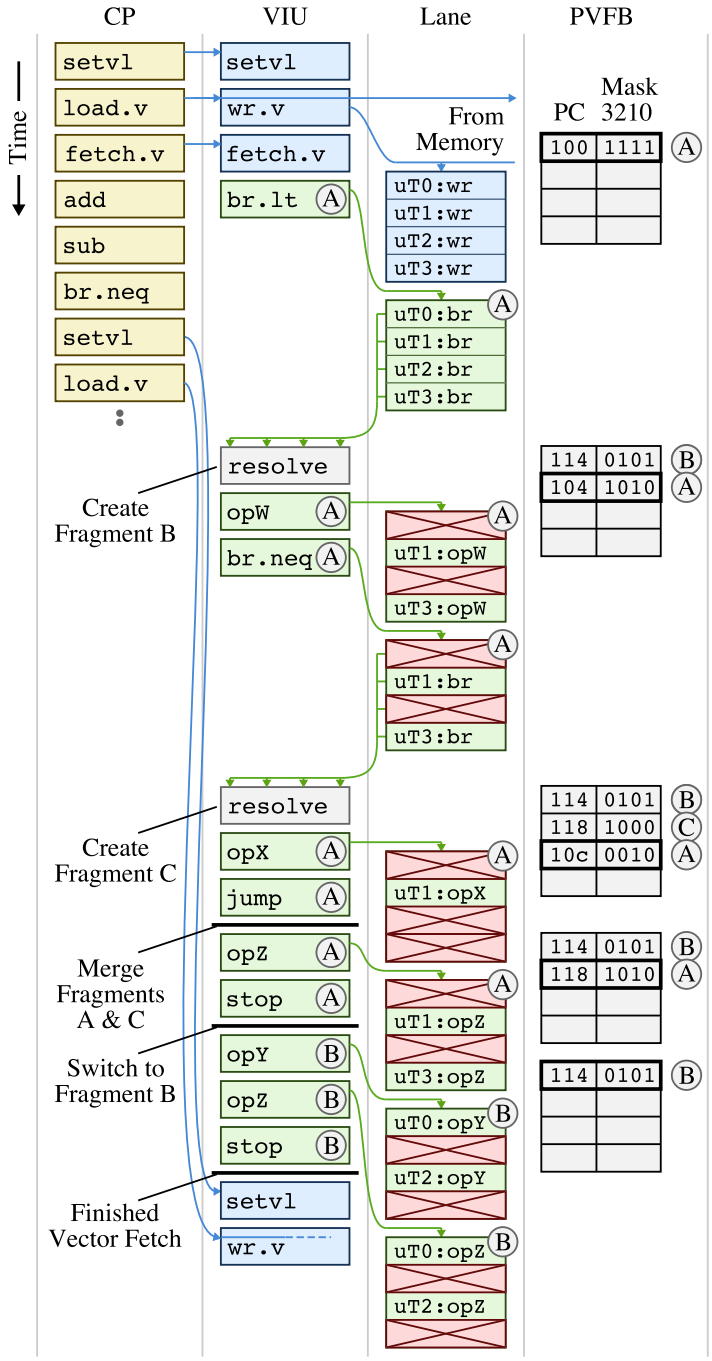
Figure 5.3 illustrates the vector fragment merging technique for the same example used in Figure 5.2. When the VIU executes the unconditional jump at address `0x110`, it searches the PVFB and notices that both the current fragment A and the pending fragment C are targeting the same program counter (`0x118`). The VIU then merges the microthread active in fragment C (μT3) with the microthread in the current fragment A (μT1), and continues executing the newly merged fragment A. Unfortunately this technique is not able to uncover all opportunities for reconvergence, since there is no way for the hardware to know that fragment B will eventually also reach the instruction at address `0x118`. Note that searching the PVFB every cycle does not increase microthread reconvergence in this example, but changing the order in which fragments are executed could help. For example, if the VIU switched to fragment B after merging fragments A and C, and the VIU also searched the PVFB every cycle, then it would successfully reconverge all four microthreads at instruction address `118`. Developing a simple yet effective heuristic for changing the fragment execution order to encourage merging is an open research question.

```
for ( i = 0; i < n; i++ )
  if ( A[i] >= 0 )
    opW;
    if ( A[i] == 0 )
      opX;
  else
    opY;
  opZ;
```

(a) Pseudocode

```
loop:
 000: setvl   vlen, n
 004: load.v  VA, a_ptr
 008: fetch.v ut_code
 00c: add     a_ptr, vlen
 000: sub     n, vlen
 014: br.neq  n, 0, loop
 ...

ut_code:
 100: br.lt   a, 0, else
 104: opW
 108: br.neq  a, 0, done
 10c: opX
 110: jump    done
else:
 114: opY
done:
 118: opZ
 11c: stop
```

(b) Pseudo-Assembly



(c) Maven Execution Diagram

Figure 5.3: Example of Vector Fragment Merging – Example pseudocode (a) and pseudo-assembly (b) is the same as in Figure 5.2. Execution diagram (c) illustrates how fragments A and C are able to reconverge when fragment A executes the uncoditional jump instruction. (Code syntax similar to that used in Figure 2.14. Vector memory unit not shown for simplicity. CP = control processor, VIU = vector issue unit, PVFB = pending vector fragment buffer.)

5.4 Vector Fragment Interleaving

One of the primary disadvantages of the basic vector-fetched scalar branch mechanism is the long branch resolution latency. The benefit of waiting is that the VIU is able to maintain partial vector-like efficiencies for vector fragments after the branch, possibly even keeping all the microthreads coherent if all branches are taken or not-taken. Unfortunately, the long branch resolution latency also leads to idle execution resources as shown in Figure 5.2. The performance impact is actually worse than implied in Figure 5.2, since Maven includes five vector functional units that will all be idle waiting for the branch to resolve. Vector fragment interleaving is a technique that hides the branch resolution latency (and possibly other execution latencies) by switching to a new fragment before the current fragment has completed. Since no two fragments contain the same microthread, fragments are always independent and their execution can be interleaved arbitrarily.

As with vector fragment merging, vector fragment interleaving only requires modification to the fetch/decode stage of the VIU. The general approach is for the VIU to choose a different fragment for execution every cycle. Standard scheduling techniques for multithreaded MIMD architectures are applicable. The VIU can obliviously rotate around the fragments in the PVFB, or can only select from those fragments that are not currently stalled due to a branch, data dependency, structural hazard, or unready memory port. The regular way in which fragments access the vector register file, even when some number of microthreads are inactive, means that the standard VIU issue stage will work correctly without modification. A simpler interleaving scheme, called *switch-on-branch*, focuses just on hiding the branch resolution latency. When the VIU encounters a vector-fetched scalar branch it immediately switches to a new fragment. The VIU only returns to the original fragment when it encounters another branch or the current fragment is completed. Vector fragment merging is equally applicable to both multi-lane and single-lane VTUs.

Figure 5.4 illustrates the vector fragment interleaving technique for the same example used in Figure 5.2. When the VIU executes the conditional branch `br.lt` at address `0x100` as part of fragment A, it still must wait for the branch to resolve because there are no other pending fragments in the PVFB. When the VIU executes the conditional branch `br.neq` at address `0x108`, then it can immediately switch to the pending fragment B. The VIU executes fragment B to completion before switching back to fragment A, which is the only remaining fragment. The VIU still needs to wait for the branch to resolve, but the vector lane is kept busy executing the micro-ops for fragment B. Eventually the branch resolves, and execution proceeds similar to Figure 5.2.

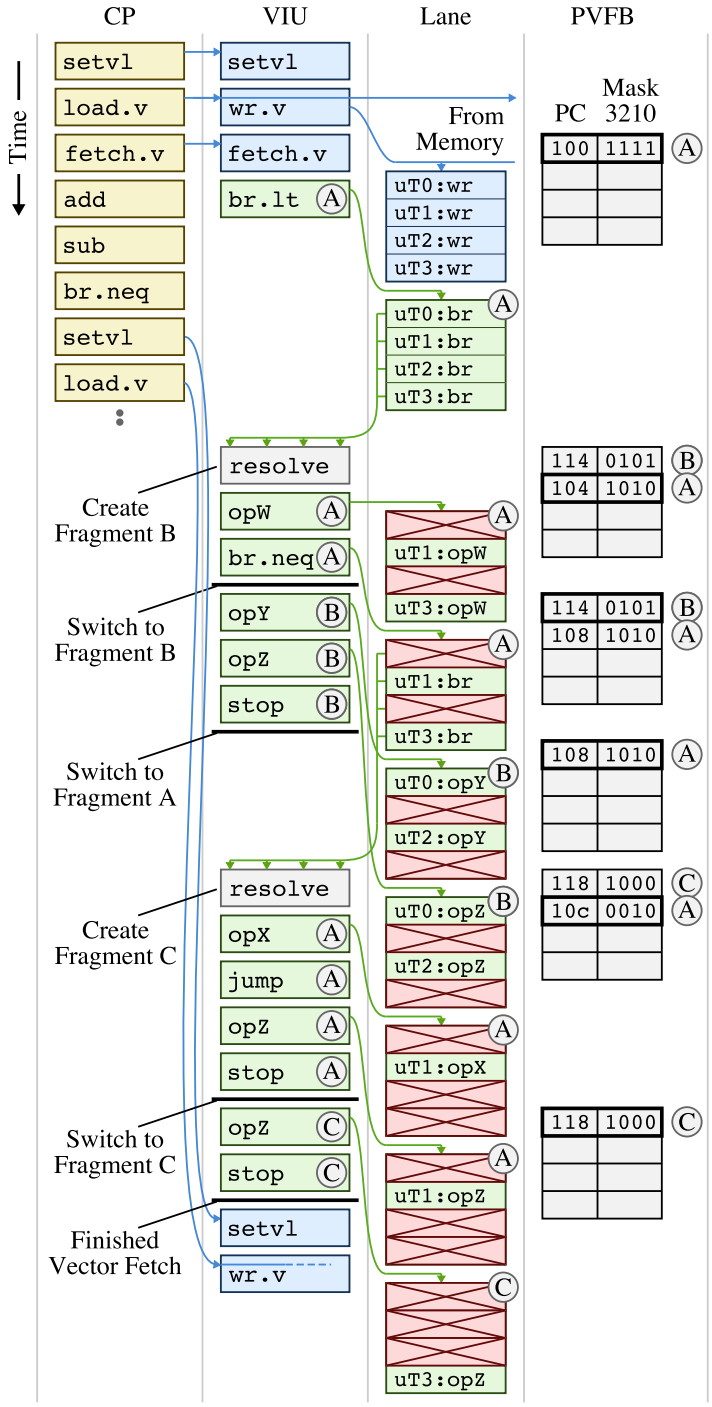
Although it is possible to combine vector fragment merging and interleaving, the fragment interleaving meant to hide execution latencies may not produce the best opportunities for merging. Even if the PVFB is searched every cycle for fragments that can be merged, the best we can hope for is serendipitous reconvergence. This reinforces the importance of statically managed reconvergence by partitioning large vector-fetched blocks into multiple smaller vector-fetched blocks. Each vector fetch instruction (or any other vector instruction) forces reconvergence across the microthreads.

```
for ( i = 0; i < n; i++ )
  if ( A[i] >= 0 )
    opW;
    if ( A[i] == 0 )
      opX;
  else
    opY;
  opZ;
```

(a) Pseudocode

```
loop:
 000: setvl   vlen, n
 004: load.v  VA, a_ptr
 008: fetch.v ut_code
 00c: add     a_ptr, vlen
 000: sub     n, vlen
 014: br.neq  n, 0, loop
 ...

ut_code:
 100: br.lt   a, 0, else
 104: opW
 108: br.neq  a, 0, done
 10c: opX
 110: jump    done
else:
 114: opY
done:
 118: opZ
 11c: stop
```

(b) Pseudo-Assembly



(c) Maven Execution Diagram

Figure 5.4: Example of Vector Fragment Interleaving – Example pseudocode (a) and pseudo-assembly (b) is the same as in Figure 5.2. Execution diagram (c) illustrates how the VIU can switch to fragment B while waiting for fragment A's branch to resolve. (Code syntax similar to that used in Figure 2.14. Vector memory unit not shown for simplicity. CP = control processor, VIU = vector issue unit, PVFB = pending vector fragment buffer.)

5.5 Vector Fragment Compression

As microthreads diverge, more of the execution resources for each fragment are spent on inactive microthreads. The implementation described so far has the nice property that fragments are always executed regularly regardless of the number of active microthreads. The dependency and structural hazard logic in the issue stage of the VIU can ignore the number of active microthreads, since the vector register file and vector functional units are always accessed in the exact same way. Unfortunately, codes with large amounts of divergence will experience poor performance and energy overheads associated with processing the inactive microthreads. Vector fragment compression is a hardware technique that skips inactive microthreads so that the fragment micro-ops execute in density-time as opposed to vlen-time.

Vector fragment compression requires more modifications to the VTU than the previous techniques. The VIU fetch/decode stage remains essentially the same, but the issue stage is slightly more complicated. The changes to the issue stage will be discussed later in this section. In addition to modifications in the VIU, the vector functional unit controllers in the vector lane need to handle compressed fragments. In a vlen-time implementation, the controllers simply increment the register read/write address by a fixed offset every cycle, but with vector fragment compression the controllers need more complicated register indexing logic that steps through sparse microthread mask to generate the appropriate offset. The VMU does not require any modifications, since it already needed to handle a variable number of microthread loads and stores which occur after the microthreads diverge.

Figure 5.5 illustrates the vector fragment compression technique for the same example used in Figure 5.2. Notice that after the VIU resolves the first vector-fetched scalar branch, fragment A only contains two microthreads. In Figure 5.2, micro-ops for fragment A still take four execution slots (vlen-time execution), but with vector fragment compression they now take only two execution slots (density-time execution). This figure illustrates, that with vector fragment compression, VIU fetch bandwidth is more likely to become a bottleneck. Divergence effectively requires fetch bandwidth proportional to the number of pending vector fragments, but this only becomes significant once we use vector fragment compression to reduce functional unit pressure. Although it is possible to implement a multi-issue VIU to improve performance on highly irregular DLP, Maven uses the simpler single-issue VIU design.

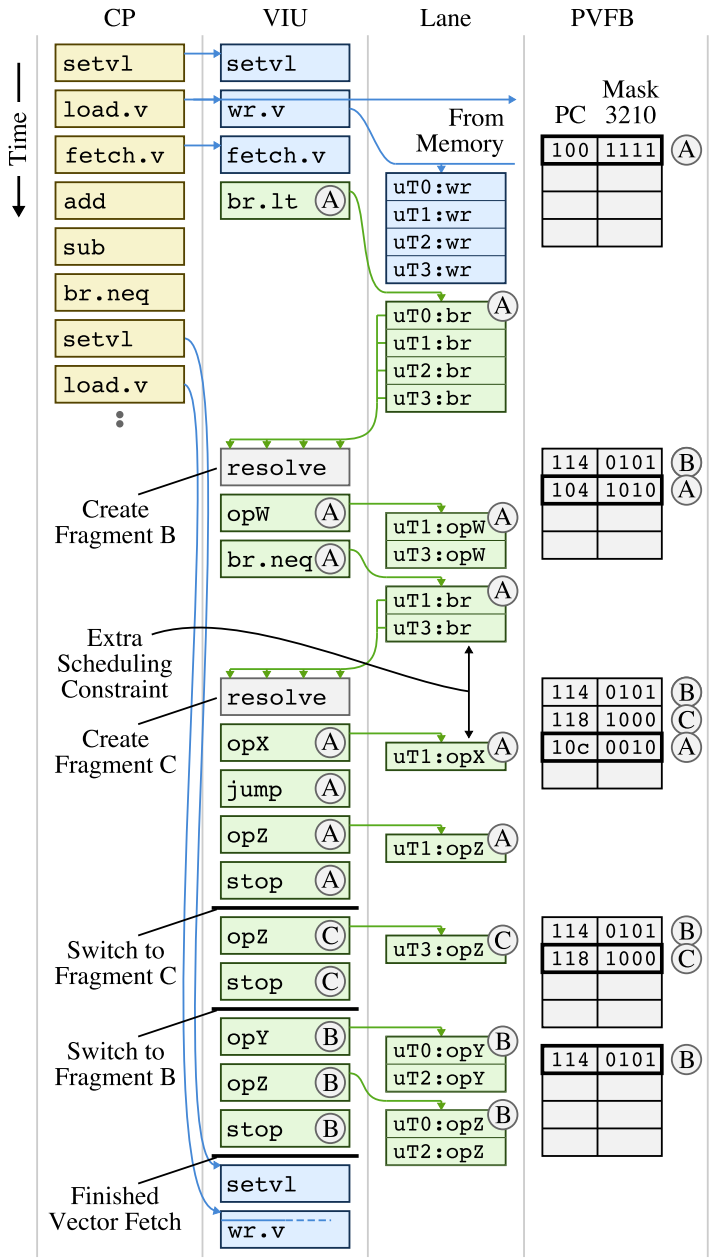
Figure 5.5 highlights that an extra scheduling constraint is required after a branch. In reality, this extra constraint is only required when the vector unit supports multiple vector functional units. Figure 5.6 shows two example executions for a vector unit similar to Maven with two vector functional units. In these figures, the read and write ports for each vector functional unit are shown as separate columns. Arrows connect the time when the read port reads the sources for a given micro-op and the time when the write port writes the corresponding result. We assume these are long-latency micro-ops. Arrows also connect dependent microthreads across micro-ops. Figure 5.6a illustrates

```
for ( i = 0; i < n; i++ )
  if ( A[i] >= 0 )
    opW;
    if ( A[i] == 0 )
      opX;
  else
    opY;
  opZ;
```

(a) Pseudocode

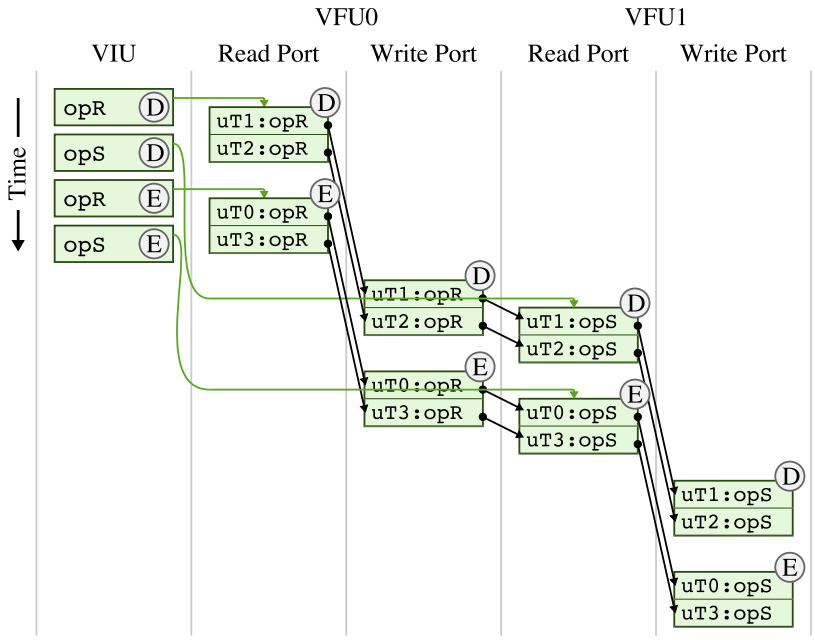
```
loop:
 000: setvl   vlen, n
 004: load.v  VA, a_ptr
 008: fetch.v ut_code
 00c: add     a_ptr, vlen
 000: sub     n, vlen
 014: br.neq  n, 0, loop
 ...

ut_code:
 100: br.lt   a, 0, else
 104: opW
 108: br.neq  a, 0, done
 10c: opX
 110: jump    done
else:
 114: opY
done:
 118: opZ
 11c: stop
```

(b) Pseudo-Assembly

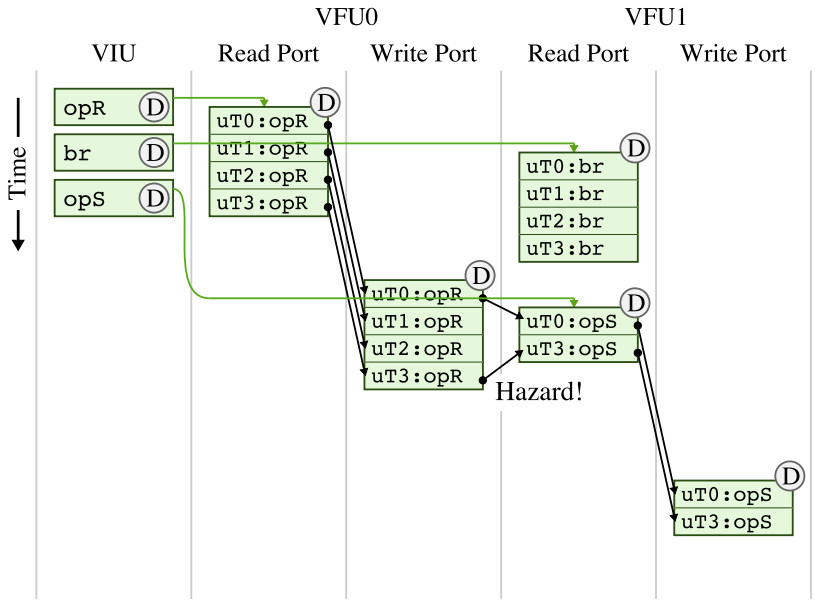


(c) Maven Execution Diagram

Figure 5.5: Example of Vector Fragment Compression – Example pseudocode (a) and pseudo-assembly (b) is the same as in Figure 5.2. Execution diagram (c) illustrates how the vector lane can skip inactive microthreads for each fragment, but there is an additional scheduling constraint between fragments that are not independent *and* are compressed differently. (Code syntax similar to that used in Figure 2.14. Vector memory unit not shown for simplicity. CP = control processor, VIU = vector issue unit, PVFB = pending vector fragment buffer.)



(a) Scheduling Multiple Independent Compressed Micro-Ops



(b) Scheduling Compressed Micro-Ops After a Branch

Figure 5.6: Extra Scheduling Constraints with Vector Fragment Compression – (a) micro-ops from the same fragment and from multiple independent fragments can go down the pipeline back-to-back; (b) micro-ops which are compressed differently after a branch as compared to before the branch must be delayed until all micro-ops before the branch have drained the pipeline. (VIU = vector issue unit, VFU = vector functional unit)

the common case. Micro-ops from the same compressed fragment (opR and opS) can go down the pipeline back-to-back, and the result from the first micro-op can be chained to the second micro-op. Micro-ops from fragment D and fragment E can also go down the pipeline back-to-back, even though they are compressed differently because they are independent. Because these two fragments contain completely different microthreads, there can be no data-dependency hazards between them. Figure 5.6b illustrates the subtle case where we must carefully schedule a compressed fragment. This occurs when a fragment is compressed differently before and after a branch. In this example, the VIU might issue micro-op opS because the dependency on the first microthread for opR is resolved. Remember, that the VIU only tracks the first microthread in a fragment to amortize interlocking and dependency tracking logic. Unfortunately, because the fragment is compressed differently after the branch, µT3 of micro-op opS can read its sources before µT3 of micro-op opR has finished writing them. To prevent this hazard and also avoid per-element dependency logic, the VIU must wait for micro-ops before a branch to drain the pipeline before issuing any micro-ops after the branch. This is simple but overly conservative, since the VIU really only needs to wait for dependent micro-ops. The VIU does not need to wait if the fragment does not diverge, since the same compression will be used both before and after the branch.

Vector fragment compression can be easily combined with fragment merging. This might also reduce fetch pressure, since fragment merging reduces the number of fragments that need to be executed. Combining fragment compression with arbitrary fragment interleaving is also possible, and can help hide the extra scheduling constraint. Vector fragment compression is difficult to implement in multi-lane VTUs, because the lanes need to be decoupled to enable independent execution. Decoupling the lanes is already a significant step towards single-lane VTUs, but fully separate single-lane VTUs are more flexibly and easier to implement. The simplicity of implementing vector fragment compression is one reason Maven uses these kind of VTUs.

5.6 Leveraging Maven VT Cores in a Full Data-Parallel Accelerator

Figure 5.7 illustrates how Maven VT cores can be combined to create a full data-parallel accelerator. Each core includes a small private L1 instruction cache for instructions executed by both the control thread and the microthreads, while data accesses for both types of threads go straight to a banked L2 cache. To help amortize the network interface and exploit inter-core locality, four cores are clustered together to form a *quad-core tile*. Each tile contains an on-chip network router and four cache banks that together form the L2 cache shared by the four cores (but private with respect to the other quad-core tiles). The four cores are connected to the four L2 cache banks via the memory crossbar, while the L2 cache banks are connected to the on-chip network router via the network interface crossbar. To balance the core arithmetic and memory bandwidth, the channels between the cores and the L2 cache banks are all scalar width (32 b/cycle), while the channels in the network interface and in the on-chip network itself are much wider (256 b/cycle).

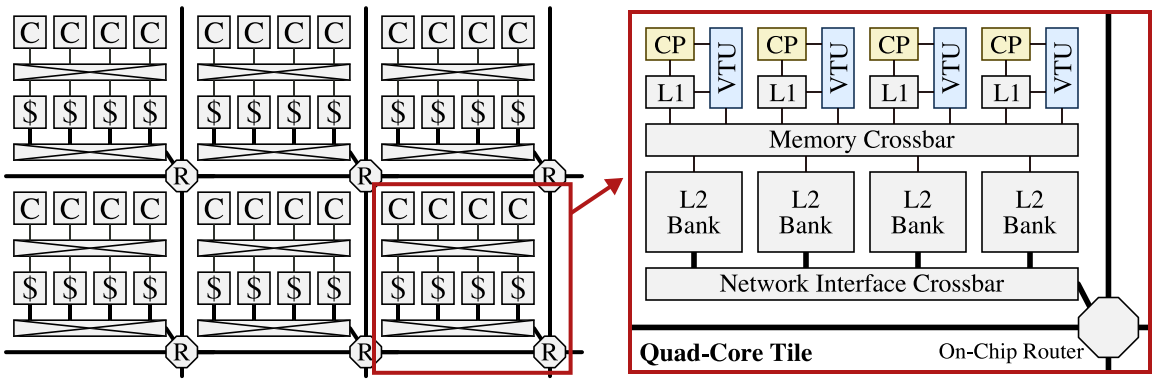


Figure 5.7: Maven Data-Parallel Accelerator with Array of VT Cores – A Maven VT core includes a control processor, single-lane VTU, and small L1 instruction cache. Four cores are combined into a quad-core tile along with four shared L2 cache banks and a shared network interface. (C = Maven GT core, $ = L2 cache bank, R = on-chip network router, CP = control processor, VTU = vector-thread unit, — = narrow 32-bit channel, ━ = wide 256-bit channel)

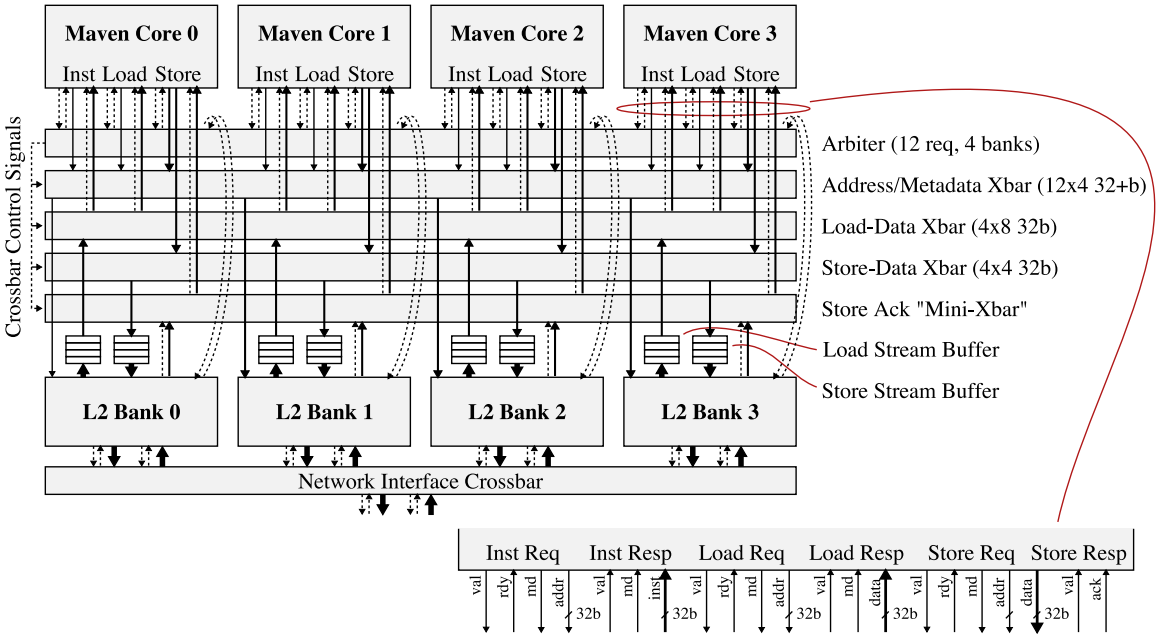


Figure 5.8: Microarchitecture for Quad-Core Tile – Four maven VT cores (each with its own control processor, single-lane VTU, and L1 instruction cache as shown in Figure 5.1) are combined into a quad-core tile with a four-bank L2 cache shared among the four lanes. Each tile shares a single wide (256 b) network interface. (Control signals shown with dashed lines. Xbar = crossbar, val = valid signal, rdy = ready signal, md = metadata, addr = address, ack = acknowledgement, *N*x*M* crossbar = *N* inputs and *M* outputs)

Figure 5.8 shows a possible microarchitecture for the quad-core tile. Each core has separate instruction, load, and store ports resulting in 12 requesters arbitrating for the four L2 cache banks. There are dedicated load/store stream buffers for each core next to each L2 cache bank to enable wide bank accesses that are then streamed to or from the cores. The memory crossbar is similar to the intra-VMU crossbar found in multi-lane vector units, but is a little simpler. The memory crossbar in Figure 5.8 requires less muxing, since the element rotation is implemented temporally in the single-lane's VMU and associated stream buffers. The design sketched in Figures 5.7 and 5.8 is just one way single-lane VT cores can be combined into a data-parallel accelerator. The on-chip cache hierarchy, on-chip interconnection network, and off-chip memory system are all important parts of data-parallel accelerators, but a full investigation of this design space is beyond the scope of this thesis. Instead, this thesis focuses on just the specialized data-parallel cores that are the primary distinguishing characteristic of such accelerators.

5.7 Extensions to Support Other Architectural Design Patterns

We would like to compare the energy-efficiency and performance of Maven to the other architectural design patterns presented in Chapter 2, but we do not necessarily want to build several completely different implementations. This section describes how the Maven microarchitecture can be extended to emulate the MIMD, vector-SIMD, and SIMT architectural design patterns for evaluation purposes. These extensions leverage the instruction set extensions described in Section 4.6.

5.7.1 MIMD Extensions to Maven Microarchitecture

Our multithreaded MIMD core is based on the Maven control processor with a larger architectural register file to handle additional microthread contexts. The larger register file always requires only two read ports and two write ports. Although the MIMD core does not include the VTU, it does include the long-latency functional units as a separate decoupled module directly attached to the same control processor embedding queues used in Maven. This allows the exact same issue logic to be used in both the Maven control processor and the MIMD core. The MIMD core is similar to the control processor in its ability to overlap short-latency instructions with decoupled loads and long-latency operations. Obviously all of the vector instruction decode logic is not present in the MIMD core, and there is no need to share the memory ports. The MIMD core also include additional thread scheduling logic to choose between the active microthreads using a simple round-robin arbiter. The ability to mark microthreads as active or inactive and thus remove them from the scheduling pool improves performance on serial codes. This also means that instructions from the same thread can flow down the pipeline back-to-back, so our implementation provides full bypassing and interlocking logic.

5.7.2 Vector-SIMD Extensions to Maven Microarchitecture

Our vector-SIMD core use a very similar microarchitecture to the one shown in Figure 5.1 with additional support for executing the vector-SIMD instruction set shown in Tables 4.4 and 4.5. The vector lane for the vector-SIMD core removes the BRMRs but adds a separate vector flag register file and a new vector flag functional unit (VFFU). The vector flag register file includes eight 32-bit entries, four read ports, and three write ports. VFU0 and VFU1 can both read an entry in the vector flag register file (to support vector arithmetic operations executed under a flag) and also write an entry in the vector flag register file to support the flag integer and floating-point compare instructions. The single-cycle VFFU requires two read ports and one write port to execute the simple vector flag arithmetic instructions (e.g., `or.f` and `not.f`).

The VIU issue stage is the same in both the Maven VT core and the vector-SIMD core. The vector-SIMD fetch/decode stage does not include the PVFB nor the extra vector-fetched instruction port. This means the control processor has a dedicated instruction port and L1 instruction cache in the vector-SIMD core. The vector-SIMD instruction set extensions were specifically designed to mirror their scalar counterparts. For example, the vector-SIMD `add.s.vv` instruction is executed in a similar way as a vector-fetched scalar `add.s` instruction. The fetch/decode stage takes vector instructions from the VIU queue and converts them into the same internal micro-op format used by the Maven VT core except without the microthread mask. The vector flag registers are read at the end of the VFU0 and VFU1 pipelines to conditionally write the results back to the vector register file as opposed to the PVFB which is at the front of the vector pipeline. This allows chaining flag values between vector instructions. The VIU queue needs to handle an extra scalar operand for vector-scalar instructions, but this space is also required in the Maven VT core for some instructions (i.e., the base address for vector memory instructions). This scalar value is propagated along with the vector instruction to the vector lane similar to Maven's vector-scale move instruction (`mov.sv`).

Unit-stride and strided vector memory instructions are executed almost identically in the vector-SIMD core as in the Maven VT core. Indexed vector memory instructions are similar to microthread loads/stores, except that a base address is included from the control-thread's scalar register file. The VMU then requires an additional address generation unit for adding the indexed base register to the offsets being streamed from the vector register file by the VAU.

In addition to a single-lane vector-SIMD core, we also want to compare our single-lane Maven VT core to multi-lane vector-SIMD cores. For these configurations, we simply instantiate multiple vector lane modules and broadcast the VIU control information to all lanes. The VIU needs to be aware of the longer hardware vector lengths, but is otherwise identical to the single-lane configuration. Unfortunately, the multi-lane VMU is different enough from the single-lane implementation to require a completely different module. The multi-lane VMU must manage a single set of address generators and multiple VAQs, VSDQs, and VLDQs such that they can be used in both unit-stride, strided, and indexed accesses. The control processor is embedded in the single-lane configuration,

and currently it is also embedded in the multi-lane configurations to simplify our evaluation. A different design might use a completely separate control processor in wide multi-lane vector-SIMD cores, to avoid wasting many execution slots across the lanes for each long-latency control processor operation.

5.7.3 SIMT Extensions to Maven Microarchitecture

The SIMT pattern is emulated by running software using a subset of the Maven VT instruction set on an unmodified Maven VT core. Although this provides a reasonable emulation, there are some important caveats. First, our SIMT core includes a control processor, yet one of the key aspects of the SIMT pattern is that is specifically lacks such a control processor. We address this discrepancy by limiting the code which executes on the control processor. Code which still runs on the control processor resembles some of the fixed hardware logic that would be found in a SIMT core's VIU but is probably less efficient than an actual SIMT core. In addition, the Maven VT core does not include any special memory coalescing hardware so the microthread accesses in our SIMT emulation are always executed as scalar loads and stores. The VT execution mechanisms do, however, reasonably capture the control-flow characteristics of a SIMT core including vector-like execution of coherent microthreads and efficient divergence management.

5.8 Future Research Directions

This section briefly describes some possible directions for future improvements with respect to the Maven microarchitecture.

Support for New Instructions – Section 4.6 discussed several new instructions that would affect the microarchitecture in non-trivial ways. Some of these extensions, such as vector compress and expand instructions and vector shared registers, are more straight-forward to implement in a single-lane implementation, since they would otherwise require cross-lane communication in a multi-lane implementation. Integrating vector segment accesses into the current Maven pipeline is an open research question, since these operations write to the register file in a non-standard way. The Scale technique leverages atomic instruction block interleaving which is not possible in the Maven VT core. Vector fetch atomic instructions could be implemented specially in the VIU. When a VIU encounters this instruction it then executes the first microthread in the current fragment by itself until it reaches a `stop` instruction. The VIU continues serializing the execution of all microthreads. When finished with the vector fetch atomic instruction, the VIU can start executing the next vector instruction as normal. A small instruction buffer in the VIU can help amplify the instruction fetch bandwidth needed for executing these atomic blocks. Combining atomic blocks with vector fragment compression might provide an efficient mechanism for handling cross-iteration dependencies via shared registers, similar to the atomic instruction blocks used in Scale.

Compiler Assistance for Vector Fragment Merging – As discussed in Section 5.3, effective hardware policies for choosing the microthread execution order to promote vector fragment merging is a difficult problem. Static compiler analysis could definitely help the hardware determine when to switch fragments. For example, the compiler could generate a reconvergence `wait` instruction at points in the instruction stream where it can statically determine that most or all of the microthreads should reconverge. When the VIU encounters a `wait` instruction it marks the corresponding fragment in the PVFB as waiting and switches to another non-waiting fragment. When the VIU encounters another `wait` instruction it merges the corresponding microthreads into the waiting fragment and switches to another non-waiting fragment. Eventually the VIU will have executed all microthreads such that they have completed or they are part of the waiting fragment. The VIU can then go ahead and proceed to execute the microthreads that have reconverged as part of the waiting fragment. A `wait` instruction that supports multiple waiting tokens enables the compiler to generate multiple independent wait points. Since this instruction is only a hint, the compiler can be optimistic without concern for dead-lock. The VIU is always allowed to simply ignore the `wait` instruction, or start executing a waiting fragment prematurely.

Vector-Fetched Scalar Branch Chaining – Although vector fragment interleaving helps hide the vector-fetched scalar branch resolution latency, it does not help when there are no pending fragments. For example, the VIU still needs to wait for the first branch to fully resolve in Figure 5.3c. A complementary technique can predict that at least one microthread will not take the branch. Since the branches for each microthread are resolved incrementally, the VIU can go ahead and issue the first fall-through instruction before the BRMR is completely written. We can essentially chain the BRMR to the instruction on the fall-through path. If the first microthread does not take the branch the result is written back into the vector register file as normal, but if the first microthread does take the branch then the writeback is disabled. The same process occurs for all microthreads in this first fall-through instruction and for later instructions on the fall-through path. When the original vector-fetched scalar branch is completely resolved the VIU will take one of three actions. If the BRMR is all zeros (no microthreads take the branch) then the VIU simply continues executing the fall-through path. If the BRMR is all ones (all microthreads take the branch) then the VIU has executed several cycles of wasted work and switches to executing the taken path. If the microthreads diverged, then the VIU creates a new pending fragment to capture the taken path and continues along the fall-through path. This technique closely resembles how vector flags are chained in the vector-SIMD core, but it also has the advantages of being a vector-fetched scalar branch. The disadvantage is wasted work when all microthreads take the branch. This technique can be combined with vector fragment merging and interleaving, but might be difficult to combine with vector fragment compression. One option is just to avoid compressing the speculative fall-through path.

Vector Register File Banking – he current Maven vector register file is quite large due to the six read and three write ports required to support the five vector functional units. We could reduce the

number of vector functional units but this would significantly impact performance. An alternative is to exploit the structured way in which vector fragment micro-ops access to the vector register file. Since a vector fragment micro-op can only access a given element on a specific cycle, we element-partition the vector register file into multiple banks each with fewer ports. This technique is common in some kinds of traditional vector-SIMD machines [Asa98]. For example, assume we implement the Maven vector register file with four banks each with two read and two write ports. Vector elements are striped across the banks such that each bank contains one-fourth of the total number of vector elements. A vector fragment micro-op would cycle around the banks as it stepped through the vector elements. Successive vector instructions follow each other as they cycle around the banks in lock-step resulting in no bank conflicts. Two read ports and one write port would be for general use by VFU0, VFU1, VAU, and VSDRU. The second write port would be dedicated for use by the VLDWU. This allows all five vector functional units to be accessing the vector register file at the same time, but with many fewer read and write ports per register bitcell. Ultimately, this results in a smaller, faster, and more energy-efficient vector register file. Vector fragment merging and interleaving can use a banked vector register file without issue. Banking does complicate vector fragment compression, since compressed fragments access the vector register file in a less structured way. The solution is to limit the amount of compression which is supported. For example, with four banks a fragment could not be compressed to less than four microthreads even if less than four microthreads were active.

5.9 Related Work

This section highlights selected previous work specifically related to the Maven VT core microarchitecture presented in this chapter.

Scale VT Processor – Scale uses a multi-lane VTU with a very different microarchitecture than the one presented in this chapter [KBH+04a]. The four Scale lanes and the four functional units within each lane are all highly decoupled. As a result, the Scale microarchitecture is more complicated but also supports a more dynamic style of execution, while the Maven microarchitecture is simpler and supports a style of execution closer to the vector-SIMD pattern. The Scale microthread control-flow mechanism uses a significantly different implementation than Maven. Upon a thread-fetch, the Scale VIU immediately begins fetching new scalar instructions for just that microthread. The Scale VIU does not wait in an effort to exploit partial vector-like efficiencies as in the Maven VIU. This is analogous to only supporting vector fragments that either contain a single active microthread (after a thread-fetch) or all active microthreads (after a vector fetch). Scale provides small instruction caches in each lane in an attempt to dynamically capture some of the locality inherent when microthreads have coherent control flow. The vector fragment mechanisms attempt to preserve as much vector-like efficiencies as possible even after the microthreads have diverged. Scale's thread-fetch mechanisms naturally produce a density-time implementation, while Maven can eventually use vector

fragment compression to eliminate inactive microthreads. Maven executes a vector-fetched scalar instruction for all microthreads before moving to the next vector-fetched scalar instruction, while Scale executes several instructions for one microthread before switching to execute those same instructions for the next microthread. This is called atomic instruction-block interleaving, and it allows more efficient use of temporary architectural state (such as functional unit bypass latches) but requires more complicated vector functional unit controllers. The Scale controllers continually toggle control lines as they step through the instructions for each microthread; in Maven, the control lines stay fixed while executing each instruction completely. Scale uses software-exposed clusters to avoid a large multi-ported vector register file, while in the future a Maven VT core will likely use a banked vector register file for the same purpose.

Cray-1 Single-Lane Vector-SIMD Unit and Embedded Control Processor – The Cray-1 vector supercomputer uses a single-lane vector unit with six functional units and eight vector registers each containing 64×64-bit elements [Rus78]. The control processor is partially embedded, since it shares the long-latency floating-point functional units with the vector lane but has its own data port to memory. In some sense, Maven's focus on single-lane VTUs is reminiscent of the earliest vector processor designs except with increased flexibility through VT mechanisms.

Cray X1 Multi-Streaming Processors – The more recent Cray-X1 can have hundreds of vector-SIMD cores *single-streaming processors* (SSPs), where each SSP is a separate discrete chip containing a control processor and a two-lane vector unit [DVWW05]. Four SSPs are packaged with 2MB of cache to create a *multi-streaming processor* (MSP) module. For irregular DLP, the SSPs can operate as independent vector-SIMD cores each supporting 32 vector registers with 64 elements each. For regular DLP, the four SSPs in a MSP can be ganged together to emulate a larger vector-SIMD core with eight lanes and a hardware vector length of 256. For efficient emulation, the Cray X1 includes fast hardware barriers across the four SSPs and customized compiler support for generating redundant code across the four control processors.

Partitioned Vector-SIMD Units – Rivoire et al. also observe that data-parallel application performance varies widely for a given number of lanes per vector unit [RSOK06]. Some applications with regular DLP see great benefit from increasing the number of lanes, while other applications with irregular DLP see little benefit. In Maven, this observation has led us to pursue single-lane VTUs, while Rivoire et al. propose a different technique that can dynamically partition a wide vector unit into multiple narrow vector units. With an eight-lane vector unit, a single control-thread can manage all eight-lanes, two control-threads can manage four lanes each, or eight control-threads can each manage a single lane. This final configuration resembles our approach, except that multiple control-threads are mapped to a single control processor instead of the embedded control processor per lane used in Maven. There are significant overheads and implementation issues associated with dynamically partitioning a wide vector unit including increased instruction fetch pressure. Maven

takes the simpler approach of only providing single-lane VTUs with embedded control processors. Temporal vector execution provides competitive energy-efficiency, while independent single-lane cores provide good flexibility. The partitioned vector-SIMD approach, where a wide vector unit is partitioned into multiple smaller vector units, can be seen as the dual of the Cray X1 MSPs, where multiple smaller vector units are ganged together to create a larger vector unit [DVWW05].

Larrabee's Multithreaded Control Processor – The Larrabee accelerator uses a multithreaded control processor with four control threads and a single 16-lane vector unit with a hardware vector length of 16 [SCS+09]. The Larrabee vector register file is large enough to support four independent sets of architectural vector registers, so effectively each control thread's logical array of microthreads are time multiplexed onto a single vector unit. An alternative design could use a single control thread and simply increase the hardware vector length to 64. Both designs require the same number of physical registers, and both designs time multiplex the microthreads. The difference is whether this time multiplexing is supported through a longer hardware vector length or through multiple control threads. Four control threads that share the same vector unit can offer more flexibility, but less energy amortization, as compared to a single control thread with a much longer hardware vector length. In addition, the extra control threads require a more aggressive control processor to keep the vector unit highly utilized, especially if each Larrabee vector instruction only keeps the vector unit busy for a single cycle. Extra control threads can help tolerate memory and functional-unit latencies, but Maven achieves a similar effect through control processor decoupling (for memory latencies) and exploiting DLP temporally (for functional unit latencies).

Single-Lane Density-Time Vector-SIMD Units – Others have proposed single-lane density-time vector units. For example, Smith et al. describe a single-lane implementation that allows the VIU to skip inactive microthreads in power-of-two increments [SFS00]. Unfortunately, such schemes significantly complicate chaining. The VIU must wait for flags to be computed first before compressing the vector instruction leading to a VIU stall similar to the vector-fetched scalar branch resolution latency. Once the VIU compresses a vector instruction there is no guarantee the next instruction can also be compressed in the same fashion meaning that the VIU may not be able to chain operations executing under a flag register. Out-of-order issue complicates the VIU but might help find an instruction with the same flag register for chaining, or an instruction with a flag register containing an independent set of microthreads for interleaving. Future Maven VT cores will be able to provide density-time execution and fragment interleaving with a much simpler implementation due to the way in which conditionals are expressed in a VT instruction set.

NVIDIA SIMT Units – The NVIDIA G80 [LNOM08] and Fermi [nvi09] families of graphics processors implement the SIMT architectural design pattern but also include microthread control-flow mechanisms similar in spirit to the vector fragment technique presented in this chapter. Although the NVIDIA SIMT units can handle microthread divergence, the exact microarchitectural mecha-

nism has not been discussed publicly. It is clear, however, that the NVIDIA SIMT units use multiple lanes without support for density-time execution. To compensate for the lack of control processor decoupling in the SIMT pattern, each SIMT unit contains architectural state for many microthread blocks. Effectively, graphics processors use highly multithreaded VIUs, and this allows the SIMT unit to hide memory and execution latencies by switching to a different microthread block. This has some similarities to providing multiple control threads. SIMT units require hundreds of microthread blocks to hide long memory latencies, while Maven control processor decoupling can achieve a similar effect with much less architectural state. NVIDIA SIMT units also support a technique similar to vector fragment interleaving, and appear to support static hints in the instruction stream for vector fragment merging.

Dynamic Fragment Formation – Vector fragment compression is difficult to implement in multi-lane implementations usually resulting in low utilization for highly divergent code. Fung et al. propose a new SIMT core design that dynamically composes microthreads from *different* microthread blocks to fill unused execution slots [FSYA09]. A similar technique is possible in vector-SIMD or VT cores that support multiple control threads by composing vector fragments from different control threads. Both approaches effectively generate new fragments at run-time that are not in the original program. This of course relies on the fact that the different microthread blocks or control threads are executing the same code. Constraints on the types of allowed composition prevents the need for a full crossbar between lanes. Unfortunately, dynamic fragment formation can interfere with memory coalescing and requires good fragment interleaving heuristics to enable opportunities for using this technique. An associative structure similar to the PVFB, but much larger to support multiple microthread blocks or control threads, is required to discover fragments that are executing the same instruction. In discussing dynamic fragment formation, Fung et al. also investigate reconvergence techniques and note the importance of good heuristics to promote vector fragment merging.

Chapter 6

Maven Programming Methodology

This chapter describes the Maven programming methodology, which combines a slightly modified C++ scalar compiler with a carefully written application programming interface to enable an explicitly data-parallel programming model suitable for future VT accelerators. Section 3.3 has already discussed how this approach can raise the level of abstraction yet still allow efficient mappings to the VT pattern. Section 6.1 gives an overview of the methodology including the VT C++ application programmer's interface and two example programs that will be referenced throughout the rest of the chapter. Section 6.2 discusses the required compiler modifications, and Section 6.3 discusses the implementation details of the VT application programmer's interface. Section 6.4 briefly outlines the system-level interface for Maven. Section 6.5 discusses how the Maven programming methodology can be leveraged to emulate the MIMD, vector-SIMD, and SIMT architectural design patterns. The chapter concludes with future research directions (Section 6.6) and related work (Section 6.7).

6.1 Programming Methodology Overview

The Maven programming methodology supports applications written in the C++ programming language. C++ is low-level enough to enable efficient compilation, yet high-level enough to enable various modern programming patterns such as object-oriented and generic programming. Maven applications can use the *vector-thread application programmer's interface* (VTAPI) to access Maven's VT capabilities. Table 6.1 lists the classes, functions, and macros that form the VTAPI. The VTAPI relies on the use of a special low-level `HardwareVector<T>` class. This class represents vector types that can be stored in vector registers or in memory (but never in control-thread scalar registers). A hardware vector contains a number of elements equal to the current hardware vector length. This class is templated based on the type of the elements contained with the hardware vector, although currently the `HardwareVector<T>` class is not fully generic. It is instead specialized for the following types: `unsigned int`, `unsigned short`, `unsigned char`, `signed int`, `signed short`, `signed char`, `char`, `float`, `T*` (any pointer type). The `HardwareVector<T>` class in-

Hardware Vector Class

`void HardwareVector<T>::load( const T* in_ptr );`	Unit-stride load
`void HardwareVector<T>::load( const T* in_ptr, int stride );`	Strided load
`void HardwareVector<T>::store( const T* out_ptr );`	Unit-stride store
`void HardwareVector<T>::store( const T* out_ptr, int stride );`	Strided store
`void HardwareVector<T>::set_all_elements( const T& value );`	`mov.sv` instruction
`void HardwareVector<T>::set_element( int idx, const T& value );`	`mtut` instruction
`T HardwareVector<T>::get_element( int idx );`	`mfut` instruction

Vector Configuration Functions

`int config( int nvregs, int app_vlen );`	Compiles into `vcfgivl` instruction
`int set_vlen( int app_vlen );`	Compiles into `setvl` instruction

Memory Fence Functions

`void sync_l();`	`void sync_g();`	Compiles into `sync.{l,g}` instruction
`void sync_l_v();`	`void sync_g_v();`	Compiles into `sync.{l,g}.v` instruction
`void sync_l_cv();`	`void sync_g_cv();`	Compiles into `sync.{l,g}.cv` instruction

Atomic Memory Operations Functions

`int fetch_add( int* ptr, int value );`	Compiles into `amo.add` instruction
`int fetch_and( int* ptr, int value );`	Compiles into `amo.and` instruction
`int fetch_or ( int* ptr, int value );`	Compiles into `amo.or` instruction

Miscellaneous Functions

`int get_utidx();`	Compiles into `utidx` instruction

Vector-Fetch Preprocessor Macro

```
VT_VFETCH( (outputs), (inouts), (inputs), (passthrus),
({
  body;
}));
```

Table 6.1: Maven VT Application Programming Interface (VTAPI) – Classes, functions, and preprocessor macros which form the Maven VTAPI. All classes and functions are encapsulated in the `vt::` C++ namespace. The templated `HardwareVector<T>` class is currently not fully generic and is instead specialized for the following types: `unsigned int`, `unsigned short`, `unsigned char`, `signed int`, `signed short`, `signed char`, `char`, `float`, `T*` (any pointer type).

cludes member functions for loading and storing elements from standard C++ arrays and for accessing the elements in the hardware vector. Additional free functions are provided for configuring the vector unit, memory fences, atomic memory operations, and accessing the current microthread's index. The key to the VTAPI is a sophisticated macro called `VT_VFETCH` that vector fetches code onto the microthreads. When using the macro, a programmer specifies the input and output hardware vectors for the vector-fetched block, and the VTAPI handles connecting the vector view of these registers with the scalar view seen by the microthreads. The VTAPI will be discussed in more detail in Section 6.3.

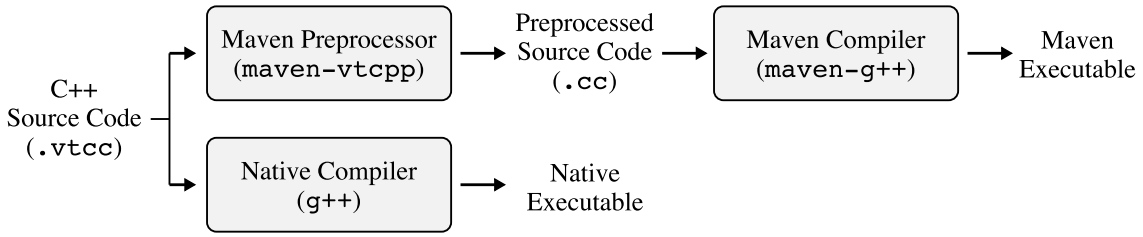


Figure 6.1: Maven Software Toolchain – Maven C++ applications can be compiled into either native executable or a Maven executable. Native executables emulate the VT application programming interface to enable rapid testing and development. A preprocessing step is required when compiling Maven executables to connect the control-thread's vector register allocation with the microthread's scalar register allocation.

Figure 6.1 illustrates the Maven software toolchain. A C++ application which uses the Maven VTAPI is first preprocessed and then compiled to generate a Maven executable (see Section 6.2 for more details on the compiler, and Section 6.3 for more details on the Maven preprocessor). The Maven executable can then be run on a functional simulator, detailed RTL simulator, or eventually a Maven prototype. A key feature of the Maven programming methodology is that all Maven applications can also be compiled with a native C++ compiler. Currently only the GNU C++ compiler is supported, since the methodology makes extensive use of non-standard extensions specific to the GNU C++ toolchain [gnu10]. Native compilation does not attempt to generate an optimized executable suitable for benchmarking, but instead generates an executable that faithfully emulates the Maven VTAPI. Native emulation of Maven applications has several benefits. Programs can be quickly developed and tested without a Maven simulator (which can be slow) or prototype (which may not be available for some time). Once a developer is satisfied that the application is functionally correct using native compilation, the developer can then start using the Maven compiler. Problems that arise when compiling for Maven can be quickly isolated to a small set of Maven-specific issues as opposed to general problems with the application functionality. In addition to faster development and testing, native executables can also be augmented with run-time instrumentation to gather various statistics about the program. For example, it is possible to quickly capture average application vector lengths and the number and types of vector memory accesses.

Figure 6.2 illustrates using the VTAPI to implement the regular DLP loop in Table 2.1c. This example compiles to the assembly code shown in Figure 4.2a. The `set_vlen` function on line 3 takes two arguments: the application vector length and the required number of vector registers. Currently, the programmer must iterate through the software toolchain multiple times to determine how many registers are needed to compile the microthread code, but it should be possible to improve the methodology such that this is optimized automatically. The `set_vlen` function returns the actual number of microthreads supported by the hardware, and then this variable is used to stripmine across the input and output arrays via the `for` loop on line 5. The additional call to `set_vlen` on line 6 allows the stripmine loop to naturally handle cases where `size` is not evenly divisible by the

```
1  void rdlp_vt( int c[], int a[], int b[], int size, int x )
2  {
3    int vlen = vt::config( 5, size );
4    vt::HardwareVector<int> vx(x);
5    for ( int i = 0; i < size; i += vlen ) {
6      vlen = vt::set_vlen( size - i );
7
8      vt::HardwareVector<int> vc, va, vb;
9      va.load( &a[i] );
10     vb.load( &b[i] );
11
12     VT_VFETCH( (vc), (), (va,vb), (vx),
13     ({
14       vc = vx * va + vb;
15     }));
16
17     vc.store( &c[i] );
18   }
19   vt::sync_l_cv();
20 }
```

Figure 6.2: Regular DLP Example Using Maven Programming Methodology – Code corresponds to the regular DLP loop in Table 2.1c and compiles to the assembly shown in Figure 4.2a. (C++ code line (3,5–6) handles stripmining, (4) copies scalar `x` into all elements of hardware vector `vx`, (9–10) unit-stride vector loads, (12) specifies input and output hardware vectors for vector-fetched block, (13–15) vector-fetched block which executes on microthreads, (17) unit-stride store, (19) vector memory fence.)

hardware vector length. Line 4 instantiates a hardware vector containing elements of type `int` and initializes all elements in the vector with the scalar value `x`. This shared variable will be kept in the same hardware vector across all iterations of the stripmine loop. Line 8 instantiates the input and output hardware vectors also containing elements of type `int`. Lines 9–10 are effectively unit-stride loads; `vlen` consecutive elements of arrays `a` and `b` are moved into the appropriate hardware vector with the `load` member function. Lines 12–15 contain the vector-fetched code for the microthreads. Line 17 stores `vlen` elements from the output hardware vector to the array `c`, and line 19 performs a memory fence to ensure that all results are visible in memory before returning from the function.

The `VT_VFETCH` macro on lines 12–15 is a key aspect of the VTAPI. The macro takes five arguments: a list of output hardware vectors, a list of hardware vectors that are both inputs and outputs, a list of input hardware vectors, a list of hardware vectors that should be preserved across the vector-fetched block, and finally the actual code which should be executed on each microthread. The programming methodology requires that the input and output hardware vectors be explicitly specified so that the compiler can properly manage register allocation and data-flow dependencies. With more sophisticated compiler analysis it should be possible to determine the input and output hardware vectors automatically. The code within the vector-fetched block specifies what operations to perform on each element of the input and output hardware vectors. This means that the C++

```
1 void idlp_vt( int c[], int a[], int b[], int size, int x )
2 {
3   int vlen = vt::config( 7, size );
4   vt::HardwareVector<int>  vx(x);
5   for ( int i = 0; i < size; i += vlen ) {
6     vlen = vt::set_vlen( size - i );
7
8     vt::HardwareVector<int*> vcptr(&c[i]);
9     vt::HardwareVector<int> va, vb;
10    va.load( &a[i] );
11    vb.load( &b[i] );
12
13    VT_VFETCH( (), (), (va,vb), (vcptr,vx),
14    ({
15      if ( va > 0 )
16        vcptr[vt::get_utidx()] = vx * va + vb;
17    }));
18
19  }
20  vt::sync_l_cv();
21 }
```

Figure 6.3: Irregular DLP Example Using Maven Programming Methodology – Code corresponds to the irregular DLP loop in Table 2.1f and compiles to the assembly shown in Figure 4.2b. Notice the vector-fetched block includes an conditional statement that compiles to a scalar branch. (C++ code line (3,5–6) handles stripmining, (4) copies scalar `x` into all elements of hardware vector `vx`, (8) copies `c` array base pointer to all elements of hardware vector `vcptr`, (10–11) unit-stride vector loads, (13) specifies input and output hardware vectors for vector-fetched block, (13–15) vector-fetched block which executes on microthreads, (16) uses `vt::get_utidx()` to write into the proper element of output array, (20) vector memory fence.)

type of a hardware vector is very different inside versus outside the vector-fetched block. Outside the block, a hardware vector represents a vector of elements and has type `HardwareVector<T>` (e.g., `va` on line 9 has type `HardwareVector<int>`), but inside the block, a hardware "vector" now actually represents a *single* element and has type `T` (e.g., `va` on line 14 has type `int`). This also means the microthread code is type-safe in the sense that only operations that are valid for the element type are allowed within a vector-fetched block. Code within a vector-fetched block can include almost any C++ language feature including stack allocated variables (the VTAPI ensures that all microthreads have their own unique stack), object instantiation, templates, function and method calls, conditionals (`if`, `switch`), and loops (`for`, `while`). The primary restrictions are that a vector-fetched block cannot use C++ exceptions nor make any system calls, thus microthreads cannot currently perform dynamic memory allocation. Note that these restrictions do not apply when compiling a Maven application natively, so it is possible to insert debugging output into a vector-fetched block during early development through native emulation.

Figure 6.3 illustrates using the VTAPI to implement the irregular DLP loop in Table 2.1f. This example compiles to the assembly code shown in Figure 4.2b. The primary difference from the

previous example is the conditional `if` statement on line 15. Also notice that this example no longer uses a unit-stride store for the output, but each microthread instead uses its own scalar store if the condition is true. The base pointer for the array `c` is copied into all elements of the hardware vector `vcptr` on line 8 and then passed into the vector-fetched block. This allows each microthread to independently calculate the proper location for their output. After compiling this example, it was observed that more registers were needed in the vector-fetched block which is why the vector-thread unit (VTU) is configured with seven registers per microthread on line 3 (instead of the five registers in Figure 6.2). The Maven functional simulator checks to verify that an application is not accessing registers that are unavailable.

6.2 VT Compiler

The Maven programming methodology attempts to leverage a standard scalar compiler as much as possible. We started with the most recent GNU assembler, linker, and C++ compiler (version 4.4.1) which all contain support for the basic MIPS32 instruction set. We then modified the assembler to support the new Maven scalar and vector instructions; modified the compiler back-end to support vector registers and intrinsics; modified the compiler instruction scheduler to help vector functional unit performance tuning; and modified the compiler front-end to support long vector types and new function attributes. Each of these is described in more detail below.

- **Modified Assembler –** The assembler was modified as follows: added support for a unified integer and floating-point register space; updated floating-point instruction encodings appropriately; added support for new Maven instructions; removed unsupported instructions and the branch delay slot.
- **Modified Compiler Back-End –** The compiler back-end needed significant changes to unify the integer and floating-point registers. Instruction templates were added for the new divide and remainder instructions to allow the compiler to correctly generate these instructions. A new vector register space and the corresponding instruction templates required for register allocation were added. Some of these modifications were able to leverage the GNU C++ compiler's built-in support for fixed-length subword-SIMD instructions. Compiler intrinsics for some of the vector instructions were added to enable software to explicitly generate these instructions and for the compiler to understand their semantics.
- **Modified Compiler Instruction Scheduler –** The compiler instruction scheduling framework was used to create two new pipeline models for Maven: one for the control thread and one for the microthreads. The two types of threads have different performance characteristics. For example, a control-thread floating-point instruction occupies one floating-point functional unit for a single cycle and has a latency of four cycles, while a microthread floating-point instruction occupies one floating-point functional unit for a number of cycles equal to the either the

```
1  void intcopy_vt( int out[], int in[], int size )
2  {
3    int vlen;
4    asm volatile ( "setvl %0, %1" : "=r"(vlen), "r"(size) );
5    for ( int i = 0; i < size; i += vlen ) {
6      asm volatile ( "setvl %0, %1" : "=r"(vlen), "r"(size-i) );
7
8      int vtemp  __attribute__ ((vector_size(128)));
9      vtemp = __builtin_mips_maven_vload_vsi( &in[i] );
10     __builtin_mips_maven_vstore_vsi( vtemp, &out[i] );
11   }
12   asm volatile ( "sync.l.cv" ::: "memory" );
13 }
```

Figure 6.4: Low-Level Example Using the Maven Compiler – Simple function which copies an array of integers. Modifications to a standard GNU C++ compiler enable vector types to be allocated to vector registers and provide vector intrinsics for common vector memory operations. (C++ code line (4,6) GNU C++ inline assembly extensions for setting hardware vector length, (8) instantiate a vector type with $32 \times$ `int` using GNU C++ subword-SIMD extensions, (9–10) use compiler intrinsics for unit-stride vector loads and stores, (12) GNU C++ inline assembly extensions for memory fence.)

vector length or the active vector length (depending on whether we have a density-time implementation) but still has a latency of four cycles (through chaining). Since pipeline descriptions must use static occupancy and latency values, we use a reasonable fixed occupancy of eight for microthread arithmetic operations. In addition, the instruction cost functions were modified to separately account for the longer branch latency in microthreads as compared to the control thread. This allows the compiler to more aggressively use conditional move instructions when compiling for the microthreads.

- **Modified Compiler Front-End –** There were relatively few modifications necessary to the compiler front-end. We used the GNU C++ compiler's function attribute framework to add new attributes denoting functions meant to run on the microthreads for performance tuning. We were able to leverage the GNU C++ compiler's built-in support for fixed-length subword-SIMD instructions to create true C++ vector types. Unlike subword-SIMD types which usually vary the number of elements with the element type (e.g, $4 \times$ `int` or $8 \times$ `short`), Maven fixes the number of elements to the largest possible hardware vector length (e.g., $32 \times$ `int` or $32 \times$ `short`). Programs can still use `setvl` to change the active vector length, but the compiler always assumes that vector types contain 32 elements. This can potentially waste stack space during register spilling but greatly simplifies the compiler modifications.

Figure 6.4 illustrates how some of these modifications are exposed at the lowest-level to the programmer. This is a simple example that copies an array of integers using vector memory operations. The example uses the GNU C++ compiler's inline assembly extensions to explicitly generate `setvl` and `sync.l.cv` instructions (lines 4,6,12). The `vtemp` variable on line 8 uses the GNU C++

compiler's subword-SIMD extensions to tell the compiler that this is a vector of 32 integers. We have modified the compiler such that variables with these kind of vector types are automatically allocated to vector registers. Note that the actual number of elements in the vector register might be less that 32 due to available hardware resources or simply because `size` is less than 32. The compiler, however, will always treat these types as having 32 elements which should be the absolute maximum hardware vector length as specified by the Maven instruction set. The intrinsic functions on lines 9–10 generate unit-stride vector loads and stores. Using intrinsics instead of inline assembly allows the compiler to understand the semantics of these operations and thus perform better memory aliasing analysis. Eventually, more Maven operations could be moved from inline assembly into compiler intrinsics.

Overall, the changes required to a standard C++ compiler to support the Maven programming methodology are relatively straight-forward. However, as the low-level example in Figure 6.4 shows, the result is far from elegant. This example is tightly coupled to the Maven architecture and obviously cannot be compiled natively. In addition, the compiler modifications do little to simplify vector fetching code onto the microthreads. The VTAPI provides an additional layer on top of these compiler modifications to enable native emulation and greatly simplify writing Maven applications.

6.3 VT Application Programming Interface

The VT application programming interface (VTAPI) is shown in Table 6.1 and was briefly discussed in Section 6.1. This section provides more details on the VTAPI implementation.

The Maven implementation of the `HardwareVector<T>` class contains a single vector type data member using the GNU C++ compiler subword-SIMD extensions, and each member function uses either inline assembly or compiler intrinsics as discussed in the previous section. A templated class is much more efficient than virtual functions and dynamic dispatch, yet at the same time it cleanly enables adding more element data types in the future. The rest of the free functions in Table 6.1 also use a combination of inline assembly and compiler intrinsics. Notice how the raw subword-SIMD vector types and intrinsics are never exposed to users of the VTAPI; this enables the native implementation to be completely library-based. The native implementation of the `HardwareVector<T>` class contains a standard C array. The hardware vector length is always set to be the minimum vector length specified in the Maven instruction set (i.e., four), and global state is used to track the active vector length.

Vector fetches are handled in the Maven VTAPI implementation by generating a separate C++ function containing the microthread code, and then using a function pointer as the target for the vector fetch instruction. This keeps the microthread code separate from the control-thread code, which avoids the control thread from having to jump around microthread code. The challenge then becomes connecting the vector registers seen by the control-thread with the scalar registers seen

within the microthread code. Although it is possible to establish a standard calling convention for the inputs and outputs of the vector-fetched microthread function, this can result in suboptimal vector register allocation. There is a tight coupling between the control-thread and the vector-fetched microthread code, so we would rather the compilation process more closely resemble function inlining. Unfortunately, this creates a contradiction: we need to keep the microthread code separate from the control-thread code, but at the same time we want the effect of inlined register allocation. Our solution is to use the two-phase compilation process shown in Figure 6.1. A C++ source file that uses the VTAPI is denoted with the special `.vtcc` file name extension. These files are first compiled by the Maven preprocessor. The preprocessor uses the Maven compiler to generate the assembly for the source file, scans the assembly for the vector register allocation used in the control-thread, and then generates a new C++ source file with this register allocation information embedded in the microthread functions. The new C++ source file is then compiled as normal. Implementing this process completely in the compiler itself would require significant effort. The two-phase process is simple to implement although it does increase compilation time for files which use the VTAPI.

The Maven implementation of the `VT_VFETCH` macro uses sophisticated C preprocessor metaprogramming to generate the microthread function and all associated register allocation information. Figure 6.5 shows the result of running the vector-fetched block from Figure 6.2 (lines 12–15) through the C preprocessor. Notice that the microthread code is contained in a static member function of a local class (lines 9–30). The address of this function (`&vp_::func`) is then used in the inline assembly for the actual vector fetch instruction on lines 33–35. The microthread function includes Maven specific function attributes (`utfunc` and `target("tune=maven_ut")` on line 9) to enable microthread specific optimizations for this function. GNU C++ compiler extensions are used to ensure that the various microthread variables are in the correct registers (lines 12,15,18). This example shows the result after we have run the Maven preprocessor, so the explicit register allocation in the microthread function also corresponds to the vector register allocation in the control-thread. The inline assembly for the vector fetch instruction includes information on which hardware vectors this vector-fetched block uses as inputs and outputs (lines 34–35). This dependency information allows the compiler to schedule vector fetch instructions effectively, and also allows us to compactly capture the control-thread's vector register allocation in the generated assembly file using an assembly comment.

The `VT_VFETCH` macro shown as part of the VTAPI in Table 6.1 takes four lists of hardware vectors as arguments. The first is a list of hardware vectors used only as outputs, the second is a list of hardware vectors used as both inputs and outputs, and the third is a list of hardware vectors used only as inputs. The final list is for hardware vectors which must be allocated to the same register both before and after the vector-fetched block is executed. This list is needed because the compiler is otherwise unaware of vector register allocation done in the control-thread when it is compiling the microthread function. It is perfectly valid for the compiler to use extra scalar

```
1  {
2    struct vp_
3    {
4      typedef __typeof__ (vc) vc_t_;
5      typedef __typeof__ (va) va_t_;
6      typedef __typeof__ (vb) vb_t_;
7      typedef __typeof__ (vx) vx_t_;
8
9      __attribute__ ((flatten,utfunc,target("tune=maven_ut")))
10     static void func()
11     {
12       register va_t_::ireg_type va_in_ __asm__ ( "2" );
13       va_t_::ielm_type va = va_t_::cast_input( va_in_ );
14
15       register vb_t_::ireg_type vb_in_ __asm__ ( "3" );
16       vb_t_::ielm_type vb = vb_t_::cast_input( vb_in_ );
17
18       register vx_t_::ireg_type vx_in_ __asm__ ( "4" );
19       vx_t_::ielm_type vx = vx_t_::cast_input( vx_in_ );
20
21       vc_t_::elm_type vc;
22
23       vc = vx * va + vb;
24
25       goto stop;
26      stop:
27       register vx_t_::elm_type vx_out_ __asm__ ( "4" ) = vx;
28       register vc_t_::elm_type vc_out_ __asm__ ( "2" ) = vc;
29       __asm__ ( "" :: "r"(vc_out_), "r"(vx_out_) );
30     }
31   };
32
33   __asm__ volatile
34     ( "vf %4 # VT_VFETCH line: 18 ovregs: %0 ivregs: %1 %2 pvregs: %3"
35       : "=Z"(vc) : "Z"(va), "Z"(vb), "Z"(vx), "i"(&vp_::func) );
36 };
```

Figure 6.5: Example of Vector-Fetched Block After Preprocessing – Code corresponds to the vector-fetched block in Figure 6.2. The VTAPI uses a combination of a local class with a static member function, function attributes, explicit casts, and inline assembly to implement the `VT_VFETCH` preprocessor macro. (C++ code line (2–30) local class, (4–7) capture types of input and output hardware vectors, (9) function attributes, (10–30) static member function of local class, (12–19) specify register allocation and cast scalar elements to proper type, (21) declare output variable, (23) actual vector-fetched work, (25–26) stop label to allow user code to exit early, (27–28) ensure outputs are allocated to correct registers, (29) empty inline assembly to prevent compiler from optimizing away outputs, (33–35) inline assembly for actual vector-fetch instruction with input and output hardware vector dependencies properly captured.)

registers for temporaries when compiling the microthread function, but since these scalar registers correspond to vector registers, they may clobber values that are live across the vector-fetched block in the control-thread. By fully specifying which hardware vectors are live into, live out of, and live across a vector-fetched block, the compiler can efficiently allocate vector registers both in the control-thread and in the microthread.

The native implementation of the `VT_VFETCH` macro also generates a separate function for the microthread code. This function is called in a `for` loop for each element of the hardware vectors using standard C++. Input and output values are simply passed as function arguments.

The complexity of the `VT_VFETCH` macro is hidden from the programmer as illustrated by the examples in Figures 6.2 and 6.3. The resulting assembly shown in Figure 4.2 shows how the compiler is able to generate efficient code with an optimized vector register allocation.

6.4 System-Level Interface

In addition to efficiently managing its array of microthreads, the Maven control thread also needs some form of system-level support for interacting with the host thread. Figure 6.6 shows the approach used in the Maven programming methodology. To run a Maven application, a user accesses the general-purpose processor running the host-thread and then starts the Maven application server. The application server is responsible for loading the program into the data-parallel accelerator's memory and then starting the control-thread's execution.

A Maven application is linked with the GNU Standard C++ Library which in turn is linked with the Redhat Newlib Standard C Library. Newlib is a lightweight implementation of the ANSI C standard library well suited to embedded platforms. It includes a narrow operating system interface to simplify porting to new architectures. We have written a small proxy kernel that runs on the control thread and handles the Newlib system calls. The proxy kernel marshals system call arguments and passes them to the application server for servicing. Once finished, the application server places the results back in the data-parallel accelerator's memory and then notifies the proxy kernel to continue execution. This simple scheme gives the Maven application the illusion that it is running on a full

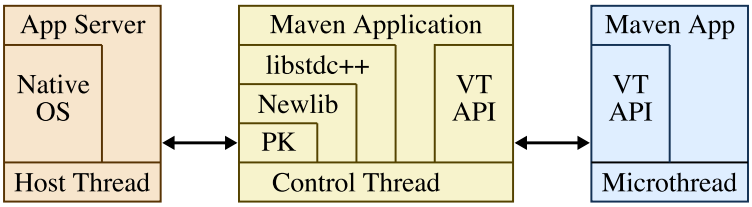


Figure 6.6: Maven System-Level Software Stack – A portion of the Maven application runs on the control thread and a different portion runs on the microthreads, with the VTAPI as the connection between the two. Maven uses the GNU standard C++ library which in turn uses the Redhat Newlib standard C library. System calls are handled through a lightweight proxy kernel that communicates with an application server running on the host thread to actually service the system call. (PK = proxy kernel)

operating system. For example, Maven applications can easily read and write files that reside as part of the native operating system running on the host-thread.

6.5 Extensions to Support Other Architectural Design Patterns

When comparing Maven to other architectural design patterns, we would like to leverage the Maven programming methodology as much as possible. This simplifies programming multiple implementations of the same application. It also enables a fairer comparison, since all software will be using a very similar programming methodology. This section describes how the Maven programming methodology can be extended to emulate the MIMD, vector-SIMD, and SIMT architectural design patterns. These extensions leverage the instruction set extensions discussed in Section 4.6 and the microarchitectural extensions discussed in Section 5.7. Unless otherwise specified, all extensions include both native and Maven implementations to enable rapid development and testing of applications through native execution regardless of the architectural design pattern being used.

6.5.1 MIMD Extensions to Maven Programming Methodology

In the MIMD pattern, the host thread is usually responsible for spawning work onto the microthreads. Unfortunately, this can have high overheads especially when the amount of work to be done per microthread is relatively small. So the Maven multithreaded MIMD extension takes a different approach where a "master" microthread on the multithreaded core is responsible for spawning the work on the other remaining "worker" microthreads. We can think of this as emulating a multithreaded MIMD accelerator where the host thread manages coarse-grain work distribution, while each master microthread manages fine-grain work distribution.

The general concept of a master microthread distributing work to the worker microthreads has some similarities to the vector-fetched blocks described earlier in this chapter with respect to wanting to cleanly express work "inline" along with the necessary inputs and outputs that need to be

```
1 void rdlp_mt( int c[], int a[], int b[], int size, int x )
2 {
3   BTHREAD_PARALLEL_RANGE( size, (c,a,b,x),
4   ({
5     for ( int i = 0; i < range.begin(); i < range.end(); i++ )
6       c[i] = x * a[i] + b[i];
7   }));
8 }
```

Figure 6.7: Regular DLP Example Using Maven MIMD Extension – Code corresponds to the regular DLP loop in Table 2.1c. (C++ code line (3) `BTHREAD_PARALLEL_RANGE` preprocessor macro automatically partitions input dataset's linear index range, creates separate function, spawns the function onto each microthread, passes in arguments through memory, and waits for the threads to finish, (5–6) each thread does the work from `range.begin()` to `range.end()` where `range` is defined by the preprocessor macro to be different for each thread.)

marshalled (of course the implementations of the VT and MIMD patterns are radically different). To this end, we first modify the Maven proxy kernel to support multiple threads of execution and then build a lightweight user-level threading library called *bthreads* on top of the proxy-kernel threads. Bthreads stands for "bare threads" because it has absolutely no virtualization. There is one bthread for each underlying hardware microthread context. The application is responsible for managing scheduling and virtualization, although the Maven MIMD extension currently maintains this one-to-one correspondence. Bthreads includes standard MIMD constructs such as mutex classes and access to the Maven atomic memory operations.

Spawning work is done with a `BTHREAD_PARALLEL_RANGE` macro as shown in Figure 6.7. Line 3 specifies the total number of elements to be distributed to the worker microthreads and a list of C++ variables that should be marshalled for each worker microthread. The final argument to the macro is the work to be done by each microthread (lines 4–7). The macro automatically partitions the input range (`0` to `size-1`), and each microthread accesses their respective range through the implicitly defined `range` object. This macro is just one part of the bthreads library. Several other classes, functions, and macros enable other ways to partition and distribute work across the microthreads.

As with the `VT_VFETCH` macro, the `BTHREAD_PARALLEL_RANGE` macro's implementation generates a separate function containing the actual work to be done in parallel. The generated function takes one argument which is a pointer to an argument structure. This structure is automatically created to hold the arguments (i.e., `c`, `a`, `b`, and `x`). When the master thread executes the `BTHREAD_PARALLEL_RANGE` macro, it goes through the following four steps: (1) partition input linear index range and marshal arguments, (2) spawn work function onto each worker thread passing the argument pointer to each, (3) execute the work function itself, (4) wait for the worker microthreads to finish. The Maven implementation of bthreads also includes support for activating and inactivating microthreads to improve performance on serial code (see 4.6.1 for more information on this mechanism). Bthreads are initially inactive until work is spawned onto them. The native implementation of bthreads is built on top of the standard `pthreads` threading library for truly parallel execution.

The bthreads library can be combined with any of the other architectural design pattern programming methodologies to enable mapping an application to multiple cores. For example, by using a `VT_VFETCH` macro inside the body of a `BTHREAD_PARALLEL_RANGE` macro we can use the bthreads library to distribute work amongst multiple control processors and the VTAPI to vectorize each core's work.

6.5.2 Vector-SIMD Extensions to Maven Programming Methodology

When emulating the vector-SIMD pattern, software can use any of the classes and functions in Table 6.1 except for `get_utidx` and the `VT_VFETCH` macro. This means that the vector-SIMD pattern uses the hardware vector class just as in the VT pattern for specifying vector values that can

```
1 void rdlp_tvec( int c[], int a[], int b[], int size )
2 {
3   int vlen = vt::config( 4, size );
4   for ( int i = 0; i < size; i += vlen ) {
5     vlen = vt::set_vlen( size - i );
6
7     vt::HardwareVector<int> vc, va, vb;
8     va.load( &a[i] );
9     vb.load( &b[i] );
10
11    vc = va + vb;
12
13    vc.store( &c[i] );
14  }
15  vt::sync_l_cv();
16 }
```

Figure 6.8: Regular DLP Example Using Maven Traditional-Vector Extensions – Code corresponds to the regular DLP loop in Table 2.1a. (C++ code line (3–5) handles stripmining, (8–9) unit-stride vector loads, (11) vector-vector add, (13) unit-stride store, (19) vector memory fence.)

be allocated to vector registers. We have modified the compiler back-end to recognize arithmetic operations on vector types, and we provide appropriate operator overloading for these operations in the `HardwareVector<T>` class.

Figure 6.8 illustrates using the vector-SIMD extensions for the simple regular DLP loop in Table 2.1a. This loop performs a vector-vector addition by first loading two hardware vectors from memory (line 8–9), adding the two hardware vectors together and writing the result into a third hardware vector (line 11), and then storing the third hardware vector back to memory (line 13). The vector-SIMD programming methodology uses the exact same stripmining process as the VT pattern.

Unfortunately, the vector-SIMD programming methodology currently has some significant limitations. Vector-scalar operations are not supported so it is not possible to map the regular DLP loop in Table 2.1c, and vector flag operations are also not supported so it is not possible to map the irregular DLP loop in Table 2.1f. For these kind of loops, the programmer must resort to hand-coded assembly. This actually illustrate some of the challenges when working with the vector-SIMD pattern. Mapping regular DLP to a vector-SIMD pattern is relatively straight-forward whether using a vectorizing compiler or the explicit data-parallel methodology described above, but mapping irregular DLP to a vector-SIMD pattern can be very challenging for assembly-level programmers, compilers, and programming frameworks.

6.5.3 SIMT Extensions to Maven Programming Methodology

The SIMT extensions are essentially just a subset of the full VT programming methodology. When emulating the SIMT pattern, software should use the control thread as little as possible,

```
1  void rdlp_simt( int c[], int a[], int b[], int size, int x )
2  {
3    int blocksz = vt::config( 5, size );
4    int nblocks = ( size + blocksz - 1 ) / blocksz;
5
6    vt::HardwareVector<int*> vcptr(c), vbptr(b), vaptr(a);
7    vt::HardwareVector<int>  vsize( size );
8    vt::HardwareVector<int>  vblocksz( blocksz );
9
10   for ( int block_idx = 0; block_idx < nblocks; block_idx++ ) {
11     vt::HardwareVector<int> vblock_idx( block_idx );
12
13     VT_VFETCH( (), (), (vblock_idx), (vcptr,vaptr,vbptr,vsize,vblocksz),
14     ({
15       int idx = vblock_idx * vblocksz + vt::get_utidx();
16       if ( idx < vsize )
17         vcptr[idx] = vx * vaptr[idx] + vbptr[idx];
18     }));
19   }
20   vt::sync_l_cv();
21 }
```

Figure 6.9: Regular DLP Example Using Maven SIMT Extensions – Code corresponds to the regular DLP loop in Table 2.1c. (C++ code line (3–4) calculate hardware vector length and number of microthread blocks, (6) copy array base pointers into all elements of hardware vectors, (7–8) copy size and block size into all elements of hardware vectors, (10–20) `for` loop emulating multiple microthread blocks mapped to the same core, (13) specifies input and output hardware vectors for vector-fetched block, (14–18) vector-fetched block, (15) each microthread calculates own index, (16) check to make sure index is not greater than array size, (17) actual work, (20) vector memory fence.)

since the SIMT pattern does not include a control thread. The code which does run on the control thread should be thought of as emulating some of the dedicated hardware in a SIMT vector issue unit. SIMT programs should not use vector memory instructions, and instead all memory accesses should be implemented with microthread loads and stores. SIMT programs should also not use the `setvl` instruction, and instead use a microthread branch to handle situations where the application vector length is not evenly divisible by the hardware vector length. Hardware vectors should be limited to initializing each microthread with appropriate scalar values.

Figure 6.9 illustrates using the SIMT extensions for the regular DLP loop in Table 2.1c. Instead of using vector loads for the input arrays, the SIMT pattern initializes the array base pointers once outside the stripmine loop (line 6) and then uses microthread loads inside the vector-fetched block (line 17). Each microthread must calculate its own offset into the input and output arrays (line 15), and also check to see if the current index is less than the application vector length (line 16). It is important to note that this additional address calculation and conditional branch closely follows the practices recommended by the CUDA programming methodology [NBGS08].

6.6 Future Research Directions

This section briefly describes some possible directions for future improvements with respect to the Maven programming methodology.

Support for New Instructions – Section 4.6 discussed several new instructions that would affect the programming methodology. Some of these extensions, such as a vector unconfiguration instruction, vector compress and expand instructions, and vector reduction instructions, would be exposed as additional free functions. Adding segments to the VTAPI would be difficult, since segments impose additional constraints on vector register allocation. One option is simply to allow the programmer to more explicitly allocate segments to vector registers, but this significantly lowers the level of abstraction. A better solution is to add a compiler optimization pass that can merge multiple strided accesses into a single segment access. To support shared hardware vectors, the VTAPI would need a way to explicitly denote this kind of vector. The compiler register allocation framework would need to be modified to efficiently allocate both private and shared hardware vectors.

Hardware Vectors of C++ Objects – Hardware vectors are limited to containing primitive types, but programs often work with arrays of objects. It would be useful to be able to easily transfer arrays of objects into vector-fetched microthread code. Currently, the programmer must "unpack" the fields of the objects using strided accesses into multiple hardware vectors, pass these hardware vectors into the vector-fetched block, and then "repack" these fields into the corresponding object in the microthread code. In addition to being cumbersome, this process also violates the object's data encapsulation by requiring data members to be publicly accessible (the vector load operations need access to the underlying addresses of these fields). Hardware vectors containing objects would greatly simplify this process, and possibly even allow the use of efficient segment accesses.

Automatically Determine Vector-Fetched Block Inputs and Outputs – Improperly specified inputs and outputs for a vector-fetched block are a common source of errors when using the Maven programming methodology. More sophisticated static analysis might allow the compiler to automatically determine these inputs and outputs. This analysis would need to be cross-procedural (i.e., from control thread function to microthread function) and possibly rely on variable naming to relate microthread scalar variables to the control-thread's hardware vector variables. Integrating the vector fetch construct into the language would probably be required to support this kind of analysis.

Optimize Number of Registers per Microthread – The number of registers per microthread is currently set manually by the programmer. The compiler could include an optimization pass that attempts to statically determine an appropriate number of registers per microthread. More registers allows the compiler more scheduling freedom at the cost of reduced hardware vector length, while less registers increases the hardware vector length at the cost of constrained scheduling and possible register spilling.

6.7 Related Work

This section highlights selected previous work specifically related to the Maven programming methodology presented in this chapter.

Scale VT Processor – Scale provides two programming methodologies. The first approach uses a standard scalar compiler for most of the control-thread code, and then manually written assembly for the critical control-thread loops and microthread code [KBH+04a]. The second approach uses a research prototype VT compiler [HA08]. Hand-coded assembly provides the highest performance, but requires the most effort. The research prototype VT compiler automatically exploits DLP for a VT architecture, but can not handle all kinds of loops and results in lower performance than hand-coded assembly. Porting the research prototype VT compiler to Maven would have required significant effort. The Maven programming methodology takes a very different approach where the programmer must explicitly specify the data-level parallelism but at a high-level of abstraction.

Automatically Vectorizing Compilers – There has been a great deal of research on compilers that can automatically extract DLP from standard sequential programming languages [BGS94, DL95, HA08]. The success of such compilation varies widely and depends on how the original program was written, the presence of true data dependencies, and the sophistication of the compiler optimization passes. Effective vectorizing compilers must manage memory aliasing, nested loops, and complex data-dependent control flow. An explicitly data-parallel programming methodology simplifies the compiler implementation by shifting more of the burden on the programmer, but can result in very efficient code generation. The Maven methodology uses a relatively high-level framework to help the programmer naturally express the data-level parallelism that is available in the application.

Data-Parallel Programming Languages – Programming languages such as Paralation Lisp, APL, and Fortran 90 include explicit support in the language for data-parallel operators (see [SB91] for a survey of early data-parallel languages). More recent examples include ZPL [CCL+98] and the Scout language for graphics processors [MIA+07]. Although all of these languages can elegantly express regular DLP and could produce efficient code on vector-SIMD architectures, they quickly become cumbersome to use when working with irregular DLP. The NESL language specifically attempts to better capture irregular DLP by providing nested data-parallel constructs [Ble96]. The explicit data-parallel programming methodology introduced in this chapter provides a much more general framework for expressing both regular and irregular DLP, although because it is mostly library-based it lacks some of the static guarantees available in true data-parallel languages.

Programming Frameworks for Graphics Processors – Programming frameworks that follow the SIMT pattern are probably the most similar to the Maven programming methodology. NVIDIA's CUDA [NBGS08], OpenCL [ope08a], and Stanford's Brook language [BFH+04] provide the ability to write microthread code as a specially annotated function, which is then spawned on the ac-

celerator from the host thread. Both the SIMT and VT patterns include a two-level hierarchy of microthreads, and this hierarchy is reflected in both programming methodologies. The SIMT pattern uses microthread blocks, while the VT pattern uses software-exposed control threads. Maven extends these earlier SIMT frameworks to include efficient compilation for the control thread.

Chapter 7

Maven Evaluation

This chapter evaluates the Maven VT core in terms of area, performance, and energy as compared to the MIMD, vector-SIMD, and SIMT architectural design patterns. Section 7.1 describes the 20 different data-parallel core configurations that we have implemented using a semi-custom ASIC methodology in a TSMC 65 nm process, and Section 7.2 describes the four microbenchmarks for studying how these cores execute both regular and irregular DLP. Section 7.3 reviews the methodology used to measure the area, performance, and energy of each core running the various microbenchmarks. Section 7.4 compares the cores based on their cycle times and area breakdown, and Sections 7.5–7.6 compares the core's energy-efficiency and performance. Section 7.7 concludes this chapter with a brief case study of how a much larger computer graphics application can be mapped to the Maven VT core.

7.1 Evaluated Core Configurations

Table 7.1 lists the 20 core configurations used to evaluate a simplified VT architecture for use in future data-parallel accelerators. There are three broad classes of configurations that correspond to three of the architectural design patterns: MIMD, vector-SIMD, and VT. There is no SIMT core configuration, since the SIMT pattern is evaluated by using the VT cores to run code written in the SIMT style. Table 7.1 also lists the number of lanes, number of physical registers, the number of supported microthreads with the default 32 registers per microthread, minimum and maximum hardware vector lengths, sustainable arithmetic throughput, and sustainable memory throughput.

The *mimd-1x** configurations adhere to the MIMD pattern with *mimd-1x1* representing a single-threaded MIMD core, and *mimd-1x2*, *mimd-1x4*, and *mimd-1x8* representing multithreaded MIMD cores with two, four, and eight microthreads per lane respectively. These MIMD cores include the instruction set and microarchitectural extensions described in Sections 4.6.1 and 5.7.1. Most notably, the multithreaded MIMD cores include support for activating and deactivating microthreads to avoid overheads on the serial portions of the microbenchmarks. Although the MIMD cores have

Pattern	Variant	Num Lanes	Num Regs	Num µTs	Min Vlen	Max Vlen	Throughput	
							Arith (ops/cyc)	Mem (elm/cyc)
mimd	1x1	n/a	31	1	n/a	n/a	1	1
mimd	1x2	n/a	62	2	n/a	n/a	1	1
mimd	1x4	n/a	124	4	n/a	n/a	1	1
mimd	1x8	n/a	248	8	n/a	n/a	1	1
vsimd	1x1	1	31	1	1†	10	1c + 2v	1l + 1s
vsimd	1x2	1	62	2	2†	20	1c + 2v	1l + 1s
vsimd	1x4	1	124	4	4	32	1c + 2v	1l + 1s
vsimd	1x8	1	248	8	4	32	1c + 2v	1l + 1s
vsimd	2x1	2	62	2	2†	20	1c + 4v	2l + 2s
vsimd	2x2	2	124	4	4	41	1c + 4v	2l + 2s
vsimd	2x4	2	248	8	4	64	1c + 4v	2l + 2s
vsimd	2x8	2	496	16	4	64	1c + 4v	2l + 2s
vsimd	4x1	4	124	4	4	41	1c + 8v	4l + 4s
vsimd	4x2	4	248	8	4	82	1c + 8v	4l + 4s
vsimd	4x4	4	496	16	4	128	1c + 8v	4l + 4s
vsimd	4x8	4	992	32	4	128	1c + 8v	4l + 4s
vt	1x1	1	31	1	1†	10	1c + 2v	1l + 2s
vt	1x2	1	62	2	2†	20	1c + 2v	1l + 2s
vt	1x4	1	124	4	4	32	1c + 2v	1l + 2s
vt	1x8	1	248	8	4	32	1c + 2v	1l + 2s

Table 7.1: Evaluated Core Configurations – Twenty data-parallel core configurations are used to evaluate three architectural design patterns: MIMD, vector-SIMD, and VT. In addition, the SIMT pattern is evaluated by using the VT cores to run code written in the SIMT style. MIMD core configuration include support for 1–8 microthreads. Vector-SIMD core configurations include 1–4 lanes with 31–248 physical registers per lane. VT core configurations all use a single lane with 31–248 physical registers. (*num regs* column excludes special register zero, *num µTs* column is the number of microthreads supported with the default 32 registers per microthread, † = these are exceptions since technically the Maven instruction set requires a minimum vector length of four, xc + yv = x control processor operations and y vector unit operations per cycle, xl + ys = x load elements and y store elements per cycle)

a decoupled load/store unit and long-latency functional units, they are fundamentally single-issue resulting in a sustainable arithmetic throughput of one operation per cycle and a sustainable memory throughput of one element per cycle.

The *vsimd-** configurations adhere to the vector-SIMD pattern with one-lane (*vsimd-1x**), two-lane (*vsimd-2x**), and four-lane (*vsimd-4x**) variants. These vector-SIMD cores include the instruction set and microarchitectural extensions described in Sections 4.6.2 and 5.7.2. They support a full complement of vector instructions including vector flag operations for data-dependent conditional control flow. The control processor is embedded in all configurations. The *vsimd-*x1*, *vsimd-*x2*, *vsimd-*x4*, and *vsimd-*x8* configurations are sized to have enough physical registers to support one, two, four, and eight microthreads respectively. This assumes each microthread requires the full 32

MIPS32 registers, although longer vector lengths are possible with reconfigurable vector registers. The special register zero is not included as a physical register, since it can be easily shared across all microthreads. The twelve vector-SIMD cores allows us to evaluate mapping microthreads in two dimensions: spatially across 1–4 lanes and temporally with 31–248 physical registers per lane. Some cores have a minimum vector length less than four, which is the minimum vector length specified by the Maven instruction set. We included these configurations for evaluation purposes, even though they are unlikely to be used in a real implementation. Also notice that the maximum vector length is limited to 32 per lane, which is sometimes less than what the number of physical registers will theoretically support (e.g., *vsimd-1x8* can theoretically support a maximum vector length of 82). This helps limit the amount of state that scales with the number of microthreads (e.g., size of each vector flag register) and enables a fair comparison against the VT cores that have similar constraints. The arithmetic throughput includes both the control processor functional unit and vector functional units, although the control processor functional unit is only used for simple control operations. Increasing the number of lanes increases the arithmetic throughput, and the memory throughput is also increased to maintain a similar balance across all configurations. For example, the *vsimd-4x** configurations' vector unit can execute eight arithmetic ops per cycle owing to two vector functional units (VFU0/VFU1) per lane and four lanes. The *vsimd-4x** configurations also include a wide VMU that supports up to four load elements and four store elements per cycle. In addition to increased data bandwidth, the multi-lane configurations also include increased address bandwidth so that the *vsimd-4x** configurations can issue four independent indexed or strided requests to the memory system per cycle. It is important to note, that this increased arithmetic and memory throughput does not require increased fetch or issue bandwidth. The vector-SIMD cores can only sustain a single instruction fetch per cycle, but the VIU can still keep many vector fetch units busy with multi-cycle vector operations.

The *vt-1x** configurations adhere to the VT pattern with a single lane and a physical register file that ranges in size from 31–248 elements. The *vt-1x** configurations are very similar to the *vsimd-1x** configurations with respect to the number of physical registers, number of supported microthreads, minimum and maximum vector lengths, and sustained arithmetic and memory throughputs. Of course the *vt-1x** configurations differ from the *vsimd-1x** configurations in their ability to support vector-fetched scalar instructions. The evaluation in this thesis is limited to just the basic Maven microarchitecture without support for vector fragment merging, interleaving, or compression. As with the vector-SIMD configurations, VT configurations with few physical registers (i.e., *vt-1x1* and *vt-1x2*) do not support very long hardware vector lengths. These configurations are useful for studying trends, but are less practical for an actual prototype.

Each configuration is implemented in Verilog RTL. There is a great deal of common functionality across configurations, so many configurations are generated simply by enabling or disabling certain design-time options. For example, the MIMD cores are similar to the control processor

used in the vector-SIMD and VT cores. The primary differences are that the MIMD cores use a dedicated decoupled floating-point functional unit instead of sharing with a vector unit, and the multithreaded MIMD cores include microthread scheduling control logic. The vector-SIMD and VT configurations are also very similar with design time options to enable or disable the vector flag register, PVFB, and other small modules that are specific to one configuration or the other. For this evaluation, all configurations include a relatively idealized memory system represented by a fixed two-cycle latency magic memory module. The L1 instruction cache is not modeled and instead the instruction ports connect directly to the magic memory module.

7.2 Evaluated Microbenchmarks

Eventually, we will experiment with large-scale data-parallel applications running on the various core configurations, but my initial evaluation focuses on four carefully crafted microbenchmarks. These microbenchmarks capture some of the characteristics of both regular and irregular DLP programs, and they are easy to analyze and understand. In addition, although the evaluation methodology used in this thesis produces detailed results, it also requires lengthy simulation times that are a better suited to shorter microbenchmarks. In the future, we can use the insight gained through working with these microbenchmarks to build higher-level models that should reduce the simulation times required for large-scale applications.

Figure 7.1 illustrates the four microbenchmarks: *vvadd* which performs a 1000-element vector-vector addition with integer elements, *cmult* which performs a 1000-element vector-vector complex multiply with floating-point imaginary and real components, *mfilt* which performs a masked convolution with a five-element kernel across a gray-scale image that is 100×100 pixels and a corresponding mask image that is also 100×100 pixels, and *bsearch* which performs 1000 look-ups

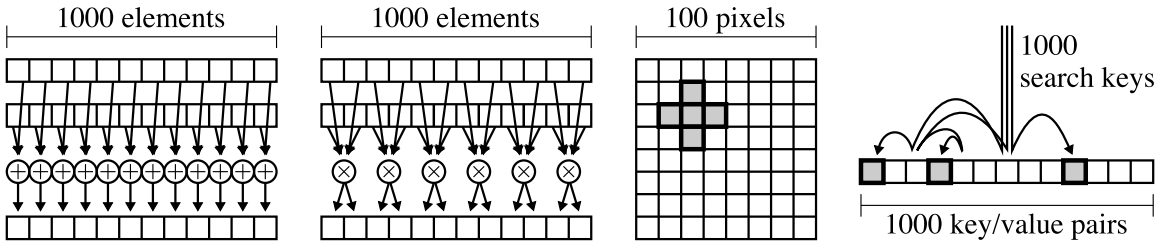


Figure 7.1: Microbenchmarks – Four microbenchmarks are used to evaluate the various architectural design patterns: (a) the *vvadd* microbenchmark performs element-wise integer addition across two input arrays and writes a third output array; (b) the *cmult* microbenchmark performs element-wise complex multiplication across two input arrays of structures with the imaginary/real components and writes a third output array; (c) the *mfilt* microbenchmark performs a masked convolution with a five element kernel on a gray-scale input image; (d) the *bsearch* microbenchmark uses a binary search to look-up search keys in a sorted array of key/value pairs.

using a binary search into a sorted array of 1000 key/value pairs. Each microbenchmark has at least four implementations that correspond to the four patterns we wish to evaluate: a multithreaded MIMD implementation that runs on the *mimd-** cores, a vector-SIMD implementation that runs on the *vsimd-** cores, a SIMT implementation that runs on the *vt-** cores, and a VT implementation that also runs on the *vt-** cores. Table 7.2 lists the number of instructions for each microbenchmark by type. Only the instructions in the inner loop are included in this table, which ignores the fact that the MIMD pattern requires significantly more start-up overhead as compared to the other patterns. The programming methodology described in Chapter 6 is used to compile all of the microbenchmarks, with the specific extensions discussed in Section 6.5 for the MIMD, vector-SIMD, and SIMT patterns. Due to the difficulty of compiling irregular DLP for the vector-SIMD pattern, the vector-SIMD implementation of the *mfilt* and *bsearch* microbenchmarks required hand-coded assembly. The *bsearch* microbenchmark also has implementations that use explicit conditional moves to reduce the number of conditional branches.

The *vvadd* and *cmult* microbenchmarks illustrate regular DLP. The *vvadd* microbenchmark uses integer arithmetic and unit-stride accesses, while the *cmult* microbenchmark uses floating-point arithmetic and strided accesses. The strided accesses are necessary, since the input and output are stored as an array of structures where each structure contains the imaginary and real part for one complex number. Neither microbenchmark includes any data-dependent control flow beyond the standard loop over the input elements. As expected, Table 7.2 shows that the MIMD implementations only use microthread scalar instructions, and the vector-SIMD implementations only use control-thread scalar and vector instructions. The SIMT and VT implementations use a combination of control-thread and microthread instructions. To capture the fact that a real SIMT core would not include a control processor, the SIMT implementations uses fewer control-thread instructions and more microthread instructions as compared to the VT implementations. The microthread branch instruction in the SIMT implementations is for checking if the current microthread index is less than the input array size. The control-thread scalar and vector memory instructions for the vector-SIMD and VT implementations are identical because both use essentially the same stripmining and unit-stride/strided vector memory accesses. Because more code runs on the microthreads in the SIMT implementations, these implementations require more registers per microthread. This in turn will result in shorter hardware vector lengths as compared to the vector-SIMD and VT implementations.

The *mfilt* microbenchmark illustrates regular data access with irregular control flow. The input pixels are loaded with multiple unit-stride accesses and different base offsets, and the output pixels are also stored with a unit-stride access. The convolution kernel's coefficients use shared accesses, and there is also some shared computation to calculate an appropriate normalization divisor. Each iteration of the loop checks to see if the corresponding boolean in a mask image is false, and if so, skips computing the convolution for that pixel. The mask image used in this evaluation includes several randomly placed squares covering 67% of the image. Each microthread in the MIMD im-

		CT Scalar		CT Vector					μT Scalar									Totals	
Name	**Pattern**	**int**	**br**	**int**	**fp**	**ld**	**st**	**misc**	**int**	**fp**	**ld**	**st**	**br**	**j**	**cmv**	**misc**	**nregs**	**CT**	**μT**
vvadd	mimd								6		2	2	1						10
vvadd	vsimd	7	1	1		2u	2u	1									4	12	
vvadd	simt	1	1					2	8		2	2	1			2	10	4	14
vvadd	vt	7	1			2u	2u	2	1							1	4	12	2
cmult	mimd								5	6	6	2	1						20
cmult	vsimd	10	1		6	4s	2s	1									4	24	
cmult	simt	1	1					2	7	6	6	2	1			2	10	4	24
cmult	vt	10	1			4s	2s	2	1	6						1	4	19	8
mfilt	mimd								34		7	2	5	1					49
mfilt	vsimd	24	2	1,9f		6u	1u	2									13	45	
mfilt	simt	4	3					3	27		7	2	2			3	26	10	41
mfilt	vt	24	2			6u	1u	3	10				1			1	13	35	12
bsearch	mimd								19		4	2	5						30
bsearch	vsimd	9	3	2,17f		1u,2x	1u	5									10	40	
bsearch	simt	1	1					2	26		4	2	5	1	1	2	16	4	41
bsearch	simt-cmv	1	1					2	28		3	2	2		4	2	26	4	41
bsearch	vt	6	1			1u	1u	5	15		3		4	1	1	2	10	13	26
bsearch	vt-cmv	6	1			1u	1u	5	17		2		1		4	1	13	14	25

Table 7.2: Instruction Mix for Microbenchmarks – Number of instructions for each microbenchmark are listed by type. The *vvadd* and *cmult* microbenchmarks have regular data access and control flow. The *mfilt* microbenchmark has regular data access but irregular control flow, and the *bsearch* microbenchmark has irregular data access and control flow. (CT = control thread, μT = microthread, int = simple short-latency integer, fp = floating-point, ld = load, st = store, br = conditional branch, j = unconditional jump, cmv = conditional move, CT vector misc = {`setvl`, `vf`, `mov.sv`}, μT scalar misc = {`utidx`, `stop`}, nregs = number of scalar registers required per μT, u suffix = unit-stride, s suffix = strided, x suffix = indexed, f suffix = vector operation writing or reading vector flag register)

plementation requires five conditional branches to iterate over a portion of the input image and to check the mask image. The vector-SIMD implementation of this data-dependent conditional control flow uses nine vector instructions that either manipulate or operate under a vector flag register. Even though there is some irregular control flow, both the vector-SIMD and VT implementations can refactor 26 scalar instructions onto the control thread and use seven unit-stride memory accesses (five to load the five pixels in the input image, one to load the mask image, and one to store the result). The SIMT and VT implementations use vector-fetched scalar branches to implement the data-dependent control flow and the hardware manages divergence as necessary. Again notice the difference between the SIMT and VT implementations: the SIMT implementation uses 41 microthread instructions and only 10 control-thread instructions, while the VT implementation uses 12 microthread instructions and 35 control-thread instructions. VT is able to better amortize overheads onto the control processor, which should improve performance and energy-efficiency.

The *bsearch* microbenchmark illustrates irregular DLP. The microbenchmark begins with an array of structures where each structure contains a key/value pair. The input is an array of search keys, and the output is the corresponding value for each search key found by searching the sorted key/value array. The dataset used in this evaluation requires less than the full number of look-ups (10) for 52% of the search keys. The microbenchmark is essentially two nested loops with an outer `for` loop over the search keys, and an inner `while` loop implementing a binary search for finding the key in the key/value array. The MIMD implementation requires five branches to implement these nested loops. The vector-SIMD implementation uses outer-loop vectorization such that the microthreads perform binary searches in parallel. The outer-loop control runs on the control thread and always executes $O(\log(N))$ (the worst case) number of times. Vectorizing the outer loop requires 17 vector instructions that either manipulate or operate under vector flag registers. Although it is possible for the control thread to check if the flag register is all zeros every iteration of the inner `while` loop, this prevents control processor decoupling and decreases performance. The vector-SIMD implementation must use indexed vector memory instructions to load the keys in the key/value array, since each microthread is potentially checking a different element in the key/value array. In the SIMT and VT implementations, each microthread directly executes the inner `while` loop, and the hardware manages divergence as necessary. The microthreads can use scalar loads to access the key/value array. The VT implementation can still amortize some instructions onto the control thread including the unit-stride load to read the search keys and the unit-stride store to write the result. Although the compiler is able to automatically replace one conditional branch with a conditional move, both the SIMT and VT implementations include many conditional branches that can potentially limit performance and energy efficiency. We also examine implementations where some of these conditional branches are replaced with explicit conditional moves inserted by the programmer.

7.3 Evaluation Methodology

Figure 7.2 illustrates the evaluation methodology used to collect area, performance, and energy results from the 20 core configurations and four microbenchmarks. The software toolflow follows the programming methodology described in Chapter 6. A C++ application can be compiled either natively for rapid development and testing, or for Maven with the Maven preprocessor and compiler. The resulting Maven binary can then be run on the instruction set simulator for functional verification. The instruction set simulator produces preliminary statistics about dynamic instruction counts and microthread branch divergence. We augment each microbenchmark with special instructions to tell the various simulators when the critical timing loop begins and ends so that our statistics only correspond to the important portion of the microbenchmark.

The hardware toolflow starts with the Verilog RTL for each of the 20 core configurations. Synopsys VCS is used to generate a fast simulator from this RTL model that can be linked to the

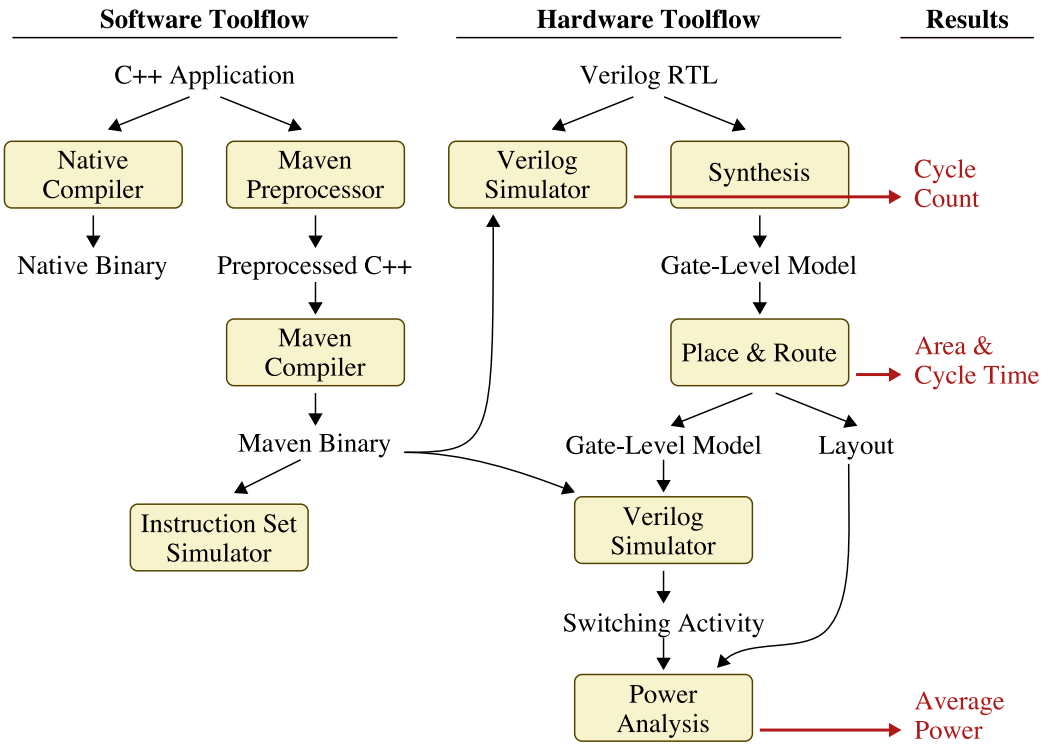


Figure 7.2: Evaluation Software and Hardware Toolflow – The software toolflow allows C++ applications to be compiled either natively or for Maven, while the hardware toolflow transforms the Verilog RTL for a data-parallel core into actual layout. From this toolflow we can accurately measure area, performance ($1/\text{cycle count} \times \text{cycle time}$), and energy ($\text{average power} \times \text{cycle count} \times \text{cycle time}$).

application server described in Section 6.4. We use an extensive self-checking assembly-level test suite that can run on both the instruction set and RTL simulators to verify the functionality of the RTL model. Since this model is cycle accurate, we can also use the RTL simulator to determine the number of cycles required to execute a specific microbenchmark.

Synopsys Design Compiler is used to synthesize a gate-level model from the higher-level RTL model. We used a 1.5 ns target cycle time (666 MHz clock frequency) for all configurations in hopes that most designs would ultimately be able to achieve a cycle time between 1.5–2 ns (666-500 MHz). All core configurations explicitly instantiate Synopsys DesignWare Verilog components to implement the larger and more complex functional units including the integer multiplier/divider and floating-point units. Automatic register retiming is used to generate fully pipelined instruction set units that meet our timing requirements. This resulted in the following functional unit latencies: four-cycle integer multiplier, 12-cycle integer divider, four-cycle floating-point adder, four-cycle floating-point multipler, eight-cycle floating-point divider, and eight-cycle floating-point square root. The RTL is crafted such that the synthesis tool can automatically generate clock-gating logic throughout the datapaths and control logic. Unfortunately, we do not currently have access to a custom register-file generator. Instead, we have written a Python script that generates gate-level

models of register files with different numbers of elements, bits per element, read ports, and write ports. Our generator uses standard-cell latches for the bitcells, hierarchical tri-state buses for the read ports, and efficient muxing for the write ports. In addition to generating the Verilog gate-level model, our generator pre-places the standard-cells in regular arrays. Our generator can also build larger register files out of multiple smaller subbanks. These pre-placed blocks are included as part of the input to the synthesis tool, but they are specially marked to prevent the tool from modifying them. This allows the synthesis tool to generate appropriately sized decode logic.

The synthesized gate-level model is then passed to Synopsys IC Compiler which handles placing and routing the standard-cells. We also use IC Compiler for floorplanning the location of the pre-placed blocks, synthesizing the clock tree, and routing the power distribution grid. The place-&-route tool will automatically insert extra buffering into the gate-level model to drive heavily loaded nets. The output of the place-&-route tool is an updated gate-level model and the final layout. We use the place-&-route tool to generate reports summarizing the area of each module in the hierarchy and the most critical timing paths using static timing analysis. From the cycle time and the cycle counts generated through RTL simulation, we can calculate each microbenchmark's performance in iterations per second for each core configuration.

We use Synopsys VCS to generate a simulator from the post-place-&-route gate-level model. This gate-level simulator is more accurate but slower than the higher-level RTL simulator. The gate-level simulator is linked to the application server allowing us to run the exact same binaries as used with the instruction set and RTL simulators. The gate-level simulator also allows us to perform additional verification. Certain types of design errors will simulate correctly in the RTL simulator but produce incorrect results in the gate-level simulator (and in a fabricated prototype). In addition, the gate-level simulator generates detailed activity statistics for every net in the design. Synopsys PrimeTime is used to combine each net's switching activity with the corresponding parasitic capacitance values from the layout. This generates average power for each module in the design. From the average power, cycle time, and cycle counts, we can calculate each microbenchmark's energy-efficiency in Joules per iteration for each core configuration.

As an example of our evaluation methodology, Figure 7.3 shows the post-place-&-route chip plot for the *vt-1x8* core configuration. The modules are labeled according to the microarchitectural diagram in Figure 5.1. Using our pre-placed register-file generator results in regularly structured vector and control-processor register files. Although this produces much better results than purely synthesizing these large register files, the pre-placed register files are still significantly larger than what would be possible with a custom register-file generator. We avoid any serious floorplanning of the vector functional units, even though there is the possibility to better exploit the structure inherent in these kinds of vector units. The benefit of these limitations is that they enable much faster development of a wide range of core configurations, and the results should still provide reasonable preliminary comparisons. Future work will iteratively refine a smaller set of core configurations to

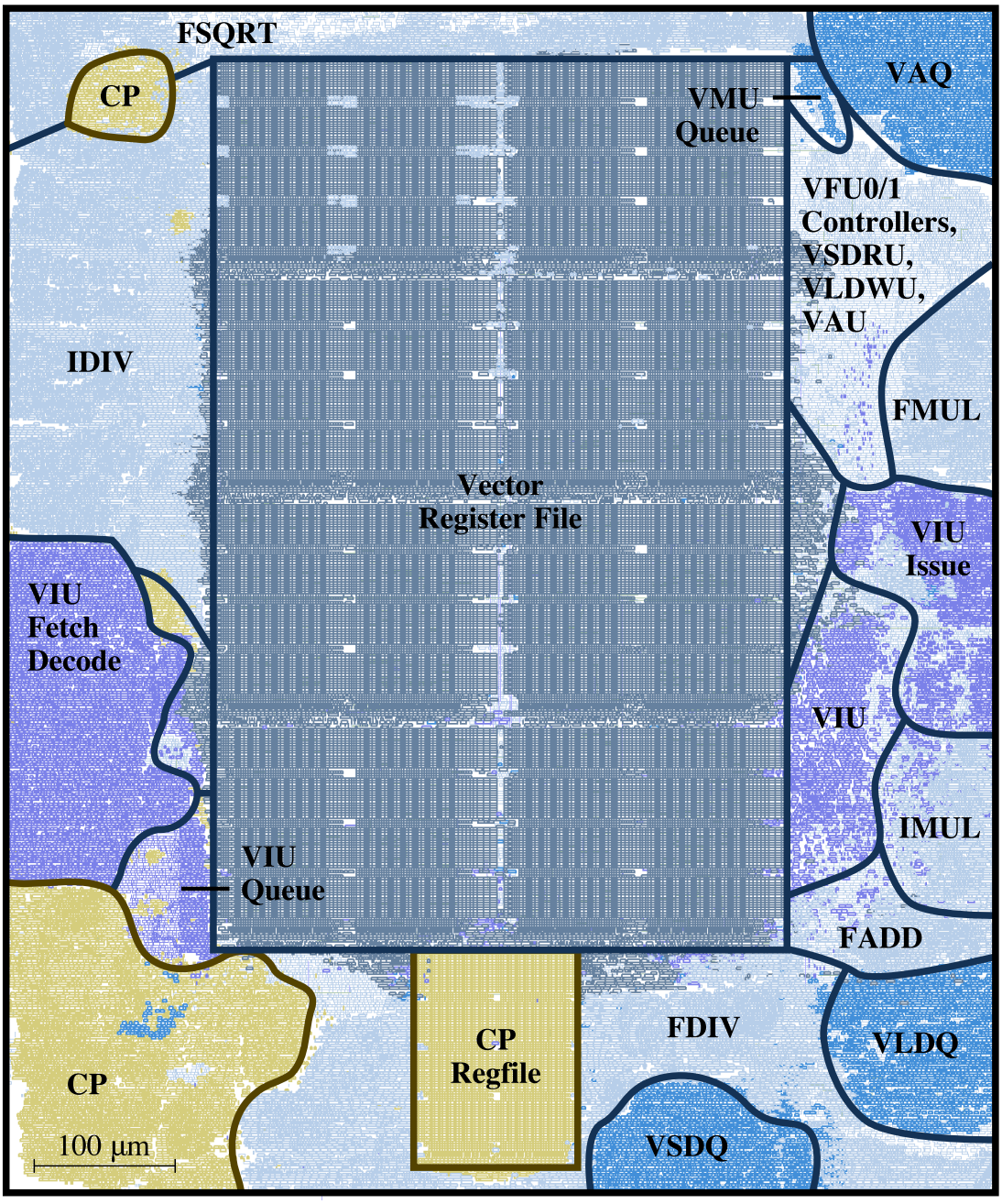


Figure 7.3: Chip Floorplan for *vt-1x8* Core – Chip floorplan showing position of each gate in the post-place-&-route gate-level model. Modules are labeled according to the microarchitectural diagram in Figure 5.1. Register files use pre-placed standard-cell latches. Total area is 690 µm × 825 µm. (CP = control processor, VIU = vector issue unit, VMU =vector memory unit, VFU = vector functional unit, VAU = vector address unit, VSDRU = vector store-data read unit, VLDWU = vector load-data writeback unit, VAQ = vector address queue, VSDQ = vector store-data queue, VLDQ = vector load-data queue, INT = simple integer ALU, IMUL/IDIV = integer multiplier and divider, FMUL/FADD/FSQRT/FDIV = floating-point multiplier, adder, square root unit, and divider.)

improve the quality of results, and eventually the designs will be suitable for fabricating a prototype.

The software and hardware toolflow is completely scripted, which allows us to quickly push new designs through the toolflow. Using a small cluster of eight-core servers, the entire set of 20 core configurations and 18 microbenchmark implementations can be pushed through the toolflow to generate a complete set of results overnight.

7.4 Cycle-Time and Area Comparison

Table 7.3 shows the cycle time for each of the 20 core configurations. The tools worked hard to meet the 1.5 ns cycle time constraint, but ultimately the cycle times range from 1.57–2.08 ns (637–481 MHz). Generally, the cycle times increase as configurations include larger physical register files, which require longer read and write global-bitlines. Analysis of the critical paths in these configurations verify that the cycle time is usually limited by accessing the register file. Although it might be possible to pipeline the register file access, a more promising direction is to move to a

		Cycle	Area (Thousands μm^2)						
Pattern	Variant	Time (ns)	Reg File	Int Dpath	FP Dpath	Mem Dpath	Ctrl Logic	CP	Total
mimd	1x1	1.57	22.2	60.8	53.4	4.4	7.4		148
mimd	1x2	1.78	44.5	59.3	51.9	4.4	8.9		169
mimd	1x4	1.83	89.0	63.8	51.9	4.4	8.9		218
mimd	1x8	2.08	179.4	69.7	51.9	4.4	8.9		314
vsimd	1x1	1.64	59.3	56.3	44.5	29.7	32.6	50.4	273
vsimd	1x2	1.79	105.3	54.9	44.5	28.2	34.1	48.9	316
vsimd	1x4	1.73	197.2	54.9	44.5	28.2	35.6	48.9	409
vsimd	1x8	1.84	390.0	54.9	43.0	28.2	38.6	47.5	602
vsimd	2x1	1.75	115.7	105.3	89.0	54.9	40.0	48.9	454
vsimd	2x2	1.68	209.1	108.2	89.0	54.9	41.5	50.4	553
vsimd	2x4	1.67	397.4	108.2	89.0	54.9	43.0	53.4	746
vsimd	2x8	1.97	784.4	112.7	87.5	54.9	47.5	53.4	1140
vsimd	4x1	1.73	229.8	215.0	176.5	109.7	53.4	51.9	836
vsimd	4x2	1.72	413.7	212.0	172.0	108.2	54.9	51.9	1013
vsimd	4x4	1.92	784.4	210.6	172.0	111.2	56.3	50.4	1385
vsimd	4x8	2.07	1546.6	212.0	170.5	114.2	68.2	53.4	2165
vt	1x1	1.67	46.0	54.9	47.5	28.2	51.9	48.9	277
vt	1x2	1.67	91.9	56.3	46.0	29.7	51.9	48.9	325
vt	1x4	1.68	183.9	57.8	46.0	29.7	54.9	50.4	423
vt	1x8	1.81	379.6	60.8	46.0	29.7	54.9	50.4	621

Table 7.3: Absolute Area and Cycle Time for Core Configurations – Cycle time as measured by static timing analysis after place-&-route. Area breakdown as reported by place-&-route tool. Area breakdown for vector-SIMD and VT cores are for the vector unit with the entire control processor grouped into a single column. See Figure 7.4 for plot of normalized area for each configuration.

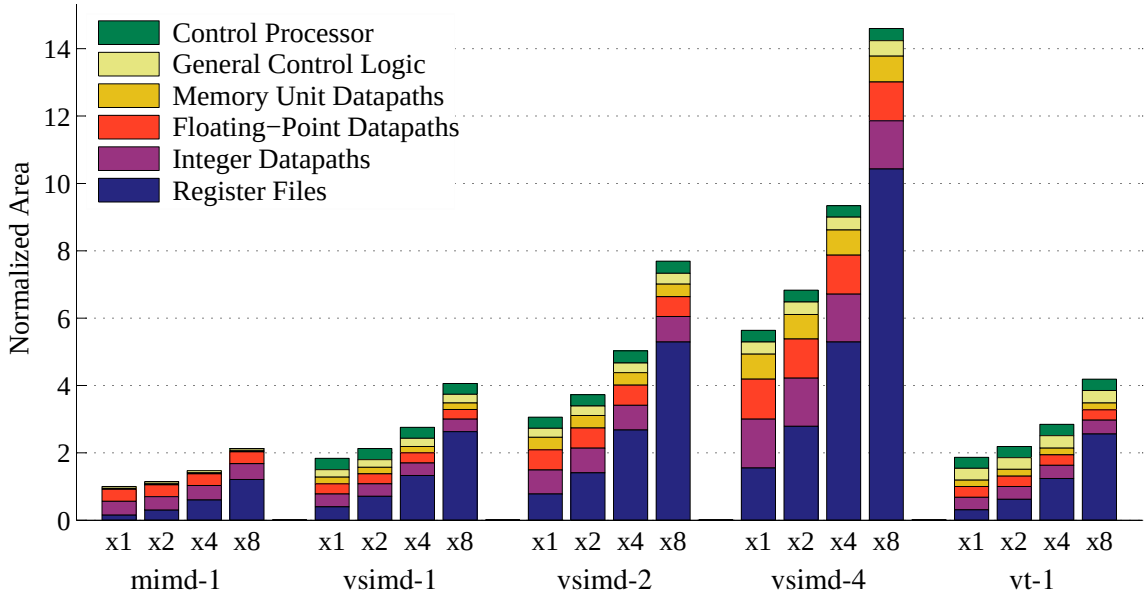


Figure 7.4: Normalized Area for Core Configurations – Area for each of the 20 core configurations listed in Table 7.1 are shown normalized to the *mimd-1x1* core. Note that the embedded control processor helps mitigate the area overhead of single-lane vector units, and the VT cores have very similar area as the single-lane vector-SIMD cores. (See Table 7.3 for absolute area numbers. Area breakdown for vector-SIMD and VT cores are for the vector unit with all parts of the control processor grouped together on top.)

banked design as described in Section 5.7. Increasing the number of lanes has less of an impact on the cycle time, since the larger vector register file in multi-lane vector units is naturally banked by lane. The cycle times vary from 6–13% for configurations with the same number of physical registers per lane, and vary 25% across all configurations.

Table 7.3 also shows the area breakdown for each of the 20 core configurations, and Figure 7.4 plots the area normalized to the *mimd-1x1* configuration. As expected the register file dominates the area particularly in *-**x8* configurations. These results can help us better understand the area impact of moving to multi-lane vector units, embedding the control processor, and adding VT mechanisms.

Increasing the number of lanes in the *vsimd-1x**, *vsimd-2x**, and *vsimd-4x** configurations obviously also increases the total area. Notice that the relative increase in the control logic area is much less than the relative increase in the register file and datapaths. For example, from the *vsimd-1x8* configuration to the *vsimd-4x8* configuration the register-file and datapath area increases by 395%, but the control logic only increases by 76%. This is because, in a multi-lane vector unit, most of the control logic is amortized across the lanes.

To evaluate control processor embedding, we can compare the area of the four-lane configurations (*vsimd-4x4*, *vsimd-4x8*) with four times the area of the single-lane configurations (*vsimd-1x4*, *vsimd-1x8*). The *vsimd-**x1* and *vsimd-**x2* configurations are less practical, since they support much shorter hardware vector lengths. The four single-lane configurations consume approximately

11–18% more area than the corresponding four-lane configurations. This comparison favors the multi-lane configurations, since we are assuming control-processor embedding is possible in all configurations. The embedded control processor area as a fraction of the total area for the *vt-1x4* and *vt-1x8* configurations ranges from 8–11%. If the control processor required its own long-latency functional units and memory ports, the area overhead would range from ≈22–30%. Even if more optimization decreases the size of the register file by a factor of 2–3×, the embedded control processor area would still be limited to 11–16%. This is a conservative estimate, because optimized register files would also decrease the area of the control processor. These results imply that control processor embedding is a useful technique, and enables single-lane vector units without tremendous area overhead.

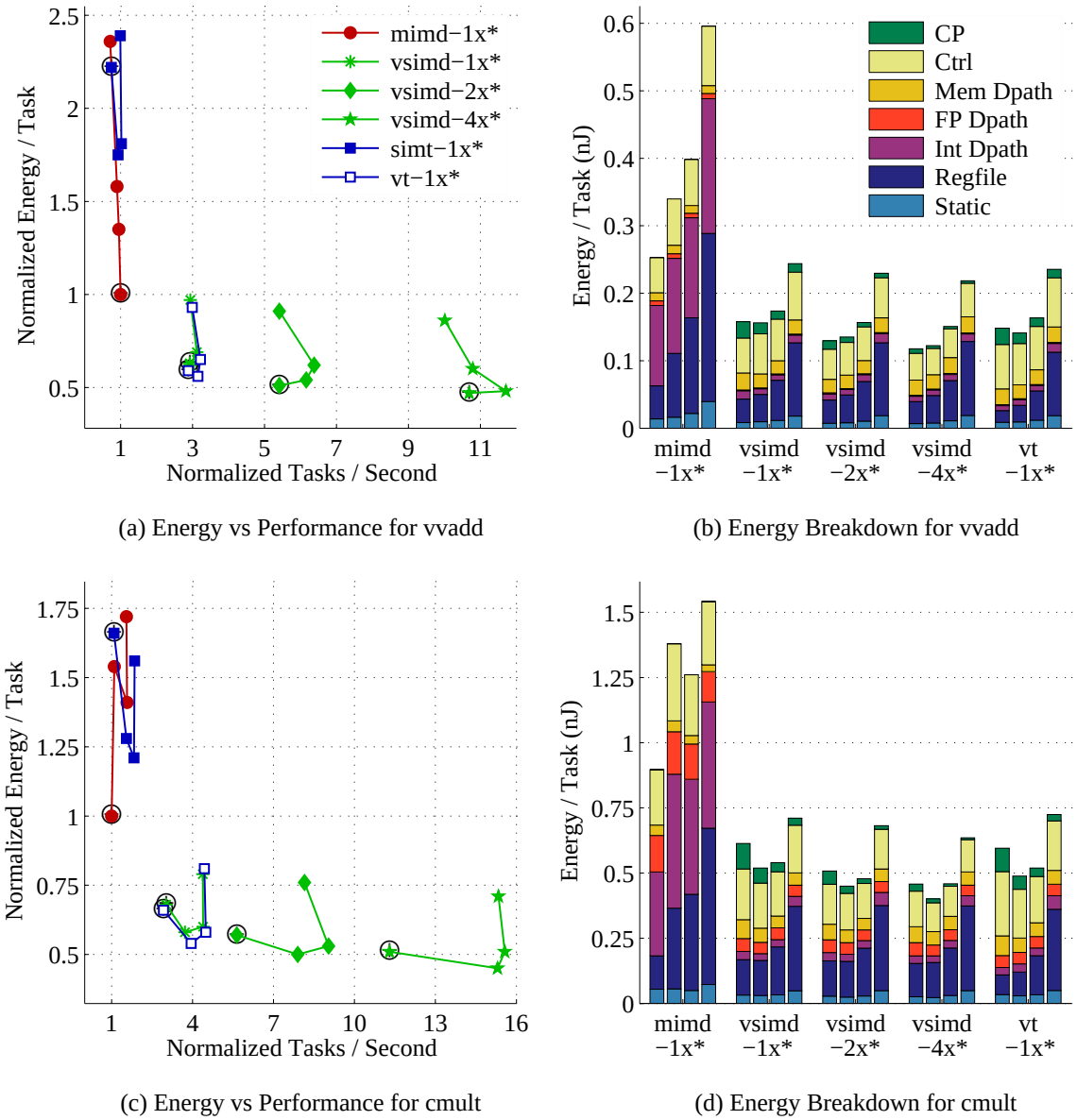
One of the design goals for the Maven VT core was to add VT capabilities with minimal changes to a vector-SIMD core. The hope is that this would allow the Maven VT core to maintain the area- and energy-efficiency advantages of the vector-SIMD core while adding increased flexibility. We can see the area impact of these changes in Table 7.3 by comparing the *vt-1x8* configuration to the *vsimd-1x8* configuration. The total area is within 3%, but this area is allocated across design differently. The largest discrepancies are in the register file and control logic. The vector-SIMD core has a larger register file, since it includes the vector flag register file. The VT core has more control logic, since it includes the PVFB. These two structures (vector flag register file and the PVFB) both help the corresponding cores manage irregular DLP, but they do so in very different ways. Ultimately, the VT cores are able to provide more flexible handling of irregular DLP with a similar total area as the single-lane vector-SIMD cores.

7.5 Energy and Performance Comparison for Regular DLP

Figures 7.5a–b show the energy and performance for the *vvadd* microbenchmark running on the 20 core configurations. Figure 7.5a shows the energy and performance normalized to the *mimd-1x1* configuration, and Figure 7.5b shows the absolute energy breakdown for each configuration. The *vvadd* microbenchmark running on the MIMD configurations has few critical execution latencies. There is a two-cycle memory latency, one-cycle branch resolution latency, and no functional unit latencies, since *vvadd* only performs single-cycle integer additions. As a consequence, there are few execution latencies to hide, and increasing the number of MIMD microthreads results in no performance improvement. Moving to eight threads (*mimd-1x8*) actually degrades performance due to the non-trivial start-up overhead required to spawn and join the microthreads. The energy increases significantly for larger MIMD cores, and Figure 7.5b shows that this is mostly due to increased register file energy. Analysis of the energy per cycle results, reveal that the increase in register energy is partially due to the larger register file required to support microthread contexts but also due to the increased number of start-up instructions required to manage more microthreads. These extra instructions also slightly increase the control and integer datapath energy per task.

For the *vvadd* microbenchmark, the vector-SIMD cores are able to achieve higher performance at lower energy per task than the MIMD cores. Figure 7.5a shows that the *vsimd-1x4* configuration is 3× faster than the *mimd-1x1* configuration, and consumes just 60% of the energy to execute the same task. The *vsimd-1x4* configuration is almost 3× larger, but area normalizing these results to larger multithreaded MIMD cores results in an even larger margin of improvement. Instantiating three *mimd-1x1* cores would help improve throughput but would not reduce the energy per task. Notice that the speedup is greater than what is implied by the *vsimd-1x4* configuration's ability to execute two integer additions per cycle using the two vector functional units. This is because the *vsimd-1x4* configuration not only executes the fundamental addition operations faster but also does less work by amortizing various overheads onto the control processor. Figure 7.5b shows the effect of increasing the number of microthreads temporally. Moving from *vsimd-1x1* (10 µTs) to *vsimd-1x8* (32 µTs) decreases the control processor energy because longer hardware vector lengths enable greater amortization. At the same time the register file energy increases because the global bitlines in the register file grow longer. In the larger configurations, the larger register file energy outweighs the smaller control processor energy. Figure 7.5b also shows the effect of increasing the number of microthreads spatially with multiple lanes. Moving from *vsimd-1x1* (10 µTs) to *vsimd-4x1* (40 µTs) decreases the control processor energy via longer hardware vector lengths. In addition, the multi-lane configuration is able to achieve better energy efficiency as compared to a single-lane configuration with a similar number of microthreads, since structures such as the control logic and register file are banked across the lanes. Ultimately, increasing the number of lanes improves performance and slightly reduces the energy per task. Multiple single-lane vector units could potentially achieve similar throughput but without the slight reduction in energy per task. Finally, note that all of the vector-SIMD results show decreased performance when moving from few microthreads per lane to many microthreads per lane (e.g., *vsimd-2x1* to *vsimd-2x8*). This is due to the longer cycle times in these larger designs. Figure 7.6a shows the energy and performance assuming all configurations run at the same cycle time. Notice that the performance is then always monotonically increasing with an increasing number of microthreads.

The SIMT results are for the SIMT implementation of the *vvadd* microbenchmark running on the *vt-1x** core configurations. Because the SIMT implementation does not use the control processor to amortize control overheads nor to execute vector memory commands, the only energy-efficiency gains are from the vector-like execution of the arithmetic operations. The results in Figure 7.5a show that this is not enough to produce any significant energy-efficiency improvement. In fact, the SIMT configurations perform much worse than the MIMD configurations due to the additional work each microthread has to perform. Each microthread must check to see if its index is less than the array size and also calculate its array index using an integer multiply operation. Although these overheads make the SIMT pattern much less attractive, more realistic SIMT implementations would include a multithreaded VIU to hide the branch resolution latency and dynamic memory coalescing



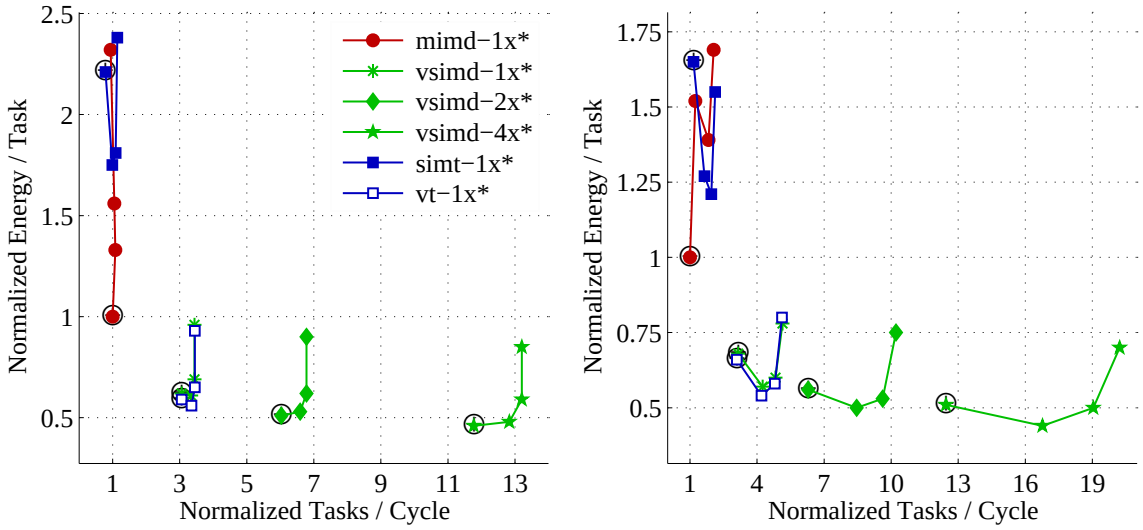
(a) Energy vs Performance for vvadd

(b) Energy Breakdown for vvadd

(c) Energy vs Performance for cmult

(d) Energy Breakdown for cmult

Figure 7.5: Results Regular DLP Microbenchmarks – Results for the 20 configurations listed in Table 7.1 running the four implementations of the *vvadd* and *cmult* microbenchmarks listed in Table 7.2. The VT cores are able to achieve similar energy-efficiency and performance as the vector-SIMD cores on regular DLP. (Energy and performance calculated using the methodology described in Section 7.3. Results in (a) and (c) normalized to the *mimd-1x1* configuration. For each type of core, the **-*x1* configuration is circled in plots (a) and (c), and the remaining three data-points correspond to the **-*x2*, **-*x4*, and **-*x8* configurations. Each group of four bars in plots (b) and (d) correspond to the **-*x1*, **-*x2*, **-*x4*, and **-*x8* configurations. Energy breakdown for vector-SIMD and VT cores are for vector unit with all parts of the control processor grouped together on top.)



(a) vvadd (b) cmult

Figure 7.6: Results for Regular DLP Microbenchmarks with Uniform Cycle Time – These plots are similar to Figures 7.5a and 7.5c, except that the performance and energy assume all core configurations can operate at the same cycle time. Notice that the performance almost always improves or stays constant with increased physical register size.

to try and recoup some of the vector-like efficiencies from microthread loads and stores.

For regular DLP such as the *vvadd* microbenchmark, we expect the VT core to have similar energy-efficiency and performance as compared to the single-lane vector-SIMD core. Figure 7.5a confirms these expectations even though the VT core has to perform some additional work compared to the vector-SIMD core. For example, the VT core must execute an extra vector fetch and microthread stop instruction. Ultimately these overheads are outweighed by the vector-like efficiencies from executing vector memory instructions and vector-fetched scalar instructions. Figure 7.5b shows a similar energy breakdown for both the *vt-1x** and *vsimd-1x** configurations. Note that the SIMT and VT results are running on the exact same hardware, but the VT configurations achieve much higher performance and lower energy per task. The only difference is that the VT implementation makes extensive use of the control thread to amortize various overheads, while the SIMT configuration requires each microthread to redundantly execute many of these control operations.

Figures 7.5c–d show the energy and performance for the *cmult* microbenchmark. These results are similar to those for the *vvadd* microbenchmark except for the floating-point arithmetic causing increased floating-point-datapath energy and decreased integer-datapath energy. The MIMD configurations are able to achieve some performance improvement as the multiple threads help hide the floating-point execution latencies. The non-monotonic energy profile of the MIMD configurations is caused by performance issues with how the microthreads are scheduled and the sharing of the second register-file write port between the floating-point units and the load writeback. The vector-

SIMD and VT cores are able to once again achieve higher performance (3–16×) at reduced energy per task (50–75%) compared to the *mimd-1x1* configuration.

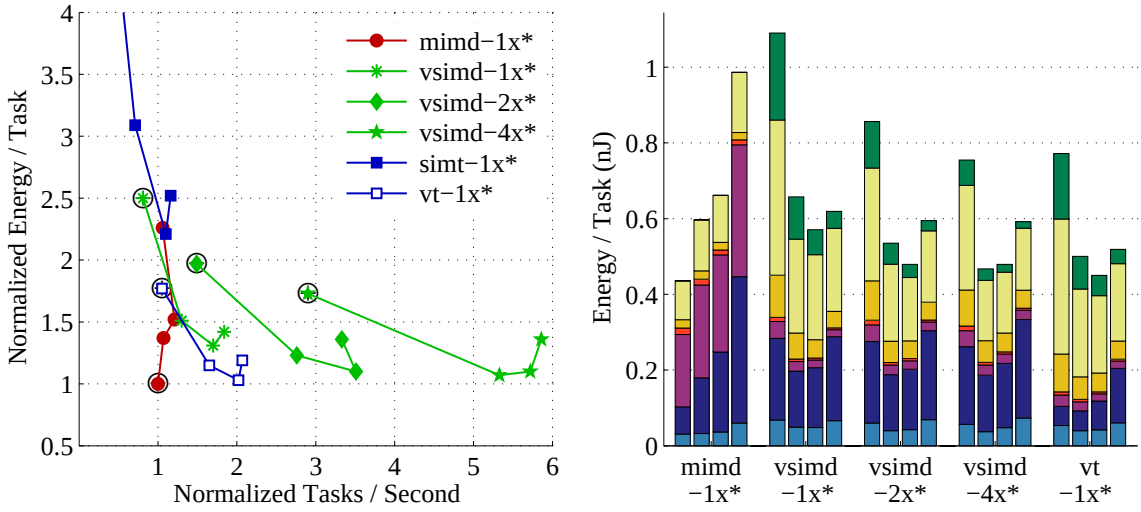
Overall, the results for both regular DLP microbenchmarks are very encouraging. The VT cores are able to achieve vector-like energy-efficiencies on regular DLP even though the VT core uses a very different VIU implementation. The multithreaded MIMD cores make less sense unless there are significant execution latencies that can be hidden by interleaving multiple threads. The SIMT configurations would require significantly more hardware support before they can be competitive with the vector-SIMD and VT cores.

7.6 Energy and Performance Comparison for Irregular DLP

Figures 7.7a–b show the energy and performance for the *mfilt* microbenchmark running on the 20 core configurations. Figure 7.7a shows that additional microthreads improve the performance of the MIMD cores slightly, mainly due to hiding the integer-division execution latency. Since the MIMD pattern executes regular and irregular DLP identically, the energy breakdown results for the MIMD configurations in Figure 7.7a are similar to those for the regular DLP microbenchmarks in Figure 7.5. More microthreads require larger register files and increase start-up overheads, which together increase the energy per task.

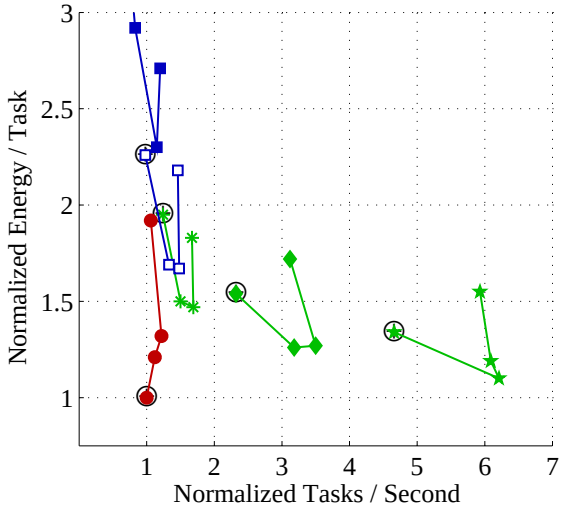
The vector-SIMD cores are less effective at executing irregular DLP. The very small *vsimd-1x1* configuration actually has lower performance and 2.5× worse energy per task as compared to the *mimd-1x1* configuration. The *vsimd-1x1* configuration only supports a hardware vector length of two, and this is too short to achieve reasonable control amortization or to effectively hide the integer-division execution latency. Larger vector-SIMD configurations are better able to improve performance resulting in a 2× speedup for the *vsimd-1x8* configuration and a 6× speedup for the *vsimd-4x8* configuration. Increasing the size of the physical register file per lane improves the energy efficiency through longer hardware vector lengths, which better amortizes the control processor and vector control overheads. Eventually this is outweighed by a larger vector register file and more wasted work due to a greater faction of inactive microthreads. Also notice that while increasing the number of lanes does reduce the energy per task (*vsimd-1x4* to *vsimd-2x4* decreases energy by ≈20%), the marginal reduction from two to four lanes is much less. There is only so much control overhead that can be amortized, and Figure 7.7b shows static energy starting to play a more significant role in the larger designs. Ultimately, the vector-SIMD cores are able to improve performance but not improve the energy efficiency as compared to the *mimd-1x1* configuration. It should also be reiterated that the vector-SIMD implementation of the *mfilt* microbenchmark required hand-coded assembly.

As with the regular DLP microbenchmarks, the SIMT configurations are unable to improve much beyond the MIMD cores. Again, this is caused by the increased number of scalar instructions that each microthread must execute even in comparison to the MIMD implementations. The *mfilt*

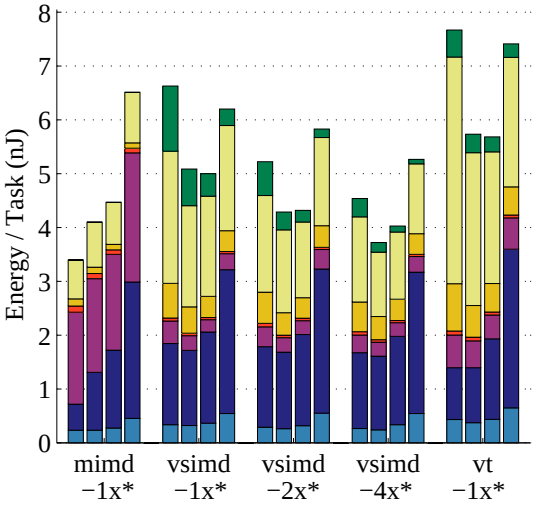


(a) Energy vs Performance for mfilt

(b) Energy Breakdown for mfilt



(c) Energy vs Performance for bsearch



(d) Energy Breakdown for bsearch

Figure 7.7: Results for Irregular DLP Microbenchmarks – Results for the 20 configurations listed in Table 7.1 running the four implementations of the *mfilt* and *bsearch* microbenchmarks listed in Table 7.2. The VT core is sometimes better and sometimes worse than the vector-SIMD core on irregular DLP. Future enhancements include vector fragment merging, interleaving, and compression are likely to improve both the performance and energy efficient of the VT core on irregular DLP. (Energy and performance calculated using the methodology described in Section 7.3. The *bsearch* benchmark for the SIMT and VT cores uses explicit conditional moves and corresponds to the *simt-cmv* and *vt-cmv* entries in Table 7.2. Results in (a) and (c) normalized to the *mimd-1x1* configuration. For each type of core, the *-*x1* configuration is circled in plots (a) and (c), and the remaining three data-points correspond to the *-*x2*, *-*x4*, and *-*x8* configurations. See Figure 7.5b for the legend associated with plots (b) and (d). Each group of four bars in plots (b) and (d) correspond to the *-*x1*, *-*x2*, *-*x4*, and *-*x8* configurations. Energy breakdown for vector-SIMD and VT cores are for vector unit with all parts of the control processor grouped together on top.)

microbenchmark is, however, straight-forward to implement using the SIMT pattern owing to the flexible programmer's model, especially when compared to the hand-coded assembly required for the vector-SIMD implementation.

The VT configurations are able to do much better than the SIMT configurations, yet they maintain the MIMD pattern's simple programming model. The primary energy-efficiency advantage of the VT versus SIMT configurations comes from using the control thread to read the kernel coefficients with shared loads, amortize the shared computation when calculating the normalization divisor, read the image pixels with unit-stride vector loads, and manage the pointer updates. To help better understand the *mfilt* microbenchmark's execution on the VT core, Figure 7.8a shows what fraction of the vector fragment micro-ops are active for all four VT configurations. More physical registers enable longer hardware vector lengths, but this also gives greater opportunity for the longer fragments to diverge. For example, approximately 8% of all fragment micro-ops have less than the maximum number of active microthreads in the *vt-1x1* configuration, but in the *vt-1x8* configuration this increases to 70%. A critical feature of the Maven VT microarchitecture is that the energy efficiency for these diverged fragments is still roughly proportional to the number of active microthreads. As Figure 7.7a shows, this allows the energy per task to decrease with increasing hardware vector lengths even though the amount of divergence increases. The VT core is actually able to do better than the vector-SIMD cores because of a key feature of the VT pattern. Since data-dependent control flow is expressed as vector-fetched scalar branches, the VT core's VIU can completely skip the convolution computation when all microthreads determine that their respective pixels are under mask. This happens 50% of the time in the *vt-1x8* configuration for the dataset used in this evaluation. This is in contrast to the vector-SIMD implementation, which must process vector instructions executing under a flag register even if all bits in the flag register are zero. Vector-SIMD cores can skip doing any work when the flag register is all zeros, but they must at least process these instructions. Thus, the VT configurations are able to execute at slightly higher performance and up to 30% less energy than their single-lane vector-SIMD counterparts. This also allows the Maven VT core to achieve a 2× speedup at equivalent energy per task as compared to the *mimd-1x1* configuration.

The current Maven VT core does not include support for vector fragment merging, interleaving, or compression. In the *mfilt* microbenchmark, there is only a single microthread branch, so vector fragment interleaving is unlikely to provide much benefit. The microthreads naturally reconverge after each iteration of the stripmine loop. Vector fragment merging can provide a very slight benefit as there are two common microthread instructions at the target of the microthread branch. Vector fragment compression could improve performance and energy efficiency, especially on those fragment micro-ops with few active microthreads per fragment.

The *bsearch* microbenchmark is the most irregular of the four microbenchmarks studied in this thesis. Figures 7.7c–d show that the MIMD configurations perform similar as in the other

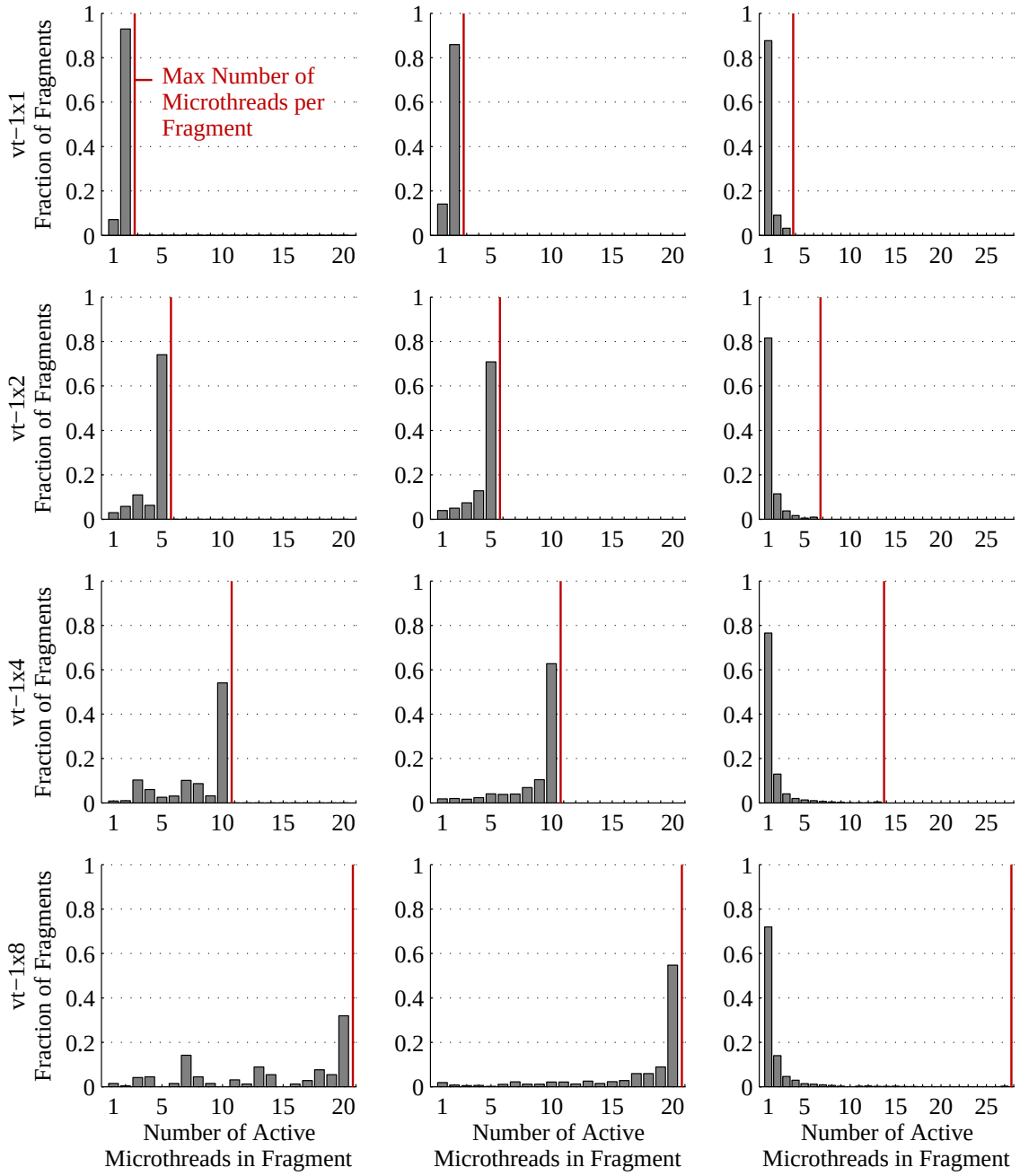


Figure 7.8: Divergence in Irregular DLP Microbenchmarks – Normalized histograms showing what fraction of vector fragment micro-ops have the given number of active microthreads. Increasing the hardware vector length leads to more possibilities for divergence in both the *mfilt* and *bsearch* microbenchmarks. Without explicit conditional moves, most of the fragment micro-ops in the *bsearch* microbenchmark are completely diverged with a single active microthread in each fragment. (Maximum number of microthreads per fragment (i.e., the hardware vector length) shown with vertical line.)

microbenchmarks. This is to be expected since the MIMD cores execute both regular and irregular DLP in the same way. As in the *mfilt* microbenchmark, the vector-SIMD configurations are able to improve performance, but there is not enough regularity in the microbenchmark to reduce the energy per task below the *mimd-1x1* configuration. As in the other microbenchmarks, the microthreads in the SIMT configurations must execute too many additional instructions to achieve any improvement in performance or energy-efficiency.

The results for the VT core in Figures 7.7c–d correspond to the *vt-cmv* row in Table 7.2. This implementation includes explicitly inserted conditional move operations to force the compiler to generate vector-fetched conditional move instructions as opposed to vector-fetched scalar branches. There is one remaining backward branch that implements the inner `while` loop. Given this optimization. the VT core has slightly worse performance and requires ≈15% more energy per task as compared to the single-lane vector-SIMD cores. Unlike the *mfilt* microbenchmark, there are less opportunities to skip over large amounts of computation. The fragment can finish early if all microthreads in the fragment find their search key before reaching the maximum number of look-ups. Unfortunately, this happens rarely, especially in the larger configurations with the longer hardware vector lengths. The energy overhead then, is somewhat due to the VIU's management of divergent fragments as microthreads gradually find their search key and drop out of the computation. Figure 7.7d clearly shows a significant increase in the control energy for the VT configurations as compared to the vector-SIMD configurations. Some of this energy is also due to the VIU's need to fetch and decode vector-fetched scalar instructions, which is handled by the control-processor in the vector-SIMD configurations. The VT configurations are limited to the MIPS32 conditional move instruction for expressing conditional execution, while the vector-SIMD configurations use a more sophisticated vector flag mechanism. These vector flags allow the vector-SIMD configurations to more compactly express the conditional execution, which also accounts for some of the performance discrepancy. Even though this is an irregular DLP microbenchmark, comparing *vt-1x1* to *vt-1x8* in Figure 7.7d shows that the VT cores are still able to amortize control processor and control overheads with increasing hardware vector lengths. In the largest configuration, the energy overhead of the larger vector register file outweighs the decrease due to better amortization.

Figure 7.8b shows the divergence for all four VT configurations. As with the *mfilt* microbenchmark, increasing the number of physical registers leads to longer hardware vector lengths, but also causes additional divergence. In the *vt-1x8* configuration, about 50% of the fragment micro-ops stay coherent before microthreads begin to find their respective keys and exit their while loops. Contrast these results to those shown in Figure 7.8c, which correspond to the *bsearch* microbenchmark without the explicit conditional moves. In this implementation the compiler generates many conditional branches (and only one conditional move), and as a result the fragment micro-ops quickly diverge. For this implementation running on the *vt-1x8* configuration, over 70% of the fragment micro-ops contain only a single microthread. Figure 7.9 shows how this additional divergence impacts the

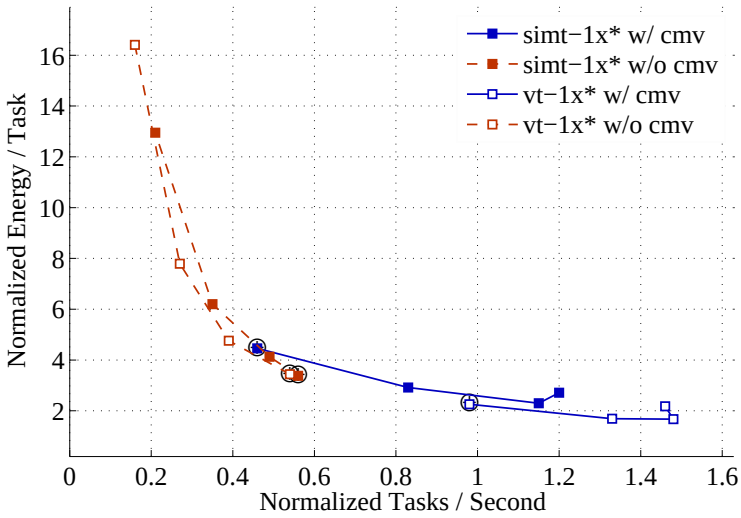


Figure 7.9: Results for *bsearch* Microbenchmark without Explicit Conditional Moves – Without explicitly inserted conditional moves, the compiler generates many data-dependent conditional branches. Unfortunately, the current Maven microarchitecture is unable to efficiently handle this number of branches with so few instructions on either side of the branch. Future Maven implementations can use vector fragment merging, interleaving, and compression to improve the efficiency of such highly-irregular DLP. (Energy and performance calculated using the methodology described in Section 7.3. The *w/ cmv* implementations correspond to the *bsearch/simt-cmv* and *bsearch/vt-cmv* rows in Table 7.2, while the *w/o cmv* implementations correspond to the *bsearch/simt* and *bsearch/vt* rows in Table 7.2. Results normalized to the *mimd-1x1* configuration. For each type of core, the **-*x1* configuration is circled and the remaining three data-points correspond to the **-*x2*, **-*x4*, and **-*x8* configurations.)

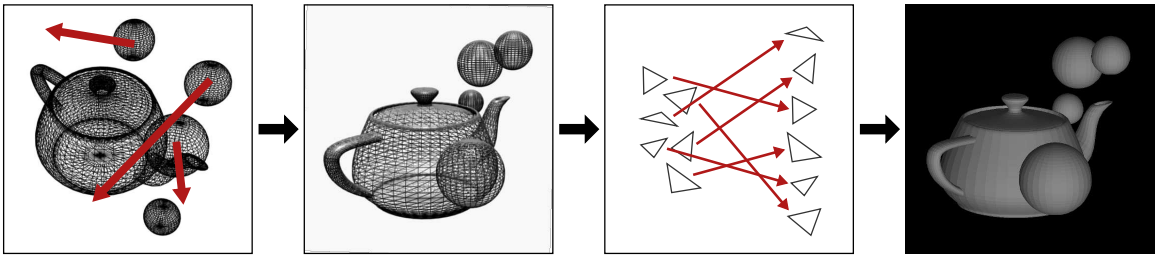
performance and energy on the SIMT and VT configurations. Increasing the number of physical registers actually causes the performance and energy to worsen considerably. In this example, longer hardware lengths simply lead to more inactive microthreads per fragment.

Overall, the results for irregular DLP are a promising first step towards a Maven VT core that can efficiently handle all kinds of irregular DLP. The results for the *mfilt* microbenchmark illustrate the advantage of completely skipping large amounts of computation through vector-fetched scalar branches. The results for the *bsearch* microbenchmark are not as compelling, but do raise some interesting opportunities for improvement. Conditional move instructions can help mitigate some of the overheads associated with vector-fetched scalar branches, and it should be possible to modify the compiler to more aggressively use this type of conditional execution. It should also be possible to use the microarchitectural techniques introduced in Chapter 5, such as vector fragment merging, interleaving, and compression, to improve the performance and energy efficiency of the *bsearch* microbenchmark specifically and irregular DLP more generally. One of the strengths, however, is the simple programming methodology that allows mapping irregular DLP to the Maven VT core without resorting to hand-coded assembly.

7.7 Computer Graphics Case Study

As a first step towards evaluating larger applications running on a Maven VT core, we have started mapping a computer graphics application using the Maven programming methodology. Figure 7.10 illustrates the four kernels that together form this application: the *cg-physics* kernel performs a simple Newtonian physics simulation with object collision detection; the *cg-vertex* kernel handles vertex transformation and lighting; the *cg-sort* kernel sorts the triangles based on where they appear in the frame buffer; and the *cg-raster* kernel rasterizes the triangles to the final frame buffer. The rendering pipeline is an example of a sort-middle parallel rendering algorithm where both the *cg-vertex* and *cg-raster* kernels are completely data-parallel, and the global communication during the *cg-sort* kernel reorganizes the triangles [MCEF94]. Table 7.4 shows the instruction mix for each kernel. The kernels include a diverse set of data-dependent application vector lengths ranging from ten to hundreds. Although some tuning is still possible, all kernels currently use the full complement of 32 registers per microthread. All four kernels were written from scratch and include a scalar, multithreaded MIMD, and VT implementation. The application requires a total of approximately 5,000 lines of C++ code.

The *cg-physics* kernel includes four phases per time step. In the first phase, the control thread creates an octree to efficiently group objects into cells that are close in space. Octree creation accounts for only a small portion of the kernel's run-time, but it may be possible to parallelize this phase in the future. The second phase determines which objects intersect. Each microthread is responsible for comparing one object to all other objects in the same octree cell. The objects are read using microthread loads, since the octree only stores pointers to objects as opposed to the objects themselves (making octree formation much faster). The microthreads perform several data-dependent conditionals to determine if the two objects under consideration collide. The second



(a) cg-physics (b) cg-vertex (c) cg-sort (d) cg-raster

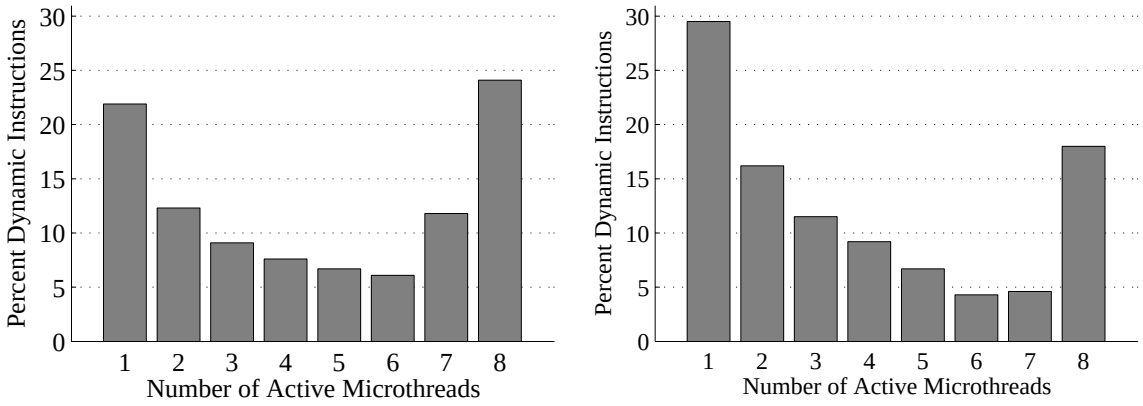
Figure 7.10: Computer Graphics Case Study – The case study is comprised of four application kernels: (a) the *cg-physics* kernel performs a simple Newtonian physics simulation with collisions on a scene of rigid bodies each made up of many triangles, (b) the *cg-vertex* kernel transforms the resulting triangles into camera coordinates, projects the triangles into the 2D screen coordinates, and updates each vertex with lighting information, (c) the *cg-sort* kernel uses a radix sort to organize the triangles based on which rows in the frame buffer they cover, and (d) the *cg-raster* kernel rasterizes the triangles into the final frame buffer. Final image is an example of the actual output from the computer graphics application.

	Control Thread						Microthread								
Name	**lw.v**	**sw.v**	**lwst.v**	**swst.v**	**mov.sv**	**vf**	**int**	**fp**	**ld**	**st**	**amo**	**br**	**j**	**cmv**	**tot**
cg-physics	6	1	12	9	15	4	5	56	24	4	0	16	0	0	132
cg-vertex	0	16	42	25	76	12	0	206	0	0	0	0	0	7	255
cg-sort	5	4	2	0	7	3	16	0	2	3	1	0	0	0	27
cg-raster	2	0	13	0	4	4	155	3	37	26	6	69	14	24	482

Table 7.4: Instruction Mix for Computer Graphics Kernels – Number of instructions for each application kernel are listed by type. The *cg-vertex* and *cg-sort* kernels have mostly regular data access and control flow, although *cg-vertex* includes conditional move instructions and *cg-sort* includes microthread load/stores and atomic memory operations. The *cg-physics* and *cg-raster* kernels have much more irregular data access and control flow. Note that the instruction counts for the *cg-raster* kernel reflect compiler loop unrolling by a factor of six. (lw.v/sw.v = unit-stride load/store, lwst.v/swst.v = strided load/store, mov.sv = copy scalar to all elements of vector register, vf = vector fetch, int = simple short-latency integer, fp = floating-point, ld/st = load/store, amo = atomic-memory operation, br = conditional branch, j = unconditional jump, cmv = conditional move)

phase is actually split into two vector-fetched blocks to allow vector memory accesses to interleave and better block the computation into vector registers. A conditional control-flow state variable is required to link the control flow in the vector-fetched blocks together. This phase can potentially experience significant divergence, since different microthreads will exit the vector-fetched blocks at different times based on different collision conditions. The third phase iterates over just the colliding objects and computes the new motion variables. Each microthread is responsible for one object and iterates over the list of objects it collides with using an inner `for` loop. The microthreads must again access the objects through microthread loads, since the collision lists are stored as pointers to objects. This phase will probably experience little divergence, since the number of objects that collide together in the same time step is usually just two or three. The final phase iterates over all objects in the scene and updates the location of each object based on that object's current motion variables. The *cg-physics* is a good example of irregular DLP. Table 7.4 shows that it includes a large number of microthread loads/stores (28) and conditional branches (16). Figure 7.11a shows the amount of divergence in this kernel when simulating a scene with 1000 small pyramids moving in random directions. Approximately 75% of the fragment micro-ops experience some amount of divergence, and 22% of the instructions are executed with only one microthread active per fragment. Although this kernel exhibits irregular DLP, it would still be simple to extend this implementation to multiple cores. Multiple cores might, however, increase the need to parallelize octree creation in the first phase. For the remaining phases, each core would work on a subset of the objects.

The *cg-vertex* kernel iterates over all triangles in the scene applying coordinate transformations and updating lighting parameters. Each microthread works on a specific triangle by first transforming the three vertices into camera coordinates, then applying light sources to each vertex via flat shading, and finally projecting the vertices into quantized screen coordinates. This kernel includes a few small data-dependent conditionals that are mapped to conditional move instructions by the



(a) cg-physics (b) cg-raster

Figure 7.11: Divergence in Computer Graphics Kernels – Normalized histograms showing what fraction of vector fragment micro-ops have the given number of active microthreads. Both irregular DLP kernels have a significant amount of divergence. Statistics for the *cg-physics* kernel processing a scene with 1000 small pyramids moving in random directions, and for the *cg-raster* kernel processing a scene of the space shuttle with 393 triangles and an image size of 1000×1000 pixels. Measurements assume a *vt-1x8* core configuration.

compiler. Otherwise, this kernel is a good example of regular DLP. Table 7.4 shows that it includes a large number of vector load/store instructions (83) and shared loads via the `mov.sv` instruction (76). Notice that this kernel uses 12 vector fetch instructions, even though it is parallelized by simply stripmining over all triangles. The large number of vector-fetched blocks are used to interleave vector load/store instructions amongst the microthread computation. This enables better use of the vector registers and prevents unnecessary microthread register spilling. It would be straight-forward to extend this kernel to multiple cores by simply having each core work on a subset of the triangle array.

The *cg-sort* kernel takes as input an array of triangles and a key for sorting. In this application, the key is the maximum y-coordinate across all vertices in a triangle. The maximum y-coordinate is computed in the previous kernel using the conditional move instructions. The kernel uses a radix sort to organize the triangles based on this key (i.e., from top to bottom in the frame buffer). The radix sort has three phases each with a different vector-fetched block. In the first phase, the kernel creates a histogram of the frequency of each digit in the key using atomic memory operations. In the second phase, the kernel calculates offsets into the final output array based on these histograms. In the final phase, the kernel writes the input data into the appropriate location based on these offsets. Vector memory fences are required between each phase to properly coordinate the cross-microthread communication. This algorithm is similar in spirit to previous approaches for vector-SIMD processors [ZB91]. One difference is the use of atomic memory operations to update the radix histograms. Table 7.4 shows that this is a mostly regular DLP kernel with a small number

of microthread loads and stores. Mapping this kernel to multiple cores is still possible, but more challenging, owing to the cross-microthread communication.

The *cg-raster* kernel takes as input the sorted array of triangles and rasterizes them to the frame buffer. The kernel includes two phases. In the first phase, the kernel stripmines over the sorted array of triangles, and each microthread creates a set of scanline chunks for its triangle. The chunk corresponds to the portion of a triangle that overlaps with a given scanline. The chunk contains a start and stop x-coordinate, as well as metadata for each coordinate interpolated from the metadata associated with each triangle vertex. The chunks are stored per scanline using atomic memory operations. In the second phase, the kernel stripmines over the scanlines, and each microthread processes all chunks related to its scanline. The corresponding vector-fetched block has a `for` loop that allows each microthread to iterate across all pixels in its respective chunk. This can cause divergence when chunks are not the same length. Microthreads manage a z-buffer for overlapping triangles. Since microthreads are always working on different scanlines, they never modify the same region of the frame buffer at the same time. This kernel is another good example of irregular DLP. Table 7.4 shows that it requires many microthread load/stores, conditional/unconditional branches, and conditional moves. Note that these instruction counts reflect compiler loop unrolling in the microthread code by a factor of six. Figure 7.11b shows the amount of divergence in this kernel when rasterizing a space shuttle scene with 393 triangles and an image size of 1000×1000 pixels. Over 80% of the fragment micro-ops experience some amount of divergence. This is mostly due to the backward branch implementing the `for` loop in the second phase. Chunk lengths vary widely and the corresponding fragments often diverge as some microthreads continue to work on longer chunks. Mapping this kernel to multiple cores involves giving each core a set of triangles that map to a region of consecutive scanlines. This is straight-forward, since the triangles have been sorted in the previous kernel. Triangles that overlap more than one region will be redundantly processed by multiple cores.

Overall, this study demonstrates that larger applications will exhibit a diverse mix of regular and irregular DLP, and those loops with irregular DLP will experience a range of microthread divergence. This helps motivate our interest in data-parallel accelerators that can efficiently execute many different types of DLP.

Chapter 8

Conclusion

Architects are turning to data-parallel accelerators to improve the performance of emerging data-parallel applications. These accelerators must be flexible enough to handle a diverse range of both regular and irregular data-level parallelism, but they must also be area- and energy-efficient to meet increasingly strict design constraints. Power and energy are particularly important as data-parallel accelerators are now deployed across the computing spectrum, from the smallest handheld to the largest data-center. This thesis has explored a new approach to building data-parallel accelerators that is based on simplifying the instruction set, microarchitecture, and programming methodology for a vector-thread architecture.

8.1 Thesis Summary and Contributions

This thesis began by categorizing regular and irregular data-level parallelism (DLP), before introducing the following five architectural design patterns for data-parallel accelerators: the multiple-instruction multiple-data (MIMD) pattern, the vector single-instruction multiple-data (vector-SIMD) pattern, the subword single-instruction multiple-data (subword-SIMD) pattern, the single-instruction multiple-thread (SIMT) pattern, and the vector-thread (VT) pattern. I provided a qualitative argument for why our recently proposed VT pattern can combine the energy efficiency of SIMD accelerators with the flexibility of MIMD accelerators.

The rest of this thesis introduced several general techniques for building simplified instances of the VT pattern suitable for use in future data-parallel accelerators. We have begun implementing these ideas in the Maven accelerator, which uses a malleable array of vector-thread engines to flexibly and efficiently execute data-parallel applications. These ideas form the key contribution of the thesis. I have outlined them and our experiences using them in the Maven accelerator below.

- **Unified VT Instruction Set Architecture –** This thesis discussed the advantages and disadvantages of unifying the VT control-thread and microthread scalar instruction sets. The hope was that a unified VT instruction set would simplify the microarchitecture and programming

methodology, and we found this to be particularly true with respect to rapidly implementing a high-quality Maven compiler for both types of thread. In addition, a unified VT instruction set offers interesting opportunities for sharing between the control thread and microthread both in terms of implementation (e.g., refactored control logic and control processor embedding) and software (e.g., common functions called from either type of thread). I suggest that future VT accelerators begin with such a unified VT instruction set and only specialize when absolutely necessary.

- **Single-Lane VT Microarchitecture Based on the Vector-SIMD Pattern –** This thesis described a new single-lane VT microarchitecture based on minimal changes to a traditional vector-SIMD implementation. Our experiences building the Maven VT core have anecdotally shown that implementing a single-lane VTU is indeed simpler than a multi-lane VTU, and that basing our design on a vector-SIMD implementation helped crystallize the exact changes that enable VT capabilities. This thesis introduced vector fragments as a way to more flexibly execute regular and irregular DLP in a vector-SIMD based VT microarchitecture. I explained how control processor embedding can mitigate the area overhead of single-lane cores, and I introduced three new techniques that manipulate vector fragments to more efficiently execute irregular DLP. Vector fragment merging can automatically force microthreads to reconverge. Vector fragment interleaving can hide execution latencies (especially the vector-fetched scalar branch resolution latency) by switching between independent fragments. Vector fragment compression can avoid the overhead associated with a fragment's inactive microthreads. I suggest that future VT accelerators take a similar approach, although there is still significant work to be done fully evaluating the proposed vector fragment mechanisms.
- **Explicitly Data-Parallel VT Programming Methodology –** This thesis presented a new VT programming methodology that combines a slightly modified C++ scalar compiler with a carefully written support library. Our experiences writing applications for Maven, particularly the computer graphics case study discussed in this thesis, have convinced us that an explicitly data-parallel framework is a compelling approach for programming future VT accelerators. Our experiences actually implementing the Maven programming methodology have also verified that, while not trivial, this approach is far simpler to develop than a similarly featured automatic vectorizing compiler. There is, of course, still room for improvement. I suggest that the methodology discussed in this thesis be used as a interim solution, possibly being replaced by a modified version of a more standard data-parallel framework such as OpenCL.

To evaluate these ideas we implemented a simplified Maven VT core using a semi-custom ASIC methodology in a TSMC 65 nm process. We have been able extend the Maven instruction set, microarchitecture, and programming methodology to reasonably emulate the MIMD, vector-SIMD, and SIMT design patterns. I evaluated these various core designs in terms of their area, perfor-

mance, and energy. The Maven VT core has similar area as compared to a single-lane vector-SIMD core, and uses control processor embedding to limit the area overhead of including a control processor per lane. To evaluate the performance and energy of a Maven VT core, I used four compiled microbenchmarks that capture some of the key features of regular and irregular DLP. My results illustrated that a Maven VT core is able to achieve similar performance and energy efficiency as a vector-SIMD core on regular DLP. I also explained how for some forms of irregular DLP, a Maven VT core is able to achieve lower energy per task compared to a vector-SIMD core by completely skipping large portions of work. My results for more irregular DLP with many conditional branches were less compelling, but the vector fragment mechanisms presented in this thesis offer one possible way to improve the performance and energy efficiency on such codes. It is important to note, that although the vector-SIMD core performed better than the Maven VT core on the highly irregular DLP, this required careful work writing hand-coded assembly for the vector-SIMD core. Maven's explicitly data-parallel programming methodology made writing software with both regular and irregular DLP relatively straight-forward. Both the VT and vector-SIMD cores perform better than the MIMD and SIMT design, with a few small exceptions. This work is the first detailed quantitative comparison of the VT pattern to other design patterns for data-parallel accelerators. These results provide promising evidence that future data-parallel accelerators based on simplified VT architectures will be able to combine the energy efficiency of vector-SIMD accelerators with the flexibility of MIMD accelerators.

8.2 Future Work

This thesis serves as a good starting point for future work on VT-based data-parallel accelerators. Sections 4.7, 5.8, and 6.6 discussed specific directions for future work with respect to the instruction set, microarchitecture, and programming methodology. More general thoughts on future work are briefly outlined below.

- **Improving Execution of Irregular DLP –** One of the key observations of this thesis is that the vector fragment mechanism alone is not sufficient for efficient execution of highly irregular DLP. This thesis has introduced vector fragment merging, interleaving, and compression as techniques that can potentially improve the performance and energy efficiency on such codes. The next step would be to implement these techniques and measure their impact for our microbenchmarks using our evaluation methodology. In addition, small slices of the computer graphics case study could be used to help further evaluate the effectiveness of these techniques. Eventually, a higher-level and faster model will be required to work with even larger Maven applications.
- **Maven Quad-Core Tile –** This thesis has focused on a single data-parallel core, but there are many interesting design issues with respect to how these cores are integrated together. In

Section 5.6, I briefly described how four Maven VT cores could be combined into a quad-core tile. The next step would be to implement area-normalized tiles for each of the design patterns. This would probably include a MIMD tile with eight cores, a vector-SIMD tile with one four-lane core, a vector-SIMD tile with four single-lane cores, and a VT tile with four single-lane cores. This would also enable more realistic modeling of the intra-tile memory system, without needing a detailed model for the inter-tile memory system. A tile might also make a reasonably sized module for including in a fabricated test chip.

- **Full Maven Data-Parallel Accelerator –** Finally, these quad-core tiles can be integrated into a model of a full data-parallel accelerator. Detailed RTL design of such an accelerator will be difficult, but higher-level models of each tile might enable reasonable research into the accelerator's full memory system and on-chip network.

Bibliography

[ABC+06] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006.

[amd08] Fusion: Industry-Changing Impact of Accelerated Computing. AMD White Paper, 2008. `http://sites.amd.com/us/documents/amd_fusion_whitepaper.pdf`

[arm09] The ARM Cortex-A9 Processors. ARM White Paper, 2009. `http://www.arm.com/pdfs/armcortexa-9processors.pdf`

[Asa96] K. Asanović. Torrent Architecture Manual. Technical report, EECS Department, University of California, Berkeley, Dec 1996.

[Asa98] K. Asanović. *Vector Microprocessors*. Ph.D. Thesis, EECS Department, University of California, Berkeley, 1998.

[ati06] ATI CTM Guide. AMD/ATI Techincal Reference Manual, 2006. `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`

[ati09] R700-Family Instruction Set Architecture. AMD/ATI Techincal Reference Manual, 2009. `http://developer.amd.com/gpu_assets/R700-family_instruction_set_architecture.pdf`

[BAA08] C. Batten, H. Aoki, and K. Asanović. The Case for Malleable Stream Architectures. *Workshop on Streaming Systems (WSS)*, Nov 2008.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2008.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, Aug 2004.

[BGS94] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, Dec 1994.

[BKGA04] C. Batten, R. Krashinsky, S. Gerding, and K. Asanović. Cache Refill/Access Decoupling for Vector Machines. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2004.

[Ble96] G. E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, Mar 1996.

[CC06] B. H. Calhoun and A. Chandrakasan. Ultra-Dynamic Voltage Scaling Using Sub-Threshold Operation and Local Voltage Dithering. *IEEE Journal of Solid-State Circuits (JSSC)*, 41(1):238–245, Jan 2006.

[CCE+09] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, Mar/Apr 2009.

[CCL+98] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, , and W. D. Weathersby. The Case for High-Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, Jul 1998.

[cle06] CSX Processor Architecture. ClearSpeed White Paper, 2006. `http://www.clearspeed.com/docs/resources/ClearSpeed_CSX_White_Paper.pdf`

[DL95] D. DeVries and C. G. Lee. A Vectorizing SUIF Compiler. *SUIF Compiler Workshop*, Jan 1995.

[dLJ06] P. de Langen and B. Juurlink. Leakage-Aware Multiprocessor Scheduling for Low Power. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr 2006.

[DM06] J. Donald and M. Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2006.

[DPT03] A. Duller, G. Panesar, and D. Towner. Parallel Processing - The PicoChip Way! *Communicating Process Architectures*, pages 125–138, Sep 2003.

[DSC+07] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An Integrated Quad-Core Opteron Processor. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2007.

[DVWW05] T. Dunigan, J. Vetter, J. White, and P. Worley. Performance Evaluation of the Cray X1 Distributed Shared-Memory Architecture. *IEEE Micro*, 25(1):30–40, Jan/Feb 2005.

[EAE+02] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2002.

[Eat05] W. Eatherton. Keynote Address: The Push of Network Processing to the Top of the Pyramid. *Symp. on Architectures for Networking and Communications Systems (ANCS)*, Oct 2005.

[ERPR95] J. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, 15:33–43, Apr 1995.

[EV96] R. Espasa and M. Valero. Decoupled Vector Architectures. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 1996.

[EV97] R. Espasa and M. Valero. Multithreaded Vector Architectures. *Int'l Symp. on Supercomputing (ICS)*, Jul 1997.

[EVS97] R. Espasa, M. Valero, and J. E. Smith. Out-of-Order Vector Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 1997.

[EVS98] R. Espasa, M. Valero, and J. E. Smith. Vector Architectures: Past, Present, and Future. *Int'l Symp. on Supercomputing (ICS)*, Jul 1998.

[FS00] A. Forestier and M. R. Stan. Limits to Voltage Scaling from the Low-Power Perspective. *Symp. on Integrated Circuits and Systems Design (ICSD)*, Sep 2000.

[FSYA09] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):1–35, Jun 2009.

[GC97] V. Gutnik and A. P. Chandrakasan. Embedded Power Supply for Low-Power DSP. *IEEE Trans. on Very Large scale Integration Systems (TVLSI)*, 5(4):425–435, Dec 1997.

[GGH97] R. Gonzalez, B. M. Gordon, and M. A. Horowitz. Supply and Threshold Voltage Scaling for Low-Power CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 32(8):1210–1216, Aug 1997.

[GHF+06] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, Mar 2006.

[gnu10] GCC, The GNU Compiler Collection. Online Webpage, 2010 (accessed January 10, 2010). `http://gcc.gnu.org`

[GTA06] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2006.

[HA08] M. Hampton and K. Asanović. Compiling for Vector-Thread Architectures. *Int'l Symp. on Code Generation and Optimization (CGO)*, Apr 2008.

[HBK06] J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. Intel White Paper, 2006. `http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`

[HP07] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[HSU+01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 5(1), Feb 2001.

[int04] Intel IXP2800 Network Processor. Intel Product Brief, 2004. `http://download.intel.com/design/network/prodbrf/27905403.pdf`

[int09] Intel Single-chip Cloud Computer. Online Webpage, 2009 (accessed Dec 4, 2009). `http://techresearch.intel.com/articles/tera-scale/1826.htm`

[JBW89] N. P. Jouppi, J. Bertoni, and D. W. Wall. A Unified Vector/Scalar Floating-Point Architecture. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 1989.

[Jes01] C. Jesshope. Implementing an Efficient Vector Instruction Set in a Chip Multiprocessor Using Micro-Threaded Pipelines. *Australia Computer Science Communications*, 23(4):80–88, 2001.

[Kal09] R. Kalla. POWER7: IBM's Next Generation POWER Microprocessor. *Symp. on High Performance Chips (Hot Chips)*, Aug 2009.

[Kan87] G. Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[KAO05] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25:21–29, Mar/Apr 2005.

[KBA08] R. Krashinsky, C. Batten, and K. Asanović. Implementing the Scale Vector-Thread Processor. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 13(3), Jul 2008.

[KBH+04a] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[KBH+04b] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *IEEE Micro*, 24(6):84–90, Nov/Dec 2004.

[KJJ+09] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2009.

[KJL+09] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, , and S. J. Patel. A Task-Centric Memory Model for Scalable Accelerator Architectures. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2009.

[Koz02] C. Kozyrakis. *Overcoming the Limitations of Conventional Vector Processors*. Ph.D. Thesis, EECS Department, University of California, Berkeley, 2002.

[KP03] C. Kozyrakis and D. Patterson. Scalable Vector Processors for Embedded Systems. *IEEE Micro*, 23(6):36–45, Nov 2003.

[Kre05] K. Krewell. A New MIPS Powerhouse Arrives. *Microprocessor Report*, May 2005.

[KWL+08] B. K. Khailany, T. Williams, J. Lin, M. Rygh, D. W. Tovey, and W. J. Dally. A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing. *IEEE Journal of Solid-State Circuits (JSSC)*, 43(1):202–213, Jan 2008.

[LK09] J. Lee and N. S. Kim. Optimizing Throughput of Power- and Thermal-Constrained Multicore Processors using DVFS and Per-Core Power-Gating. *Design Automation Conference (DAC)*, Jul 2009.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computer Architecture. *IEEE Micro*, 28(2):39–55, Mar/Apr 2008.

[LSFJ06] C. Lemuet, J. Sampson, J. Francios, and N. Jouppi. The Potential Energy Efficiency of Vector Acceleration. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, Nov 2006.

[Mar09] T. Maruyama. SPARC64 VIIIfx: Fujitsu's New Generation Octo-Core Processor for PETA Scale Computing. *Symp. on High Performance Chips (Hot Chips)*, Aug 2009.

[MCEF94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering, Jul 1994.

[MHM+95] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 1995.

[MIA+07] P. McCormicka, J. Inmana, J. Ahrensa, J. Mohd-Yusofa, G. Rothb, and S. Cumminsa. Scout: A Data-Parallel Programming Language for Graphics Processors. *Parallel Computing*, 33(10–11):648–662, Mar 2007.

[Mic09] Graphics Guide for Windows 7: A Guide for Hardware and System Manufacturers. Microsoft White Paper, 2009.
`http://www.microsoft.com/whdc/device/display/graphicsguidewin7.mspx`

[mip09] MIPS32 Architecture for Programmers, Volume II: The MIPS32 Instruction Set. MIPS Technical Reference Manual, 2009.
`http://www.mips.com/products/architectures/mips32`

[MJCP08] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. Tradeoffs in Designing Accelerator Architectures for Visual Computing. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 2008.

[MSM05] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[NBGS08] J. Nickolls, I. Buck, M. Garland, and K. Skadron. OpenMP Application Program Interface. *ACM Queue*, 6(2):40–53, Mar/Apr 2008.

[NHW+07] U. G. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, and A. Kumar. An 8-Core 64-Thread 64 b Power-Efficient SPARC SoC. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2007.

[nvi09] NVIDIA's Next Gen CUDA Compute Architecture: Fermi. NVIDIA White Paper, 2009. `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`

[OH05] K. Olukotun and L. Hammond. The Future of Microprocessors. *ACM Queue*, 3(7):26–29, Sep 2005.

[ope08a] The OpenCL Specification. Khronos OpenCL Working Group, 2008. `http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf`

[ope08b] OpenMP Application Program Interface. OpenMP Architecture Review Board, 2008. `http://www.openmp.org/mp-documents/spec30.pdf`

[Oya99] Y. Oyanagi. Development of Supercomputers in Japan: Hardware and Software. *Parallel Computing*, 25(13–14):1545–1567, Dec 1999.

[QCEV99] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a Vector Unit to a Superscalar Processor. *Int'l Symp. on Supercomputing (ICS)*, Jun 1999.

[RDK+98] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. *Int'l Symp. on Microarchitecture (MICRO)*, Nov 1998.

[Rei07] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[RSOK06] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector Lane Threading. *Int'l Conf. on Parallel Processing (ICPP)*, Aug 2006.

[RTM+09] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Kottapalli. A 45 nm 8-Core Enterprise Xeon Processor. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2009.

[Rus78] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[SB91] J. M. Sipelstein and G. E. Blelloch. Collection-Oriented Languages. *Proc. of the IEEE*, 79(4):504–523, Apr 1991.

[SCS+09] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *IEEE Micro*, 29(1):10–21, Jan/Feb 2009.

[SFS00] J. E. Smith, G. Faanes, and R. Sugumar. Vector Instruction Set Support for Conditional Operations. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2000.

[SKMB03] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger. Universal Mechanisms for Data-Parallel Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2003.

[SNL+03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2003.

[SS00] N. Slingerland and A. J. Smith. Multimedia Instruction Sets for General Purpose Microprocessors: A Survey. Technical report, EECS Department, University of California, Berkeley, Dec 2000.

[Swe07] D. Sweetman. *See MIPS Run Linux*. Morgan Kaufmann, 2007.

[TKM+03] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, , and A. Agarwal. A 16-Issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2003.

[TLM+04] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[URv03] T. Ungerer, B. Robič, and J. Šilc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, Mar 2003.

[VHR+07] S. Vangali, J. Howard, G. Ruhi, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyerl, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskotel, and N. Borkarl. 80-Tile 1.28 TFlops Network-on-Chip in 65 nm CMOS. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2007.

[WAK+96] J. Wawrzynek, K. Asanović, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan. Spert-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, Mar 1996.

[Wil08] S. Williams. *Auto-tuning Performance on Multicore Computers*. Ph.D. Thesis, EECS Department, University of California, Berkeley, 2008.

[WL08] D. H. Woo and H.-H. S. Lee. Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. *IEEE Computer*, 41(12):24–31, Dec 2008.

[YBC+06] V. Yalala, D. Brasili, D. Carlson, A. Hughes, A. Jain, T. Kiszely, K. Kodandapani, A. Varadharajan, and T. Xanthopoulos. A 16-Core RISC Microprocessor with Network Extensions. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2006.

[ZB91] M. Zagha and G. E. Blelloch. Radix Sort for Vector Multiprocessors. *Int'l Conf. on High Performance Networking and Computing (Supercomputing)*, 1991.