Broadening the Applicability of FPGA-based Soft Vector Processors

by

Aaron Severance

Bachelor of Science in Computer Engineering, Boston University, 2004

Master of Science in Electrical Engineering, University of Washington, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

March 2015

Abstract

A soft vector processor (SVP) is an overlay on top of FPGAs that allows data-parallel algorithms to be written in software rather than hardware, and yet still achieve hardware-like performance. This ease of use comes at an area and speed penalty, however. Also, since the internal design of SVPs are based largely on custom CMOS vector processors, there is additional opportunity for FPGA-specific optimizations and enhancements.

This thesis investigates and measures the effects of FPGA-specific changes to SVPs that improve performance, reduce area, and improve ease-of-use; thereby expanding their useful range of applications. First, we address applications needing only moderate performance such as audio filtering where SVPs need only a small number (one to four) of parallel ALUs. We make implementation and ISA design decisions around the goals of producing a compact SVP that effectively utilizes existing BRAMs and DSP Blocks. The resulting VENICE SVP has 2$\times$ better performance per logic block than previous designs.

Next, we address performance issues with algorithms where some vector elements 'exit early' while others need additional processing. Simple vector predication causes all elements to exhibit 'worst case' performance. Density time masking (DTM) improves performance of such algorithms by skipping the completed elements when possible, but traditional implementations of DTM are coarse-grained and do not map well to the FPGA fabric. We introduce a BRAM-based implementation that achieves 3.2$\times$ higher performance over the base SVP with less than 5% area overhead.

Finally, we identify a way to exploit the raw performance of the underlying FPGA fabric by attaching wide, deeply pipelined computational units to SVPs through a custom instruction interface. We support multiple inputs and outputs, arbitrary-length pipelines, and heterogeneous lanes to allow streaming of data through large operator graphs. As an example, on an n-body simulation problem, we show that custom instructions achieve 120$\times$ better performance per area than the base SVP.

Preface

The following publications have been adapted for inclusion in this dissertation:

- **VENICE: A Compact Vector Processor for FPGA Applications** [57]

  Published in the 2012 International Conference on Field-Programmable Technology (FPT 2012). Authored by Aaron Severance and Guy Lemieux. Appears in Chapter 3.

  I performed all of the design and implementation and did the primary benchmarking and analysis presented in this paper. Guy Lemieux served in an advisory fashion. A separate work [46] detailing a compiler for VENICE was published earlier; the included paper is a distinct contribution detailing the design and architecture of the VENICE SVP.

- **Wavefront Skipping using BRAMs for Conditional Algorithms on Vector Processors** [61]

  Published in the 2015 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA 2015). Authored by Aaron Severance, Joe Edwards, and Guy G.F. Lemieux. Appears in Chapter 4.

  I performed the design, implementation, and benchmarking in this paper with the following exceptions:

  - Joe Edwards developed the scalar and basic vector (without density-time masking) code for Viola-Jones face detection and FAST9 feature detection.
  - Guy Lemieux developed the scalar and basic vector (without density-time masking) code for Mandelbrot set generation.

Guy Lemieux also served in an advisory fashion.

- **Soft Vector Processors with Streaming Pipelines** [60]

  Published in the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA 2014). Authored by Aaron Severance, Joe Edwards, Hossein Omidian, Guy Lemieux. Appears in Chapter 5.

  I performed the design, implementation, and benchmarking done in this paper with the following exceptions:

  - Guy Lemieux and Joe Edwards developed the scalar and basic vector (without custom instructions) code for the N-Body problem.
  - Hossein Omidian designed and implemented the high-level synthesis approach for creating custom instructions. His contribution to the paper was removed from the body of this dissertation.

  Guy Lemieux also served in an advisory fashion.

Table of Contents

List of Tables

List of Figures

Glossary

The following acronyms are used in this dissertation:

ALM	adaptive logic module
ALU	arithmetic logic unit
AoS	array of structures
API	application programming interface
ASIC	application specific integrated circuit
BRAM	block RAM
CAD	computer aided design
CAM	content addressable memory
CGRA	coarse-grained reconfigurable array
CMOS	complementary metal-oxide semiconductor
CMOV	conditional move
CPU	central processing unit
CVI	custom vector instruction
DCT	discrete cosine transform
DMA	direct memory access

DLP	data level parallelism
DSP	digitial signal processing
DFG	data flow graph
DRAM	dynamic random-access memory
DSPBA	Altera's DSP Builder Advanced Blockset for Simulink
DTM	density-time masking
eALM	equivalent ALM
ESL	electronic system level
FF	flip-flop
FFT	fast Fourier transform
FIFO	first-in first-out
FIR	finite impulse response
f_{max}	maximum operating frequency
FPGA	field-programmable gate array
FU	functional unit
GDB	the GNU debugger
GUI	graphical user interface
GPU	graphics processing unit
HDL	hardware description language
HLS	high level synthesis
IDE	integrated development environment
IC	integrated circuit

ILP	instruction level parallelism
IP	intellectual property
ISA	instruction set architecture
LAB	logic array block
LUT	look-up table
MAC	multiply accumulate
MMVL	maximum masked vector length
MSB	most significant bit
MVL	maximum vector length
MXP	A SVP from VectorBlox Computing, Inc. used in this dissertation
NOP	no operation
OoOE	out-of-order execution
PC	program counter
PE	processing element
PVFB	pending vector fragment buffer
RTL	register transfer level
SAD	sum of absolute differences
SIMD	single-instruction multiple-data
SIMT	single-instruction multiple-thread
SoA	structure of arrays
SoC	system on chip
SRAM	static RAM

SVP	soft vector processor
VARF	vector address register file
VEGAS	A second generation SVP from The University of British Columbia
VENICE	A new SVP that is part of this dissertation
VESPA	A first generation SVP from University of Toronto
VIRAM	The VIRAM project; a hard vector processor from Berkeley
VIPERS	A first generation SVP from The University of British Columbia
VL	vector length
VLIW	very-long instruction word
VRF	vector register file

Acknowledgments

Papa, thanks for the support (moral and otherwise) and advice. Mama, thanks for helping me through the tough times. Sister, thanks for being a tiny potato. Sean, you are a great sounding board and full of useful advice. Thanks for helping me find reasons to try.

Thanks Guy; you were the impetus for this whole thing and were there the whole way for me. Steve, thanks for taking over; you're a nice guy to a fault. Joe, you showed me it was good to be excited (and your work on the benchmarks was great). Hossein, thanks for your small part in this; you'll go on to great things I'm sure. Alex, Ameer, Chris, Chris, Dave, Doug, Keith, Mike, Tom, Usman, and Zhiduo, it was great working with you.

Thanks to NSERC, MITACS, and VectorBlox Computing for funding, and to Altera for donating hardware and software licenses.

To whom it may concern.

Chapter 1

Introduction

What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from. — T. S. Eliot

1.1 Motivation

A field-programmable gate array (FPGA) is a configurable logic device that can be programmed at bit-level granularity. Instead of running a sequential or parallel software program like a central processing unit (CPU) or graphics processing unit (GPU), it is programmed with a bitstream that tells its logic and interconnect how to behave. This extremely fine granularity allows FPGAs to expose massive amounts of parallelism to the designer and provide better performance and efficiency than CPUs and GPUs on certain applications [13]. At the same time, this flexibility also makes it difficult for applications programmers to implement algorithms on FPGAs.

Since FPGAs can be used to emulate digital circuits, they are commonly programmed the same way: using a register transfer level (RTL) description. RTL is a textual description of a circuit in which behavioral and structural statements describe elements which execute in parallel and communicate through signals. For programmers familiar with the sequential computing model used for software (on top of which parallelism can be added, but at a much coarser granularity) RTL design can be difficult to learn and understand. Even for programmers accustomed

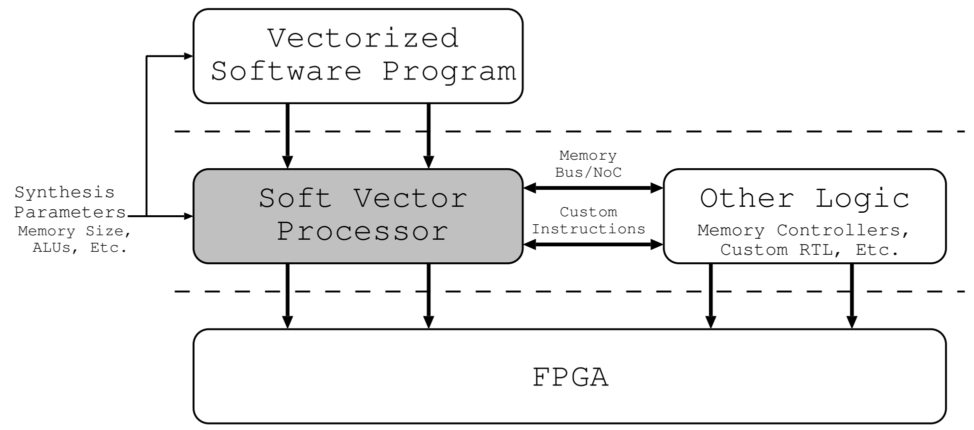


Figure 1.1: An FPGA System with a Soft Processor

to writing RTL, the design process is time-consuming. An RTL design will take from several minutes to many hours to synthesize for an FPGA depending on its size and complexity, versus seconds to compile software. Additionally debug is more difficult for RTL designs; there is limited visibility when they are implemented on an FPGA, and simulations run at several orders of magnitude slower than the actual design. An additional complexity is that modern FPGAs are not homogenous; in addition to configurable logic there are hardened memories (block RAMs (BRAMs)) and arithmetic units (digitial signal processing (DSP) blocks) that the programmer must utilize to take full advantage of the FPGA.

A goal for researchers and industry has thus become to achieve close to software levels of productivity for FPGA design. There are many approaches to this, with some trying to improve productivity by improving synthesis speed [47, 69] or debug visibility [26], some trying to make RTL more software-like [7, 51], and some using a software program as input to generate RTL [11, 19]. This work concentrates on soft processors: processor-style intellectual property (IP) blocks instantiated on FPGAs that offer a more software-like design flow.

Figure 1.1 shows a conceptual view of a soft processor. It has three levels: the FPGA hardware, the soft processor (which is implemented on the FPGA hardware), and the user program (which runs on the soft processor). Soft processors bring the afforementioned benefits of software programmability and debugability

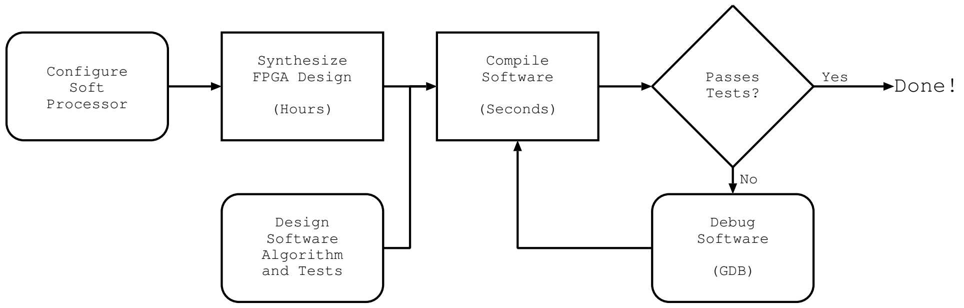

Configure Soft Processor
Synthesize FPGA Design (Hours)
Compile Software (Seconds)
Passes Tests?
Yes
Done!
No
Design Software Algorithm and Tests
Debug Software (GDB)

Figure 1.2: Design Flow using a Soft Processor

to FPGAs. In Figure 1.2 the soft processor design flow is shown. First, the designer has to instantiate a soft processor into their FPGA design and synthesize the design (taking minutes to hours). Next, changes to the algorithm running on the soft processor can happen in seconds; the designer simply has to recompile the software and update the program memory. Debugging can proceed with tools familiar to software programmers such as the GNU debugger (GDB). By contrast, in an RTL design each time the design is changed FPGA resynthesis must occur. This means the designer must wait hours to see the results of their changes. Additionally, if there is not enough information visible to diagnose a bug, inserting additional debug capability requires another run of FPGA synthesis.

Another benefit of implementing a design using a soft processor is the ability to easily reuse the same hardware for multiple functions or applications. To reuse FPGA hardware that is not programmable (such as fixed RTL IP blocks) on the same hardware, some FPGAs provide the ability to reconfigure parts of the device at runtime. However, even the fastest partial reconfiguration times are on the order of a millisecond [54]. By contrast, a soft processor need only switch instruction streams, which can be done orders of magnitude faster.

Soft processors are already familiar to FPGA programmers; each major FPGA vendor offers a soft CPU [4, 43, 72]. Soft CPUs are useful for controlling datapath logic and interfacing with low speed I/O interfaces. However, emulating a scalar CPU on an FPGA has certain challenges [71]; techniques that provide traditional CPUs with high performance (e.g., superscalar, out-of-order execution) are expensive to implement in FPGA fabric. Because of this, different approaches to organizing processors (or extremely simple processors known as processing elements (PEs)) have been proposed for FPGAs. Multithreaded [42] and very-long instruction word (VLIW) [52] processors can provide additional performance, and multiple PEs can be put together into 2D-grids as coarse-grained reconfigurable arrays (CGRAs) [12]. In contrast, this dissertation is concerned with another arrangement of PEs into a 1D single-instruction multiple-data (SIMD) array with vector processor-style control, the soft vector processor (SVP).

1.2 Soft Vector Processors

SVPs draw from both SIMD processors and traditional vector processing models. SIMD processors execute the same instruction on multiple data elements at the same time; for example, Intel's latest SIMD extention to the x86 instruction set architecture (ISA) can perform an operation on sixteen single-precision or eight double-precision floating-point operands with one instruction [27]. Multiple copies of each functional unit (FU) are needed to process the data in parallel, and if the amount of data to be processed is larger than the number of FUs the processing must be broken up into multiple instructions (a process known as strip mining [70]). Traditional vector processors also operate on multiple data elements with a single instruction, using a single FU and streaming the data through sequentially over multiple clock cycles. The number of elements is configurable through a vector length (VL) register; strip mining is not needed unless data is larger than some maximum vector length (MVL) threshold.

Vector-SIMD hybrids are an extension of traditional vector processors to use multiple parallel FUs to speed up computation. The vector-SIMD paradigm has been shown to map well to embedded media processing applications [37]. Existing vector-SIMD hybrids were the basis for initial SVP implementations [74, 78]. While these initial implementations were closely modeled on 'hard' vector processors, they had the benefit of being 'soft'; i.e., programmer-configurable. The programmer has the option of setting multiple parameters when instantiating an SVP such as the number of parallel arithmetic logic units (ALUs) and the size of the local memory/register file. This configurability allows the programmer to customize the processor for their application, only using the resources required to meet their performance target.

1.3 SVP Weaknesses

However, SVPs face many challenges before being viable as a mainstream option for implementing algorithms on FPGAs. First, there is a large penalty in speed and/or area compared to an RTL design. This penalty can vary greatly depending on the application; a comparison of an RTL implementation of a motion estimation kernel (used in video encoding) ran 21× faster than a SVP implementation given similar resource usage [78], while a study across several embedded benchmarks found that an SVP with 16 FUs was 17× slower than hardware but used 64× more area than a hardware implementation [77]. Reducing this penalty is essential if SVPs are to be a viable choice for many applications. To do this, the design of SVPs must be rethought with FPGAs in mind; some initial work has been done in this area [15] but there are still many performance and area optimizations that can be made. FPGAs have a very different cost model than application specific integrated circuits (ASICs) for memories and multipliers due to the hardened on-chip BRAMs and DSP blocks. SVPs should be designed such that they can make as much use of these hardened blocks as possible.

Also, the performance penalty can be reduced by better targetting likely user applications. Targetting large designs that can fill up multi-thousand dollar FPGAs makes for impressive benchmark numbers but is an unlikely use-case for SVPs. Rather, targetting smaller applications where the goal is to get 'good-enough' performance quickly is more realistic. An example is accelerating the audio decoding portion of video playback; the video decoding core(s) may be need preconfigured IP or custom RTL to achieve the necessary performance. However, an SVP can easily support a multitude of audio codecs, switching between them at runtime in microseconds. Though the area of the SVP may be larger than a single audio codec designed in RTL, one SVP replaces several codecs and will not be large compared to the video codec(s).

Another challenge is expanding the number of different types of applications that SVPs can handle. An algorithm with a simple data flow graph (DFG) can already be implemented via RTL or other methods; an SVP should allow a programmer to implement algorithms with that would be difficult to implement otherwise. One example is algorithms with data-dependent behavior. Current SVPs allow a limited form of data-dependent behavior. Data dependent branches can be skipped only if every element on the vector has not taken that branch. By allowing skipping within vectors, rarely taken branches can be accelerated. This is important for algorithms that do most of their useful computation on only a small subset of their data.

Finally, to truly take advantage of the FPGA, the programmer needs to be able to interface with external logic. Current SVPs only share data through main memory, which has limited bandwidth and is cumbersome to synchronize. Current FPGA programmers may therefore be wary of an SVP implementation; if it is unable to meet their performance goals there is no graceful way to implement logic to increase performance, so the SVP design may have to be abandoned completely. Giving users a tightly coupled interface to external logic allows for a smoother transition between software and RTL.

1.4 Goals

The goal of this dissertation is to broaden the overall capabilities of SVPs by making them more area-efficient, achieve higher performance on many applications, remain general enough for general computations, and provide a low-level interface which allows advanced users a way to access the underlying FPGA fabric. By harnessing these gains in efficiency, it will become more feasible to implement applications in the SVP framework instead of the conventional approach of using custom RTL. At the same time, we wish to ensure the SVP becomes easier to use from a programmer's perspective. The overall improved ease-of-use should make SVP a much more compelling framework for solving the types of processing problems encountered by FPGA users.

1.5 Approach

Our approach is threefold. First, we address the area/performance penalty of SVPs by developing VENICE, an SVP specifically targetted for moderate performance applications. Scalar soft processors are not adequate for such applications, but a small SVP (with one to four lanes) can meet the desired peformance levels. We provide a direct comparison of VENICE to previous SVPs to show the impact of these design choices.

Second, we investigate accelerating data-dependent algorithms as a way to further differentiate SVPs from RTL. We investigate density-time masking (DTM) [63] (also known as wavefront skipping), a technique from traditional vector processors, but implemented in an FPGA-specific way. This allows us to have much more efficient FU usage, which we demonstrate on applications including Viola-Jones face detection [68].

Third, we add a new interface for sharing data between the SVP and external logic. Our custom vector instructions (CVIs) have an external channel for data to flow to and from logic the programmer has created using either RTL or visually with Altera's DSP Builder Advanced Blockset for Simulink (DSPBA). By implementing an N-body simulation application using these CVIs, we show how the main kernel of an application can be greatly accelerated with a small amount of RTL programming. The programmer does not have to manage data movement and synchronization, and non-critical pre-processing/post-processing can be done using normal SVP instructions.

1.6 Contributions

The contributions of this dissertation are:

1. A demonstration of the benefits of designing SVPs for performance density rather than scalability and emulation of traditional vector processors. The VENICE SVP is able to achieve 2× better performance per logic block than previous SVPs. The main contributions that combine to achieve this are:

   (a) The vector address register file was removed to save area; addresses are stored in scalar registers and transferred with each instruction.

   (b) 2D and 3D vector addressing modes were added to increase performance on code that contains loops with fixed increments.

   (c) Alignment networks were added in the pipeline, removing the need for separate vector align/rotate instructions. This is especially important for sliding window/stencil filter algorithms.

Several other minor contributions were made and will be discussed within the body of this dissertation.

2. A demonstration of a method for run-time skipping of masked-off vector elements in SVPs. A novel implementation called wavefront skipping takes advantage of FPGA BRAMs. On early exit algorithms, such as Viola-Jones face detection, this gives up to an 3× increase in performance with a maximum of 5% area overhead. Additionally, partitioned wavefront skipping showed additional performance benefits for an extra cost in area.

3. A demonstration of a method for adding streaming pipelines to SVPs as custom vector instructions (CVIs). An end user can create a CVI with minimal effort compared to a full hardware algorithm implementation, and achieve performance similar to custom RTL while still having software programmable control. Specific contributions are:

   (a) Expansion of the custom instruction to variable width vector units.

   (b) Allowing an arbitrary number of CVI lanes to be used (from one to the number of SVP lanes) without adding additional multiplexing or buffering. This allows the user to more finely tune the area/performance tradeoff of using CVIs.

   (c) Creation of a method for time multiplexing inputs and outputs for complex CVIs. Instead of simple two-input, one-output instructions, complex CVIs can have 2*N inputs and N outputs where N ranges from 1 to the number of CVI lanes. This allows instructions to replace large data flow graphs (DFGs) that would otherwise take multiple instructions to be implemented.

   (d) Demonstration of alternate design menthods that do not require RTL design skills. Users can design a CVI within MATLAB's Simulink graphical programming environment using DSPBA.

1.7 Dissertation Organization

Chapter 2 presents the background of SVPs, including traditional vector processors. Chapter 3 details the design of VENICE, our SVP targetting narrow, FPGA-centric design. Chapter 4 details our implementation of wavefront skipping for conditionally divergent data-parallel algorithms. Chapter 5 details our CVI implementation method and results. Finally, Chapter 6 provides some conclusions about what has been done and what has yet to be done.

Chapter 2

Background

Those who cannot remember the past are condemned to repeat it.
— George Santayana

This chapter provides the necessary background information for this dissertation. First, it presents an overview of FPGAs architecture and design flow to introduce the problem. Next, it provides is a discussion of overlays on FPGAs and why they are a useful tool for designers. To give the necessary background to understand soft vector processors (SVPs), vector processing and SIMD execution are then introduced. This leads to a discussion of existing SVPs. Finally, some more specific detail into alternatives to and predecessors of the contributions of this dissertation is given.

2.1 FPGAs

FPGAs are integrated circuits (ICs) that are used to implement digital logic functions. The difference between an FPGA and an application specific integrated circuit (ASIC) is that the FPGA is *field-programmable*; that is, the logic functions must be programmed after the device has been manufactured. An ASIC implements digital logic functions via the placement and connections of transistors, which cannot change once the chip has been fabricated. An FPGA implements digital logic functions with configurable basic logic elements (typically look-up tables (LUTs)) and configurable interconnection between them. Most modern

FPGAs are programmed by loading data into static RAM (SRAM) cells, which can be reconfigured practically an unlimited number of times [41]. These SRAM cells, as well as the logic elements and routing are created with normal IC technology, so it is reasonable to consider an FPGA a type of ASIC that can emulate other ASICs. FPGAs have traditionally found use in applications such as ASIC prototyping, telecommunications equipment (where low volumes and changing standards make ASICs less attractive), and as interfaces between other ICs ("glue-logic"). The cost of programmability is that an FPGA implementation of a circuit performs approximately $4\times$ slower than an ASIC implementation, and requires $14\times$ as much dynamic power and $35\times$ as much area [40].

A note on terminology: in relation to FPGAs the adjective *soft* refers to a logic function or digital circuit implemented by configuring the FPGA, while *hard* refers to something implemented at the transistor level. For instance, an FPGA configured with a CPU circuit would be said to have a soft CPU; implementing the same CPU circuit on an ASIC (*hardening* it) would make it a hard CPU.

2.1.1 Architecture and Design Flow

A simple model of an FPGA is a two-dimensional grid of blocks connected by a mesh routing network. The blocks may consist of programmable logic or *hard blocks*. The hard blocks are functions that are frequently used and would be much more expensive to implement in soft logic than as hardened structures. The most common of these are memories (block RAMs (BRAMs)) and multipliers or other expensive arithmetic functions (digitial signal processing (DSP) blocks). Figure 2.1 shows an example architecture with columns of logic blocks, memory blocks, and DSP blocks. Signals between blocks are carried by a configurable routing network; a common configuration is to have horizontal and vertical routing channels with programmable switch blocks where they meet. Additionally, the wire within a routing channel to which a block input or output connects is configurable. The input/output blocks with pins that connect the FPGA to the outside world are also configurable, supporting multiple voltage levels and signalling styles.

The architecture is therefore relatively generic; user logic can be implemented

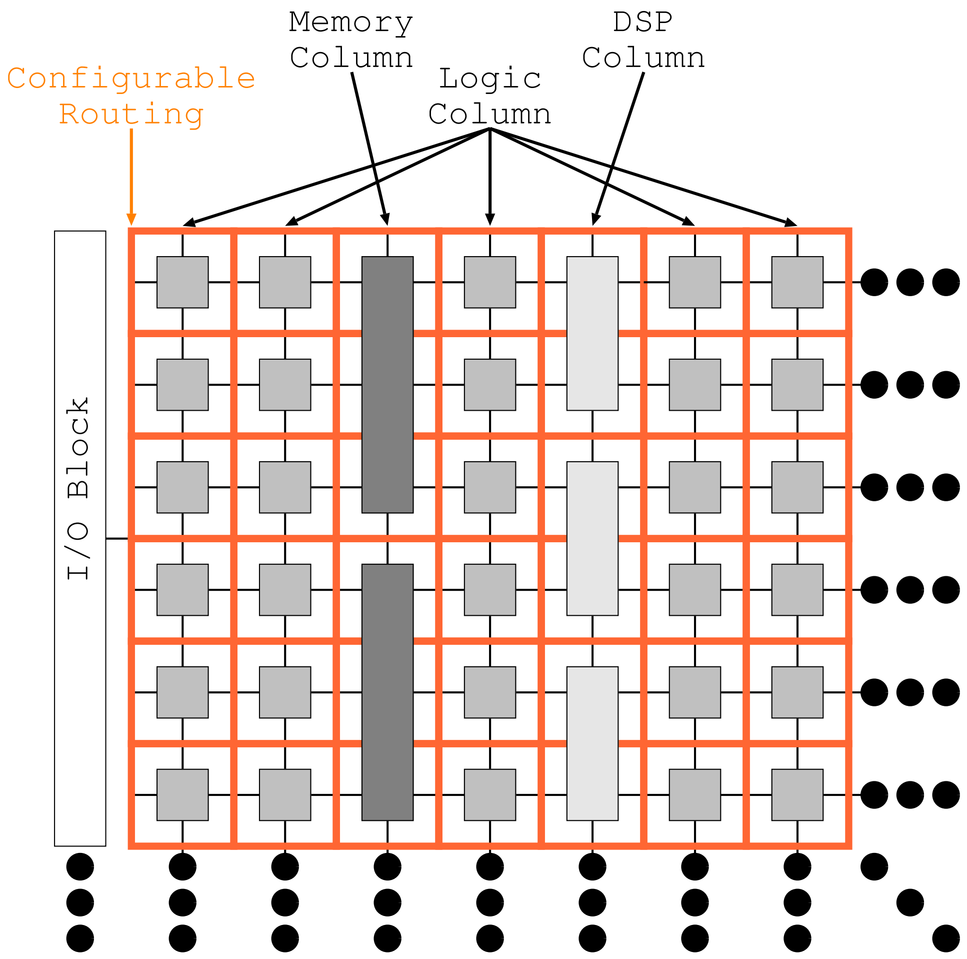


Figure 2.1: Example FPGA Architecture

in any programmable logic block and if hardened blocks are specified there are multiple locations they could be placed. It is not a straightforward problem to automatically map a large design to a large FPGAs; to appreciate the problem size consider that the largest Altera Stratix V FPGA has 359,200 adaptive logic modules (ALMs) (its basic programmble logic element), 2,640 BRAMs, and 352 DSP blocks [5].

The computer aided design (CAD) flow for translating a design to an FPGA configuration bitstream varies for different vendors' CAD tools; a survey can be found in [14]. The differences are not important for the purposes of this dissertation, but it is necessary to understand a CAD flow to understand the reasons the traditional FPGA design cycle is long compared to the design cycle for software. Basic CAD steps include RTL synthesis, technology mapping, placement, routing, and assembly. RTL synthesis is the process of translating the input circuit as specified by the user to a netlist of boolean functions and macros such as hard blocks. The boolean functions get technology-mapped to FPGA programmable logic blocks. Placement then selects locations for each of these units, and routing determines how to configure the communication network to connect logic block inputs and outputs. Finally, assembly creates the bitstream that is used to program the FPGA. Note the term synthesis is often used to refer to the entire processes; saying it takes hours to synthesize a design includes the time taken by all of the steps.

Each of these steps is time consuming, but placement is particularly so. For instance, the Stratix V FPGA mentioned has 359,200 ALMs, which are grouped into logic array blocks (LABs) of 10 ALMs before placement. Any LAB that can be placed can therefore go into one of 35,920 locations. LABs that communicate with each other should be placed close together, as longer distance between them results in longer circuit delays (reducing the maximum operating frequency (f_{max}) of the circuit) and higher routing resource usage. For an arbitrary circuit, finding the optimal placement (according to a metric such as f_{max}) is not computationally feasible. In practice, methods such as simulated annealing [35] are used to try to find a high quality placement.

Simulated annealing is a stochastic method which works by swapping the placement of LABs if they meet some threshold of a cost function. Initially the threshold is set to allow swaps even if the cost function worsens somewhat; as the algorithm progresses the threshold is slowly lowered until eventually only swaps that improve the cost function are allowed. This slow convergence prevents the placement from being stuck in local optima (as a greedy algorithm would be) but requires a huge number of potential swaps. In turn this means that placement on large FPGAs can run for many hours. There are techniques to reduce the CAD time, such as incremental compilation [3] and parallel placement [47, 69], but they still do not provide software-like design cycles.

2.2 Overlays

FPGA overlays are an intermediate layer between the programmable FPGA logic and the programmer's input, designed to simplify design and improve productivity. The overlay itself is a premade IP block and may be distributed as configurable RTL or a pre-synthesized netlist. What separates overlays from other IP blocks is that an overlay provides an additional level of programmability; the overlay can be configured for different applications without having to resynthesize the IP block. Instead, the overlay is programmed by changing some configuration state, typically the contents of one or more BRAMs. An overlay can be added to a design with external logic via some communication channels, or in special cases it may be the majority or entirety of the design. These special cases include designs with significant time-to-market constraints and those for which only low volumes are needed; given enough time and volume it would make more sense to make a design that was only an overlay into an ASIC.

Overlays improve user productivity by allowing programming and debug at a higher level, shortening design cycle times, and speeding up reconfiguration. An overlay that implements a CPU, for instance, can reuse tools that have been designed for CPUs, such as compilers, integrated development environments (IDEs), and debuggers. Not only is the level of abstraction in programming and debugging raised, but compiling for a CPU is orders of magnitude faster than synthesizing an FPGA design. By making a coarser layer which is less flexible than the underlaying FPGA fabric, CAD tools have less design space to explore, shortening compile time. Also, in general, an overlay is only suited to a particular class of applications, and so the mapping from application to overlay is more straightforward.

Overlays can implement many functions, such as networks [32], simulators [31], or even a different reconfigurable logic fabric (a meta-FPGA) [8].

The overlays dealt with in this work are SVPs; before discussing them directly it is useful to know how other soft processors are designed and why they are designed that way. A canonical example of a scalar soft processor is Altera's Nios II processor [4]. Nios II is actually a family of processors, from a multicycle economy variant (Nios II/e) to a six-stage pipelined high performance variant (Nios II/f). The ability to drop in a different ISA-compatible variant allows the designer to allocate more or less resources in exchange for more or less performance without having to rewrite the application software running on top of the soft processor. Additionally, each variant has a multitude of configuration options such as cache sizes and whether to use hardware for certain instructions or trap to software emulation.

Implementing a soft processor does not involve the same tradeoffs as creating a custom CMOS CPU or even a processor embedded within an ASIC. A summary of the differences in FPGA and custom CMOS with respect to processor design can be found in [71]. They find that designs that simply map custom CMOS CPUs to FPGAs have a delay 18 to 26× greater and use 17 to 27× more area. Key to the differences are the relatively higher costs of multipexors and content addressable memorys (CAMs), which are necessary to implement data forwarding nextworks and reordering structures such as those used in out-of-order execution (OoOE). Multiported register files are also relatively more expensive in FPGAs, which makes it more difficult to implement superscalar or VLIW techniques where multiple operands are read and written each cycle.

We are therefore motivated to look at other processor organizations that can offer increased peformance without requiring structures that are difficult or costly to implement on an FPGA. Particularly well-suited organizations include vector processing and SIMD processing, which use deep pipelines and replicated PEs respectively.

2.3 Vector Processing and SIMD Overview

There are many different parallel processor organizations; a good overview can be found in [20]. SVPs take core concepts from both the vector and SIMD paradigms.

Vector processing has been applied in supercomputers on scientific and engineering workloads for decades [21]. It exploits the data-level parallelism readily available in scientific and engineering applications by performing the same operation over all elements in a vector or matrix. It is also well-suited for image processing. The first commercially succesful (and certainly most iconic) vector processor was the Cray-1 [56]. The Cray-1 improved on prior vector processors in several ways, most notably by having a vector register file (VRF) instead of streaming vector data from memory. Each vector register could hold up to 64 elements of data

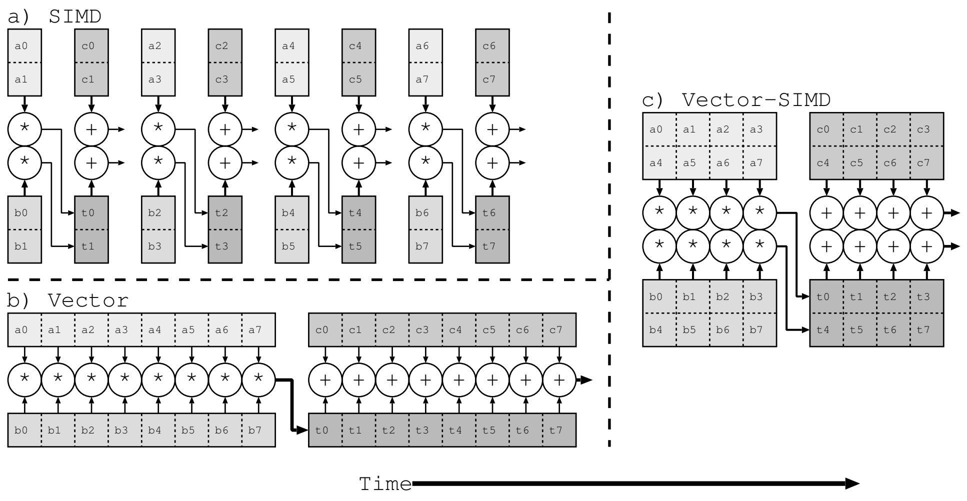


Figure 2.2: Data Parallel Execution on Different Architectures

which could be streamed through a FU at the rate of one element per cycle. The number of elements actually processed by any given instruction was controlled by the vector length (VL) register. Only one instruction could issue per clock cycle, but because each vector instruction would take multiple clock cycles, more than one instruction could be executing at the same time. In the case of dependent vector operations, the output of one FU could feed into the input of another FU; this is a process called chaining. Multiple FUs could be chained together this way, with each FU operating at the same time to provide parallel execution.

SIMD processing, by contrast, utilizes multiple parallel FUs executing the same instruction at the same time. Modern microprocessors are augmented with SIMD processing instructions to accelerate data-parallel workloads. These operate on short, fixed-length vectors (e.g., only 128b, or four 32b words). Significant overhead comes from instructions to load/pack/unpack these short vectors and looping (incrementing, comparisons, and branches are still necessary). Intel's Haswell processors implement AVX2, the latest version of their mainstream SIMD extensions, with a maximum vector length of 256b (8x32b words) [27]. In contrast, the Intel Many Integrated Core (MIC) microarchitecture Xeon Phi supports 512b vector lengths (16x32b words) [28].

Figure 2.2 shows data being processed in a vector processor (with instruction

```
//a) Scalar
for i in 0..7
  temp = a[i] * b[i]
  d[i] = temp + c[i]

//b) 2-wide SIMD
for i in 0..3  //2 elements per instruction, so only 4 iterations
  temp[0:1]      = a[2*i:(2*i)+1] * b[2*i:(2*i)+1]
  d[2*i:(2*i)+1] = temp[0:1] + c[2*i:(2*i)+1]

//c) Vector and Vector-SIMD
set_vector_length(8)   //Subsequent instructions process 8 elements
temp[] = a[] * b[]     //Elementwise vector multiply
d[]    = temp[] + c[]  //Elementwise vector add
```

Figure 2.3: Psuedocode for Different Parallel Paradigms

chaining), a SIMD processor, and a hybrid vector-SIMD processor. Pseudocode for this example is shown in Figure 2.3. The code multiplies two eight-element arrays together and adds them to a third eight-element array. In the code for a scalar processor (Figure 2.3a), this requires eight iterations through a loop; in each iteration a multiply and add instruction are executed. Additionally, the data elements are read from and stored back to memory in eight separate memory operations per array. If data is not available in the processor's cache or there are hazards then there will be additional cycles in which no work is being done. Thus, even though there are 16 cycles of computation (eight multiplies and eight adds) a scalar processor may take several times that many cycles to execute the whole loop.

Figure 2.3b shows the code for a 2-wide SIMD implementation. The code closely matches scalar code except that every operation works on two elements, so only four loop iterations are needed. Most SIMD architectures also have SIMD load and store instructions, so the overhead of issuing memory instructions is similarly reduced. Figure 2.2a shows how the data moves through a SIMD processor for this code; there are eight SIMD instructions which will take at least eight cycles to execute. The total execution time for SIMD code rarely scales perfectly with the number of parallel ALUs; hazards and stalls are not reduced by increasing SIMD width and so they quickly dominate runtime.

Figure 2.3c shows the vector code. Instead of having a preset amount of data processed by each instruction, there is a VL register that controls the amount of data processed. After it is set to eight, the subsequent instructions will process eight data elements each. Figure 2.2b shows the data movement in a classic vector processor; there are only two vector instructions to execute. Because the control for looping and addressing is in the vector processor's hardware, once the instruction has started, executing the computation can proceed at one cycle per operation until the entire array has been processed. Vector processors have vector-load and vector-store instructions to address memory in large chunks, which can allow for better memory bandwidth utilization.

Figure 2.2c shows the code running on a vector-SIMD hybrid; it combines the efficiency of vector execution and the parallelism of SIMD. While there are the same number of execution cycles as a SIMD implementation, only two instructions need to be issued and there is no control flow or address manipulation in software. Note that the same code runs on a classic vector processor or vector-SIMD regardless of SIMD width; the vector hardware executes for as many cycles as is needed depending on the current VL and the number of ALUs.

The vector-SIMD model allows parallelism in a more scalable manner than chaining. Chaining and SIMD are not orthogonal; for instance, the T0 vector processor could chain together three instructions on eight vector pipelines to achieve 24 operations per cycle [6]. Chaining takes advantage of instruction level parallelism (ILP), while SIMD takes advantage of data level parallelism (DLP). While measuring the amount of ILP present in a program is non-trivial [55], in practice vector processors have been limited to chaining three instructions, and wide superscalar CPUs such as Intel's latest Haswell architecture can only commit four operations per cycle [30]. Not only is there a limit to how much ILP can be extracted, exploiting it requires reading and writing multiple operands per cycle from a flat register file. In SIMD execution, by contrast, each ALU writes back to its own independent bank of the vector register file, so parallelism is achieved without needing multiple read and write ports [36].

The case for embedded vector processors was made by the VIRAM [36] project. It showed that vector processing was more efficient than VLIW or superscalar techniques for embedded media applications. VIRAM used an existing MIPS CPU to handle control flow, and dispatched vector instructions to a coprocessor vector core. The vector core consisted of four 64-bit lanes, which could alternately process eight 32-bit or sixteen 16-bit elements per cycle. This ability to split lanes and process more elements of narrow widths is common in SIMD architectures and is referred to as intra-lane SIMD or sub-word SIMD. VIRAM was a load/store architecture and connected to memory through a crossbar connected to eight DRAM banks. VIRAM was the architecture the first SVPs were modeled upon.

2.4 Soft Vector Processors (SVPs)

The first SVPs were developed in parallel, VIPERS [78] from The University of British Columbia and VESPA [74, 76] from the University of Toronto.

VIPERS demonstrated that programmers can explore the area-performance tradeoffs of data-parallel workloads without any hardware design expertise. The results of three benchmark kernels demonstrate a scalable speedup of 3–30× over the scalar Nios II processor. Additionally, a speedup factor of 3–5× can be achieved by unrolling the vector assembly code. VIPERS uses a Nios II-compatible multithreaded processor called UT-IIe [22], but control flow execution is hindered by the multithreaded pipeline. The UT-IIe is also cacheless; it contains a small on-chip instruction memory and accesses all data through vector read/write crossbars to fast on-chip memory. VIPERS instructions are largely based on VIRAM. VIPERS also offered a comparison to high level synthesis (HLS) (Altera's C2H compiler [1]) and custom RTL circuits. The HLS comparison showed VIPERS to be larger in area but faster; they were roughly even in terms of performance per area. The RTL comparison only used one (motion estimation) benchmark, for which VIPERS was 82× slower at the same amount of area, though that could be reduced to 21× with minor ISA enhancements.

VESPA is a MIPS-compatible scalar core with a VIRAM compatible vector coprocessor. The original VESPA at 16 lanes can acheive an average speedup of 6.3× the scalar core over six EEMBC benchmarks. Furthermore, an improved VESPA achieves higher performance by adding support for vector chaining with a banked register file and heterogeneous vector lanes [75]. Over the 9 benchmarks tested, the improved VESPA averages a speedup of 10× at 16 lanes and 14× at 32 lanes. The MIPS core uses a 4kB instruction cache, and shares a data cache with

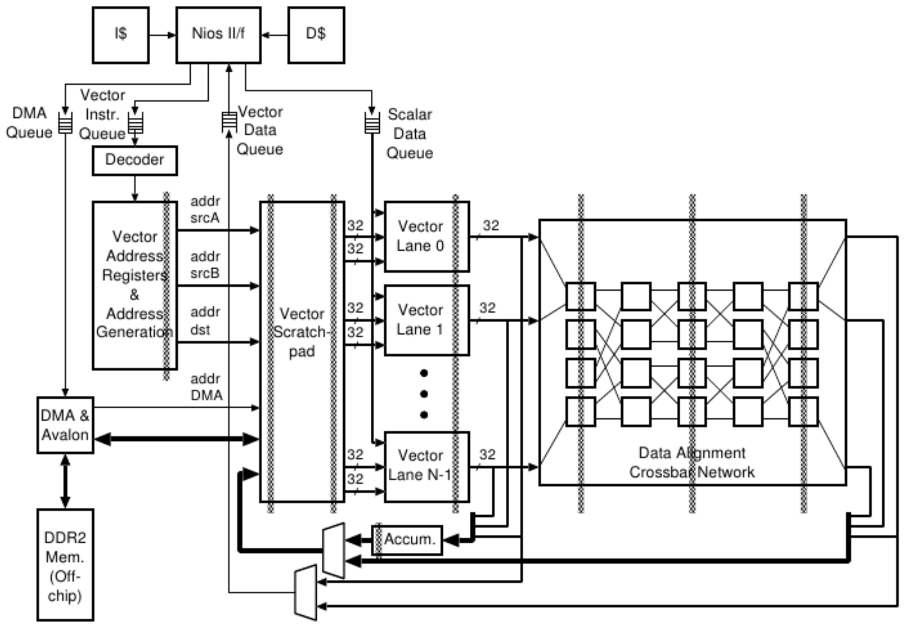


Figure 2.4: The VEGAS Soft Vector Architecture [15]

the vector coprocessor. Smaller (1- or 2-lane) vector coprocessors use an 8kB data cache, while larger ones use 32kB.

Both VIPERS and VESPA offer a wide range of configurability. For example, the parallel vector lanes can be specified at FPGA compile-time to be 8, 16 or 32 bits wide. However, when mixed-width data is required, the vector engine must be built to the widest data. Therefore, when processing smaller data, load instructions will zero-extend or sign-extend to the full width, and store instructions will truncate the upper bits. Since the vector register file must store all data (even byte-sized data) at the widest width, VIPERS and VESPA can be very inefficient: byte-wide data is stored in the on-chip main memory or data cache, then expanded to word-wide inside the register file. On top of that, to implement dual read ports, VIPERS and VESPA duplicate the vector register file. Hence, a single byte of data may occupy up to 9 bytes of on-chip storage (including the copy in the data cache).

The VEGAS soft vector architecture [15] uses a Nios II/f with a soft vector accelerator. Figure 2.4 shows the architecture of VEGAS. Each vector instruction is encoded into the free bits of a Nios custom instruction. Instead of vector data registers, VEGAS uses an 8-entry vector address register file (VARF) that points into a large, banked scratchpad memory. The scratchpad can be partitioned into any number of vectors of any length. During execution, the ALUs in VEGAS can be fractured to support subword arithmetic. This means a fixed-width vector engine can operate on more elements if they are halfword or byte sizes. Also, it makes more effective use of the scratchpad memory, since smaller operands are not expanded to fill an entire word as done with prior architectures. The execution pipeline is followed by a data alignment network that is used when operands start in different banks of the scratchpad. If the source operands are aligned to each other but the destination has a different alignment, this occurs in a pipelined fashion. However, if the source operands are not aligned to each other the vector core must stall the current instruction and insert a vector move operation to correct the alignment mismatch.

Both VIPERS and VESPA also share a similar memory-system design. They use vector load and vector store instructions to transfer blocks of data, with optional strides, from main memory to the vector data register file. Separate read and write crossbars shuffle data during loads and stores.

In contrast, VEGAS does not support vector load and store instructions. Instead, it uses direct memory access (DMA) block-read and block-write commands to copy between the scratchpad and main memory. All vector alignment / byte shuffling is done at runtime in the writeback pipeline stages by passing vector results through a shuffle network. Except for lengthening the pipeline, there is no run-time penalty if the destination vector is misaligned. However, if vector source operands are misaligned, VEGAS must first copy one operand to a temporary location in the scratchpad. This extra copy operation cuts performance roughly in half.

The vector register file is connected to an on-chip memory (VIPERS) or on-chip data cache (VESPA) through *separate* read and write crossbars. These crossbars are used when striding through memory during vector load and store instructions; they must shuffle bytes/halfwords/words from their byte-offset in memory into word size at a new word-offset in the vector register file. The size of the crossbars are constrained on one end by the overall width of the vector register file, and on the other end by the overall width of the on-chip memory/cache. As the vector processor is scaled to contain more lanes, one end of the crossbars increases in size while the other end is fixed by the memory width. To quickly load a large register file, the on-chip memory/cache width must be similarly increased to match. The area to implement these crossbars is significant, and grows as the product of the two widths.

While VEGAS serves as the starting point for our work, there have been other SVPs or SVP-like processors implemented as well. The FPVC, or floating point vector coprocessor, was developed by Kathiara and Leeser [33]. It adds a floating-point vector unit to the hard Xilinx PowerPC cores which can exploit SIMD parallelism as well as pipeline parallelism. The FPVC fetches its own VIRAM-like instructions and has its own private register file. Unlike most other vector architectures, it can also execute its own scalar operations separate from the host PowerPC.

Convey's HC-2 [18] is a vector computer built using several FPGAs. It is targetted at high performance applications such as geological exploration and utilizes the entirety of the FPGA rather than being a building block in an embedded system like the other SVPs mentioned.

Bluevec [50] was developed to perform neural-network simulations on FPGAs. The authors created a custom SVP in Bluespec [51]. They compared a custom pipeline (also written in Bluespec) to the SVP implementation and found that performance was within a factor of two given similar resource usage, and that further performance improvement was not possible with either design due to memory bandwidth limitations.

Bluevec also has a C++ compiler designed to allow programming at a higher level than vector assembly or C macros [49]. Their compiler is a run-time system that compiles and downloads a kernel to the SVP; the first time a kernel is encountered it is compiled, and then that compiled kernel can be reused if the kernel is executed again. This contrasts to a compiler for the VENICE processor (to be presented in Chapter 3) which does a source-to-source translation of Microsoft's Accelerator object-based langauge [65] to VENICE C code [46].

Additionally, the work on VENICE lead to a commercial SVP, the VectorBlox MXP [59]. MXP is used for the experiments performed in Chapter 4 and Chapter 5. MXP is similar to VENICE in design, but with added features such as fixed-point

```
//a) Scalar
for i in 0..7
  if mask[i] /= 0  //Skip computation if mask element is not zero
    a[i] = (b[i] * c[i]) + (d[i] * e[i])


//b) Vector and Vector-SIMD
set_vector_length(8)   //Subsequent instructions process 8 elements
temp[] = (b[] * c[]) + (d[] * e[]) //Compute result for all elements

      //select elements of temp[] if the corresponding mask is not zero
      //otherwise keep the same value in a[]
a[] = merge_nonzero(mask[], temp[], a[])
```

Figure 2.5: Psuedocode for Divergent Control Flow

arithmetic, 2D-DMA support, and a C++ object based application programming interface (API) for higher level programming.

2.5 Divergent Control Flow

Chapter 4 deals with a method of speeding up execution of algorithms with divergent control flow. Figure 2.5 gives an example of code that has control flow divergence. The scalar version has a branch within the inner loop (the if statement). For each iteration i of the loop, if the mask[i] element is nonzero a calculation is performed and the result stored into a[i]. If the mask[i] element is zero, nothing happens and the scalar processor can jump to the next iteration of the loop. Figure 2.5b gives the code example for a vector processor. Instead of performing the calculation only for the elements where mask[i] is nonzero, the calculation is performed for all elements and stored in temp[i]. Then a merge instruction is used to select whether to write temp[i] to a[i] or leave the value unchanged (by writing back the current value of a[i]). Predicated instructions can also be used. Predicated instructions use a writeback enable signal to prevent the update of the result vector depending on the value of a flag vector that was previously set.

Support for divergent control flow in vector processors is well researched; a good summary is [63]. For short conditionals, implementations that use merge or predicated operations can give good performance. These operations are performed on all data elements, with only the elements that pass some conditional test written back; the rest of the results are discarded. For longer conditional branches, how-

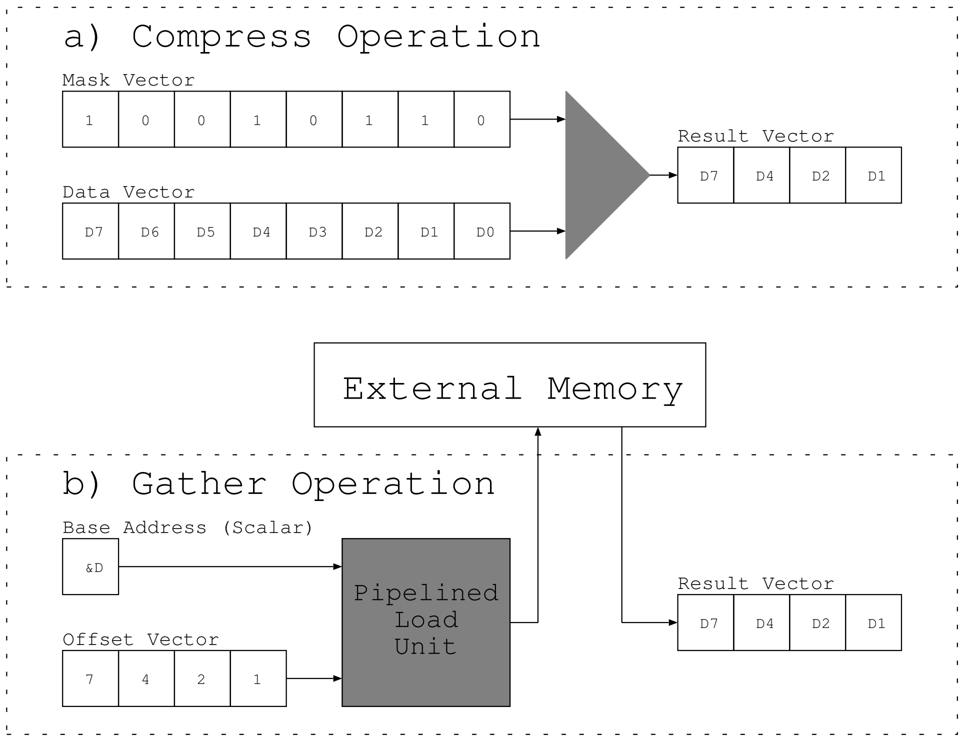


Figure 2.6: Compress and Gather Operations

ever, this can lead to a large percentage of execution time spent on unused results. In these instances, it is desirable to skip elements that are not on the current branch. Three strategies can be employed (separately or together): compress/expand, scatter/gather, and wavefront skipping (also referred to as density-time masking).

Compress vector operations are a way to take a source vector and a mask and produce a new vector that only contains the valid (unmasked) elements without gaps, while expand vector operations take a compressed source vector and a mask and fill in the unmasked slots in the destination vector with the source data [39]. Figure 2.6a shows an example of a compress operation. The compress operation takes in a mask vector and a data vector, and outputs a vector with only the data elements that correspond to locations in the mask vector that are set. Locations that are masked off (mask vector equal to zero) are discarded. VIRAM implemented VCOMPRESS and VEXPAND operators. During a long branch, the source operands can be compressed, followed by processing only the shorter (compressed) vector operations, and finally the result expanded. Since execution time is proportional to vector length, operations on compressed vectors are potentially faster than predicated operations on the uncompressed vectors. The main drawback to this approach is that all source operands must be compressed, and all destination operands must be expanded. In a MxN conditional stencil filter computation, for instance, the MxN input pixels would all have to be separately compressed.

Scatter/gather is a method of performing indexed memory accesses (indexed store is a scatter, indexed load is a gather), either to the local memory store or external memory [29]. Along with a compress operation or special index calculating instructions, scatter/gather can be used to speed up conditional execution. Figure 2.6b shows an example of using a gather operation. A scalar base address along with a vector of offsets is used to calculate the addresses of each data vector element in external memory. The resulting addresses pass through a memory unit and are loadied into the desired result vector. For the conditional stencil filter example, the indices of pixels to be processed could be compressed, and then the pixel data needed for each location could be loaded using gather operations. The main drawback of this approach is that parallelizing scatter and gather operations requires parallel memory acceses, which are nontrivial. VESPA could perform parallel scatter gather accesses within a single cache line. A special throughput cache [58] was developed for MXP to support scatter/gather operations, but even in the best case where all data could fit in a statically allocated multiple-bank onchip memory, speedups were modest.

Wavefront skipping, by contrast, uses knowledge of the mask register to skip elements that do not need to be processed. Early implementations scanned the mask register during instruction execution to determine if subsequent wavefronts could be skipped. This introduces enough latency that a way to reduce the overhead by only skipping powers of 2 elements was patented [62]. Though this reduces latency, some extra work is done because the skipping is not exact. Given that an FPGA implementation is already slower than a hard processor and we are double-clocking our scratchpad to provide additional ports, we wished to avoid the additional latency in our design. Our implementation uses a special mask setup instruction to store the offsets of valid wavefronts in one or more BRAMs within the FPGA. Prior work [63] suggested the idea that each lane can skip forward individually rather than lockstep as entire wavefronts, but no implementation was created.

A similar concept exists in GPUs based computing [53]. GPUs divide up a kernel (consisting of hundreds of threads) into warps, which are analogous to our wavefronts, usually 16 or 32 threads. Each warp maintains a program counter (PC) and a mask of active elements. Inside a kernel, branch instructions allow different threads to diverge. If threads within a warp take different paths because of a branch instruction, then a separate PC and active mask must be maintained for both branches. However, if all threads take the same path, the PC for the not-taken path can be discarded. This is somewhat analagous to unpartitioned wavefront skipping in vector processors, where a wavefront that is completely masked-off does not need to be executed.

There has been some work published about, but to our knowledge no actual implementations of, GPUs skipping at a finer grain than the warp level. Warp compaction for single-instruction multiple-thread (SIMT) processors and thread block compaction were simulated [23, 24]. This approach works within blocks of warps and moves threads between warps to reduce the amount of unnecessary work done. A similar approach uses 'large warps' which are much larger than the SIMD width of the actual FUs and then executes elements from different subwarps that are all active [48].

Similarly, an architectural study involving the vector-thread architectural paradigm looked at divergence, including density-time execution [45]. Vector-thread architectures can be thought of as a hybrid between vector processors and GPUs. A single control thread executes per core, and this control thread can issue a vector-fetch instruction that starts parallel execution. This vector-fetch is not a single instruction, but the start of a kernel of *microthreads* that can include control flow and diverge. When diverged, an active mask is used similar to a GPU, with pending PCs placed in a pending vector fragment buffer (PVFB). Microthreads do not reconverge in the baseline, but keep executing until all encounter a stop instruction and the microkernel finishes. With a simple first-in first-out (FIFO) PVFB, adding density-time execution reduced energy use by more than $2\times$. Adding a dynamic reconverge scheme lessens the impact of density-time execution. This work only considered single lane implementations of density-time execution.

2.5.1 Execution Pipeline Customization

Chapter 5 explores our method for customizing the SVP pipeline with custom operators created in FPGA logic. There has been some related work in data parallel processors with configurable or extensible pipelines.

Some scalar processors have support for extensible pipelines. The Tensilica Xtensa [25] is a synthesizable CPU for system on chip (SoC) designs which reserves certain opcodes for custom instructions. Some soft processors, such as Altera's Nios II [4], support custom instructions directly in their execution pipeline. Others, such as Xilinx's MicroBlaze [72] implement tightly coupled queues to interface to custom logic in a similar manner.

VESPA included support for heterogeneous vector lanes [75]; e.g., having fewer multipliers than general-purpose ALUs. Due to the mismatch between the vector register file width and execution unit width, a parallel load queue was used to buffer a vector for heterogeneous operations, and a separate output queue was used to buffer results before writeback. The amount of customization was limited to subsetting the existing ISA.

Work by Cong et al. [16] created composable vector units. At compilation time, the DFG of a vector program was examined for clusters of operations that can be composed together to create a new streaming instruction that uses multiple operators and operands. This was done by chaining together existing functional units using an interconnection network and multi-ported register file. This is similar to traditional vector chaining, but it was resolved statically by the compiler (not dynamically in the architecture) and encoded into the instruction stream. This provided pipeline parallelism, but was limited by the number of available operators and available register file ports. The reported speedups were less than a factor of two.

The Convey HC-2 [18] can adopt one of several 'personalities', each of which provides a domain-specific vector instruction set. User-developed personalities are also possible. Designed for high-performance computing, the machine includes a high bandwidth, highly interleaved, multi-bank DRAM array to reduce strided access latency. Since the personalities come preconfigured it may be more useful to think of the HC-2 as a family of processors rather than a processor with a extensible execution units.

Chapter 3

VENICE: Optimizing for Small but Capable

It takes a tough man to make a tender chicken. — Frank Perdue

Our first approach to broadening the applicability of SVPs was to reduce the area/performance penalty of using SVPs to make them more competitive with hand-crafted RTL designs. In modest performance applications where only a small SVP is needed, the area overhead incurred (compared to RTL) will be small relative to the area used in the rest of the system. This contrasts to high performance applications, where the area penalty will dominate the area of the whole system, and therefore there is more incentive to spend time developing a custom RTL solution. This lead us to investigate an SVP design that targetted a small number of lanes; we envision that such a solution that could be a building block in a larger system rather than a stand-alone processor that takes up the entire FPGA.

3.1 Introduction

This chapter presents VENICE, an SVP designed for maximum throughput with a small number (one to four) of ALUs. VENICE demonstrates that applications that need somewhat higher performance than a scalar soft processor can provide can be implemented while using only modestly more resources. VENICE can achieve over 2x better performance-per-logic block than VEGAS, the previous best SVP as of the time of its development (2011-2012). It achieves this through a combination of increasing clockspeed, eliminating bottlenecks in ALU utilization, and reducing area through FPGA-specific design. While VENICE can scale to a large number of ALUs, a multiprocessor system of smaller VENICE SVPs is shown to scale better for benchmarks with limited inner-loop parallelism. VENICE is also simpler to program than previous SVPs, since its instructions are C macros using pointers into a scratchpad memory rather than requiring assembly and/or manually allocating vector registers.

VENICE is smaller and faster than all SVPs published before it. Not only is it roughly $2\times$ better performance per logic block (speedup per ALM) than VEGAS [15], but it is also $5.2\times$ better than Altera's fastest Nios II/f processor. As a result, less area is needed to achieve a fixed level of performance, reducing device cost or saving room for other application logic. Alternatively, it enables larger multiprocessors to be built using VENICE, resulting in greater computational throughput for the fixed area of a specific device.

The key contributions that lead to this are:

- Removal of vector address register file (area)
- Use of 2D and 3D vectors (performance)
- Operations on unaligned vectors (performance)
- New vector conditional implementation (area)
- FPGA optimized fracturable multiplier (area)

Programming VENICE requires little specialized knowledge, utilizing the C programming langauge with simple extensions for data parallel computation. Changes to algorithms require a simple recompile taking a few seconds rather than several minutes or hours for FPGA synthesis. In particular, the removal of the vector address register file makes VENICE easier to program than VEGAS. In addition, VENICE does not suffer a performance penalty when using unaligned operands, freeing programmers from worrying about the placement of data in memory. Finally, as described in a separate publication, Zhiduo Liu developed

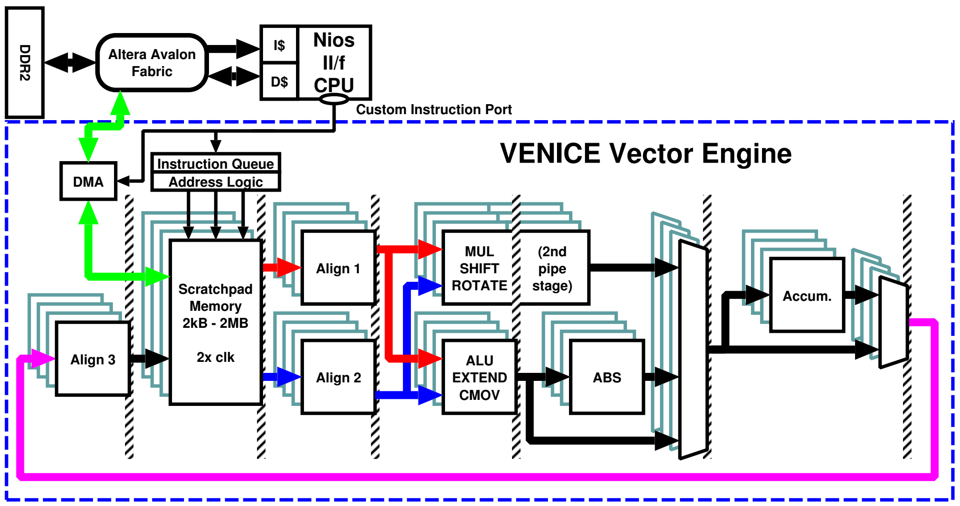


Figure 3.1: VENICE Architecture

a compiler for VENICE based on Microsoft Accelerator [46], demonstrating that high level code can be translated to VENICE's programming model.

3.2 Design and Architecture

A block diagram of the VENICE (Vector Extensions to NIOS Implemented Compactly and Elegantly) architecture is shown in Figure 3.1. Similar to previous SVPs, VENICE requires a scalar core as the control processor; in this case, a Nios II/f executes all control flow instructions. In our work, Nios is configured to use a 4kB instruction cache and a 4kB data cache. Unfortunately, Nios lacks support for hardware cache coherence, so the programmer must sometimes explicitly flush data from the cache to ensure correctness.

VENICE vector instructions are encoded into one or two tandem Nios custom instructions and placed inline with regular instructions. These custom instructions are written into a 4-entry vector instruction queue. Typically, after writing the vector instruction to the queue, the Nios core can continue executing its next instruction. However, the Nios core will stall if the instruction queue is full, or if the custom instruction synchronizes by explicitly waiting for a result.

The VENICE vector engine implements a wide, double-clocked scratchpad memory which holds all vector data. Operating *concurrently* with the vector ALUs and the Nios core, a DMA engine transfers data between the scratchpad and main memory. The DMA engine also has a 1-entry control queue, allowing the next transfer to be queued.

It is very important for the DMA engine to operate concurrently with the vector ALUs, because this hides the memory latency of prefetching the next set of data elements. All of our benchmarks implement some form of double-buffering, allowing us to hide most of the memory latency.

All vector ALU operations are performed memory-to-memory on data stored in the scratchpad. A configurable number of vector lanes (32-bit vector ALUs) provides additional parallelism. Each 32-bit ALU supports subword-SIMD operations on halfwords or bytes, thus doubling or quadrupling the parallelism with these smaller data types, respectively.

3.3 VENICE Implementation

Below, we will describe each of the key improvements made to VENICE. As a result of these and other optimizations, the design was pipelined to reach 200MHz+ (roughly 50–100% higher than previous SVPs). This also allows the SVP to run synchronously at the same full clock rate as the Nios II/f, simplifies control, adds the predictablity of a fully synchronous design, and reduces resource usage.

3.3.1 Removal of Vector Address Register File

One major difference from VEGAS is removal of the vector address register file (VARF). In VEGAS, the vector instructions emitted from the scalar core include two source register numbers and one destination register number. They are indices into the VEGAS 8-entry VARF, which produces addresses that index into the VEGAS scratchpad memory. Three VARF entries need to be read for every VEGAS instruction, and possibly modified due to instructions that automatically increment address registers. To support three reads and three writes concurently the VARF was not implemented in BRAM , but rather using LUTs and flip-flops (FFs). The result was that the VARF used 377 ALMs in a V1 VEGAS implementation, which is 10% of its total ALM usage.

```
// VEGAS code to add two vectors already in scratchpad
// Clobbers VARF registers V1, V2, V3
void vegas_add(int length, int *v_a, int *v_b, int *v_dest){

  //Set the vector length
  vegas_set( VCTRL, VL, length );

  //Copy the pointer values from the scalar register file to the VARF
  vegas_set( VADDR, V1, v_a );
  vegas_set( VADDR, V2, v_b );
  vegas_set( VADDR, V3, v_dest );

  //Perform the operation (VARF values index into scratchpad)
  vegas_vvw( VADD, V3, V1, V2 );
}


// VENICE code to add two vectors already in scratchpad
void venice_add(int length, int *v_a, int *v_b, int *v_dest){

  //Set the vector length
  vector_set_vl( length );

  //Perform the operation (pointers index directly into scratchpad)
  vector( VVW, VADD, v_dest, v_a, v_b);
}
```

Figure 3.2: VEGAS Code (Requiring Vector Address Register Setting) vs VENICE Code

Instead, VENICE relies upon the Nios register file to contain the vector addresses. Instead of emitting register numbers, the vector instruction is emitted with two source addresses and one destination address; these addresses directly index the scratchpad memory. Nios instructions can run concurrently with vector operations, calculating addresses for the next vector instruction while the current one executes. Additionally, the 2D and 3D vector instructions (which will be explained in Section 3.3.2) replace the functionality of automatically incrementing address registers.

In VEGAS, copying values into the VARF and spilling/filling them is a cumbersome task for a programmer. Code demonstrating how the VARF is used in VEGAS along with the corresponding code for VENICE is shown in Figure 3.2 (a more complete explanation of the VENICE programming model will be given in Section 3.4). Both VEGAS and VENICE work with pointers into the scratchpad memory, but programming VEGAS requires copying pointers into local vector address registers (V1, V2, and V3 in this example). Vector operations then use the vector address register number rather than the pointer, making code less readable and harder to maintain. Additionally, if more than eight vectors are needed the VARF values will have to be spilled and filled between vector operations, hurting performance. In VENICE there is no need to do this extra work. The lack of VARF also makes the task of writing a compiler for VENICE easier.

One drawback of this approach is that increased instruction issue bandwidth is required between the Nios and VENICE vector engine. VEGAS only has to transmit three constant 3-bit VARF indices along with its opcode and modifier bits for each instruction, allowing a VEGAS instruction to be encoded as a single Nios II/f custom instruction. VENICE instructions, by contrast, need to read three operand addresses from the Nios II/f register file. Since a Nios II/f custom instruction can only access two register values per cycle, two Nios II/f custom instructions are needed to issue each VENICE instruction.

3.3.2 2D and 3D Vector Instructions

The execution of some programs is limited by the instruction dispatch rate, especially when short vectors are used. This makes it difficult to achieve high vector ALU utilization.

To get around this limitation, VENICE implements 2D and 3D vector instructions. A 1D vector is an instruction applied to a configurable number of elements, which can be thought of as columns in a matrix. The 2D vector extends this to repeat the 1D vector operation across a certain number of rows. In between each row, the operand address will be incremented by a configurable stride value, allowing for selection of arbitrary submatrices. Each operand uses its own unique stride value. When executing, the vector core dispatches a separate vector instruction for each row, and in parallel adds the strides for each operand to determine the address of the next row. As a result, VENICE can issue up to 1 row per cycle. This has a direct extension to 3D instructions for operations on 3D data, which can process multiple 2D matrices with a single instruction.

An example of programming using 2D vector instructions will be given in Section 3.4.

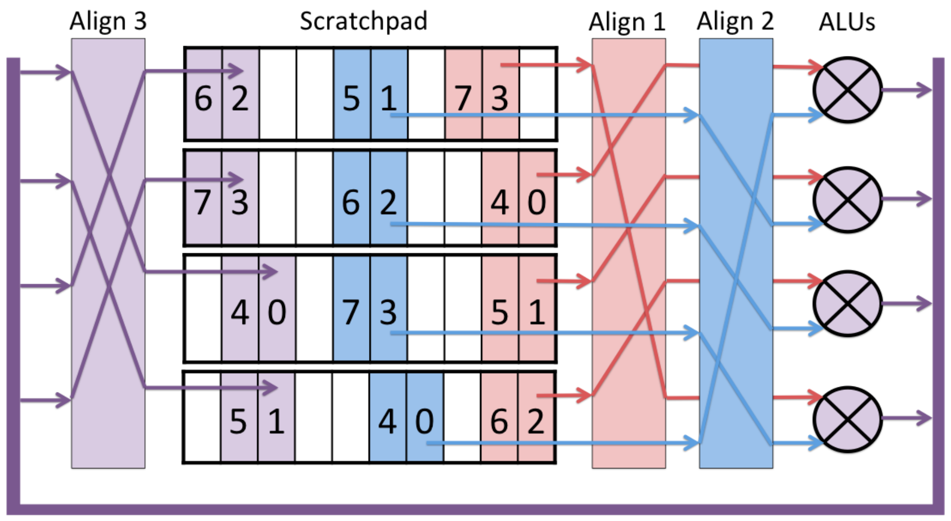


Figure 3.3: Example of Misaligned Operation

3.3.3 Operations on Unaligned Vectors

When input and/or output vectors are not aligned, data needs to be shuffled between lanes. For example, Figure 3.3 shows how VENICE uses three alignment networks to shuffle unaligned data. A four bank scratchpad memory is shown, with Bank 0 at the top of the figure. The three operand vectors are striped across the scratchpad memory, with the first input starting in Bank 1, the second input starting in Bank 3, and the destination vector starting in Bank 2. When an instruction is issued using these operands, the two source operands are first aligned into a canonical position such that Element 0 is moved to the top ALU. Next, the vector data is processed by the ALUs. Finally, the result is re-aligned, moving Element 0 of the result vector into the scratchpad memory bank that corresponds to the start of the destination operand target position. Doing vector alignment in the processors pipeline is especially useful when doing convolution, as one operand's starting address is continuously changing and seldom aligned.

Note that only two alignment networks are required to align the two input operands and one output operand; the extra alignment network was put in place to allow for future work when operand data sizes mismatch. The ability to do operand resizing in the pipeline was not implemented in VENICE (though it was in the commercial MXP processor [59]), so the third alignment network could be removed from VENICE to save area without affecting performance.

The older VEGAS processor uses a single alignment network because this is the only component that grows super-linearly with the number of vector lanes. This is a concern as the number of lanes increases, but not for the small number of lanes that VENICE is designed for, as we will show in Section 3.5.1. The drawback of VEGAS's single alignment network is that it introduces a performance penalty when input operands are not aligned. In VEGAS, when the two input operands were not aligned, a copy operation is automatically inserted to align them. These extra copies can potentially halve performance on code where inputs are mostly unaligned, such as sliding window algorithms. Additionally, space is reserved at the end of the scratchpad memory for these temporary copies, meaning that the full scratchpad can not be used by the program unless all operations are provably aligned.

3.3.4 New Vector Conditional Operations

Previous SVPs based on the VIRAM architecture used eight vector mask registers to store flags which could be used to perform conditional operations. VEGAS used a similar approach, storing flags in separate BRAMs from the scratchpad.

VENICE takes a novel approach to data-conditional operations. Vector flags for compare instructions and arithmetic overflow are written alongside each byte, using the 9th bit in BRAMs that have an extra data bit for parity or optional storage (such as the M9K in Altera Stratix devices). This reduces BRAM usage, but does not allow for a VIRAM-style register file of flags. Instead, conditional move operations that read the flag bits as one of the input operands are used to implement simple predication.

The stored flag value depends upon the operation: unsigned addition/subtraction stores the carry-out/borrow, while signed addition/subtraction stores the overflow. The flag and most significant bit (MSB) result bits can thus be used to check out-of-range results, less-than/greater-than (using the negative-result flag), or extended precision (64-bit) arithmetic.

3.3.5 Streamlined Instruction Set

VENICE implements a simple instruction set, consisting of 24 basic operations versus over 50 instructions in VEGAS. VENICE also has several mode bits to indicate more complex operations. Because two Nios custom instructions are required to dispatch a VENICE instruction, there are enough usable bits to encode the ISA while allowing any combination of mode bits, making the ISA fully orthogonal. Absolute value, 2D/3D instructions, and accumulation reduction can all be combined with any instruction and each other. This allows for complex instructions to be built that perform multiple operations between reading and writing to the scratchpad. All operations can take a scalar operand as one input, and the scalar always replaces operand A, in contrast to VEGAS where the scalar would replace operand A or B depending on the operation.

3.3.6 FPGA Architecture-Specific Optimizations

Figure 3.4 shows VENICE's method of implementing fracturable multipliers versus the partial products method used in VEGAS. The previous method used four 18-bit multipliers. Byte and halfword operations could be performed directly using the 18-bit multipliers, while word operations used the four multipliers to compute partial products which were then added together. Since this addition was not performed for all operations, the adder could not be implemented using one of the DSP Block's internal adders. The VENICE method, by contrast, uses a simpler parallel implementation where there is one 36-bit multiplier that can perform a word/halfword/byte multiply, one 18-bit multiplier that can perform a halfword/byte multiply, and two 9-bit multipliers that can perform byte multiplies. The VENICE method uses the same number of DSP Blocks as the partial products method, but does not require additional adders or multiplexers on the inputs. This leads to both lower ALM usage and lower delay in Stratix IV FPGAs. The lower ALM usage of the multiplier is the primary reason for the size difference of the ALUs of VEGAS and VENICE that will be shown in Section 3.5.1.

The only drawback of the new multiplier organization is that a single lane's multipliers cannot be packed into a single Stratix IV DSP Block due to configuration limitations. However, two lanes worth of multipliers may be packed into two

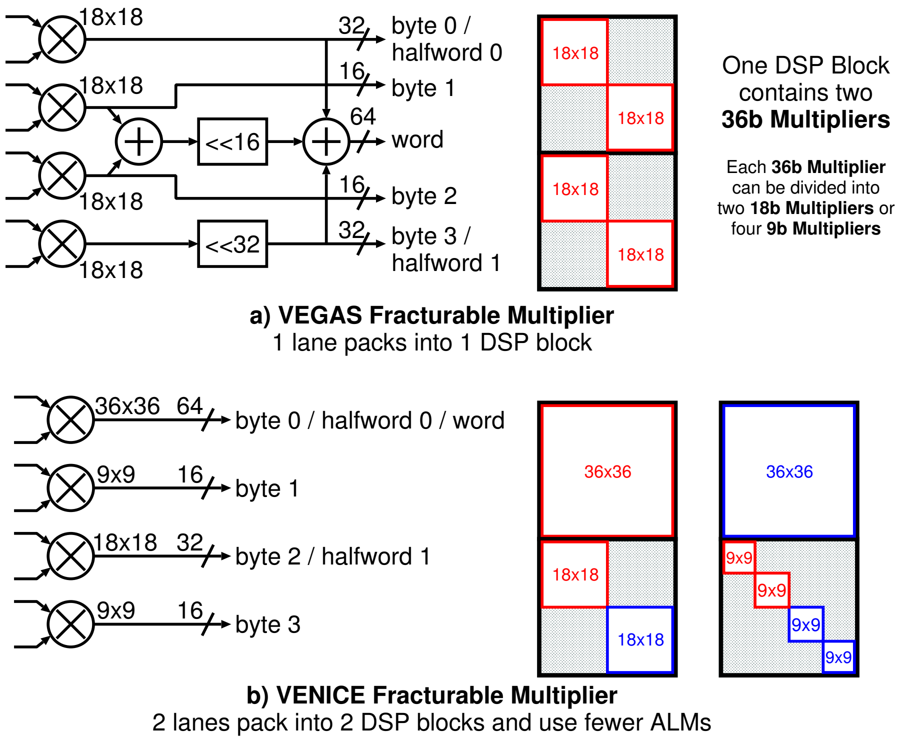

18x18
18x18
18x18
18x18
<<16
<<32
32
16
64
16
32
byte 0 / halfword 0
byte 1
word
byte 2
byte 3 / halfword 1
18x18
18x18
18x18
18x18
One DSP Block contains two **36b Multipliers**
Each **36b Multiplier** can be divided into two **18b Multipliers** or four **9b Multipliers**
a) VEGAS Fracturable Multiplier
1 lane packs into 1 DSP block
36x36
64
byte 0 / halfword 0 / word
9x9
16
byte 1
18x18
32
byte 2 / halfword 1
9x9
16
byte 3
36x36
18x18
18x18
36x36
9x9
9x9
9x9
9x9
b) VENICE Fracturable Multiplier
2 lanes pack into 2 DSP blocks and use fewer ALMs

Figure 3.4: Fracturable Multiplier Styles

DSP Blocks. Hence, V1 may use an additional half DSP Block, but larger designs use the same amount as VEGAS.

Despite its lower latency compared to previous multiplier implementations, it is necessary to pipeline the multiplier over two stages to achieve high freqency operation. This leaves an extra cycle for processing simple (non-multiplier) ALU operations which can complete in a single cycle. A general absolute value stage was added after the integer ALU, allowing operations such as absolute difference in a single instruction. When followed by VENICE's reduction accumulators, operations such as sum-of-absolute-differences (used for motion estimation) and multiply-accumulate instructions (for matrix multiply) can be implemented in a single instruction.

```
#include "vector.h"

int main()
{
    const int length = 8;
    int A[length] = {1,2,3,4,5,6,7,8};
    int B[length] = {10,20,30,40,50,60,70,80};
    int C[length] = {100,200,300,400,500,600,700,800};
    int D[length];

    // if A,B,C are dynamically modified,
    // then flush them from the data cache here

    const int num_bytes = length * sizeof(int);

    // alloc space in scratchpad, DMA from A to va
    int *va = (int *) vector_malloc( num_bytes );
    vector_dma_to_vector( va, A, num_bytes );

    // alloc and DMA transfer, in one simple call
    int *vb = (int *) vector_malloc_and_dmacpy( B, num_bytes );
    int *vc = (int *) vector_malloc_and_dmacpy( C, num_bytes );

    // setup vector length, wait for DMA
    vector_set_vl( length );
    vector_wait_for_dma();   // ensure DMA done

    vector( VVW, VADD, vb, va, vb );
    vector( VVW, VADD, vc, vb, vc );

    // transfer results from vc to D
    vector_instr_sync(); // ensure instructions done
    vector_dma_to_host( D, vc, num_bytes );
    vector_wait_for_dma();

    vector_free();
}
```

Figure 3.5: VENICE Code to Add 3 Vectors

3.4 Native Programming Interface

The native VENICE API is similar to inline assembly in C, using C preprocessor macros. The use of macros for instruction encoding was introduced in VEGAS, and makes vector instructions look like C functions without any run time overhead. Previous SVPs had required the programmer to write assembly directly. VENICE further improves on the VEGAS programming model by not having a vector address register file (VARF) (as explained in Section 3.3.1), so no register managment is needed in VENICE code. An example of VENICE code to add three vectors together is shown in Figure 3.5.

Each macro dispatches one or more vector assembly instructions to the vector engine. Depending upon the operation, these may be placed in the vector instruction queue, or the DMA transfer queue, or executed immediately. A macro that emits a queued operation will finish as soon as the operation is enqueued, and subsequent macros may run before the enqueued operation is executed. Some macros are used to restore synchrony and explicitly wait until the vector engine or DMA engine is finished.

Programs using the VENICE programming model follow seven steps:

1. Allocation of memory in scratchpad
2. Optionally flush data in data cache
3. DMA transfer data from main memory to scratchpad
4. Setup for vector instructions (e.g., the vector length)
5. Perform vector operations
6. DMA transfer resulting data back to main memory
7. Deallocate memory from scratchpad

The basic instruction format is `vector(MODE, FUNC, VD, VA, VB)`, where the values of `MODE` and `FUNC` must be predefined symbols, while the values of VD and VB must be scratchpad pointers and VA can be a scalar value or scratchpad pointer.

For example, to add two unsigned byte vectors located in the scratchpad by address pointers `va` and `vb`, increment the pointer `va`, and then store the result at address pointer `vc`, the required macro would be:

```
vector( VVBU, VADD, vc, va++, vb);
```

In the example, the MODE specifier of `VVBU` refers to a ‘vector-vector’ operation (`VV`) on byte-size data (`B`) that is unsigned (`U`). The vector-vector part can

```
void vector_fir(int num_taps, int num_samples,
  int16_t *v_output, *v_coeffs, *v_input){

  // Set up 2D vector parameters:
  vector_set_vl( num_taps  );             // inner loop count
  vector_set_2D( num_samples,             // outer loop count
                1*sizeof(int16_t),               // dest gap
                (-num_taps  )*sizeof(int16_t),   // srcA gap
                (-num_taps+1)*sizeof(int16_t) ); // srcB gap

  // The instruction below repeats a 1D dot-product (ie,
  // 1D accumulate) operation once for every output sample.
  // After each 1D dot product, it adds the gap values (set
  // by vector_set_2D) to each of the three pointers.
  vector_acc_2D( VVH, VMULLO, v_output, v_coeffs, v_input );
}
```

Figure 3.6: FIR Kernel Using 2D Vector Instructions

instead be scalar-vector (`SV`), where the first source operand is a scalar value provided by Nios instead of an address. These may be combined with data sizes of bytes (`B`), halfwords (`H`) and words (`W`). A signed operation is designated by (`S`) or by simply omitting the unsigned specifier (`U`). For example, computing a vector of signed halfwords in `vresult` by subtracting a vector `vinput` from the scalar variable *k* would be written as `vector( SVH, VSUB, vresult, k, vinput).`

Space can be allocated in vector scratchpad memory using `vector_malloc( num_bytes )` which returns an aligned pointer. As it is common to allocate space and immediately DMA a vector from main memory to scratchpad memory, the `vector_malloc_and_dmacpy()` function combines both operations into a single function call. The `vector_free()` call frees *all* previous scratchpad allocations; a single macro which frees all allocated memory was provided since the common case is to utilize the scratchpad for one kernel/function after which it can be reused for the next kernel/function. DMA transfers and instruction synchronization are handled by macros as well.

Figure 3.6 gives an example of using 2D vector instructions: a kernel that performs a finite impulse response (FIR) filter with 16-bit input and output data. This function can be performed in a single VENICE instruction using the reduction accumulators. Without 2D instructions, the scalar processor must repeat a 1D vector

Table 3.1: Resource Usage Comparison

	VEGAS				VENICE			
Device or CPU	ALMs	DSPs	M9Ks	F_{max}	ALMs	DSPs	M9Ks	F_{max}
Stratix IV EP4SGX530	212,480	128	1,280	–	212,480	128	1,280	–
Nios II/f	1,223	1	14	283	1,223	1	14	283
Nios II/f + V1 (8kB)	3,831	2	35	131	2,529	2.5	27	206
Nios II/f + V2 (16kB)	4,881	3	49	131	3,387	3	40	203
Nios II/f + V4 (32kB)	6,976	5	77	130	5,096	5	66	190

instruction (of `num_taps` length) using a `for()` loop of length `num_samples`. This requires the scalar processor to increment the source and destination addresses and count the number of iterations.

With 2D instructions, the loop counting and operand address incrementing is performed in hardware. The `vector_set_2D()` call specifies the number of iterations (first parameter), and three gaps for the destination and two source operands, respectively. Each gap is a distance, measured in bytes, between the end of the row and the start of the next row. Hence, a gap of 0 implies a packed 2D matrix, while a negative gap produces a sliding window effect as required by the FIR filter. The gap for operand A, the FIR coefficients, produces a stationary window. The destination gap is 2 bytes because each 1D operation is accumulated, producing a result row which contains just 1 element (i.e., the dot product).

3.5 Evalution Results

All soft processor results in this paper are measured on an Altera DE4-530 development system using Quartus II version 11.0. All software self-verifies itself against a sequential C solution.

3.5.1 Area and Clock Frequency

The overall resource usage and clock frequency for VENICE compared with VEGAS is in Table 3.1. In this table, the overall Stratix IV-530 device capacity is shown in terms of ALMs, DSP Blocks, and M9K BRAMs. It is important to note that an Altera DSP Block is a compound element consisting of two 36b multipliers. Alternatively, each 36b multiplier can be statically configured as two 18b multipli-

Table 3.2: Area Breakdown (ALMs)

	VEGAS	VENICE	Savings
Fracturable ALU	771	471	300
Control/Pipeline	1200	538	662
DMA	501	181	320
Alignment (V1)	136	116	20
Alignment (V4)	448	855	-407

ers or four 9b multipliers. Altera literature usually quotes device capacity as the number of internal 18b multipliers; there are eight 18b multipliers internally in a DSP Block, but only four can be used indepedently.

Table 3.1 also gives area and clock frequency results for several VEGAS and VENICE configurations, each of which includes a Nios II/f base processor. The V1, V2, and V4 notation indicates one, two and four 32b vector lanes, respectively. The (8kb), (16kB), and (32kB) notation indicates the scratchpad sizes used in each configuration. VENICE uses fewer ALMs and fewer M9Ks than VEGAS in all configurations. Except for the smallest V1 configuration, VEGAS and VENICE use the same number of DSP Blocks. In the V1 configuration, VENICE is using 2.5 DSP Blocks, but these are not filled to capacity; in addition to the 0.5 DSP block being available, there is also room for one 18b and two 9b multipliers. The clock frequency achieved with VENICE is also 50% higher than VEGAS. Except for the V1 DSP block anomaly, VENICE completely dominates VEGAS in area and clock frequency.

Figure 3.7 gives a more detailed area breakdown of VEGAS and VENICE. VENICE has consistently lower area for everything but the alignment network (when using multiple lanes). Precise area values for the V1 configuration are shown in Table 3.2. The savings in each area is primarily due to:

- Fracturable ALU savings is primarily due to the new parallel multiplier, which uses fewer adders and requires less input and output multiplexing. Additional savings were obtained by streamlining the ISA. Note that this savings is per lane.

- Control/Pipeline savings is primarily due to removal of the 8-entry vector address register file. Since each operand must support an auto-increment

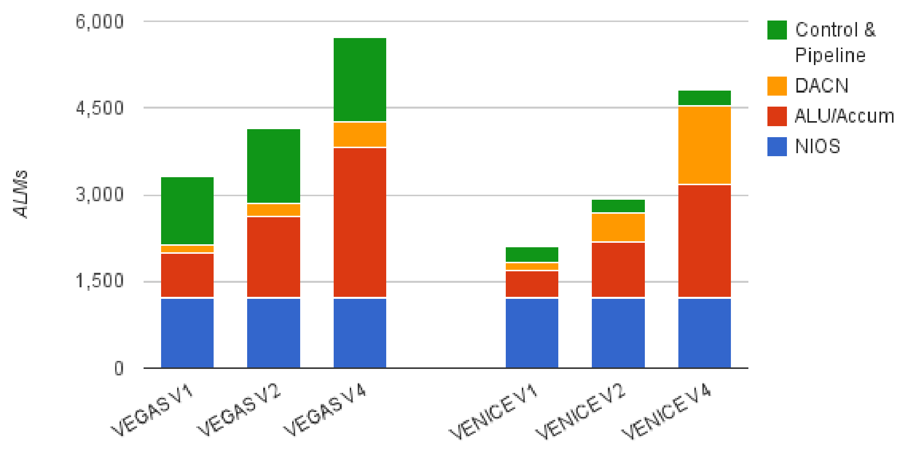


Figure 3.7: Area Breakdown (ALMs)

mode, VEGAS implemented these in ALMs and FFs.

- The VEGAS DMA engine includes an alignment network, which allows data to be loaded into an aligned position to help avoid the misalignment performance penalty. This is not necessary in VENICE, since it runs full-speed on unaligned data. Instead, VENICE performs DMA alignment in software, so unaligned DMA operations may exhibit a slowdown.

- The VEGAS alignment network was designed to scale to a large number of lanes, while the VENICE alignment network was optimized for only a few lanes.

3.5.2 Benchmark Performance

The characteristics of nine application kernels are reported in Table 3.3. The input and output data types for each kernel are shown in the Data Type column, including whether a higher precision is used during intermediate calculations. The overall input data set size is also shown, along with the size of a filter window in the Taps column. Some of these kernels come from EEMBC [17], others are from VIRAM,

Table 3.3: Benchmark Properties

	Data Type		Benchmark Properties		
Benchmark	In/Out	Intermed.	Data Set Size	Taps	Origin
autocor	halfword	word	1024	16	EEMBC
rgbcmyk	byte	-	896×606		EEMBC
rgbyiq	byte	word	896×606		EEMBC
imgblend	halfword	-	320×240		VIRAM
filt3x3	byte	halfword	320×240	3×3	VIRAM
median	byte	-	128×21	5×5	custom
motest	byte	-	32×32	16×16	custom
fir	halfword	-	4096	16	custom
matmul	word	-	1024×1024		custom

Table 3.4: Benchmark Performance

	Performance (Million elements per sec)				Speedup		
Benchmark	Nios II/f	V1	V2	V4	V1	V2	V4
autocor	0.46	5.94	11.11	18.94	12.9	24.2	41.2
rgbcmyk	4.56	17.68	21.41	22.72	3.9	4.7	5.0
rgbyiq	5.20	6.74	11.09	15.61	1.3	2.1	3.0
imgblend	4.83	77.63	145.57	251.18	16.1	30.1	52.0
filt3x3	2.11	16.82	26.95	36.42	8.0	12.7	17.2
median	0.10	0.74	1.45	2.69	7.3	14.4	26.6
motest	0.09	2.37	4.18	6.29	27.4	48.2	72.4
fir	3.32	20.11	34.95	41.67	6.1	10.5	12.5
matmul	11.7	148.20	322.22	593.75	12.6	27.4	50.6
				Geomean	7.95	13.8	20.6

and others are written by the authors. The ‘median’ benchmark performs a 5x5 median filter on a greyscale input image. The ‘motest’ benchmark performs (luma only) motion estimation of a 16x16 reference block across a 32x32 search range. Note that ‘motest’ only computes the set of sum of absolute differences (SAD) values, it does not include the time to scan and find the lowest SAD location. The ‘fir’ benchmark is a 16-tap, 16-bit FIR filter. The ‘matmul’ benchmark performs 32-bit integer dense matrix multiplication.

The Nios II/f processor was run at 283MHz with a 200MHz Avalon interconnect and 200MHz DDR2 controller (i.e., at the limit of the DDR2-800 SODIMM). The VENICE V1 and V2 configurations were run synchronously at 200MHz for everything, including the Nios II/f, VENICE engine, Avalon interconnect, and DDR2 controller, while the V4 configuration was run with everything at 190MHz.

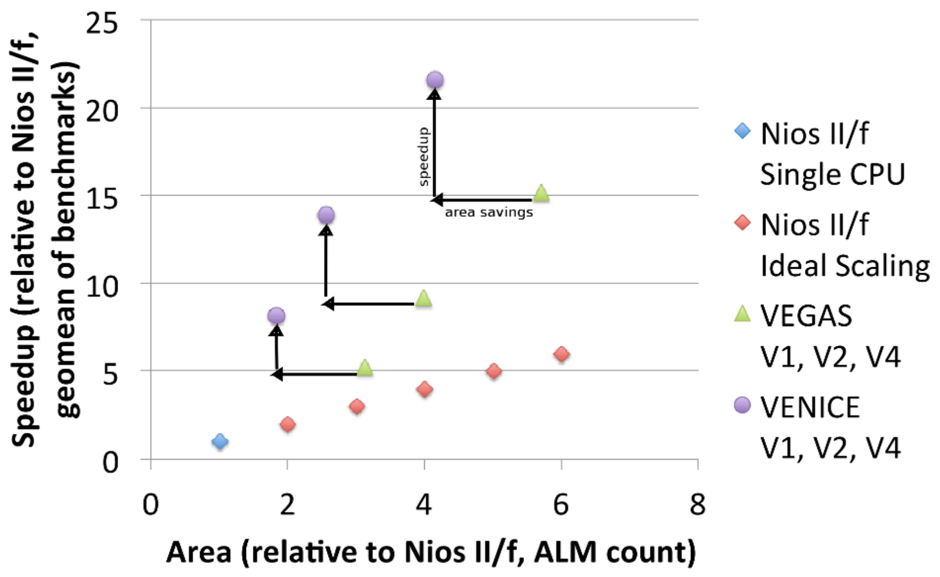


Figure 3.8: Speedup (Geometric Mean of 9 Benchmarks) vs Area Scaling

The performance of these kernels is shown in Table 3.4. The results are in units of millions of elements computed per second. For the first eight kernels, an element is in units of the output data type. This is meaningful because the amount of compute work is linear to the number of output bytes. For example, the motest kernel computes 0.09 million SAD calculations per second, where each calculation results in one byte of output. In matrix multiply, however, the amount of computation is not a linear factor of the number of output elements. For that kernel only, we report performance as millions of multiply accumulates (MACs) per second (MAC/s). The peak performance of VENICE V4 at 190MHz is 760 million MAC/s, so our matrix multiply code is running at 78% of peak performance.

3.5.3 Speedup versus Area

The VENICE processor is designed to offer higher performance and use less area than VEGAS. Figure 3.8 demonstrates this with a speedup versus area plot. The speedup and area results are normalized to the Nios II/f results, and these are used to compute a geometric mean across nine benchmark programs. Area results ac-

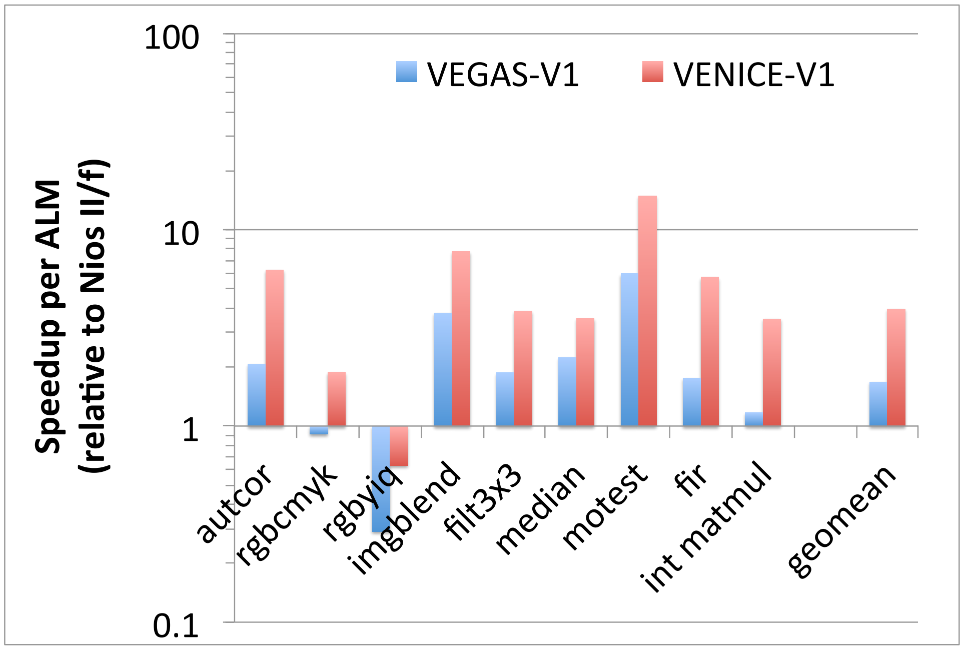


Figure 3.9: Computational Density with V1 SVPs

count for ALMs only; VENICE uses fewer M9K's but a similar number of DSP Blocks.

Results for a single Nios II/f processor are presented at the <1.0,1.0> position on the graph. Diamond-shaped data points extrapolate the performance and area if multiple Nios processor instances are created (with no interconnect overhead) and run perfectly in parallel. Triangular-shaped data points for V1, V2, and V4 configurations of VEGAS clearly outperform Nios II/f, achieving a maximum speedup of 17.2× at an area overhead of 5.7×. Circular-shaped data points for similar configurations of VENICE show it dominates VEGAS in both area and speed, achieving a maximum speedup of 20.6× at an area overhead of 4.0×.

Speedup divided by area produces a metric of performance per unit area. This can also be considered a measure of the computational density of an FPGA. Figure 3.9 compares the computational density for VEGAS and VENICE using a small V1 configuration.

Since the configuration (and therefore area) used is the same across all nine benchmarks, computational density differences between them correspond only to performance differences. Simple benchmarks such as `rgbcmyk`, `rgbyiq`, `imgblend` and `median` achieve the smallest performance increase over VEGAS. These benchmarks have large vector lengths and no misaligned vectors, and so the speedup comes mostly from the clock speed increase. Convolution benchmarks like fir and autocor benefit from the lack of misalignment penalty on VENICE. The 2D vectors accelerate autocor, motest, and fir. On matmul, using 3D vectors and the accumulators allow VENICE to achieve 3.2× the performance of VEGAS.

For one application, `rgbyiq`, the computational density falls below 1.0 on VENICE, meaning Nios II/f is better. This is because the area overhead of 1.8× exceeds the speedup of 1.3×. The limited speedup is due to a combination of memory access patterns (r,g,b triplets) and wide intermediate data (32b) to prevent overflows. However, on average, VENICE-V1 offers 3.8× greater computational density than Nios II/f, and 2.3× greater density than VEGAS-V1.

Comparing V4 configuration results (not shown), the computational density of VENICE is 5.2×, while VEGAS is 2.7× that of Nios.

3.5.4 Case Study: DCT

VENICE was designed to exploit vector parallelism, even when vectors are short. By remaining small, VENICE can be efficiently deployed in multiprocessor systems to efficiently exploit other forms of parallelism (eg, thread-level) on top of the vector-level parallelism.

In this section, we use VENICE to perform a 4x4 discrete cosine transform (DCT) with 16-bit elements on a total of 8192 different matrices. Each DCT is implemented using two matrix multiplies followed by shifts for normalization. Vectors are short, limited to four halfwords for the matrix multiply.

In Figure 3.10, the first set of bars shows the benefit of using 2D and 3D vector operations with a V2 VENICE configuration. In the 1D case, VENICE is issue-rate limited and gets poor speedup. To compute one element in the output matrix (i.e., compute the dot product of one row with a transposed row), the Nios processor must compute the addresses and issue two custom instructions. Still, it manages to get a speedup of 4.6× over the Nios II/f. With 2D instructions, an entire row

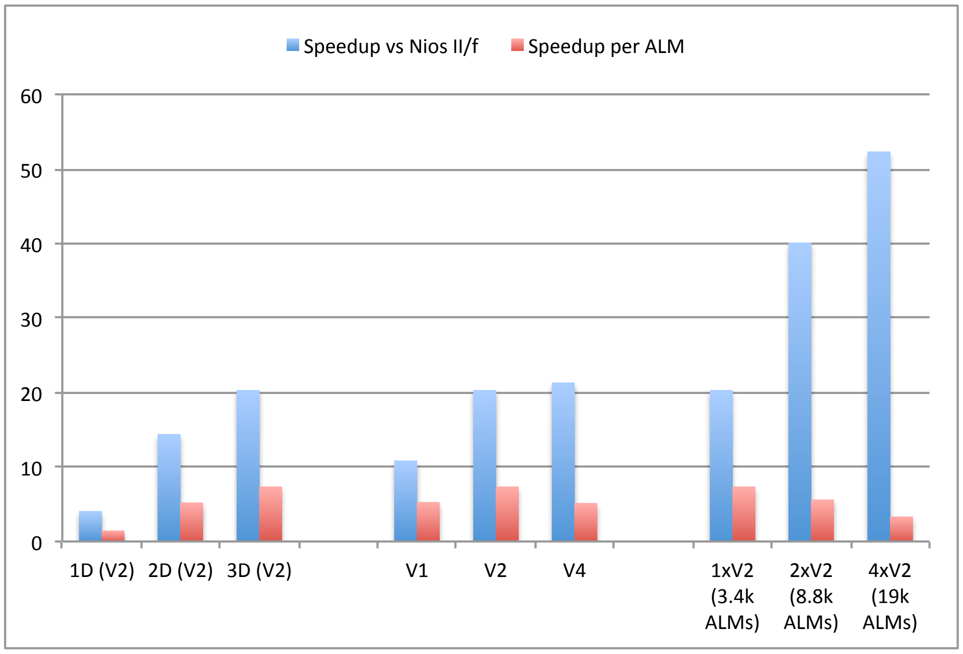


Figure 3.10: 16-bit 4x4 DCT Varying 2D/3D Dispatch, SVP Width, and # of SVPs

in the output matrix can be computed with a single VENICE instruction, giving a speedup of 3.6× over the 1D case. With 3D instructions, it can compute the entire output matrix with one instruction, running 1.4× faster than the 2D case. With 3D instructions, one DCT takes 43.9 cycles on average. The theoretical minimum is 40 cycles, giving 91% ALU utilization.

The second set of bars shows performance scaling from one to four vector lanes. A V1 VENICE can achieve a speedup of 10.8× Nios II/f, and the benchmark scales well to V2 (1.87× faster than V1). However, V4 is only 1.05× faster than V2. During the matrix multiply step, the maximum vector length is 4 halfwords, so it does not benefit from more than 2 lanes; only the normalization step benefits.

The final set of bars shows the results of a simple multiprocessor VENICE system, consisting of one, two, and four V2 VENICE processors. This system was constructed by connecting together several Nios II/f processors, with VENICE accelerators, using the default Altera Avalon fabric. Scaling from one to two processors leads to a system that is 1.98× faster. Scaling from two to four processors achieves a speedup of only 1.31×. This is lower than expected, because we are not yet bandwidth-limited (the memory bandwidth limit is 8 cycles per DCT, but we are achieving only 17.0 cycles per DCT at this point). Further investigation is required.

The set of red bars in the figure indicates speedup per ALM. The multiprocessors use a large number of additional ALMs in the Avalon fabric. This greatly deteriorates the speedup-per-ALM advantage, and suggests that Avalon is not the most efficient way to build a VENICE multiprocessor.

3.6 Summary

This chapter investigates ways to reduce SVP area and increase SVP performance by targetting modest performance applications and using FPGA-specific optimizations. VENICE is designed to accelerate applications that require higher performance than a scalar soft processor can provide, but do not need to scale to a large, expensive FPGA.

Speedups over 70× a Nios II/f were demonstrated, with 3.8× to 5.2× better performance per logic block from V1 to V4. VENICE is also both smaller and faster than the VEGAS, the previous best SVP at the time of development. VENICE offers over 2× the performance per logic block of VEGAS while using fewer BRAMs and the same number of DSP blocks. The use of 2D and 3D instructions allows for high ALU utilization (91% in DCT) even with small vector lengths.

VENICE can be used on its own to implement applications with modest performance needs, giving FPGA designers a tool that is quick to iterate designs with. Additionally, VENICE could be conceived as a building block in a heterogeneous systems such as a vector/thread hybrid solution [38].

Chapter 4

Wavefront Skipping on Soft Vector Processors

Let us redefine progress to mean that just because we can do a thing, it does not necessarily mean we must do that thing. — Federation President, Star Trek VI

In the previous chapter, we discussed how VENICE can replace RTL for modest-performance data-parallel applications with much lower overhead than previous SVPs. This gives designers a tool for quickly developing applications that have straightforward and regular data parallelism. To enable more applications on SVPs, we investigated broader classes of algorithms. This chapter presents a method for accelerating certain irregular data parallel applications, such as face detection. We utilize the cheap and abundant FPGA BRAMs to avoid wasted computational cycles.

4.1 Introduction

SVPs process multiple data elements in parallel; the data processed in a single cycle is called a wavefront. If the vector length is longer than the number of lanes, the instruction will sequentially process one wavefront per cycle until the whole vector instruction has completed. For algorithms that use conditional execution, i.e., branching, the vector processor must execute both paths of the branch and

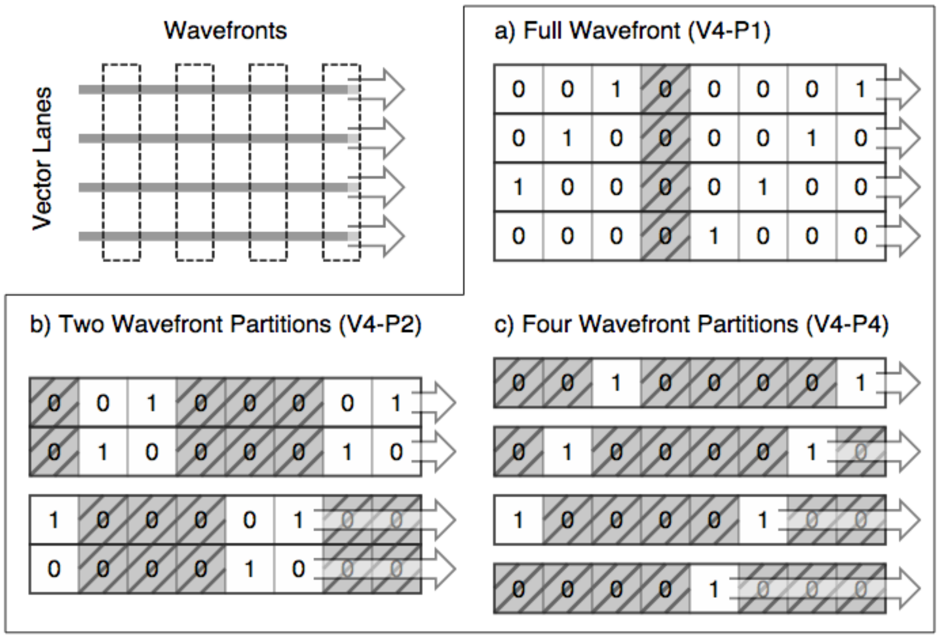

Wavefronts
Vector Lanes
a) Full Wavefront (V4-P1)
b) Two Wavefront Partitions (V4-P2)
c) Four Wavefront Partitions (V4-P4)

Figure 4.1: Wavefront Skipping on a 4 Lane SVP

mask off writes for elements not on the active branch. This can result in a large portion of execution unit results that are not used, especially for algorithms that can stop processing some data elements early depending on a conditional check.

In order to speed up these conditional algorithms, masked-off elements must not utilize execution slots. It is possible to compress vectors in order to remove masked-off elements [63], but we will show later that this is costly for algorithms such as stencil filters. Rather, we would like to skip over elements without rearranging data, which led us to implement wavefront skipping. In wavefront skipping, a wavefront where all the elements are masked off is skipped. We also implemented partial wavefront skipping, by which the wavefront is divided into partitions that can skip independently. This fine-grained skipping can lead to performance increases, at the cost of requiring more resources.

Figure 4.1 helps illustrate how wavefront skipping works. The values shown are the mask bits corresponding to each data element; a zero indicates that element is masked off. In Figure 4.1a one of the eight wavefronts can be skipped since all of its elements are masked off. Normally this instruction would take eight cycles to process, but with wavefront skipping it can complete in seven. Figure 4.1b shows two wavefront partitions; the finer granularity means more partial wavefronts can be skipped, and so the instruction can run in only four cycles (the partition with only three sub-wavefronts of active data ends up being idle during the last cycle). Figure 4.1c shows four wavefront partitions, at which point the partition size is a single element, and the instruction can execute in two cycles.

This chapter gives the first implementation of wavefront skipping on SVPs. It uses a different approach than is used on fixed vector processors, where the mask register is read out in parallel and the number of leading masked-off elements is computed each cycle. Instead, we take advantage of the relative abundance of BRAMs on FPGAs by computing wavefront offsets beforehand in a setup instruction and storing them. Additionally, we implement partitioned wavefront skipping, where instead of entire wavefronts skipping together, partial wavefronts (down to individual bytes) can skip by different amounts. To our knowledge this is the first implementation of this idea in any vector processor. The full wavefront skipping implementation requires no more than 5% increase in area and a single BRAM and achieves speedups of up to 3.2× for the early exit algorithms we have tested. The addition of partial wavefront skipping provides additional gains of up to 5.3× but requires up to 27% additonal area. While this might not be a good tradeoff in a hard vector processor, it may make sense as a configuration option for an SVP. Because the SVP can be configured differently for each design, partial wavefront skipping may make sense if the design has some unused BRAMs within the device or the algorithm is particularly sensitive to wavefront partitioning.

4.2 BRAM Based Wavefront Skipping

This work builds on the VectorBlox MXP [59], a VENICE-like commercial SVP. An architectural diagram of MXP can be seen in Figure 4.2. Prior to this work MXP supported predicated execution through conditional move (CMOV) instructions only. This differs from wavefront skipping in that the CMOV operation does both a condition check and move in the same instruction, and it operates on the entire vector instead of just the valid wavefronts. The CMOV instruction checks

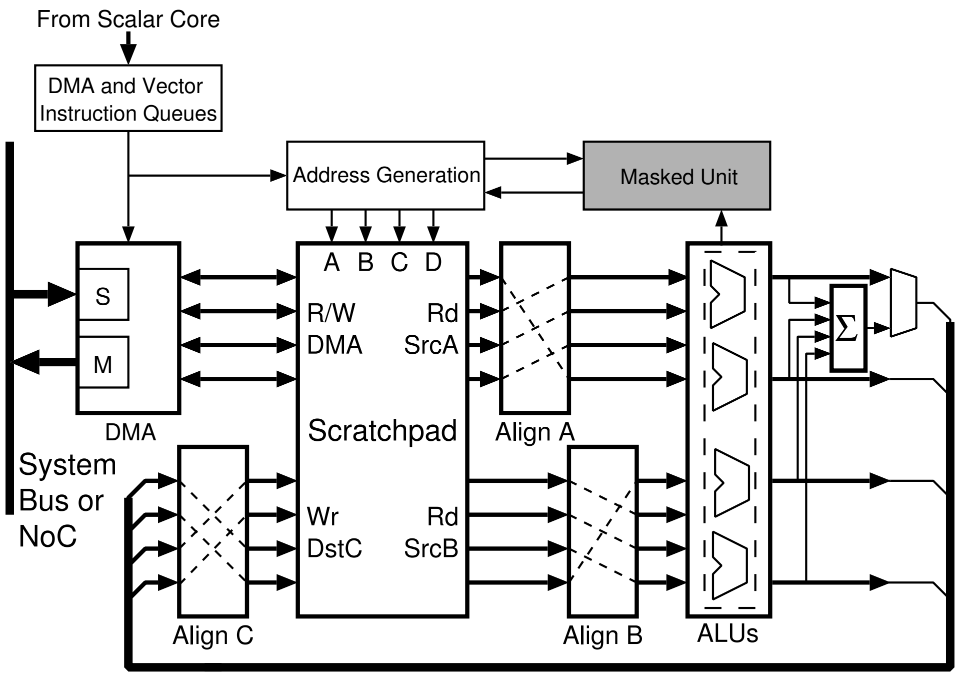


Figure 4.2: VectorBlox MXP with Four 32-bit Lanes

conditions using a flag bit associated with each element; each 9-bit wide scratchpad BRAM stores 8-bits of data and one flag bit. The flag bit is set by earlier vector instructions; for instance an add instruction stores overflow while a shift right stores the bit shifted out. The flag bit is used along with whether the element is zero or negative to perform several different CMOV operations. The most common CMOV operations first perform a subtraction-based comparison; the result is predicated based on whether the result is less than zero (LTZ), less than or equal to zero (LTE), etc. The CMOV hardware is part of the ALUs.

4.2.1 Full Wavefront Skipping

Figure 4.3 gives an example of how to use wavefront skipping to perform strided operations, such as operating on every fifth element as in Figure 4.1. The first loop sets every fifth element of v_temp to be 0 (this could be done with vector divide or modulo instructions if the SVP supports them, but is shown in scalar code for

```
#define STRIDE 5

//Toy functition to double every fifth element
//in the vector v_a
void double_every_fifth_element( int *v_a,
                                 int *v_dest,
                                 int *v_temp
                                 int vector_length )
{
  int i;

  //Initialize every fifth element of v_temp to zero
  for( i = 0; i < vector_length; i++ ) {
    v_temp[i] = i % STRIDE;
  }

  //Set the vector length which will be processed
  vbx_set_vl( vector_length );

  //Set mask for every element equal to zero
  vbx_setup_mask( CMOV_Z, v_temp );

  //Perform the wavefront skipping operation:
  //multiply all non-masked off elements of v_a by 2
  //and store the result in v_dest
  vbx_masked( SVW, MUL, v_dest, 2, v_a );
}
```

Figure 4.3: Code Example: Double Every Fifth Element of a Vector

clarity). The vbx_setup_mask instruction takes as input a comparison operation (CMOV_Z, or use the CMOV hardware to test for equal to zero) and a pointer to the vector operand (v_temp, which is a pointer to a vector of 32-bit data).

After the mask is set, the vbx_masked instruction is executed. The 'SVW' type specifier indicates that it operates on scalar (2) and vector (v_a) inputs and that the values are words (32-bit). The elements in the destination vector (v_dest) that are masked-off will not be written, while the valid elements will be set to the corresponding element of v_a multiplied by 2. Although this is a trivial example and is not faster than using a conditional move, complex algorithms that reuse the mask several times and/or have sparse valid elements can give significant speedup.

In order to support wavefront skipping, we need to alter the scratchpad address generation logic of MXP. For normal vectors, the addresses of the operands are incremented by one wavefront every cycle. So, with a V2 MXP (meaning it has two 32-bit vector lanes), each address is incremented by 8 bytes (the width of one wavefront) each cycle. To support wavefront skipping, we want to add a variable number of wavefronts to the operand addresses depending on the value of a mask that was observed earlier. We accomplish this by storing the offsets of wavefronts that have valid elements in a BRAM inside the 'masked unit'. Because a wavefront contains multiple elements, we also have to store a valid bits for each element. These are stored per byte, and during subsequent execution become byte enables upon writeback. Finally, the length of a wavefront skipped vector may be shorter than the length of the full vector, so we have to mark the last wavefront. We use an extra bit to mark the wavefront that is the end of the skipped vector. This is an implementation detail that was convenient in our design and could be replaced by storing the number of wavefronts in a register.

BRAMs are limited in depth, however, and MXP's scratchpad can hold vectors of any length. To allow for wavefront skipping of vectors of the whole scratchpad length, the masked unit would need a BRAM as deep as the scratchpad itself. This would be wasteful for many applications, so we allow the user to configure a maximum masked vector length (MMVL) as an SVP build option. The minimum depth of BRAMs in the Stratix IV FPGAs we tested on is 256 words, so setting MMVL to be less than 256 will result in underutilized BRAMs. The fact that wavefront skipping instructions have a length restriction must be known by the programmer. In real applications data does not fit entirely in scratchpad memory and so is operated on in chunks (also known as strip-mining [70]) even without an MMVL; the MMVL only affects the size of the chunks.

To generate the wavefront offsets, we added the mask setup instruction. This mask setup instruction uses the conditional move logic already in MXP to generate the valid bits set in the mask BRAM. Figure 4.4a shows the data written the mask BRAM during the mask setup instruction. For each wavefront processed, the offset and valid bits are sent to the masked unit. If no valid bits are set, as in wavefront 4 (data elements 16-19), the masked unit does not write into its BRAM. If any of the valid bits are set, the masked unit writes the current offset and the valid bits to its current BRAM write address and increments the write address. The final wavefront causes the end bit to be written to the mask BRAM along with its offset and byte enables, provided there are any byte enables set. In this example the final wavefront has valid byte enables, but in instances where the final wavefront has no

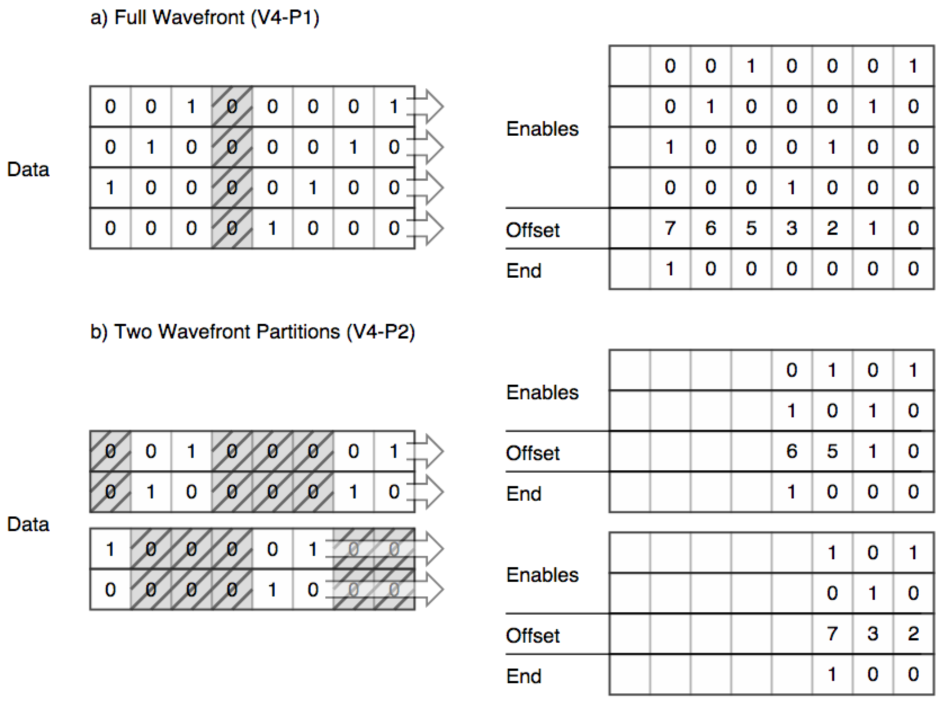

a) Full Wavefront (V4-P1)
Data
Enables
Offset
End
b) Two Wavefront Partitions (V4-P2)
Data
Enables
Offset
End
Enables
Offset
End

Figure 4.4: Data Written to Mask BRAM (Every Fifth Element Valid)

byte enables set we redo the write of the last valid wavefront with the end bit set.

In the case of algorithms with multiple early exit tests, the set of elements being masked off grows with each test. In this case, we wish to generate a new mask that is based on the previous mask, except that a superset of elements will be masked off. In this situation, it is desirable to have a masked 'setup_mask' instruction. This is a trivial extension of the normal mask setup instruction; instead of the wavefront numbers progressing linearly, they are taken from the output of the masked unit. In this way the number of valid elements can decrease until either the algorithm finishes or no more valid elements are left. When no valid elements are left, any wavefront skipping instructions will execute as no operations (NOPs), taking only a single cycle in the vector engine to execute. However, dispatching the vector instructions and calculating vector addresses still takes time in the scalar engine, so it is desirable to exit the algorithm completely if no elements are valid. For this case, we have a mask status register that the scalar core can query to determine if any valid elements are left.

4.2.2 Wavefront Partitioning

So far we have only discussed skipping whole wavefronts. When dealing with wide SVPs, it may be of little value to skip whole wavefronts, since it will be rare that the *entire* wavefront can be skipped. As a trivial example, the code from Figure 4.3 will skip 4/5 of wavefronts on a V1, 1/5 of wavefronts on a V4, and no wavefronts on wider MXP's (every wavefront will contain at least one valid word). To speedup this code on a wide MXP, we need to support skipping at a narrower granularity than the wavefront. We support this by partitioning the wavefront into narrower units which each have separately controllable BRAMs; for instance, a V4's masked unit can have 1 BRAM (whole wavefront skipping), 2 BRAMs (pairs of lanes can skip together), or 4 BRAMs (each lane can skip individually). If halfword or byte operations are frequently done, the masked unit can even have 8 or 16 BRAMs, respectively. Each BRAM requires the same number of bits to store offsets and the end-of-masked-vector bit, but the byte-enable bits are specific to each partition.

Figure 4.4b shows the data written to the 2 mask partition BRAMs for the code from Figure 4.3. This mask will take 4 cycles to execute (the maximum of the depth written to all of the partitions), compared to the 7 cycles needed for the single partition shown in Figure 4.4a.

One complication with wavefront partitioning in our architecture is the mapping from partitions to scratchpad BRAMs. In an architecture with a simple register file or a scratchpad that did not allow unaligned accesses, each BRAM would get its offset from a fixed partition; in a V2 with two partitions, Lane 1 would get its offset from Partition 1 and Lane 2 from Partition 2. However, our scratchpad supports unaligned addresses. This means that partitions are not directly associated with a scratchpad BRAM; instead, depending on the alignment of the vector operands a scratchpad BRAM address may come from any of the partitions. This means we had to implement an offset mapping network to map wavefront offsets to scratchpad BRAMs. Note that there is no such overhead with full wavefront skipping (single partition) because the same offset goes to all BRAMs, which is just a broadcast of the offset which requires no additional logic.

```
for stage in classifier:
    for row in image:
        vector::init-mask

        for feat in stage:
            for rect in feat:
                vector::feat.sum += vector::image[rect]

            if vector::feat.sum > feat.threshold:
                vector::stage.sum += feat.pass
            else:
                vector::stage.sum += feat.fail

        if vector::stage.sum < stage.threshold:
            vector::update-mask
        if all elements masked:
            exit
```

Figure 4.5: Pseudocode for Viola-Jones Face Detection

4.2.3 Application Example: Viola-Jones Face Detection

Figure 4.5 gives high level pseudocode for one of the benchmarks we have implemented, Viola-Jones face detection. Face detection attempts to detect a face at every (x,y) pixel location in an input image. To vectorize this, we will test several possible starting locations in parallel: a 2D vector of starting locations characterized by (x+i,y+j). In this way, we will be testing for $i * j$ face locations in the vector simultaneously.

Viola-Jones face detection must compute several thousand values, called Haar features, for each pixel location. Features are grouped into stages. The features in a stage must pass a threshold test if a face is to be detected at that location. If any stage fails this threshold test, there is no point in testing further features at that location. Hence, each stage is an 'early exit' test for each pixel starting location.

Features are calculated in-order across this vector of locations. When utilizing predicated instructions, the algorithm must continue processing each feature for all locations, even though some locations may have already failed an earlier threshold test and do not need to participate in further calculations. If even a single location in the 2D vector may still have a face, the algorithm must continue to compute for all locations. Here, long vector lengths work less efficiently. In this model, parallelism due to vectorization runs against the ability to exit early; longer vectors are

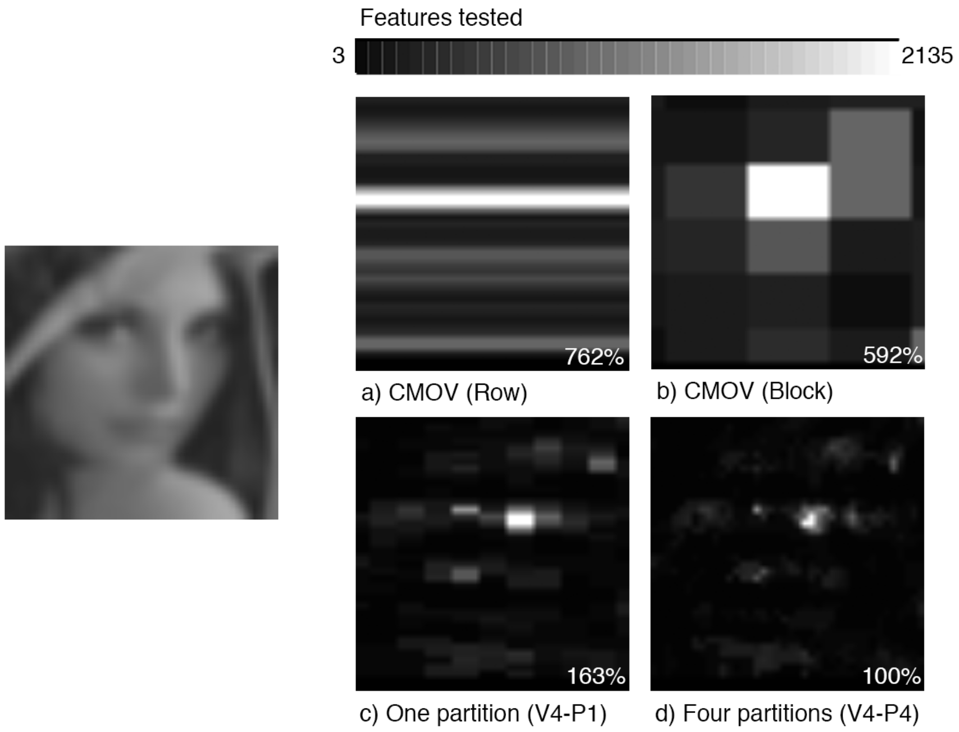


Figure 4.6: Haar Features Calculated at Each Candidate Face Location for Different Groupings (Percentage Work Done is Calculated Relative to Minimum)

more likely to contain locations requiring computation of many features, leading to extensive processing for all elements in the vector. By using wavefront skipping, we can avoid processing elements that are already known not to contain a face. Further features are computed only for those locations still in question, effectively shortening the vector length to the relevant elements.

The only differences between the predicated/CMOV version (without wavefront skipping) and the wavefront skipping version are the init-mask and update-mask instructions and the low level details of the test checking whether all elements are masked. Porting an existing predicated algorithm to wavefront skipping is therefore straightforward and requires minimal effort.

Figure 4.6 compares the amount of work done for different strategies. Each plot shows the number of Haar features calculated at each candidate face location, from the minimum of three (in black) to a maximum of 2135 (in white). Different strategies result in doing different amounts of work at each location; the less work that is done (less white and more black in each heatmap) the faster execution can be. The total work is also calculated for each strategy and shown as a percent relative to the minimum. The fact that the difference between the minimum and maximum number of features that must be calculated is three orders of magnitude shows the need for some smarter form of predication than simply computing the worst case on all pixels.

In order to run the application on an SVP, groups of pixels must be operated on as parallel vectors. Larger groupings provide more parallelism, but without wavefront skipping (only using CMOV instructions) every pixel in a group must pass the maximum number of Haar feature tests of any pixels within the group. It can be difficult to balance the CMOV implementation; it is tempting to use the minimum vector length possible to reduce the amount of work done, but below a certain level the lack of parallelism means runtime actually increases. The results in Figure 4.6 are from a 4 lane MXP implementation, and we found the runtime optimal vector length for CMOV implementations empirically.

A naive vector layout would be to have every row be a vector (Figure 4.6a), which does almost $8\times$ as much work as the optimal. A slightly better method uses rectangular blocks to get better spatial locality (Figure 4.6b); getting sufficient parallelism still requires vectors long enough that almost $6\times$ as much work as the minimum is done. In contrast, simply by changing the implementation to use wavefront skipping, the effective grouping is the wavefront partition width (Figure 4.6c). Wavefront skipping removes the need to profile an application to determine the best vector length to use; the wavefront skipping implementation can use the longest vector length possible and will always achieve at least the performance of the CMOV implementation. Finally, a fully partitioned design performs the minimal number of feature tests (Figure 4.6d), which is the same number of features that a scalar implementation would require.

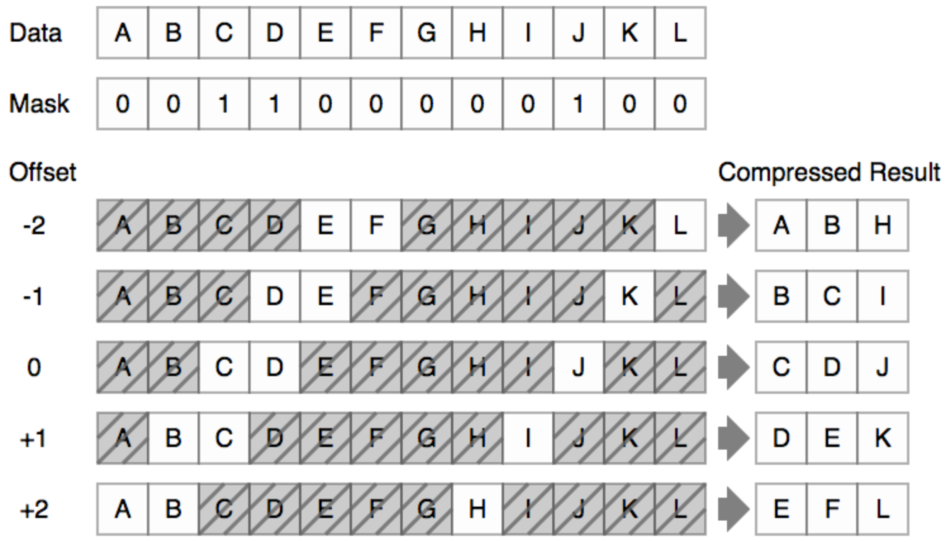


Figure 4.7: Vector Compress Operations Needed for 5x1 Stencil Filter

4.2.4 Comparison with Vector Compress

An alternative method mentioned in Section 2.5 is a vector compress operation. The compress operation removes masked-off elements from a vector, creating a shorter vector as a result. Figure 4.7 demonstrates how this would work for a simple 5x1 stencil filter. The stencil filter takes as inputs shifted versions of the input data, from an offset of -2 to +2. Before running the stencil filter, each of the shifted versions must be separately compressed. The five versions of the data with different offsets must be compressed into five scratchpad locations, each of which uses as much storage as the original in the worst case. After compressing the inputs, subsequent vector operations can operate on the shorter vectors at full speed.

The drawbacks of requiring a compress instruction and additional storage space for each input make it impractical in many stencil filter algorithms. For instance, on the Viola-Jones face detection application the amount of work and extra space needed would be prohibitive. The Haar feature tests operate on a sliding 20x20 window; each feature selects a subset of the window. Compressing the inputs would require compressing each location, or 400 compress operations and 400

Table 4.1: Resource Usage

MXP Vector Lanes - Wavefront Partitions	Logic ALMs	Memory M9Ks	DSP Blocks	Total Area eALMs	Area Increase % Over CMOV	fmax MHz
(CMOV) V1-P0	4,697	96	2	7,502		206
V1-P1	5,034	97	2	7,877	5.0%	200
(CMOV) V4-P0	9,732	152	5	14,243		173
V4-P1	10,210	153	5	14,750	3.6%	176
V4-P4	13,276	156	5	17,902	25.7%	183
(CMOV) V32-P0	63,334	414	33	76,198		144
V32-P1	63,027	418	33	76,005	-0.3%	144
V32-P4	78,698	422	33	91,791	20.5%	149
V32-P32	83,220	446	33	97,002	27.3%	146
Nios II/f	1,370	19	1	1,945		283
DE4-230 Maximum	91,200	1,235	161	131,434		–

temporary vectors. The wavefront skipping method does not need to manipulate the input data, meaning MXP only needs one copy of the chunk of the image being scanned.

Compress operations also need the inverse vector expand operation to restore the output. This is less critical for algorithms like stencil filters where the number of outputs is smaller than the number of inputs. Note that the compress operation is only of use when the vector being compressed is used multiple times, because compressing the vector takes an extra instruction. Setting up a mask for wavefront skipping also takes an extra instruction, but the same mask can be used for multiple offsets in a stencil filter.

4.3 Results

Our results were obtained using Altera Stratix IV GX230 FPGAs on the Terasic DE4-230 development board. FPGA builds were done using Quartus II 13.0sp1. We used one 64-bit DDR2 channel as our external memory.

4.3.1 Area Results

Table 4.1 shows the resources used and maximum frequency achieved for various configurations of MXP. In addition to the actual FPGA resources used (ALMs, M9Ks, and DSP blocks), we also report the area in equivalent ALMs (eALMs) [71]. By factoring the approximate silicon area of all of the resources used, eALMs are a convenient way to compare architectures that use different mixtures of logic, memory, and multipliers. For the Stratix IV family of FPGAs each M9K memory block counts as 28.7 eALMs and each DSP Block counts as 29.75 eALMs (in addition to each ALM, which counts as one eALM).

All MXP configurations shown have a maximum masked vector length (MMVL) of 256 wavefronts (the effect of changing MMVL will be investigated later). MXP configurations are listed as V*X* P*Y* where *X* is the number of 32-bit vector lanes and *Y* is the number of mask partitions. P0 means that masked instructions are disabled and only CMOV instructions can be used for conditionals. V1s are configured with 64kB of scratchpad memory, V4s are configured with 128kB of scratchpad memory, and V32s are configured with 256kB of scratchpad memory. The area numbers for MXP include the Nios scalar core used for control flow and vector instruction dispatch, as well as prefix sum and square root custom vector instructions that are used in the face detection benchmark.

The eALM area penalty is 5.0% from no wavefront skipping to wavefront skipping with full wavefront skipping on a V1. Most of this area penalty is in ALMs, as only a single extra BRAM is used and no extra DSP Blocks. Since the number of ALMs for full wavefront skipping is roughly constant with respect to the number of lanes, the overhead drops to 3.6% on a V4 and less than 1% on a V32. For multiple partitions, the area overhead is much higher (up to 27.3% more eALMs in the V32-P32 case), because of the partition to scratchpad BRAM mapping needed (as explained in Section 4.2.2). We implemented this mapping using both a switching network and multiplexers; the area results were similar but the multiplexer implementation had lower latency and so was used for our results.

4.3.2 BRAM Usage

BRAM usage is minimal for the V1 and V4 as each mask partition uses just 1 BRAM. For the V32, a single partition uses 4 BRAMs since it has to store 128-bits worth of byte enables, eight bits of offset, and one end bit, which fits in four 36-bit wide M9Ks. With more partitions, the number of byte enables per partition is reduced, so that the V32-P32 only uses one BRAM per partition. The

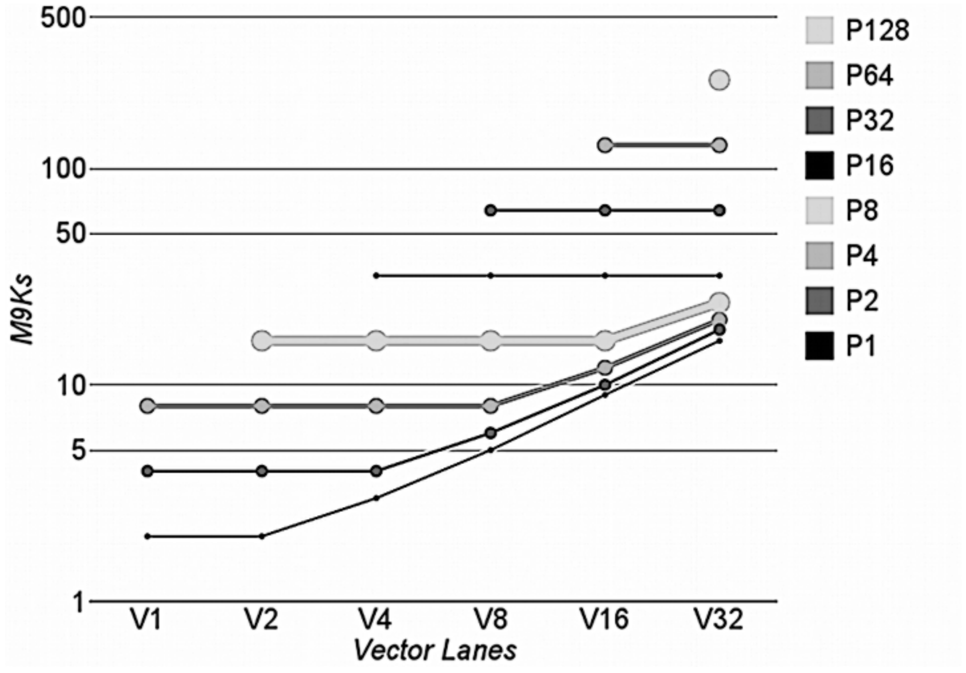


Figure 4.8: BRAM Usage vs Wavefront Partitions (MMVL = 1024)

masked unit has no critical timing paths; increasing the number of BRAMs used does make the placement job harder for the CAD tools, but the critical path always remains outside the masked unit. Since the contribution of this work is not affecting f_{max} directly, we decided to remove this variability from our results by running all benchmarks shown here at 125MHz.

Figure 4.8 shows how many BRAMs are used in our partitioned wavefront skipping with a maximum masked vector length (MMVL) of 1024 wavefronts. With a small number of partitions, as the vector processor gets wider, multiple BRAMs per partition are required to store the byte enables. With a large number of partitions, the byte enables are divided up into small enough chunks that they always fit in a single BRAM per partition. There is a wide range in the number of BRAMs that can be used by MXP. A designer has the freedom to allocate BRAMs on the FPGA device to achieve the best performance with their algorithm, either by increasing scratchpad capacity, or by increasing flexibility for conditional execution.

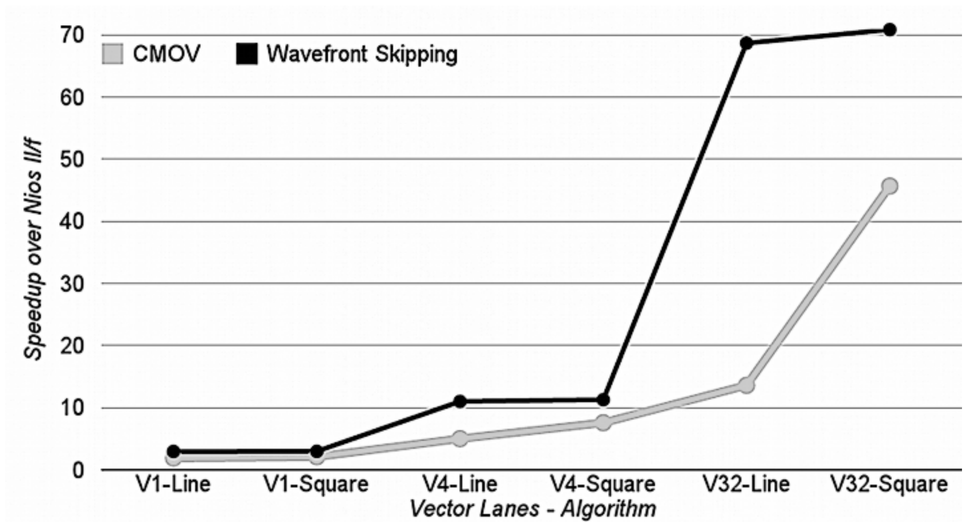


Figure 4.9: Mandelbrot Benchmark

4.3.3 Mandelbrot Benchmark

Figure 4.9 shows speedup results for computing the Mandelbrot set (geometric mean of 23 frames shown in a visual demo). Results are shown compared to a scalar version run on the Nios II/f as a baseline. The Mandelbrot computation iterates a complex valued equation at each pixel until either the pixel reaches a set condition (the early exit) or else a maximum iteration count is reached. Without masked instructions (CMOV configurations) we can only exit early after all pixels in the group agree to exit early. We show results for two versions of the algorithm: the 'line' implementation which naively computes pixels in raster order (row by row), and the 'square' implementation which computes a 2D block of pixels at a time. The 'line' implementation is more straightforward. However, since the early exit pixels are correlated spatially, selecting a group of pixels closer together in a block results in less wasted computation and therefore higher performance.

For the CMOV configurations, the difference between the line and square implementations is vast; for V32 the square implementation has a speedup that is almost a 4$\times$ higher than the line implementation. With the masked implementations, the difference between line and square are much less–at most 10%. The masked implementations always outperform the CMOV implementations. Parti-

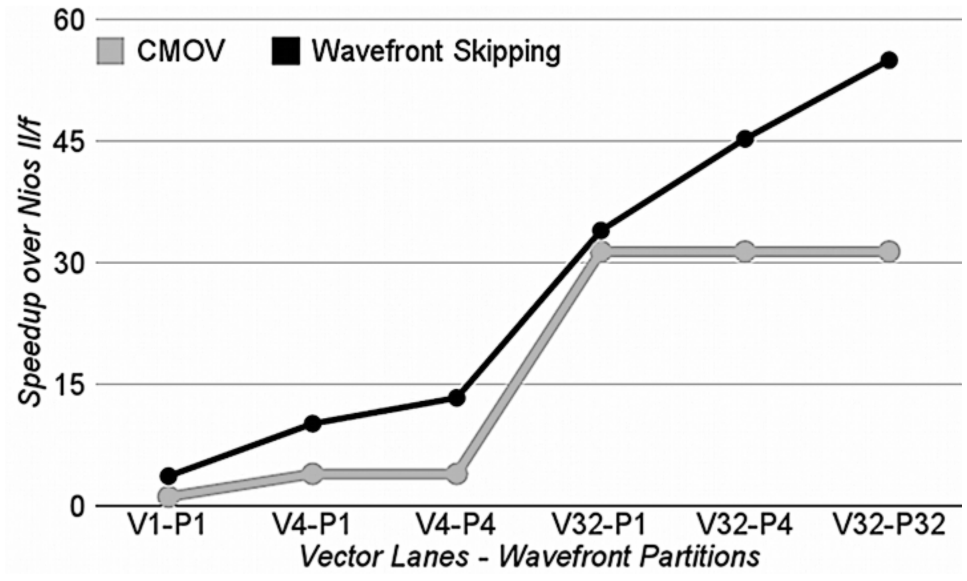


Figure 4.10: FAST9 Feature Detection

tioned wavefront skipping has little effect in Mandelbrot and so is not shown; the data is grouped together in such a way that wavefronts exit at the same or nearly the same time. The CMOV implementation also uses a vector length determined by profiling; too long and early exits are not helpful, and too short and instruction dispatch rate and data hazards reduce performance. The masked implementation just uses the maximum vector length it can, which is either determined by the MMVL of the masked instructions or the size of the scratchpad and number of vectors needed. Between not having to profile to find the best vector length, and being able to use the naive 'line' implementation with little performance penalty, the masked implementation is likely much easier for the programmer.

4.3.4 FAST9 Feature Detection

Figure 4.10 shows the results for FAST9 feature detection. FAST9 determines if a pixel is a feature by examining a circle of pixels around the target location and looking for a consecutive series of pixels that are darker or lighter by a threshold. An early exit can happen if certain conditions hold, such as if neither of two pixels on opposite sides are above or below the threshold. All operations are byte-wide, taking advantage of MXP's subword-SIMD which executes byte operations 4× faster than word operations. While running FAST9 on simple input data with the CMOV code, we found checking early exit (as done in Mandelbrot) to be helpful. However, on a real image (Lenna), the early exit code only helped on a V1. Thus, the CMOV code is doing the full calculation for all pixels on V4 and V32. In contrast, the masked code is able to achieve speedup by skipping at a more fine-grained level. However, if the mask is too coarse, such as at V32-P1 which is 128 bytes wide, very little is gained. Only with 4 or 32 partitions is speedup achieved. It may be possible to gain even more speed by using byte-wide partitions, but our MXP test configurations had a minimum partition size of one 32-bit lane wide.

4.3.5 Viola-Jones Face Detection

For Viola-Jones face detection, we used the same algorithm settings as those used in an FPGA implementation created in RTL [10]. No reference image was specified in the publication, so we used the standard 'Lenna' image. Their hardware implementation with 32 PEs was able to achieve 30fps. Our SVP implementation has an inner loop containing 9 or 13 instructions (depending on the Haar classifier), instead of being fully pipelined as in a hardware implementation, so it is expected to be somewhat slower. Also, our implementation includes full data movement, from input image to frame buffers driving a DVI display.

Figure 4.11 shows the results for CMOV only and masked implementations including partial wavefront skipping. The masked implementations with a single partition are up to 3× faster than CMOV, and partitioning increases it to up to 4.1×. For a V4 processor, partitioning gives a 29% improvement from P1 to P4, and for a V32 processor, partitioning gives a 35% improvement from P1 to P4 and an additional 22% improvement from P4 to P32 for an overall 65% improvement. While the impact of partitioning is not as dramatic as the impact of switching from CMOV instructions to wavefront skipping, it provides a performance increase without having to rewrite software.

Figure 4.12 presents the results as speedup versus area (eALMs). MXP without wavefront skipping is roughly the same the performance per area as a Nios II/f (which in practice can not scale ideally to achieve higher performance). Wavefront

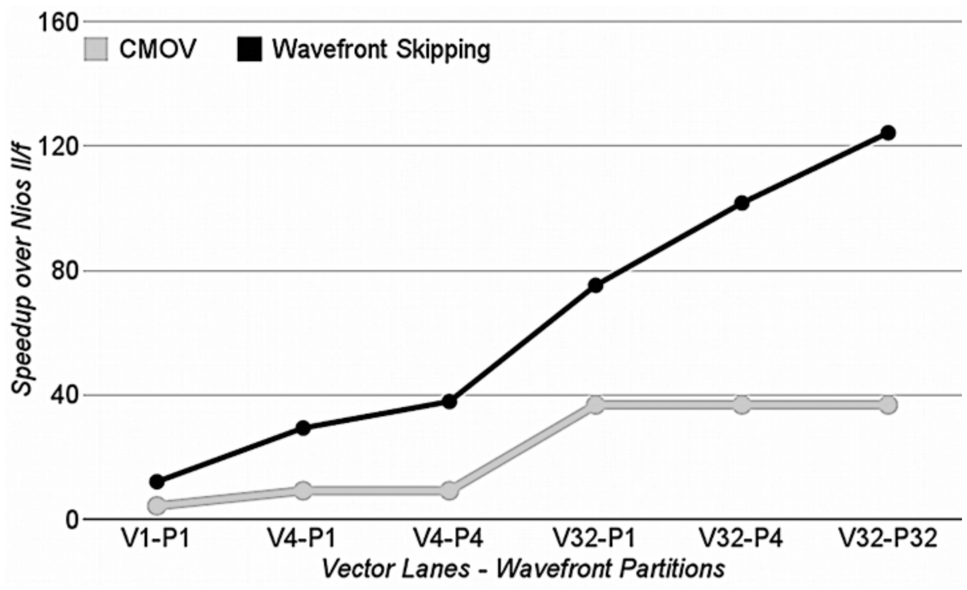


Figure 4.11: Viola-Jones Face Detection Speedup

skipping provides significantly higher performance per area, since the area impact is minimal compared to the performance gained. The highest performance per area is 4.1× that of Nios II/f (37.8× faster with 9.2× more area for the V4-P4 configuration).

With our fastest configuration, we achieve 3 frames per second, or about 1/10th that of the previous hardware result, on the Lenna input image. Not all of the processing time of our implementation is in Haar classifier tests; there is some overhead such as loading data from memory and resizing images. To test the amount of overhead we used a blank image (where all pixels exit early), which achieved 60 frames per second.

4.3.6 MMVL Tradeoffs

So far we have examined tradeoffs in additional logic and BRAMs when using multiple partitions, but it's also possible to use more BRAMs to allow for longer masked vector instructions. Each BRAM needs to store an offset of width $log_2(MMVL)$ as well 4 byte-enables per lane and an end bit. With multiple partitions, the byte enables are split between BRAMs, but the other data is replicated.

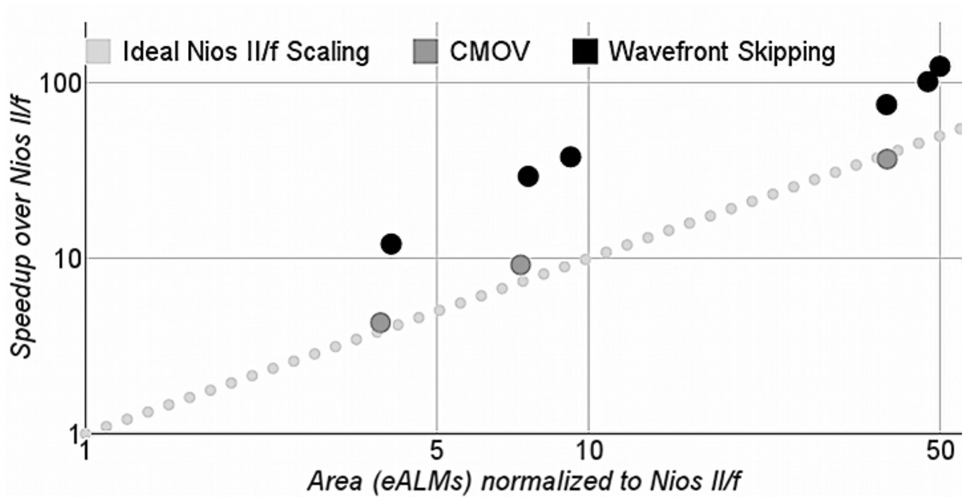


Figure 4.12: Viola-Jones Face Detection Speedup Vs Area

Figure 4.13 shows the BRAM usage for two configurations of MXP as the MMVL is varied. As explained in Section 4.2, we default to a MMVL of 256 wavefronts since that is the minimum depth of M9Ks. Hence, a MMVL of 128 or 256 wavefronts almost always has the same resource usage, except on a V16 with one partition where the width of the BRAM data goes from 72 and 73 bits and can no longer fit in two 36-bit wide M9Ks. Increasing MMVL to 512 does have a cost in BRAMs once the width of the BRAM data gets past 18-bits. An MMVL of 1024 requires even more resources, as BRAMs are only 9-bits wide at that depth.

Figure 4.14 shows the results of changing MMVL on the face detection benchmark. For V1, there is a reasonable gain when increasing MMVL; there is 16% better performance as the number of wavefronts is increased from 128 to 256, with an additional 13% improvement as the number of wavefronts is increased from 256 to 512. The results are even more dramatic on a V4 with 4 mask partitions: 100% faster as the number of wavefronts is increased from 128 to 1024. The fact that a V4-P4 with MMVL 128 is actually slower than a V4-P1 with MMVL 256 suggests that using BRAM for increased depth can be better than using it for more partitions. The reason that increased MMVL makes such a large difference is that as masked instructions have fewer and fewer valid wavefronts, they start taking so few cycles that efficiency drops (either data hazards or instruction dispatch rates

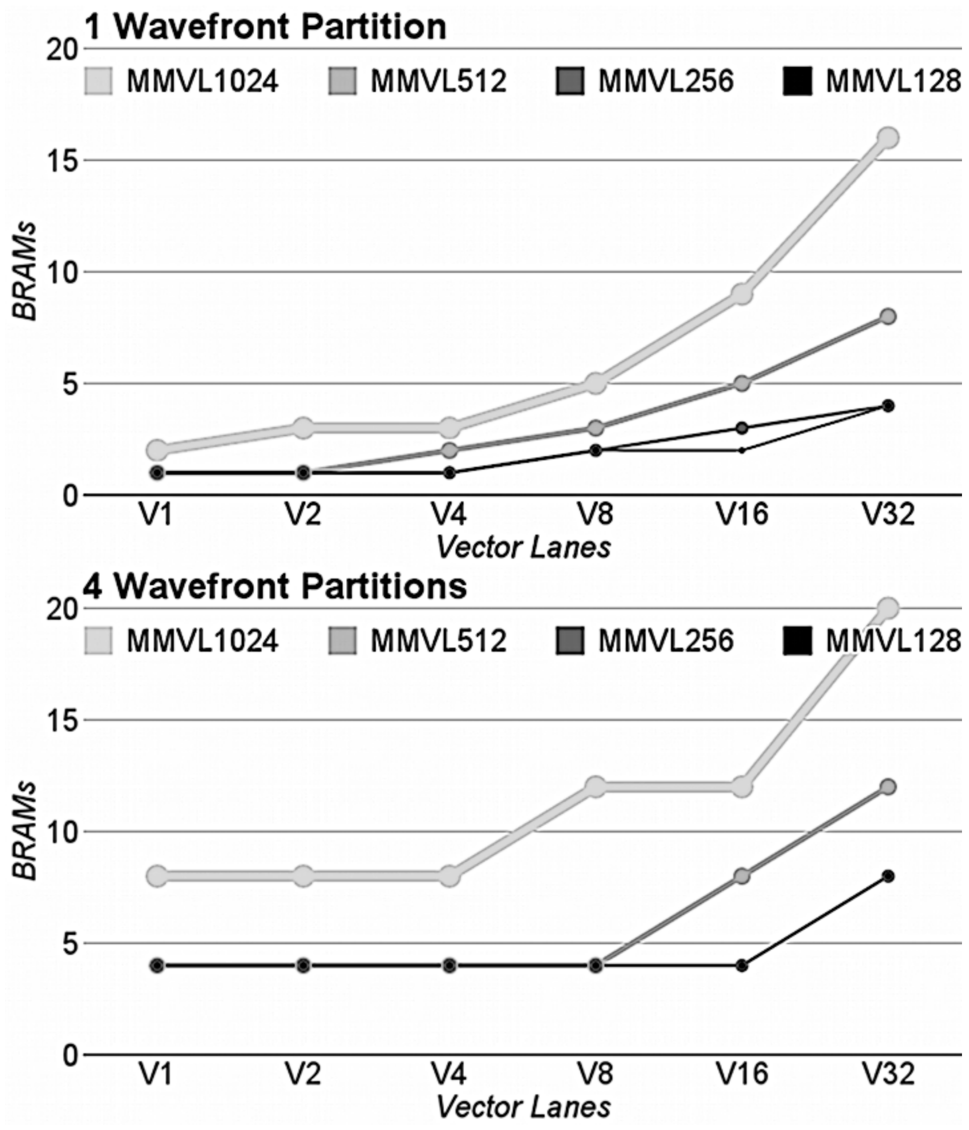


Figure 4.13: BRAM Usage for Masks When Varying MMVL for 1 (top) and 4 (bottom) Partitions

dominate). Longer MMVL means that fewer masked instructions are needed, and less time is spent in this low efficiency regime. However, in the V32 processor, the masked vector length is no longer limited by MMVL but by the number of vectors needed and the size of the scratchpad. In this case, changing MMVL had no effect

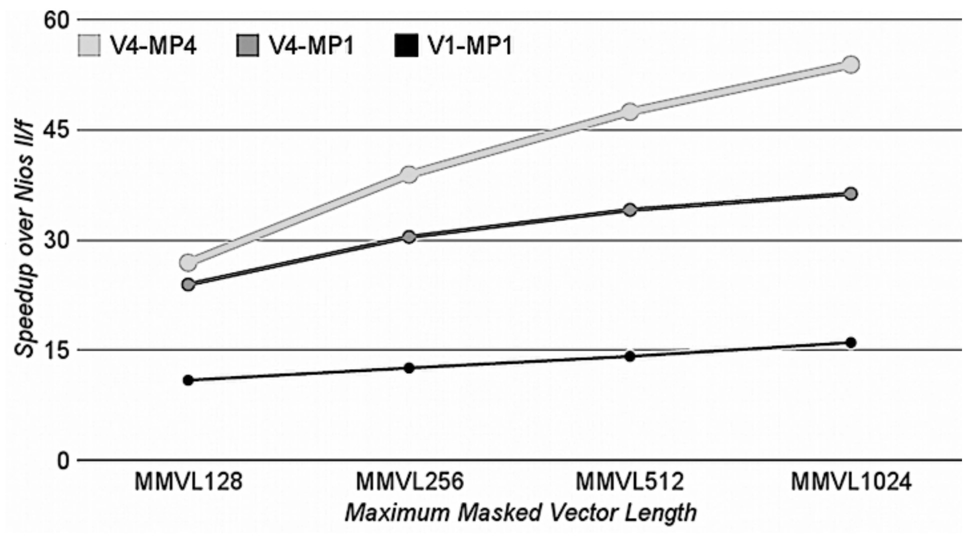


Figure 4.14: Effect of Changing MMVL on Viola-Jones Face Detection

on the results so they are not shown. The trend for V1 and V4 does suggest that increased vector length (in this case by having a larger scratchpad) would provide higher performance on this application.

4.3.7 Results Summary

Figure 4.15 summarizes the speedup of wavefront skipping over the previous CMOV implementation on our three benchmarks, while Figure 4.16 shows the speedup per area (eALMs). The maximum speedups for a single partition is 3.3× on FAST9 (V1-P1) and 3.2× on Viola-Jones (V4-P1). The largest speedup from partitioning over full wavefront skipping is 1.65× on Viola-Jones (V32-P32 vs V32-P1). All of the benchmarks gain at least as much in speedup as the cost in area of the wavefront skipping implementation, with the minimum gain being 1.1× better performance per area for FAST9 on a V32 with one wavefront partition. Although an increased number of partitions causes a corresponding increase in the area of MXP, for FAST9 and Viola-Jones the performance per area continues to increase as the number of partitions increases. Wavefront skipping is therefore useful for all the benchmarks we tested, and partitioning is useful for two of the three.

For a programmer implementing a conditional algorithm, the configuration of

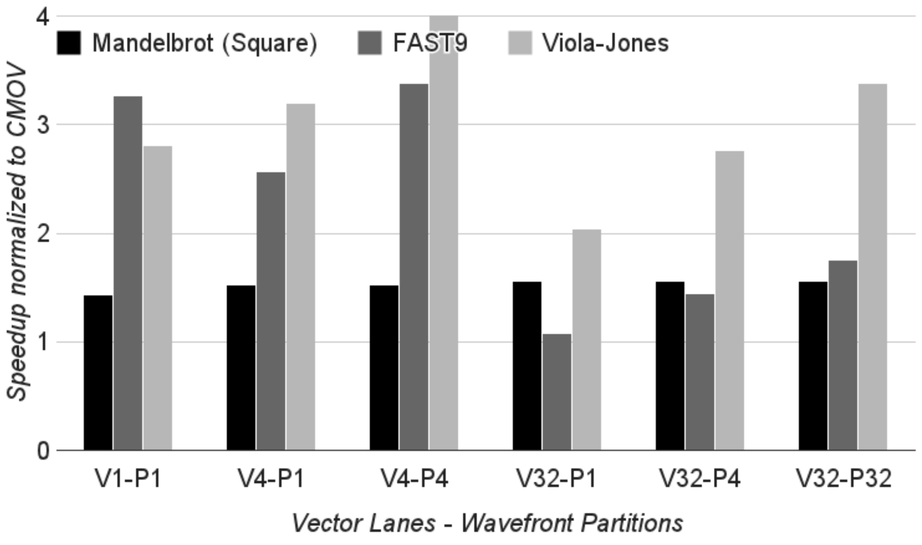


Figure 4.15: Speedup from Wavefront Skipping

BRAM usage for scratchpad size, wavefront partitions, and maximum masked vector length will depend on the application. The designer will generally want to devote as much BRAM as possible to scratchpad data, as longer vectors will benefit all parts of an application. However, for heavily conditional applications like Viola-Jones, having masked vectors that are long enough to keep the vector core busy as the number of valid wavefronts shrink is also important. Few BRAMs are needed for just a single partition; using multiple partitions adds several BRAMs, but this may be justified when performance is absolutely necessary or the targetted FPGA has leftover BRAMs with a given design.

4.4 Summary

This chapter shows how a class of irregular data parallel algorithms that are difficult or inefficient to implement with other approaches can be accelerated using wavefront skipping. Our implementation of wavefront skipping in soft vector processors (SVPs) can be done efficiently in terms of logic and BRAM usage. Wavefront skipping allows for higher performance due to skipping masked off elements. In addition, from our experience, masked execution is also much easier to use

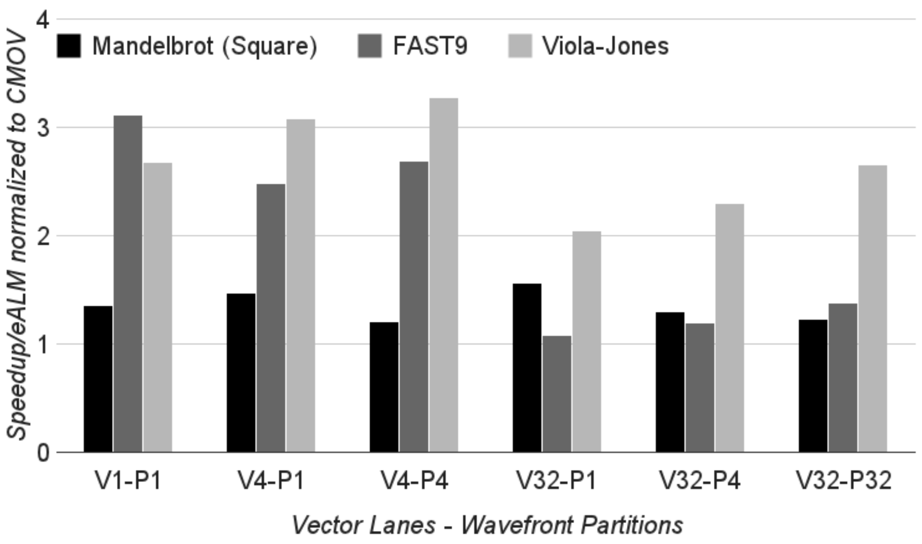


Figure 4.16: Speedup per Area (eALMs) from Wavefront Skipping

than checking early exit conditions on blocks of elements using predicated/CMOV instructions. Our implementation stores offsets in BRAMs, which are relatively plentiful and high-performance in FPGAs. The alternative, which uses a count-leading-zeros operation, would have higher latency and also limit the number of wavefronts skipped in one cycle. Our approach keeps the mask logic simple and off the critical path, and can also skip an arbitrary number of wavefronts.

When not partitioned, our wavefront skipping implementation uses less than 5% extra area and gives 3.2× better performance on Viola-Jones face detection. Extra logic and BRAMs can be used to gain additional performance by partitioning, allowing parts of a wavefront to have different offsets. Although costly in terms of area, partitioning gives up to 1.65× extra performance on face detection. Partitioned wavefront skipping may not be a reasonable design tradeoff in a fixed vector processor. In an FPGA, partitioned wavefront skipping gives a designer an extra tool to tradeoff additional logic and BRAM for application specific performance.

Chapter 5

Attaching Streaming Pipelines to Soft Vector Processors

The expert knows more and more about less and less until he knows everything about nothing. — Mahatma Gandhi

So far, this dissertation has shown how to better optimize SVPs for increased performance per area and how to enable applications that are traditionally difficult to accelerate. These approaches help make SVPs more attractive for applications requiring modest performance or those with divergent control flow. However, they do not take full advantage of the underlying FPGA fabric, and they do not offer a path for migrating a design from an SVP program to an RTL pipeline. This chapter explores the challenges and benefits of interfacing custom pipelines to an SVP. The SVP is able to manage data movement and perform computations that are less performance critical, while dispatching heavy computation to custom engines that can fully exploit the underlying FPGA fabric. In this manner, applications can migrate fixed parts of their algorithms to get the full performance the FPGA can provide while maintaining software design and control at each step.

5.1 Introduction

FPGAs are most useful when designs exploit deep pipelines, utilize wide data parallelism, and perform operations not found in a typical CPU or GPU. This typically requires a hardware designer to design a custom system in an RTL such as VHDL or Verilog. Recently, the emergence of electronic system level (ESL) tools such as Vivado HLS and Altera's OpenCL compiler allow software programmers to produce FPGA designs using C or OpenCL. However, since all ESL tools translate a high-level algorithm into a hardware description language (HDL), they share common drawbacks: changes to the algorithm require lengthy FPGA recompiles, recompiling may run out of resources (eg, logic blocks) or fail to meet timing, debugging support is very limited, and high-level algorithmic features such as dynamic memory allocation are unavailable. This suggests ESL users need some degree of hardware design expertise. Hence, ESL tools may not be the most effective way to make FPGAs accessible to software programmers.

An SVP achieves high performance through wide data parallelism, efficient looping, and prefetching. The main advantages of an SVP over RTL are scalable performance and a traditional software programming model. Performance scaling is achieved by adding more ALUs, but beyond a certain point (eg, 64 ALUs) the increases in parallelism are eroded by clock frequency degradation. Additionally, even if performance scales perfectly, performance per area will lag behind that of a custom design.

To increase performance of SVPs even further, they must also harness deep pipeline parallelism. For example, most types of encryption (such as AES) need deep pipelines to get significant speedup. Although processors (and SVPs) are not built to exploit deep pipeline parallelism, FPGAs support it very well.

Several questions then arise: How can a SVP be augmented to exploit both wide and deep parallelism? What differs in interfacing an SVP to external logic versus a scalar soft processor? How can control flow be coordinated between deep custom pipelines and the fixed execution stages of the SVP? Can more complex instructions than the standard two-input, one-output format be supported, and how much benefit do they provide?

We investigated these questions using the VectorBlox MXP SVP. We devised a way to add custom vector instructions (CVIs) to the processor, shown in Figure 5.1. Such CVIs can be simple operations, or they may contain wide and deep pipelines. The interface is kept as simple as possible so that software programmers can eventually develop these custom pipelines using C; we also show that a simple

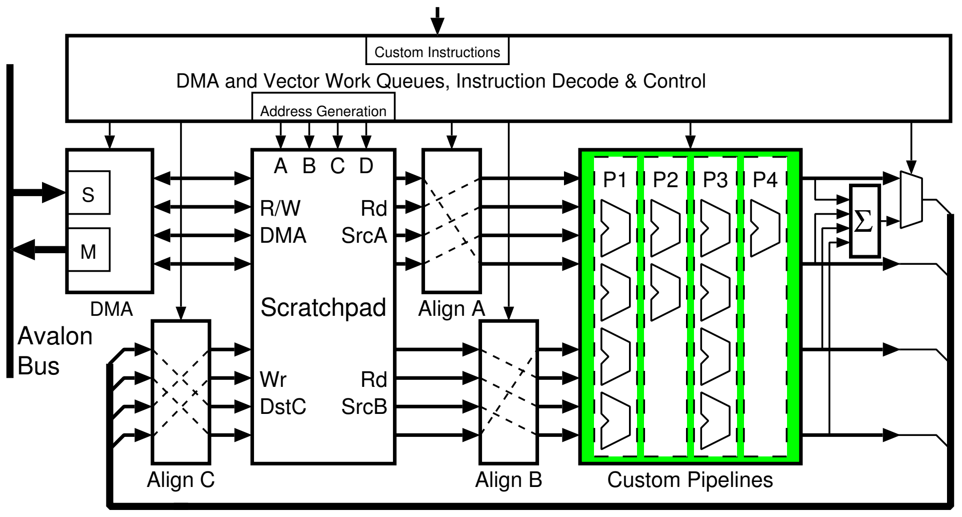


Figure 5.1: Internal View of VectorBlox MXP

high-level synthesis tool can be created for this purpose. To demonstrate speedups, we selected the N-body gravity problem as case study. In this problem, each body exerts an attractive force on every other body, resulting in an $O(N^2)$ computation. The size and direction of the force between two bodies depends upon their two masses as well as the distance between them. Solving the problem requires square root and divide, neither of which are native operations to the MXP. Hence, we start by implementing simple custom instructions for the reciprocal and square root operations. Then, we implement the entire gravity equation as a deep pipeline.

The main contribution of this chapter is the introduction of a modular way of allowing users to add streaming pipelines into SVPs as *custom vector instructions* to get significant speedups. On the surface, this appears to be a simple extension of the way custom instructions are added to scalar CPUs such as Nios II. However, there are unique challenges that must be addressed to enable streaming data from multiple operands in a SVP. Also, scalar CPU custom instructions are often data starved, limiting their benefits. We show that SVPs can provide high-bandwidth data streaming to properly utilize custom instructions.

5.2 Custom Vector Instructions (CVIs)

The VectorBlox MXP was designed to have a minimal core instruction set. It is important to keep this instruction set minimal in an SVP because the area required by an operation will be replicated in every vector lane, thus multiplying its cost by the number of lanes. In MXP, multiply/shift/rotate instructions are included as core instructions because they share the use of the hard multipliers in the FPGA fabric. However, divide and modulo are not included as core instructions because a pipelined implementation requires more than three pipeline stages and more logic than all other operators combined.

There are many instructions that could be useful in certain applications; some are large and complex but there are also simple, stateless 1- or 2-input, single-output operators with no state. These include arithmetic (e.g., divide, modulo, reciprocal, square root, maximum, minimum, reciprocal square root), bit manipulation (e.g., population count, leading-zero count, bit reversal), and encryption acceleration (e.g., s-box lookup, byte swapping, finite field arithmetic). Some of these operators require very little logic, while others demand a significant amount of logic. However, supporting all such operators is prohibitively expensive. Also, it is unlikely that a single application would make use of almost all of these specialized instructions. Finally, even when they are used by an application, these instructions may not appear frequently in the dynamic instruction mix. For this reason, the base MXP ISA excludes many operations that could potentially be useful, but were considered to be too specialized.

Instead, we would like to add such specialized instructions on a per-application basis. We would also like to allow the user to decide how many of these operators should be added to the pipeline, since it may not make sense to replicate large operators in every lane. Additionally, operators with deep pipelines should be able to co-exist along with the existing three stage execution pipeline without lengthening the number of cycles taken to process normal instructions.

5.2.1 CVI Design Approach

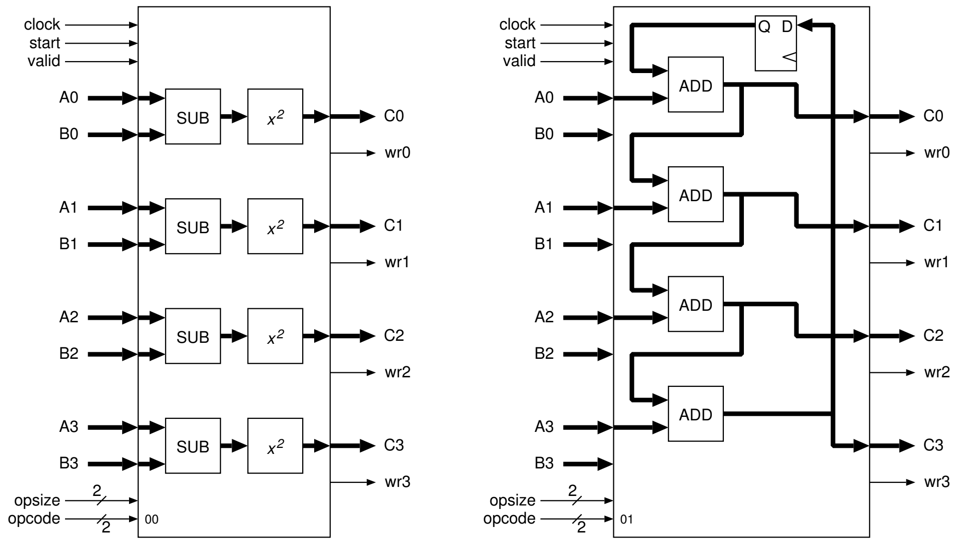
Our add-on custom vector instruction (CVI) approach is different from the VESPA approach [74], which allows selective instruction subsetting from a master instruction set. Subsetting allows a reduction in area when an application does not use specific instructions, but does not allow for further customization. Additionally, since no base instruction set is defined, it may be difficult to change an alogirthm running on a subsetted processor without resynthesizing the processor to add back needed instructions.

In contrast, the MXP approach defines a core set of instructions to increase software portability, while user-specified CVIs can be added to accelerate application-specific operations that are rarely needed by other applications. CVIs use an external port interface to MXP, allowing the addition to be done without modifying the processor source HDL. This modularity may also make it easier to add CVIs using run-time reconfiguration. Some CVIs will be application specific, while others (such as square root) may be resuable.

5.2.2 CVI Interface

A typical CVI is executed in MXP like a standard arithmetic instruction, with two source operands and one destination operand. The main difference is that data is sent out of MXP through a top-level port, processed by the CVI, then multiplexed back into the MXP pipeline before writeback. One individual CVI may consist of many parallel execution units, processing data in both a parallel and a streaming fashion.

In Altera's Qsys environment, CVIs are implemented as Avalon components with a specific conduit interface. Qsys conduits are a way to bundle signals together when connecting modules in the graphical Qsys interface. Vector data and control signals are exported out of the top level of MXP and connected to the CVI automatically through the conduit. The signals in the conduit interface can be seen in Figure 5.2. The left side (Figure 5.2a) shows an example of a simple CVI, the 'difference-squared' operation. This CVI does the same action on all data, so its individual execution units are simply replicated across the number of CVI lanes. In the simplest case, the number of CVI lanes will match the number of MXP lanes. This is adequate if the CVI is small, or there is sufficient area available. In Section 5.2.3, we will consider the area-limited case when there must be fewer CVI lanes than MXP lanes.



a) Custom instruction within lanes

b) Custom instruction prefix sum across lanes

Figure 5.2: Examples of Custom Vector Instructions

The right side (Figure 5.2b) shows a more complicated example, a prefix sum, where data is communicated across lanes. The prefix sum calculates a new vector that stores the running total of the input vector appearing on operand A. This makes it a very different type of operation than the difference-squared operation, which does not have communication between lanes. As a result, computing a prefix sum is a difficult operation for wide vector engines; it is best implemented in a streaming fashion. Since a vector may be longer than the width of the SVP, it is important to accumulate the value across multiple clock cycles in time. To support this, the CVI interface provides a clock and vector start signal. Furthermore, a data valid signal indicates when each wave of input data is provided, and individual data-enable input signals (not shown for clarity) are provided for each lane.

Additional signals in the CVI interface include an opsize (2 bits) that indicates whether the data is to be interpreted as bytes, halfwords, or words. Also, output byte-enable signals allow the instruction to write back only partial vector data, or to implement conditional-move operations, or write back a last (incomplete) wave. Finally, an opcode field is provided to allow the selection of multiple CVIs.

Alternatively, the opcode can be passed to a single CVI and used as a mode-select for different functions, such as sharing logic between divide and modulo, or to implement different rounding modes. The opcode field is shown as two bits, but this can be easily extended.

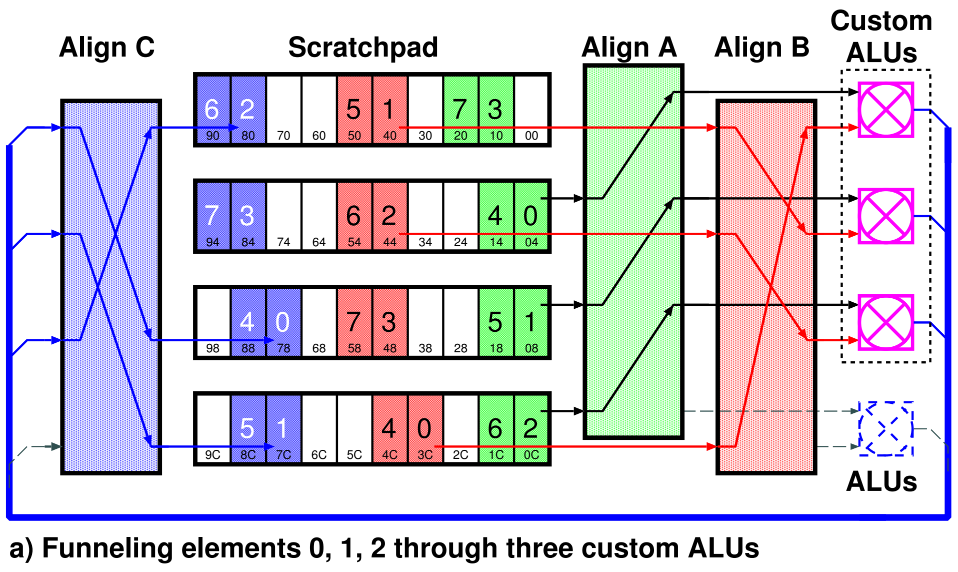
One notable exclusion from the interface is a stall signal or other means of providing back pressure. Allowing a CVI to stall the MXP front end was considered undesirable from a clock speed standpoint. More precisely measuring and addressing this limitation is left to future work.

5.2.3 Large Operator Support

Custom operators may be prohibitively large to add to each vector lane. For example, a fully pipelined 32-bit fixed-point divider with 16 fractional bits (Q16.16 format) requires 2,652 ALMs to implement in a Stratix IV FPGA. This is more logic than an entire vector lane in the MXP. Thus, it may be desirable to use fewer dividers than the number of lanes (depending upon the number of divides in the dynamic instruction mix). We have designed an interface that allows using narrower CVIs with minimal overhead by reusing the existing address generation and data alignment logic.

Figure 5.3 shows how CVIs with a different number of lanes are added to the existing MXP datapath. During normal operation, the address generation logic increments each address by the width of the vector processor each cycle. For the example shown, i.e., an MXP with four 32-bit lanes (written as a 'V4'), each source and destination address is incremented by 16 bytes, until the entire vector is processed. MXP's input alignment networks align the start of both source vectors to lane 0 before the data is processed by the ALUs. After execution, the destination alignment network is used to align the result to the correct bank in the scratchpad for writeback.

During a CVI, the address generation logic increments the source and destination addresses by the number of CVI lanes times the (4-byte) width of each lane. In the example shown, the addresses are incremented by 12 bytes for each wave, regardless of the SVP width; operand A starts at address 0x04 on the first cycle (Figure 5.3a) and increments to 0x10 on the second cycle (Figure 5.3b). As in



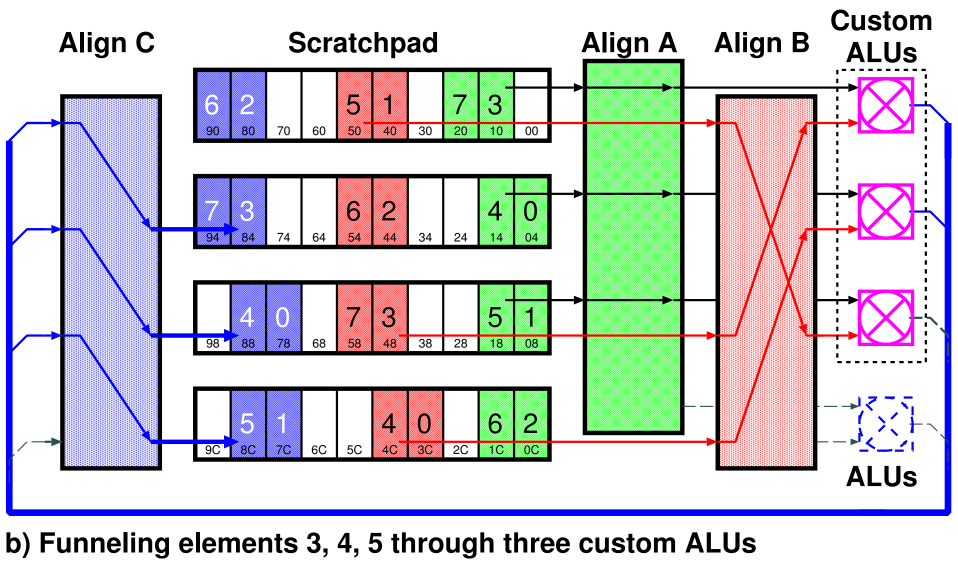


Figure 5.3: Custom Vector Instructions With Fewer Lanes Than the SVP

normal execution, the alignment networks still align source data to start at lane 0 before data is processed in the custom ALUs. In this case, the fourth lane would not contain any data, so its data-enable input would be inactive. After execution, the CVI result is multiplexed back into the main MXP pipeline, and finally the resulting data is aligned for writeback into the scratchpad. On CVI writeback, the output byte enables are then used to write out data for only the first 12 bytes of each wave; the destination alignment network then repositions the wave to the correct target address.

5.2.4 CVIs with Deep Pipelines

MXP uses an in-order, stall-free backend for execution and writeback to achieve high frequencies. The CVIs are inserted in parallel to the regular 3-stage arithmetic pipeline of MXP, which means they can also have 3 internal register stages. If fewer stages are needed, it must be padded to 3 stages.

Some operations, such as divide or floating point, require much deeper pipelines. If the user naively creates a pipeline that is longer than 3 cycles, the first wave of data would appear to the writeback stage later than the writeback address, and the last wave of data would not reach the writeback stage at all.

To address the latter problem, we have devised a very simple strategy for inserting long pipelines. In software, we extend the vector length to account for the additional pipeline stages (minus the 3 normal stages). This solves part of the problem, allowing the last wave of vector data to get flushed out of the pipeline and appear at the writeback stage. During the last cycles, the pipeline will read data past the end of the input operands, but their results will never be written back. However, the beginning of the output vector will have garbage results.

To eliminate this waste of space, the MXP could simply delay the writeback address by the appropriate number of clock cycles. A second approach is to allow the CVI itself to specify its destination address for each wave. This requires the MXP to inform the CVI of the destination addresses, and rely upon the CVI to delay them appropriately. We have chosen this latter technique, as it allows for more complex operations where the write address needs to be controlled by the CVI, such as vector compression. Because this can write to arbitrary addresses, any CVI using this mode must set a flag which tells MXP to flush its pipeline after the CVI has completed. The flag is set as a top-level parameter to the MXP instance.

Although this approach does not require additional space, it still requires extending the length of the vector and flushing the pipeline. The extended vector length makes the instruction take a number of additional cycles equal to the number of pipeline stages in the CVI, while the MXP pipeline flush costs the depth of the MXP pipeline (eight to ten stages, depending on the MXP configuration). During normal execution instructions can issue back-to-back and multiple instructions can be in the MXP pipeline in parallel, but deep-pipeline CVIs cannot execute concurrently with other instructions in our approach.

5.3 Multi-Operand CVI

The CVIs described in the previous section are intended for one or two input operands, and one destination operand. However, the DFGs of large compute kernels may require multiple inputs and outputs, and require both scalar and vector operands. In this section, we describe how to support multiple-input, multiple-output CVIs. As a motivating example, we have chosen the N-body gravitational problem. We have modified the problem slightly to produce a pleasing visual demonstration: keep calculations to only 2 dimensions, use a repelling force rather than an attracting force, and allow elastic collisions with the screen boundary.

5.3.1 N-Body Problem

The traditional N-body problem simulates a 3D universe, where each celestial object is a body, or particle, with a fixed mass. Over time, the velocity and position of each particle is updated according to interactions with other particles and the environment. In particular, each particle exerts a net force (i.e., gravity) on every other particle. The computational complexity of the basic all-pairs approach we use is $O(N^2)$. Although advanced methods exist to reduce this time complexity, we do not explore them here.

In our modified version, we consider a 2D screen rather than a 3D universe. The screen is easier to render than a 3D universe, but it also has boundaries. Also, we change the sign of gravity so that objects repel each other, rather than attract. (Attractive forces with screen boundaries would result in the eventual collapse into a moving black hole, which is not visually appealing.) Like the traditional N-body problem, we also treat particles as point masses, i.e., there are no collisions

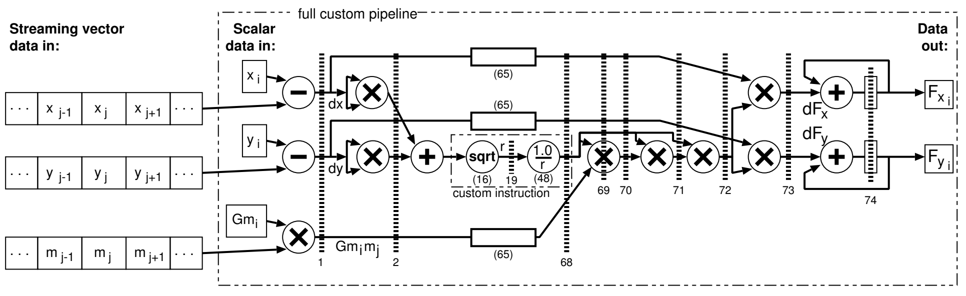


Figure 5.4: Force Summation Pipeline

between particles. We have also adjusted the gravitational constant to produce visually pleasing results.

The run-time of the N-body problem is dominated by the gravity force calculation, shown below:

$$\vec{F_{i,j}} = G\frac{M_i M_j}{r^2} = 0.0625\frac{M_i M_j}{|\vec{P_i} - \vec{P_j}|^3}(\vec{P_i} - \vec{P_j})$$

where $\vec{F_{i,j}}$ is the force particle i imposes on particle j, $\vec{P_i}$ is the position of particle i, and M_i is the size or 'mass' of particle i. When computing these forces, we chose a fixed-point Q16.16 fixed-point representation, where the integer component of $\vec{P}$ represents a pixel location on the screen.

When a particle reaches the display boundary, its position and velocity are adjusted to reflect off the edge (towards the center) after removing some energy from the particle. These checks do not dominate the run-time as they are only $O(N)$.

An implementation of the gravity computation as a streaming pipeline is shown in Figure 5.4. This is a fixed-point pipeline with 74 stages; the depth is dominated by the fixed-point square root and division operators which require 16 and 48 cycles, respectively.[1] For each particle, its x position, y position, and mass (premultiplied by the gravitational constant) are loaded into scalar data registers within the instruction. This is the reference particle, P_i. Then, three vectors representing the

[1]We used Altera's LPM primitives for these operators. The pipeline would benefit from a combined reciprocal square root operator, but it does not exist in the Altera library.

x position, y position and mass of all particles, P_j, are streamed through the vector pipeline. The pipeline integrates the forces exerted by all these particles, and accumulates a net force on the reference particle.

Overall, the pipeline requires three scalar inputs (reference particle properties) and three vector inputs (all other particles). It also produces two vector outputs (an x vector and a y vector), although the output vectors are of length 1 because of the accumulators at the end of the pipeline. Hence, this gravity pipeline is a 3-input, 2-output CVI.

All of the MXP vector instructions, including the custom type, only have two inputs and one output. This is a limitation in the software API, where only two inputs and one output can be specified, as well as the hardware dispatch, where only two source vector addresses and one destination vector address can be issued. Loading of scalar data can be accomplished by using vector operations with length 1, and either using an opcode bit to select scalar loading versus vector execution, or by fixed ping-ponging between scalar loading and vector execution. We use the ping-pong approach to save opcodes.

Supporting multiple vector operands is not as simple, however, and will be discussed below.

5.3.2 Multi-Operand CVI Dispatch

Figure 5.5 shows two approaches to dispatching multi-operand CVIs. A ‘wide’ approach requires data to be laid out spatially, such that operand A appears as vector element 0, operand B appears as vector element 1, and so forth. This is shown in Figure 5.5a. In other words, the operands are interleaved in memory as if packed into a C structure. To stream these operands as vectors, an array of structures (AoS) is created. Ideally, the input operands would precisely fit into the first wave; with two read ports, the amount of input data would be twice the vector engine width. If more input data is required, then multiple waves will be required, which will be similar to the depth approach below. If less input data is required, then the CVI does not need to span the entire width of the SVP. In this case, it may be possible to provide multiple copies of the pipeline to add SIMD-level parallelism to the CVI.

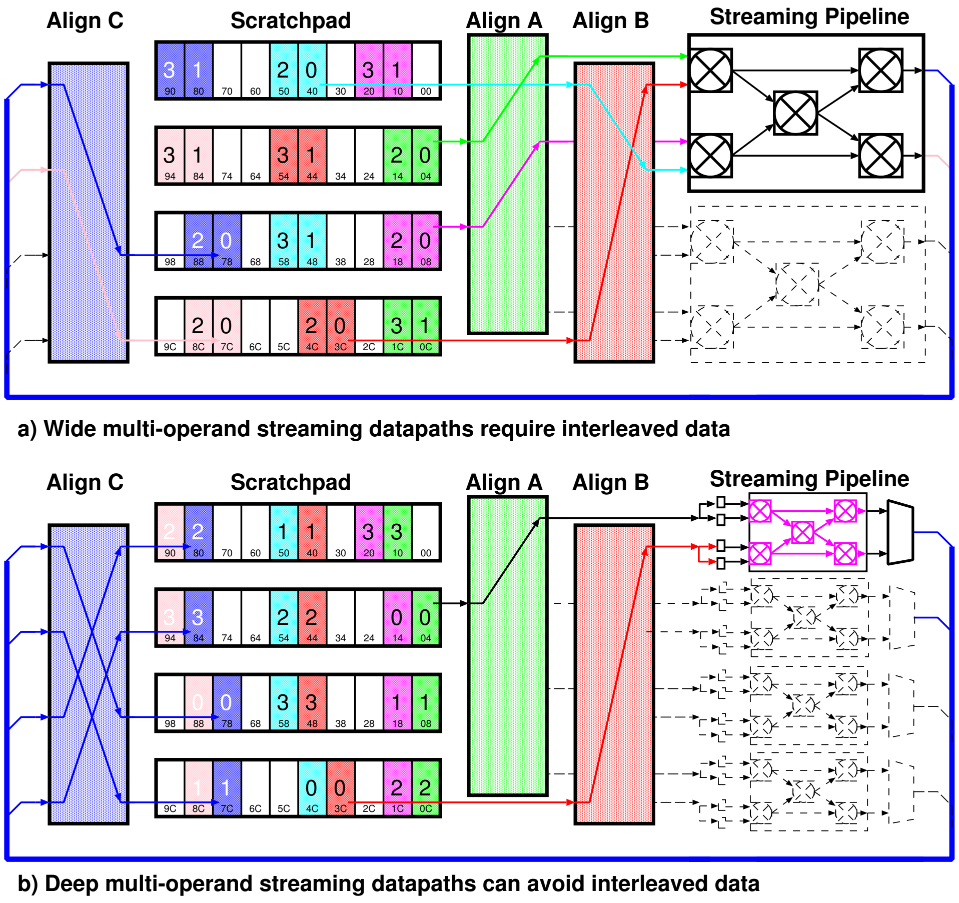
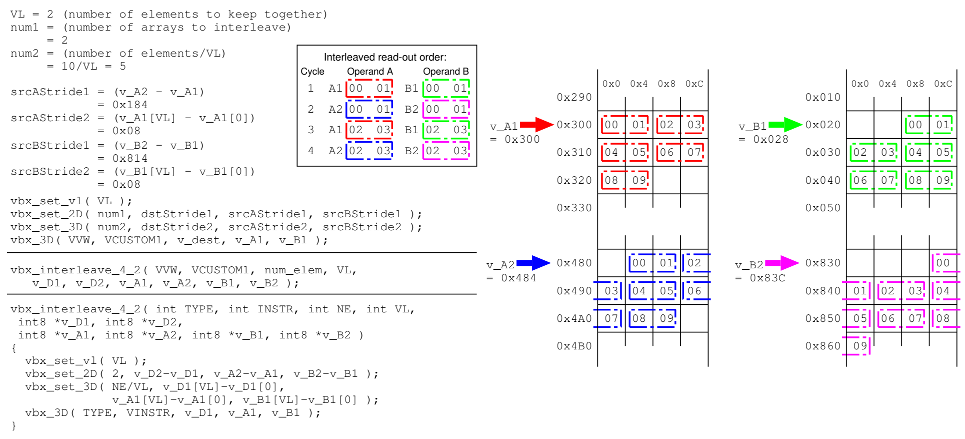

Align C
Scratchpad
Align A
Align B
Streaming Pipeline
a) Wide multi-operand streaming datapaths require interleaved data
Align C
Scratchpad
Align A
Align B
Streaming Pipeline
b) Deep multi-operand streaming datapaths can avoid interleaved data

Figure 5.5: Multi-Operand Custom Vector Instructions

The main drawback of the wide approach is that the data must be interleaved into an AoS. In our experience, SVPs work better when data is arranged into a structure of arrays (SoA). The SoA layout assures that each data item is in its own array, so SVP instructions can operate on contiguoly packed vectors.

For example, suppose image data is interleaved into an AoS as {r,g,b} triplets. With this organization, it is difficult to convert the data to {y,u,v} triplets because each output data item requires a different equation. When the image data is blocked as in a SoA, it is easy to compute the {y} matrix based upon the {r}, {g}, and {b} matrices. Furthermore, converting between AoS and SoA on the fly requires data



```
VL = 2 (number of elements to keep together)
num1 = (number of arrays to interleave)
     = 2
num2 = (number of elements/VL)
     = 10/VL = 5

srcAStride1 = (v_A2 - v_A1)
            = 0x184
srcAStride2 = (v_A1[VL] - v_A1[0])
            = 0x08
srcBStride1 = (v_B2 - v_B1)
            = 0x814
srcBStride2 = (v_B1[VL] - v_B1[0])
            = 0x08
vbx_set_vl( VL );
vbx_set_2D( num1, dstStride1, srcAStride1, srcBStride1 );
vbx_set_3D( num2, dstStride2, srcAStride2, srcBStride2 );
vbx_3D( VVW, VCUSTOM1, v_dest, v_A1, v_B1 );
```

```
vbx_interleave_4_2( VVW, VCUSTOM1, num_elem, VL,
   v_D1, v_D2, v_A1, v_A2, v_B1, v_B2 );
```

```
vbx_interleave_4_2( int TYPE, int INSTR, int NE, int VL,
 int8 *v_D1, int8 *v_D2,
 int8 *v_A1, int8 *v_A2, int8 *v_B1, int8 *v_B2 )
{
  vbx_set_vl( VL );
  vbx_set_2D( 2, v_D2-v_D1, v_A2-v_A1, v_B2-v_B1 );
  vbx_set_3D( NE/VL, v_D1[VL]-v_D1[0],
              v_A1[VL]-v_A1[0], v_B1[VL]-v_B1[0] );
  vbx_3D( TYPE, VINSTR, v_D1, v_A1, v_B1 );
}
```

Interleaved read-out order:

Cycle	Operand A		Operand B	
1	A1	00 01	B1	00 01
2	A2	00 01	B2	00 01
3	A1	02 03	B1	02 03
4	A2	02 03	B2	02 03

Figure 5.6: Using 3D Vector Operations for Multi-Operand Dispatch

copying and can be time consuming. Hence, it is better for regular SVP instructions to use SoA format.

An alternative 'deep' approach to multiple-operand CVIs requires data to be interleaved in time. This is shown in Figure 5.5b, where a streaming datapath only has access to two physical ports, operands A and B of one vector lane. This can be combined with wide parallelism by replicating the deep pipeline. It is not desirable to simply fully read two input vectors and then read the third input, though, as the CVI would have to buffer the full length of the instruction. In MXP, vector lengths are limited only by the size of the scratchpad, so the buffering could be costly. Rather, it is desirable to only buffer a single cycle's worth of inputs.

We accomplish this in MXP by using its 2D and 3D instruction dispatch to issue a single wavefront of data from each input on alternating cycles. The 2D instructions work by first executing a normal (1D) vector instruction to read one wavefront of data, then applying a different stride to each of the input addresses and output address and to switch among input arrays and thus interleave the wavefronts. The strides and repetitions can be set at runtime using a separate set_2D instruction. The 3D instructions are an extension of this, where 2D instructions are repeated using another set of strides.

Figure 5.6 illustrates how these 2D/3D ops are used to dispatch CVIs with multiple operands. In this example, a CVI with 4 inputs (A1, A2, B1, and B2) and

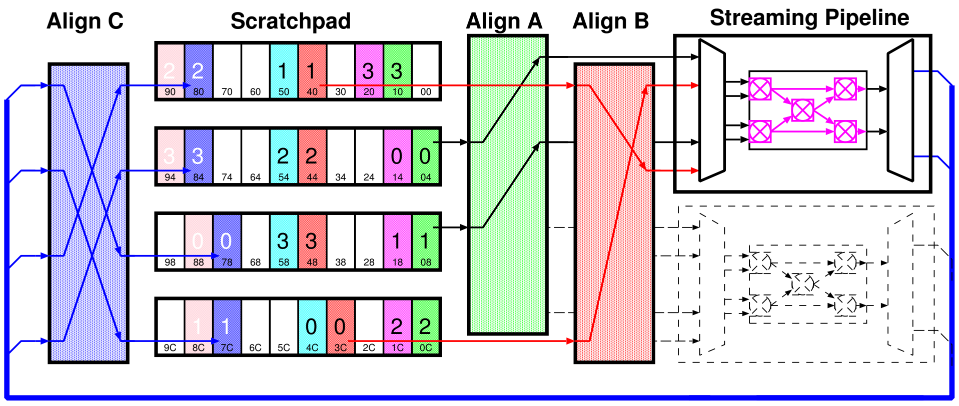


Figure 5.7: Multi-Operand Custom Vector Instruction Funnel Adapters

2 outputs (D1 and D2) is to be executed. The desired result is that the CVI will alternate A1/B1 and A2/B2 inputs each cycle, and alternate D1/D2 outputs each cycle.

To get this outcome, first the 1D vector length (VL) is set to the number of CVI lanes, and the 2D strides are set to the difference between input addresses (A2-A1, B2-B1) and output addresses (D2-D1). Since the inner vector length is the same as the number of custom instruction lanes, each row is dispatched as one wave in a single cycle, followed by a stride to the next input. The 2D vector length is set to the total number of cycles required (max(inputs/2, outputs/1)). Note that if more than 2 cycles (4 inputs or 2 outputs) are needed, sets of additional inputs and outputs will need to be laid out with a constant stride from each other.

Since a 2D operation merely alternates between sets of inputs (and outputs), a 3D instruction is used to stream through the arrays of data. Each 2D instruction processes one wavefront (of CVI lanes) worth of data, so the 3D instruction is set to stride by the number of CVI lanes. The number of these iterations (the 3D length) is set to the data length divided by the number of CVI lanes.

In Figure 5.6, the complex setup routine (top) can be abstracted away to a single function call, `vbx_interleave_4_2()` (middle). One possible implementation of this call is shown at the bottom of the figure.

On the hardware side, data is presented in wavefronts and needs to be multiplexed into a pipeline. Because a new set of inputs only arrives every max(inputs/2,

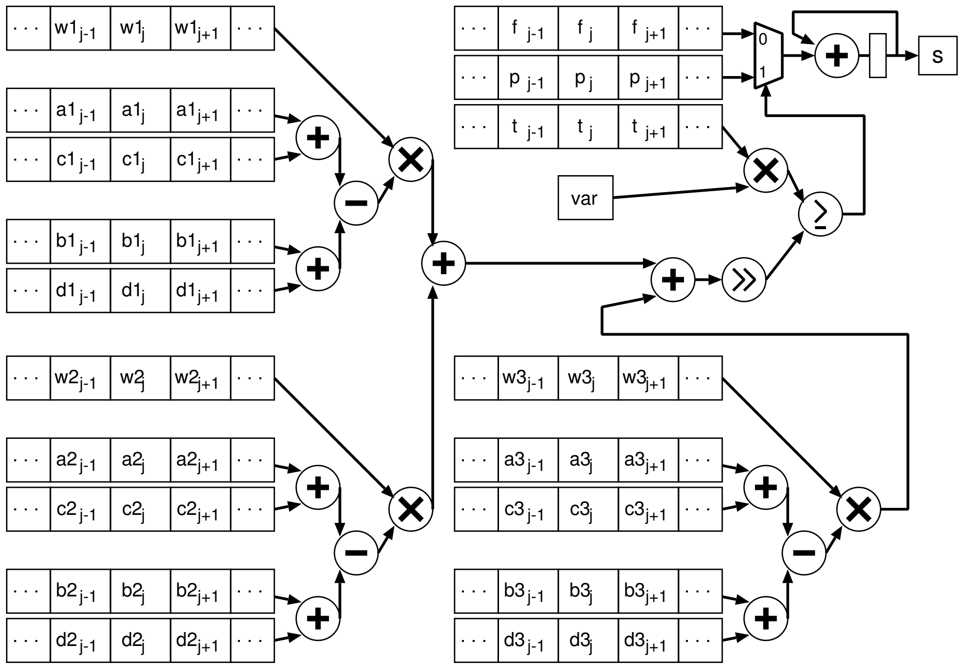


Figure 5.8: Face Detection Pipeline

outputs) cycles, the pipeline would be idle part of the time if it had the same width and clockrate as the CVI interface. We can recover the lost performance, and save area, by interleaving two or more logical streams into one physical pipeline. To do this, we have created 'funnel adapters' which are used to accept the spatially distributed wave and feed it to the pipeline over time. This is illustrated in Figure 5.7.

The funnel adapter for our 3-input, 2 output particle physics pipeline, which has inputs arriving every 2 cycles (and outputs leaving every 2 cycles), allows two MXP lanes worth of data to share a single physical streaming pipeline.

5.3.3 Face Detection CVI Example

As another example, we have also outlined the design of a multiple-input/output CVI for Viola-Jones face detection. The face detection pipeline is shown in Figure 5.8. Unlike the gravity pipeline, the face detection requires far more inputs – a total of 18 vector inputs and 1 scalar input. It produces a single vector output.

Using regular SVP instructions, this face detection requires a total of 19 instructions, requiring 19 clock cycles per wave of data. In contrast, due to the large number of vector input operands, the face detection pipeline takes 9 clock cycles per wave of data. Hence, the best-case speedup expected from this custom pipeline is $\frac{19}{9}$, or roughly 2 times. Even though face detection contains a large number of operators, the number of input operands limits the overall speedup. Hence, not all applications will benefit significantly from custom pipelines.

5.4 CVI Limitations

Our CVI implementation was designed to address a wide range of operations and support different use cases, but it is not without limitations. These include the inability of a CVI to provide flow control or back pressure to the vector processor, inability to write longer vectors than the inputs, loss of scratchpad bandwidth when using narrow CVIs, and the inability to pipeline deep CVIs with regular vector instructions.

There are situations where it may be useful to provide back pressure from the CVI to the vector processor. For example, a CVI that accesses an external memory or network interface may need to stall for a variable number of cycles. However, our CVI interface only works with a fixed-length pipeline. Adding back pressure would require changing MXP's entire pipeline, which uses a fixed-length, stall-free execution backend. The benefits of this would not be measureable unless we had a CVI which needed this feature, and the costs would not only include increased area but also possibly decreased frequency, as any stall signal would have to be propagated to all lanes of the vector processor. Still, there may be situtations where it would be useful to have back pressure.

Most of the CVIs we have implemented write back an output vector of the same length as the inputs, and we have provided a write address port so that the CVI can write a shorter vector (as in a vector compress operation) or write to arbitrary addresses. However, this is not general enough to implement operations that require more write cycles than read cycles. For instance, a parallel vector scatter operation may require writing back multiple wavefronts given a single cycle of reading addresses and data to scatter. This could be implemented if back pressure on reading data in was available while still writing output data.

When using fewer CVI lanes than the width of the vector processor, more data is read from and scratchpad memory than is used, and less data is written to scratchpad memory than is possible. Since we have more scratchpad bandwidth than is needed for the CVI, it is reasonable to investigate if we can do something useful with that extra read and write bandwidth. For instance, if there are half the number of CVI lanes as vector lanes, it could be possible to issue a half-speed vector instruction using the other half of the scratchpad memory bandwidth. How best to utilize the extra bandwidth (if it can be utilized usefully at all) is left as an open question.

Finally, as mentioned in Section 5.2.4, when executing deep pipeline CVIs we extend the vector length so that the instruction takes a number of cycles equal to the pipeline depth. This means in an N-stage pipeline CVI, the first N cycles are spent reading data into the pipeline, and no data is written to the scratchpad memory. In the last N cycles, no more data is needed to be read into the pipeline while the CVI writes back results already in its pipeline. Currently we deal with the N cycle penalty of deep pipeline CVIs by amortizing this pipeline filling overhead over very long vector lengths, but a solution that allowed other instructions to execute using the unutilized scratchpad write and read cycles would be preferable. However, this is not trivial, especially since write bandwidth is available at the beginning of the CVI execution and read bandwidth at the end.

5.5 CVI Design Methodologies

While implementing a CVI to accelerate a SVP program is much easier than writing a complete accelerator, implementing them in HDL is not desirable for our target users, software programmers. Hence, we have explored an additional method for generating CVI pipelines that uses a high-level tool from Altera.

Altera's DSP Builder Advanced Blockset for Simulink (DSPBA) [2] is a block-based toolset integrating into Matlab and Simulink to allow for push-button generation of RTL code. Figure 5.9 shows a floating-point version of our physics pipeline implemented in DSPBA. DSPBA was able to create the entire pipeline, including accumulation units, and design was significantly faster than manually building the fixed-point version in VHDL. Although we were not able to create a fixed-

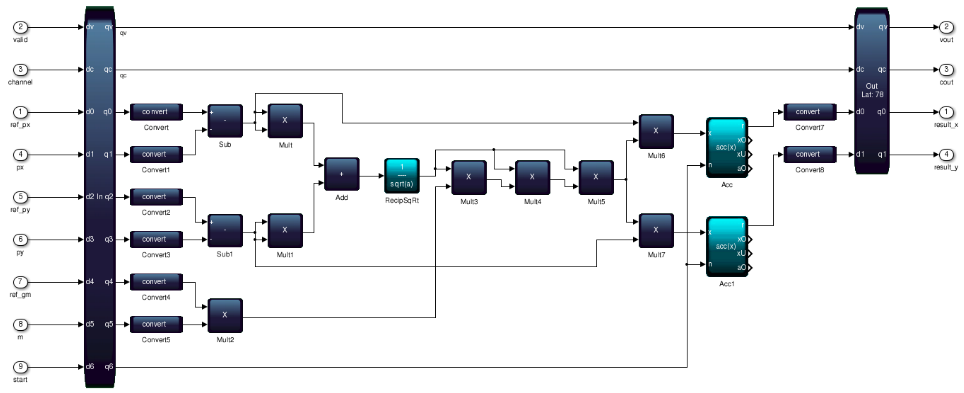


Figure 5.9: FLOAT Custom Vector Pipeline in Altera's DSP Builder

point version of our pipeline in DSPBA because it lacks fixed-point reciprocal and square root, we generated a floating-point version as an additional data point.

Glue logic was needed to integrate the pipeline into a CVI, however, because our CVI pipelines require a clock enable signal, which DSPBA-generated logic does not have. Rather than attempt to modify the output of DSPBA (including libraries used), we built a FIFO buffer to retime data appropriately, which adds minimal logic and uses one additional M9K memory per lane. This glue logic is sufficiently generic to allow any DSPBA-generated pipeline to be integrated into a CVI.

An additional method of creating CVIs was designed by Hossein Omidian using HLS techniques to generate RTL from a C function. More details can be found in [60].

5.6 Results

All FPGA results are obtained using Quartus II 13.0 and a Terasic DE4 development board which has a Stratix IV GX530 FPGA and a 64-bit DDR2 interface. For comparison, Intel Core i7-2600 and ARM Cortex-A9 (from a Xilinx Zynq-based ZedBoard) performance results are shown. Both fixed-point (fixed) and floating-point (float) implementations were used. MXP natively supports fixed-point multiplication in all lanes. The Nios II/f contains an integer hardware multiplier and hardware divider; additional instructions are required to operate on fixed-point

Table 5.1: Results with MXP Compared to Nios II/f, Intel, and ARM Processors

Processor	Area ALMs	DSPs 18-bit	f_{max} MHz	s/frame	GigaOp/s	pairs/s	Speedup
Nios II/f (fixed)	1,223	4	283	231.6	0.004	0.3M	1.0
Cortex A9 (fixed)	–	–	667	52.1	0.02	1.3M	4.5
Cortex A9 (float)	–	–	667	14.0	0.07	4.8M	16.6
Core i7-2600 (fixed)	–	–	3400	6.5	0.15	10.3M	35.6
Core i7-2600 (float)	–	–	3400	1.6	0.63	41.9M	144.8
MXP V32 (fixed)	46,250	132	193	73.8	0.14	9.1M	31.4
MXP V32+16FLOAT	115,142	644	122	0.041	24.6	1,326M	5,669
MXP V32+16FIXED	86,642	740	153	0.032	31.3	2,087M	7,203

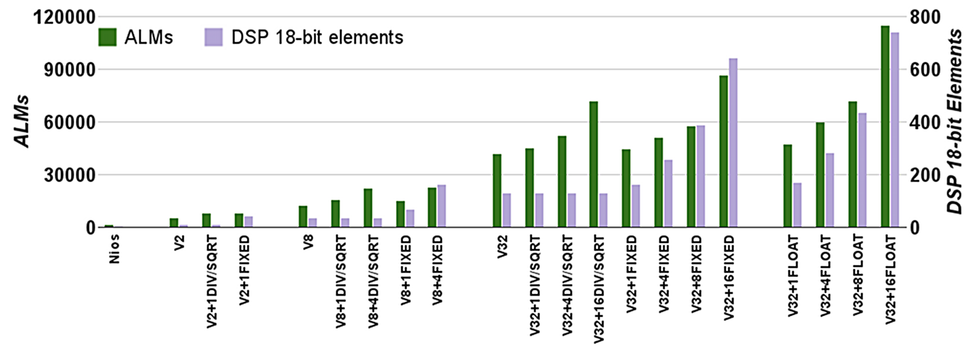


Figure 5.10: Area of Gravity Pipeline Systems

data. The Intel, ARM and Nios II versions are written with the same C source using libfixmath [9]. We developed a vectorized version of this library for use with MXP. Nios II/f and MXP results use gcc-4.1.2 with '-O2'. The Core i7 results use gcc-4.6.3 and '-O2 -ftree-vectorize -m64 -march=corei7-avx'. ARM results use gcc-4.7.2 and reports the best runtime among '-O2' and '-O3'. Our Intel and ARM code is typical of what a C programmer would start with, not highly optimized code.

When gathering the MXP results we varied the number of SVP lanes (V2, V8, and V32) and the number of CVI lanes. Three types of CVIs are generated: one containing separate fixed-point divide and square root instructions (DIV/SQRT), one containing a manually generated fixed-point gravity pipe (FIXED), and an DSPBA pipe (FLOAT).

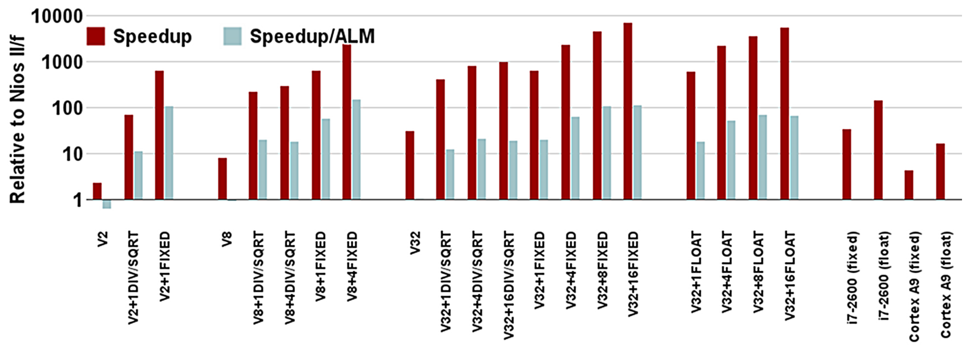


Figure 5.11: Performance and Performance-per-Area of Gravity Pipeline

Figure 5.10 shows the area, in Adaptive Logic Modules (ALMs) on the left and DSP Block 18-bit elements on the right. The DIV/SQRT configurations take roughly the same area (in ALMs) as the FIXED pipeline. However, FIXED requires more multipliers. The FLOAT pipelines require about 5,500 ALMs and 38 DSP elements per lane versus 3,000 and 32 per lane for FIXED.

Figure 5.11 shows the speedup of an algorithm to solve the N-body problem with 8192 particles. Speedup is relative to a Nios II/f soft processor, and is shown for the various MXP configurations as well as a 3.4GHz Intel Core i7-2600 and a 667MHz ARM Cortex A9. As mentioned earlier, the Intel and ARM code is basic C code and not highly optimized, but implements the algorithm exactly as our MXP code does. A highly optimized single-core AVX implementation of the N-Body problem for i7-2600 matches our best MXP performance at 2×10^9 pairs per second [64]. An exact performance comparison between MXP and an Intel processor is not intended; rather, this shows that the level of performance of an SVP with CVIs is on par with that of a hard processor.

Comparing results within the MXP designs shows the usefulness of CVIs. Without any CVIs, MXP achieves comparable performance to Nios II/f per lane, and its performance scales nearly linearly from V2 to V32. MXP is running the fixed-point divide and square root operations in software for these builds, which hampers its overall performance. Adding in divide and square root CVIs greatly improves performance, to the extent that a V2 with a single lane of divide and square root CVIs (V2+1DIV/SQRT configuration) outperforms a V32 without any CVI. Adding more lanes of the divide and square root CVIs only slightly increases performance; from V8+1DIV/SQRT to V8+4DIV/SQRT and from V32+4DIV/SQRT to V32+16DIV/SQRT performance per area decreases.

An additional order of magnitude more performance is seen going from the V2+1DIV/SQRT configuration to the V2+1FIXED configuration which has the full N-body CVI. This configuration has two orders of magnitude higher performance per area than Nios II/f. Performance of the V8+1FIXED and V32+1FIXED is essentially the same as the V2+1FIXED, showing that the kernel of the computation is running entirely on the N-body CVI, with only some housekeeping operations running on the SVP. Additional lanes of the N-body CVI increase performance to 7200× that of Nios II/f for the V32+16FIXED configuration. The FLOAT configurations created with DSPBA have slightly higher area and slightly lower performance (due to having a longer pipeline) compared to the FIXED configurations.

5.7 Summary

This chapter has presented a method of attaching custom vector instructions (CVIs) to SVPs, allowing the SVP to take advantage of custom data processing pipelines implemented in the FPGA fabric. This approach broadens the design space in which SVPs are useful by allowing the designer to acheive increased performance by migrating the most compute-intensive parts of the algorithm into custom logic. Instead of designing entirely in software or RTL, the designer is able to start in software on an SVP and create a working solution first, and only then incrementally add CVIs as needed.

Our approach reuses existing structures in SVPs to attach a variable number of streaming pipelines with minimal resource overhead. These can be accessed in software as an extension to the SVP's ISA. Logic-intensive operators, such as fixed-point divide, should not be simply replicated across all vector lanes. Doing so wastes FPGA area unnecessarily. Instead, it is important to consider the frequency of use of the specialized pipeline, and add enough copies to get the most speed-up with minimal area overhead. Methods for dispatching complex CVIs were presented, including a time-interleaved method that allows an arbitrary number of inputs and outputs using funnel adapters.

The performance results achieve speedups far beyond what a plain SVP can accomplish. For example, a 32-lane SVP achieves a speedup of 31.4, whereas a CVI-optimized version is another 230 times faster, with a net speedup of 7,200 versus Nios II/f. As a point of comparison, this puts the MXP roughly at par with reported results for an AVX-optimized Intel Core i7 implementation of the N-Body problem [64].

Chapter 6

Conclusions

What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from. — T. S. Eliot

This work has expanded the applicability of soft vector processors (SVPs), making them more efficient for applications they could already handle, and enabling new classes of applications to execute on them. Soft processors have advantages and drawbacks compared to RTL design. The software style design flow is easier to understand and faster to debug and iterate. In particular, SVPs are a good match for embedded media applications, where data can be processed as large vectors. In contrast, RTL design gives full control to do exactly what the algorithm needs with the only practical constraint being area and power budgets.

We started from the results of previous work which showed that SVP are useful, and improved upon that work with better results and new features and applications. It is important to step back and look at this work at its conclusion and try to gauge what its impact may be. In addition to the academic work (ours and the parallel work mentioned in Chapter 2), there is a commercial entity started as a result of this work. VectorBlox Computing Inc. [67] was founded in 2011, and the author has worked with that company through NSERC and MITACS scholarships. The ability of this startup company to get funding and employ multiple engineers speaks to the current relevance of SVPs, and the research work presented here in particular. That said, this work is an academic thesis and stands on its own as a set of ideas, execution of those ideas, and observations and conclusions that are useful to those who might build upon them.

6.1 Contributions

This work has demonstrated three main contributions. First, the area and performance penalty is reduced; our VENICE processor has $2\times$ the performance per area of the previous best SVP. This is done through a combination of FPGA-specific optimizations and architectural enhancements that focus on efficiency rather than maximum performance/scaling. Instead of implementing an existing vector ISA as our starting point, we designed VENICE around the FPGA hardware and thereby were able to both increase performance and reduce area.

Second, we enabled a new class of applications with wavefront skipping. For applications with divergent control flow, such as those which do a variable amount of computation depending on the input data, wavefront skipping allows for much higher performance; $3\times$ higher performance than the base SVP was observed with less than 5% area overhead. This enables SVPs to tackle computer vision algorithms such as object and face detection without wasting cycles doing unnecessary processing. These types of algorithms are also difficult to implement efficiently in hardware due to the divergent control flow, making SVPs an attractive target.

Finally, we directly addressed the gap between SVPs and RTL by allowing the use of custom vector instructions (CVIs). These allow the user to extend the SVP by connecting additional processing pipelines, created using a small amount of RTL or using a GUI. Our CVIs can attach arbitrary length pipelines to the fixed-length pipeline of a simple SVP, and can be used with an arbitrary number of inputs and outputs with little additional buffering required. This allows a user to put the most compute-intensive parts of their applications into a fixed pipeline, while still being able to do auxillary processing in software. We show how on an N-body problem we can gain speedups of $230\times$ that of the base SVP by implementing simple CVIs to perform the main force calculation, while doing all of the particle movement, bounds checking, etc. with software.

6.2 Future Work

6.2.1 Initial Work

Our current SVPs use an existing scalar core to perform address calculations, high level control flow, etc. VENICE uses Altera's Nios II/f, while MXP can use Nios II/f, Xilinx's MicroBlaze, or ARM's Cortex-A9. This was a pragmatic design decision; we can take advantage of the existing toolchain and get industry standard scalar performance without having to redesign a high performance scalar core. However, this means that no resources (such as DSP blocks) can be shared between the scalar core and vector core. Additionally, vector instructions are issued using multiple scalar instructions, which limits vector instruction issue speed and ties up the scalar processor when it could be making other progress. Fully integrating the vector and scalar cores would allow for more sharing and better instruction stream optimization. This would require either modifying an existing open-source soft processor (such as OpenRISC []) or implementing a new processor compatible with an open toolchain (such as the RISC-V project [66]).

Our wavefront skipping work was based upon the vision algorithms we studied, but it could be made more general. We mentioned in Section 4.3.6 that it would be possible to support multiple masks at the same time. Early exit algorithms only need a single mask, as elements are either being processed currently or else finished and do not need to be revisited. However, arbitrary branching in algorithms would currently require saving and restoring the contents of the mask BRAM. It may be useful to support multiple masks instead of having to save and restore when switching between branches. With a short enough maximum masked vector length (MMVL) each mask is not as deep as a full BRAM, so multiple masks could share a BRAM.

We may also be able to repurpose our multiple partition addressing logic to increase the speed of transposed matrix accesses. We can already set up strided accesses using masked vector instructions; they will run at full speed provided they do not cause bank conflicts (by being a power of two, for instance). We could also do transposed accesses in this manner for matrices that are of the correct dimensions ($2^N \pm 1$) or were padded to the correct dimensions. However, in this case, we would want to write out the destination at different offsets than we read the input, so we would have to either have a mixed addressing mode (mask inputs, flat addressed outputs) or else use multiple mask RAMs to store different mask offsets for the inputs and the outputs.

We might also want to directly accelerate strides that are powers of two. Indexing into scratchpad banks is done by taking the address modulo the width of the scratchpad, which is a power of two. Thus, power of two strides always fall within the same scratchpad bank and cannot be accelerated via wavefront skipping. Power of two strides are useful for algorithms such as fast Fourier transforms (FFTs) as well as matrix transposition. To allow parallel memory access to elements at power of two strides, they cannot be stored in the same memory bank. Prime number banking [44] can achieve this at the cost of requiring complex addressing logic. For fixed sizes, Latin squares [34] allow fully parallel access to row/column/diagonals of a matrix. Either of these alternatives could be applied to scratchpad banking or a future banked external memory system.

Regarding CVIs, larger companies such as Xilinx and Altera are seeking to move to higher level design through high level synthesis (HLS) tools such as OpenCL compilers [19, 73]. These tools allow designers to specify data flow algorithms as a series of multithreaded kernels which can be mapped into FPGA logic. While this gives a data parallel method of programming FPGAs, it still requires performing FPGA synthesis during the algorithm design and debug stages. It may be possible to combine the best of both OpenCL HLS and SVPs. Compiling OpenCL to an SVP should be relatively straightforward, though additional features would have to be added (e.g., atomic memory operations) for full support. Integrating SVPs and OpenCL HLS could be done through a custom vector instruction (CVI) interface or something similar.

6.2.2 Long Term Directions

A major point this thesis does not address is accelerating scatter/gather and power of two stride support on SVPs. Scatter/gather is the vector parlance for indexed (data-dependent) memory operations; a scatter takes an address vector and 'scatters' data to memory while a gather uses an address vector to 'gather' data from various locations in memory and load it into a packed data vector. These operations are necessary for algorithms such as sorting, graph traversal, and sparse matrix manipulation. However, fulfilling multiple memory requests per cycle is non-trivial, though. A first step towards accelerating scatter/gather operations was done with TputCache [58] which is a high-throughput cache that allowed MXP to increase performance on scatter/gather benchmarks. TputCache does not perform any memory access coalescing, but a future version could have multiple cache banks share a wide back-end that interfaces to external memory, increasing bandwidth utilization.

Longer term ambitions might include creating a wider family of soft parallel processors that can be targetted by the same software and looking at architectural support in the FPGA fabric.

One direction towards increasing performance would be to keep the vector programming model and creating a family of processors that could even better exploit existing parallelism than the straightforward SVPs we have used. We have not explored superscalar or VLIW type approaches on top of the vector paradigm, which would allow for multiple functional units (FUs) to be utilized simultaneously. Vector chaining is not straightforward on our SVPs due to the use of scratchpad pointers rather than vector registers, but would be possible for some instruction sequences. We treat our SVPs and its banked scratchpad monolithically, dispatching a single instruction per cycle, but it may be advantageous to fracture it into multiple running threads which could synchronize when possible.

Support in the FPGA fabric could come in a number of forms. Hardening of individual functions, such as having a full integer ALU as an element within the FPGA fabric, would be useful in reducing area and power, though less flexible than using soft logc. Fully hardening the vector processor itself and keeping the tightly coupled CVI interfaces to the FPGA fabric would reduce power and increase performance even more, though with even less flexibility. A hardened vector core could operate at higher frequency than the FPGA logic, but the vector model makes it straightforward to use funnel adapters where the CVI would operate at one half or one fourth of the frequency of the vector core, but use two or four times as many lanes.

6.3 Summary

This dissertation demonstrates that SVPs can be applicable in a range of applications for which they were previously not well suited. VENICE significantly reduces the performance/area penalty of using SVPs compared to RTL, showing that SVPs need not use large amounts of area to accelerate modest-performance applications. Our work on wavefront skipping shows how SVPs can be used to accelerate applications that may be difficult to implement in RTL. Our CVI interface gives a way of interfacing deep pipelines with multiple inputs and outputs to SVPs, enabling designers to harness the power of the FPGA fabric directly while retaining the software control that the SVP provides. Together, these contributions help SVPs to be a much more useful tool for implementing data parallel applications on FPGAs.

Bibliography

[1] Altera Corporation. Nios II compiler user guide, . URL http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf. Accessed December 2009. → pages 19

[2] Altera Corporation. DSP builder, . URL http://www.altera.com/technology/dsp/advanced-blockset/dsp-advanced-blockset.html. Accessed September 2013. → pages 92

[3] Altera Corporation. Increasing productivity with Quartus II incremental compilation, . URL http://www.altera.com/literature/wp/wp-01062-quartus-ii-increasing-productivity-incremental-compilation.pdf. Accessed September 2014. → pages 13

[4] Altera Corporation. Nios II processor: The world's most versatile embedded processor, . URL http://www.altera.com/devices/processor/nios2/ni2-index.html. Accessed September 2014. → pages 3, 14, 27

[5] Altera Corporation. Stratix V FPGA family overview, . URL http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/overview/stxv-overview.html. Accessed September 2014. → pages 12

[6] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998. Technical Report UCB-CSD-98-1014. → pages 18

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a Scala embedded language. *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225, 2012. → pages 2

[8] A. Brant and G. Lemieux. Zuma: An open FPGA overlay architecture. *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 93–96, 2012. → pages 14

[9] B. Brewer. libfixmath - cross platform fixed point maths library. URL http://code.google.com/p/libfixmath/. Accessed September 2013. → pages 94

[10] B. Brousseau and J. Rose. An energy-efficient, fast FPGA hardware architecture for OpenCV-compatible object detection. *International Conference on Field-Programmable Technology*, pages 166–173, Dec 2012. → pages 68

[11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems*, 13(2):24:1–24:27, 2013. → pages 2

[12] D. Capalija and T. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. *International Conference on Field Programmable Logic and Applications*, pages 1–8, Sept 2013. → pages 4

[13] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. *Symposium on Application Specific Processors*, pages 101–107, 2008. → pages 1

[14] D. Chen, J. Cong, and P. Pan. FPGA design automation: A survey. *Foundations and Trends in Electronic Design Automation*, 1(3):139–169, 2006. → pages 13

[15] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux. VEGAS: Soft vector processor with scratchpad memory. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 15–24, 2011. → pages x, 5, 20, 30

[16] J. Cong, M. A. Ghodrat, M. Gill, H. Huang, B. Liu, R. Prabhakar, G. Reinman, and M. Vitanza. Compilation and architecture support for customized vector instruction extension. *Asia and South Pacific Design Automation Conference*, pages 652–657, 2012. → pages 27

[17] T. E. M. B. Consortium. EEMBC - the embedded microprocessor benchmark consortium. URL http://www.eembc.org/benchmark/products.php. Accessed January 2015. → pages 44

[18] Convey Computer. The Convey HC-2 computer: Architectural overview. URL http://www.conveycomputer.com/files/4113/5394/7097/Convey_HC-2_Architectual_Overview.pdf. Accessed September 2013. → pages 22, 27

[19] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. *International Conference on Field Programmable Logic and Applications*, pages 531–534, 2012. → pages 2, 101

[20] R. Duncan. A survey of parallel computer architectures. *IEEE Transactions on Computers*, 23(2):5–16, 1990. → pages 15

[21] R. Espasa, M. Valero, and J. E. Smith. Vector architectures: Past, present and future. *International Conference on Supercomputing*, pages 425–432, 1998. → pages 15

[22] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown. A multithreaded soft processor for SoPC area reduction. *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 131–142, 2006. → pages 19

[23] W. Fung and T. Aamodt. Thread block compaction for efficient SIMT control flow. *International Symposium on High Performance Computer Architecture*, pages 25–36, Feb 2011. ISSN 1530-0897. → pages 26

[24] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. *IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007. → pages 26

[25] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE/ACM International Symposium on Microarchitecture*, 20(2):60–70, 2000. → pages 27

[26] E. Hung and S. J. Wilton. Accelerating FPGA debug: Increasing visibility using a runtime reconfigurable observation and triggering network. *ACM Transactions on Design Automation of Electronic Systems*, 19(2): 14:1–14:23, 2014. → pages 2

[27] Intel Corporation. Intel instruction set extensions, . URL https://software.intel.com/en-us/intel-isa-extensions. Accessed December 2014. → pages 4, 16

[28] Intel Corporation. Intel Xeon Phi coprocessor: Datasheet, . URL http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html. Accessed December 2014. → pages 16

[29] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. An architecture of highly parallel computer AP 1000. *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 13–16, 1991. → pages 25

[30] D. Kanter. Intel's Haswell CPU microarchitecture. URL http://www.realworldtech.com/haswell-cpu/. Accessed December 2012. → pages 18

[31] N. Kapre and A. DeHon. Accelerating SPICE model-evaluation using FPGAs. *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 37–44, 2009. → pages 14

[32] N. Kapre, N. Mehta, M. Delorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. Dehon. Packet switched vs. time multiplexed FPGA overlay networks. *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 205–216, 2006. → pages 14

[33] J. Kathiara and M. Leeser. An autonomous vector/scalar floating point coprocessor for FPGAs. *IEEE Symposium on Field Programmable Custom Computing Machines*, pages 33–36, 2011. → pages 22

[34] K. Kim and V. K. Prasanna. Latin squares for parallel array access. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):361–370, Apr. 1993. ISSN 1045-9219. → pages 101

[35] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, et al. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983. → pages 13

[36] C. Kozyrakis. *Scalable Vector Media Processors for Embedded Systems*. PhD thesis, University of California at Berkeley, May 2002. Technical Report UCB-CSD-02-1183. → pages 18

[37] C. Kozyrakis and D. Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. *IEEE/ACM International Symposium on Microarchitecture*, pages 283–293, 2002. → pages 4

[38] R. Krashinsky, C. Batten, M.Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. *International Symposium on Computer Architecture*, pages 52–63, June 2004. → pages 50

[39] D. J. Kuck and R. Stokes. The Burroughs Scientific Processor (BSP). *IEEE Transactions on Computers*, C-31(5):363–376, May 1982. ISSN 0018-9340. → pages 24

[40] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007. → pages 11

[41] I. Kuon, R. Tessier, and J. Rose. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008. → pages 11

[42] C. E. LaForest and J. G. Steffan. OCTAVO: an FPGA-centric processor family. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 219–228, 2012. → pages 4

[43] Lattice Semiconductor Corporation. LatticeMico32 open, free 32-bit soft processor. URL http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx. Accessed December 2014. → pages 3

[44] D. H. Lawrie and C. Vora. The prime memory system for array access. *IEEE Transactions on Computers*, C-31(5):435–442, May 1982. ISSN 0018-9340. → pages 101

[45] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *ACM Special Interest Group in Computer Architecture News*, 39(3):129–140, 2011. → pages 26

[46] Z. Liu, A. Severance, S. Singh, and G. G. Lemieux. Accelerator compiler for the VENICE vector processor. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 229–232, 2012. → pages iv, 22, 31

[47] A. Ludwin and V. Betz. Efficient and deterministic parallel placement for FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 16 (3):22:1–22:23, 2011. → pages 2, 13

[48] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU performance via large warps and two-level warp scheduling. *IEEE/ACM International Symposium on Microarchitecture*, pages 308–317, 2011. → pages 26

[49] M. Naylor and S. W. Moore. Rapid codesign of a soft vector processor and its compiler. *International Conference on Field Programmable Logic and Applications*, pages 1–4, 2014. → pages 22

[50] M. Naylor, P. Fox, A. Markettos, and S. Moore. Managing the FPGA memory wall: Custom computing or vector processing? *International Conference on Field Programmable Logic and Applications*, pages 1–6, Sept 2013. → pages 22

[51] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. *ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 69–70, 2004. → pages 2, 22

[52] K. Ovtcharov, I. Tili, and J. Steffan. TILT: A multithreaded VLIW soft processor family. *International Conference on Field Programmable Logic and Applications*, pages 1–4, Sept 2013. → pages 4

[53] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. → pages 26

[54] K. Papadimitriou, A. Dollas, and S. Hauck. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems*, 4(4):36:1–36:24, 2011. → pages 3

[55] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993. → pages 18

[56] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978. ISSN 0001-0782. → pages 15

[57] A. Severance and G. Lemieux. VENICE: A compact vector processor for FPGA applications. *International Conference on Field-Programmable Technology*, pages 261–268, Dec 2012. → pages iv

[58] A. Severance and G. Lemieux. TputCache: High-frequency, multi-way cache for high-throughput FPGA applications. *International Conference on Field Programmable Logic and Applications*, pages 1–6, Sept 2013. → pages 25, 102

[59] A. Severance and G. G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. *International Conference on Hardware/Software Codesign and System Synthesis*, pages 6:1–6:10, 2013. → pages 22, 36, 53

[60] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft vector processors with streaming pipelines. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 117–126, 2014. → pages v, 93

[61] A. Severance, J. Edwards, and G. Lemieux. Wavefront skipping using BRAMs for conditional algorithms on vector processors. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb 2015. → pages iv

[62] J. E. Smith. Density dependent vector mask operation control apparatus and method, 1996. URL http://www.google.com/patents/US5940625. US Patent 5,940,625. → pages 25

[63] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. *ACM Special Interest Group in Computer Architecture News*, 28(2):260–269, May 2000. ISSN 0163-5964. → pages 7, 23, 26, 52

[64] A. Tanikawa, K. Yoshikawa, K. Nitadori, and T. Okamoto. Phantom-GRAPE: numerical software library to accelerate collisionless N-body simulation with SIMD instruction set on x86 architecture. *arXiv.org*, Oct. 2012. → pages 95, 97

[65] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. Technical Report MSR-TR-2005-184, Microsoft Research, October 2006. URL http://research.microsoft.com/apps/pubs/default.aspx?id=70250. → pages 22

[66] University of California, Berkeley. RISC-V. URL http://riscv.org/. Accessed December 2014. → pages 100

[67] VectorBlox Computing, Inc. VectorBlox - embedded supercomputing. URL http://vectorblox.com/. Accessed September 2014. → pages 98

[68] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1:511–518, 2001. → pages 7

[69] C. Wang and G. Lemieux. Scalable and deterministic timing-driven parallel placement for FPGAs. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 153–162, 2011. → pages 2, 13

[70] M. Weiss. Strip mining on SIMD architectures. *International Conference on Supercomputing*, pages 234–243, 1991. → pages 4, 56

[71] H. Wong, V. Betz, and J. Rose. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 5–14, 2011. → pages 3, 15, 64

[72] Xilinx, Inc. MicroBlaze soft processor core, . URL http://www.xilinx.com/tools/microblaze.htm. Accessed September 2014. → pages 3, 27

[73] Xilinx, Inc. All programmable abstractions, . URL http://www.xilinx.com/content/xilinx/en/products/design-tools/all-programmable-abstractions/#software-based. Accessed December 2014. → pages 101

[74] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, 2008. → pages 4, 19, 78

[75] P. Yiannacouras, J. G. Steffan, and J. Rose. Fine-grain performance scaling of soft vector processors. *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 97–106, 2009. → pages 19, 27

[76] P. Yiannacouras, J. G. Steffan, and J. Rose. Data parallel FPGA workloads: Software versus hardware. *International Conference on Field Programmable Logic and Applications*, pages 51–58, 2009. → pages 19

[77] P. Yiannacouras, J. G. Steffan, and J. Rose. Data parallel fpga workloads: Software versus hardware. *International Conference on Field Programmable Logic and Applications*, pages 51–58, 2009. → pages 5

[78] J. Yu, C. Eagleston, C. H. Chou, M. Perreault, and G. Lemieux. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems*, 2(2):1–34, 2009. → pages 4, 5, 19