Accelerator Compiler for the VENICE Vector Processor

by

Zhiduo Liu

B.A.Sc, Harbin Institute of Technology, 2009

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

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

September 2012

Abstract

This thesis describes the compiler design for VENICE, a new soft vector processor (SVP). The compiler is a new back-end target for the Microsoft Accelerator, a high-level data parallel library in C/C++ and C#. This allows automatic compilation from high-level programs into VENICE assembly code, thus avoiding the process of writing assembly code used by previous SVPs. Experimental results show the compiler can generate scalable parallel code with execution times that are comparable to human-optimized VENICE assembly code. On data-parallel applications, VENICE at 100MHz on an Altera DE3 platform runs at speeds comparable to one core of a 2.53GHz Intel Xeon E5540 processor, beating it in performance on four of six benchmarks by up to 3.2×. The compiler also delivers near-linear scaling performance on five of six benchmarks, which exceed scalability of the Multi-core target of Accelerator.

Preface

[1] Zhiduo Liu, Aaron Severance, Satnam Singh and Guy G. F. Lemieux, “Accelerator Compiler for the VENICE Vector Processor," in *Proceedings of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 229 - 232.

Portions of chapter 3 and 5 have been published at FPGA 2012 [1].

Table of Contents

List of Tables

List of Figures

Glossary

ALU Arithmetic Logic Unit 3

ARBB Array Building Blocks 3

DAG Directed Acyclic Graph 26

IR Intermediate Representation 8

LIR Linear Intermediate Representation 28

CSE Common Subexpression Elimination 27

PA Parallel Array 25

SVP Soft Vector Processor 3

GPU Graphics Processing Unit 23

ISA Instruction Set Architecture 82

SOC System on Chip 3

CUDA Compute Unified Device Architecture

FPGA Field Programmable Gate Array 3

API Application Programming Interface 15

VL Vector Length 17

DMA Direct Memory Access 4

GPR General Purpose Register 80

SSE Streaming SIMD Extensions 24

SIMD Single Instruction, Multiple Data 1

Acknowledgments

I would like to thank my supervisor Dr. Guy Lemieux for all his support, fanancially and academically, over the past three years. I am very lucky to become a student of Professor Lemieux and being involved in cutting edge FPGA technology research projects. This thesis would not become possible without his wealth of experience and knowledge, his innovative ideas, and his insights on the project.

I want to give special thanks to Microsoft Accelerator group for accepting me as an intern and teaching me everything about the Accelerator system. They are also very generous and granted me access to part of the Accelerator source code, which makes it possible for me to conduct this study.

Thanks to all the instructors of all the courses I've taken at UBC. I really appreciate their fascinating lectures, their patience for students, and their great kindness and help. All I've learned in the past three years have been solid foundation for the completion of my thesis, and will become a precious gift in the future of my life.

Thanks to all the members of the SoC lab for the support and help. To David Grant, who inspired me and helped me on several course projects. To Aaron Severance, who provided valuable suggestions and continuously efforts on vector processor design and maintainance. To Chris Wang, who made jokes all the time and brought laughters to the lab.

In the end, I want to give my appreciation to my parents who teach me right, and always support me in all aspects. To my husband Zheng, I feel so blessed for marrying you and having you always on my side taking care of me.

Chapter 1

Introduction

1.1 Motivation and Research Goals

As modern processors have hit the power wall, we are embracing the parallel processing era with a wide range of hardware architectures, parallel programming languages, and auto-parallelizing compilers being extensively studied across the world.

Today's hardware offerings range from general-purpose chips with a few cores, to graphic processors that support large-scale data-parallel applications, to reconfigurable hardware with massive bit-level parallelism. Additionally, several specialized Single Instruction, Multiple Data (SIMD) platforms such as vector processors are being actively investigated.

The fast growth of novel architectures with diverse components and increasing levels of parallelism, either for general acceleration or specialized for certain domain applications, raises great challenges for software developer's to effectively program them.

Using traditional sequential programming languages such as C/C++ to program modern parallel platforms creates substantial difficulty for compiler developers. It requires the compiler to identify the parallelizable part, analyze complex data dependencies, and orchestrate data movement and synchronization among parallel units in the target hardware architecture. This approach is complex and usually renders limited performance.

Several specialized library languages, such as CUDA and OpenMP, have demonstrated some success. Most of these languages expose low-level details of the device architecture. The user has to perform operations such as explicitly specifying data movement between different memory spaces, creating a certain number of threads, and synchronizing among threads to better exploit the processing resources. These explicit operations provide a great help to the compiler towards parallelizing the kernel. Many publications have shown that these languages can yield much easier compiler development and good runtime performance. However, they require users to apply knowledge of hardware details and advanced programming skills. These languages usually target only one platform, being either GPU, or Multi-core/Multi-threaded architecture, or computer clusters. Users have to learn a new language or even several new languages for each new device coming to the market. In addition, in order to optimize for performance, users are usually required to learn hardware architecture in detail. This is obviously not a sustainable solution.

The ideal solution is to move towards hardware independent, portable library languages based on existing primitive ones today. Hardware-independent languages abstract away hardware details and present the user with a simple and high-level perspective. This higher abstraction is considered to be more portable and

less error-prone and it delivers higher programmer productivity. In addition, programmers would prefer these languages to completely new ones because they are more similar to current primitive languages.

There are several such projects being actively researched around the world. To name a few: Lime from IBM is a Java based language that can be ported among GPU, Multi-core and FPGA devices; Array Building Blocks (ARBB) from Intel is a C++ based library that can be compiled to Multi-core and heterogeneous many-core architectures; and Microsoft Accelerator supports multiple primitive languages including C, C++, C#, and F#. Accelerator can be compiled to GPU, Multi-core, and FPGA using the same source code.

These projects open the opportunity to provide effective, scalable solutions for future parallel architectures. For this thesis, it is compelling to investigate the feasibily of targeting one of these new high-level abstract languages to vector processors. In particular, the VENICE architecture [2] is one of the newest and unique vector architectures.

VENICE is a research project conducted at the System on Chip (SOC) lab of the University of British Columbia. It has evolved through several generations since 2007 [2–5] and is presently being commercialized. VENICE inherited the SIMD model from general vector processing that provides massive data parallelism. It also provides great flexibility with low cost as being a Soft Vector Processor (SVP) running on Field Programmable Gate Arrays (FPGAS). It can be configured to instantiate multiple vector lanes, where each lane contains a 32-bit Arithmetic Logic Unit (ALU). In addition, it has several novel features:

- It is smaller and faster than all previously published SVPs.

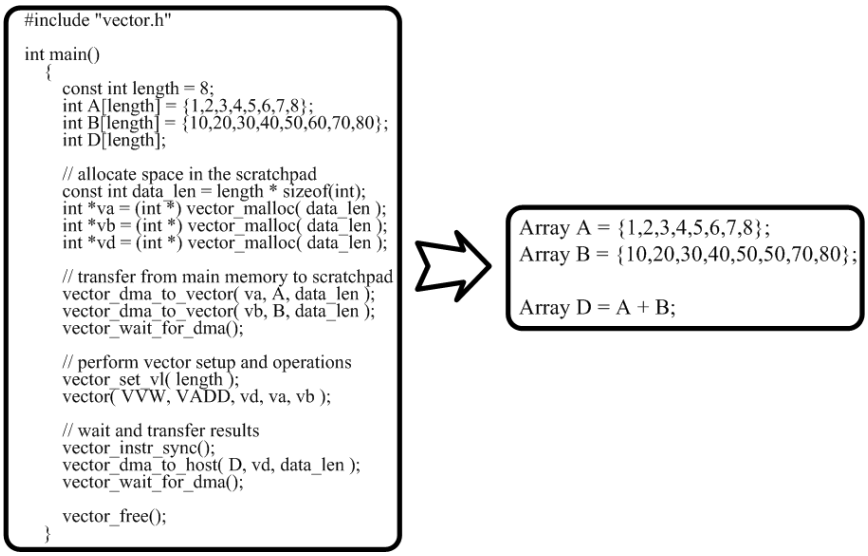

Figure 1.1: Design goal for VENICE compiler

- It uses a multi-ported scratchpad memory for concurrent data accesses by the ALUs and the Direct Memory Access (DMA) engine.
- Each ALU can be further fractured into sub-word ALUs that work concurrently, enabling better performance on vectors of bytes or half words.
- It can be programmed with a C-like inline assembly library system.
- It provides user-level flexibility with multiple configurable components.
- It uses a novel predicated move and flag system to handle data overflow and comparisons.

Although the C-like inline assembly somewhat eases the programmability of VENICE compared to previous generations, it can still be difficult to program. For example, the two code fragments in figure 1.1 show native VENICE code on the left adding two vectors, and an idealized array-based object system similar to

Microsoft Accelerator describing the same computation on the right. The cumbersome assembly programming style, detailed hardware manipulation, and difficulty of debugging create the need for a higher-level programming model to provide simplicity, efficiency, and convenience. As explained, developing a new high-level language specially for VENICE is not practical. Programming VENICE with an existing portable higher-level abstraction could save development effort, and allows the user to write simple and short expressions at the same time.

1.2 Contributions

This work presents a compiler design that serves as a vectorizing compiler/back-end code generator for VENICE based on Microsoft's Accelerator framework. Our main contributions are as follows:

1. To our knowledge, this is the first compiler designed for a vector processor with a scratchpad memory.

   VENICE operates on a scratchpad memory, which can contain a variable number of vectors with arbitrary vector length. The multi-banked scratchpad memory allows concurrent read/write by multiple ALUs and the DMA engine. The compiler simplifies the challenges raised by the novel features of VENICE by treating the scratchpad memory as a 'virtual vector registerfile'.

2. The compiler produces highly optimized VENICE assembly, and achieves robust end-to-end performance, which is close to or even better than hand-optimized code.

The compiler applies a double-buffering technique to effectively hide memory-transfer overhead. It uses a modified Sethi-Ullman algorithm to determine the evaluation order of sub-expressions, and a reference-counting method to precisely calculate the exact number of vector registers required by a computation. A profiling approach is adopted to tune compiler performance.

3. The compiler greatly improves the programming and debugging experience for VENICE.

   Before this work, VENICE can only be programmed using C-like inline assembly. Accelerator is a high-level abstraction language that eliminates the manual effort of assembly programming on performing operations such as memory transfers and synchronization of vector instructions. Visual Studio debugger is available for Accelerator, which saves the process of downloading to an FPGA board to check results using print statements.

Although data management for architectures with scratchpad memory has been studied before, and it is still an on-going research area, such work focuses exclusively on scalar architectures (single ALU) co-existing with data caches. There are also some works on compiler support for hard vector processors, such as VIRAM's vcc compiler and Cray's compiler, but these vector processor designs have a fixed-size register file which is divided into exact N_1 vectors with exact N_2 maximum elements each. In contrast, the scratchpad memory in VENICE is flexible and can

be of arbitrary size. It supports an arbitrary number of vectors, each of an arbitrary length, subject only to the maximum capacity. Further more, the DMA engine connecting the scratchpad and main memory has a dedicated read/write port to the scratchpad. The DMA queue and vector instruction queue in VENICE allows control processor to return immediately after dispatching a DMA or vector instruction. These allow the scalar core, the DMA engine, and the vector engine to operate in parallel. Compiler design for an architecture with such novel features has never been studied before.

The experimental results show that across a set of selected benchmarks, the VENICE compiler delivers a speedup (using a Nios II processor as the baseline) up to 362× using the biggest VENICE configuration. Compared to a modern Intel processor, it provides speedup of up to 3.2×. Furthermore, the compiler-generated code can achieve speedups between 0.81× and 2.24× of hand-tuned VENICE assembly code. In addition, when compared to the Accelerator Multi-core target, the compiler reveals almost linear scaling performance from 1 to 64 ALUs, whereas the Multi-core target performance saturates beyond 4 cores.

1.3 Approach

When searching for a suitable compiler framework for VENICE, ease-of-use and ease-of-compiler-development were the two most important factors under consideration. In particular, it is desirable to have a clean implementation language for the user, and have parallelism as explicit as possible in order to avoid complex dependency analysis algorithms within the compiler.

In comparison with other systems, Microsoft Accelerator has a number of features that make it ideal for our goals and it is chosen as the target high-level lan-

guage for our compiler:

1. It operates as an embedded library in C/C++, C# and F#. Therefore, it provides the user with a certain amount of flexibility in choosing a language.

2. It has a vector-like description model which matches the VENICE architecture extremely well. In many cases, an almost direct translation approach can be used by the compiler. Special parallel-array data types simplify the task of identifying the parallelizable part and managing data movement between main memory and scratchpad memory.

3. The built-in Accelerator front-end provides an Intermediate Representation (IR) with sufficient static information for the back-end target to analyze, optimize, and parallelize.

4. It has built-in support for pluggable 3rd-party developed targets (back-end compilers), so adding a new target is easy. Most other systems do not open this option for 3rd-party developers.

5. It provides a sufficiently rich expressiveness to efficiently implement our benchmarks, and yet restricts users from using operations that would destroy data parallelism through unstructured array accesses.

6. Source code is accessible directly from Microsoft.

In this thesis, the compiler operates as a source-to-source translation compiler. It compiles the Accelerator code into the low-level native VENICE code. The resulting C code needs to be compiled with gcc before being downloaded to an FPGA board.

As previously mentioned, the scratchpad memory is treated as a 'virtual vector registerfile'. However, the number of vector registers and vector length can be arbitrary. They are dynamically decided by the compiler based on user input data array sizes and computation tree structure. A modified Sethi-Ullman algorithm is adopted to minimize the number of vector registers used by the generated code. A double-buffering technique is used to hide memory latencies caused by transferring data between scratchpad memory and main memory. Highly-efficient usage of the scratchpad memory is the main optimization goal of this work.

1.4 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 presents background knowledge of the VENICE architecture, the VENICE native programming interface, the Microsoft Accelerator system, basic compiler algorithms this work is based on, and related works. Chapter 3 describes design details of this compiler including platform-specific optimizations and Chapter 4 shows how compiler performance is further improved. Chapter 5 describes the benchmarks used as well as experimental methods and results. Chapter 6 lists some future works under consideration. In the end, Chapter 7 concludes the whole thesis.

Chapter 2

Background

This chapter will first introduce the VENICE vector processor architecture, with an emphasis on key features that are important to compiler design. Then it will discuss some state-of-art parallel programming languages and projects similar to Microsoft's Accelerator, including Intel's ArBB, IBM's Lime, MATLAB's parallel computing toolbox, and the VIRAM vcc compiler. The Microsoft Accelerator system and programming model will then be described. This chapter will also introduce some general compiler algorithms related to this thesis.

2.1 Vector Processors

Vector processing has been applied on scientific and engineering workloads for decades [6, 7]. It exploits the data-level parallelism readily available in applications by performing the same operation over all elements in a vector or matrix. It is also well-suited for image processing, signal processing, and multi-media applications.

The Cray-1 is the main ancestor of modern vector processors. Developed during the early 1970s, Cray was the first vector processor to adopt a RISC-like load/s-

tore architecture, where all vector operations are executed register-to-register. The Cray processor spanned several hundred circuit boards filling a large six-foot tall cabinet.

VIRAM is a widely-cited embedded vector processor developed in the research labs at the University of California Berkeley [8]. It also uses a load/store architecture like Cray, but it is implemented on a single chip.

2.1.1 Soft Vector Processors

Compared to hard processors or other complex heterogeneous architectures, soft vector processors running on FPGAs offer a fast, low-cost, and configurable solution with ultra-high performance. They are designed to do data-parallel processing with less development effort than hardware design using VHDL or Verilog.

The VIPERS soft vector processor by Yu from the University of British Columbia [3, 4] is the first published soft vector processor with configurable parameters. It serves as a general-purpose data-parallel accelerator on an FPGA. It uses a Nios II-compatible multi-threaded processor called UT-IIe as control processor.

The VESPA soft vector architecture developed by Yiannacouras at the University of Toronto [9, 10] implements a MIPS-compatible scalar core with a VIRAM-compatible vector coprocessor.

Like Cray and VIRAM, both VIPERS and VESPA adopt a load/store architecture to transfer blocks of data into a vector data register file. In each case, the register file storage is divided evenly among a fixed number of vector registers. Hence, each vector register contains a fixed number of words. In order to offer dual read ports for simultaneous reading of two operands, the vector register file is

duplicated, which is costly for precious on-chip memory.

The VEGAS project [5] made a significant departure from past vector architectures. VEGAS shares most similarities with VENICE, which is the target architecture of this thesis. It uses a Nios II/f as a control processor. Instead of vector data registers, VEGAS uses a large multi-banked scratchpad memory. Vectors in scratchapd are indexed by an 8-entry vector address register file. This organization allows for a large number of vectors in the scratchpad, but only 8 of them can be accessed immediately by any instruction; accessing the other vectors involves spilling the vector address register file contents.

VEGAS uses DMA block-read and block-write commands to copy between the scratchpad and an off-chip main memory (DDR2). The scratchpad memory completely removes the load/store latencies and data duplication caused by using vector data registers. As a result, VEGAS operates memory-to-memory on the scratchpad, and can store a large number of vectors of arbitrary length up to the maximum scratchpad capacity. Furthermore, the ALUs can be fractured to support sub-word arithmetic. The ability to use arbitrarily long vectors and have more parallelism on byte or half word data make VEGAS much more efficient.

2.2 VENICE Architecture

The VENICE soft vector processor was developed as an improvement to the VEGAS [5] architecture. A block diagram of the VENICE architecture is shown in figure 2.1.

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.

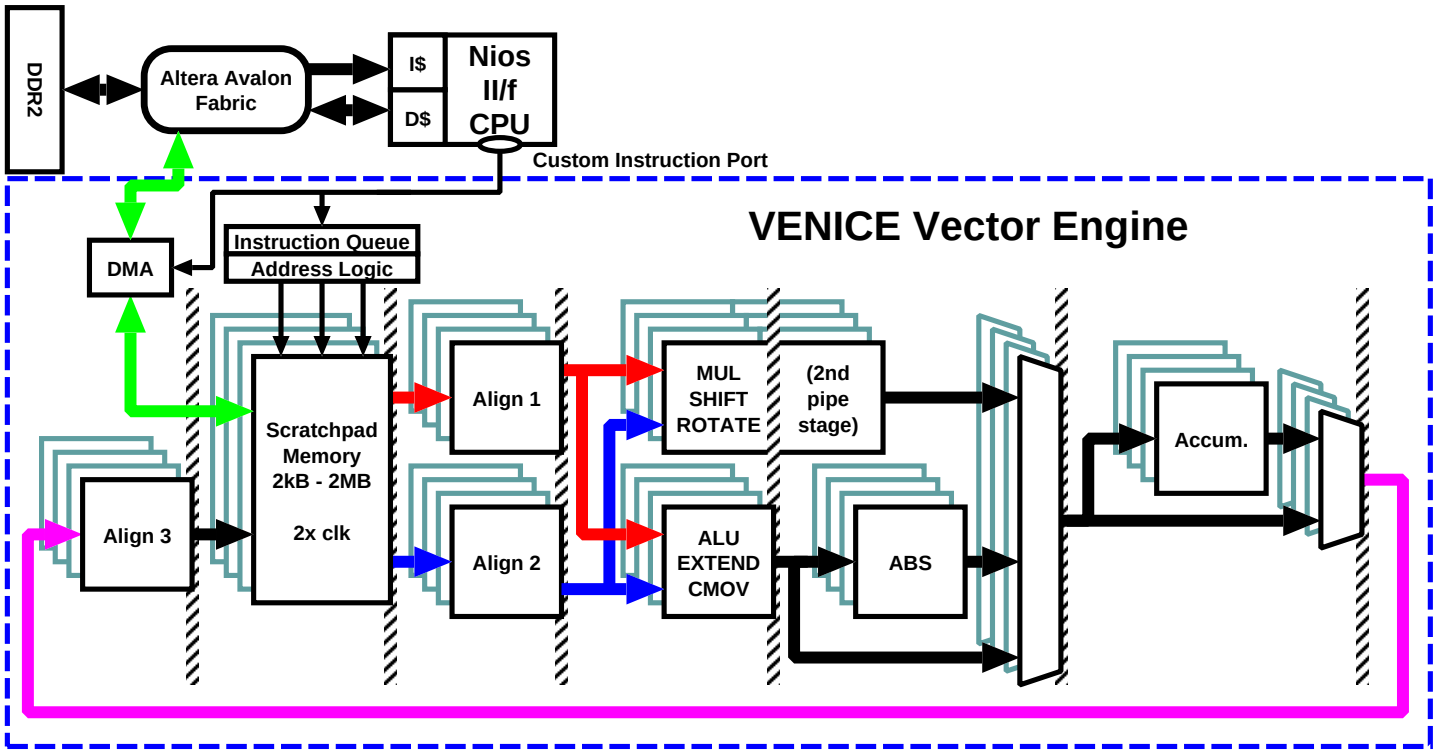

Figure 2.1: VENICE architecture (gray vertical bars are pipeline registers)

VENICE vector operations are dispatched through the Nios custom instruction interface to the vector accelerator. Similar to VEGAS, the vector engine implements a wide multi-ported scratchpad memory, in which vector operations take place, along with a DMA engine for controlling transfers to and from main memory. The scratchpad memory has two read and one write ports dedicated to each vector lane (32-bit ALU). This allows reading two operands for ALU execution and writing one result back in every cycle. It also has a dedicated read/write port for the DMA engine, which allows reading/writing data in parallel with ALU operations. The configurable number of vector lanes is the main form of parallelism for VENICE. All vector lanes receive the same vector instruction from the instruction queue and perform the operation concurrently. Similar to VEGAS, the 32-bit ALUs can be collectively fractured to support sub-word SIMD parallelism, enabling operations on vectors of bytes or half words with more parallelism. Memory-level parallelism and instruction-level parallelism are attained through concurrent operations of the scalar core, the DMA engine, and the vector core. The arbitrary vector length, as well as the ability to achieve memory-level and instruction-level parallelism raise interesting challenges for compiler design.

Besides the inherited features, VENICE makes the following improvements upon the VEGAS architecture:

- Unlike VEGAS, all vector address registers are removed. This saves the programmer from tracking and spilling the vector address registers. It would also be an extra layer of work for compiler. Instead of the vector address registers, C pointers are directly used as operands for vector instructions.

- Scratchpad-based designs require alignment networks to allow for cases

where the input operands and output are not aligned. VENICE uses 3 alignment networks in the pipeline to remove the performance penalty in VEGAS, which has only one alignment network.

- Support for 2D data structures achieves a high instruction dispatch rate even in the case of short vectors. These 2D instructions eliminate the need for the auto-increment mode of the vector address registers in VEGAS. Furthermore, since the 2D instruction can perform in a strided manner, it is easy to extract sub-matrices.

- The shared multiplier/shift/rotate structure requires two cycles operational latency, allowing a general absolute value stage (the ABS block in figure 2.1) to be added after the integer ALU.

As a result of these and other optimizations, the design was pipelined to reach speeds of 200MHz, which is 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. All of these unique features of VENICE are under consideration during the development of the compiler.

2.2.1 Native VENICE Programming Interface

Using inline C function to program VENICE

In order to simplify programming, C macros are used to make VENICE instructions look like C functions without adding any runtime overhead. The sample code in figure 2.2 adds three vectors together using the native VENICE Application Programming Interface (API).

```
1    #include "vector.h"
2
3
4    int main()
5    {
6        const int length = 8;
7        int A[length] = {1,2,3,4,5,6,7,8};
8        int B[length] = {10,20,30,40,50,60,70,80};
9        int C[length] = {100,200,300,400,500,600,700,800};
10       int D[length];
11
12       // allocate space in the scratchpad
13       const int data_len = length * sizeof(int);
14       int *va = (int *) vector_malloc( data_len );
15       int *vb = (int *) vector_malloc( data_len );
16       int *vc = (int *) vector_malloc( data_len );
17
18       flush_cache_all();
19
20       // transfer from main memory to scratchpad
21       vector_dma_to_vector( va, A, data_len );
22       vector_dma_to_vector( vb, B, data_len );
23       vector_dma_to_vector( vc, C, data_len );
24       vector_wait_for_dma();
25
26       // perform vector setup and operations
27       vector_set_vl( length );
28       vector( VVW, VADD, vb, va, vb );
29       vector( VVW, VADD, vc, vb, vc );
30
31       // wait and transfer results
32       vector_instr_sync();
33       vector_dma_to_host( D, vc, data_len );
34       vector_wait_for_dma();
35
36       vector_free();
37   }
```

Figure 2.2: Native VENICE API to add 3 vectors

Each macro dispatches one or more vector 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 may return immediately before the operation is finished. This allows several instructions to be executed concurrently if they operate on different components. In addition, some macros are used to restore synchrony and explicitly wait until the vector engine or DMA engine is finished.

As implied by figure 2.2, after initializing input data array (line 7-10), the VENICE programming model follows a few general steps:

1. Allocation of memory in scratchpad (line 14-16)
2. Optionally flush data in data cache (line 18)
3. DMA transfer data from main memory to scratchpad (line 21-23)
4. Wait for DMA transaction to be completed (line 24)
5. Setup for vector instructions, e.g., the Vector Length (VL) (line 27)
6. Perform vector operations (line 28-29)
7. Wait for all vector operations to be completed (line 32)
8. DMA transfer resulting data back to main memory (line 33)
9. Wait for DMA transaction to be completed (line 34)
10. Deallocate memory from scratchpad (line 36)

The basic vector instruction format is `vector(VVWU, FUNC, VD, VA, VB)`. The `VVWU` specifier refers to 'vector-vector' operation (`VV`) on 32-bit integer type data (`W`) that is unsigned (`U`). The vector-vector part can instead be scalar-vector (`SV`), where the first source operand is a scalar value provided by Nios.

These may be combined with data sizes of bytes (`B`), half words (`H`) or words (`W`). A signed operation is designated explicitly by using the signed specifier (`S`) or implicitly by omitting the unsigned specifier (`U`).

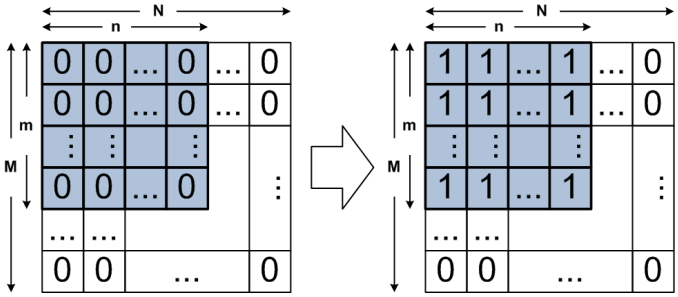

Figure 2.3: Extracting sub-matrix from a 2D array

One important feature of VENICE mentioned in the previous section is the ability to operate on two-dimensional arrays. This allows a smooth flow of vector operations on a 2D array instead of individual 1D operations on each row. The user can also operate on a sub-matrix of a 2D array by specifying strides of each vector operand or destination vector. For example, to extract the shaded `m×n` sub-matrix from the original `M×N` matrix in figure 2.3, the following vector setup instructions are needed:

```
vector_set_vl(n);
vector_setup_2D(m, 4*(N-n), 0, 4*(N-n));
```

The vector length is specified by the set_vl instruction indicating `n` elements will be processed for each row. In the setup_2D instruction, the first number `m` indicates the number of rows that will be involved in the following 2D vector instructions. The second number is the row stride for destination vector, meaning `4*(N-n)` bytes (`N-n` words) will be skipped at the end of each row after a vector length of `n` is written into the destination vector. The third number indicates there

is no stride for the first source operand. The last number `4*(N-n)` means the number of bytes will be skipped at the end of each row for the last source operand. Now, adding a scalar value of 1 to the sub-matrix to obtain the right-side matrix in figure 2.3 can be as simple as :

```
vector(SVW, VADD, VA, 1, VA);
```

In addition to basic arithmetic, conditional moves of individual vector elements are achieved via flag bits. These flags are efficiently encoded into the 9th bit of every byte after a vector arithmetic instruction.[1] The stored flag value depends upon the operation: unsigned addition stores the carry-out, whereas signed addition stores the overflow.

Optimizing for performance

The latest VENICE architecture can be configured up to 64 vector lanes (V64) on a Stratix III chip. With sufficient data parallelism in the application, a V64 could have a 64× speedup over a V1 configuration in the ideal case.

Since VENICE is built upon an FPGA, transferring data from and to main memory could be an expensive overhead. From our experience, a simple and effective technique to hide this memory latency is double-buffering. As stated in the previous section, user data has to be sent to the on-chip scratchpad memory in order to perform vectorized operation. Double-buffering allocates two buffers in the scratchpad memory for each user input array. First, input arrays are partitioned into several pieces. Then the first piece of input array is sent to scratchpad memory. After this is done, vector processing on the first piece of data will happen at the same time as transferring a second piece of input data. By the time the results

[1] FPGA memory blocks are normally 9 bits wide.

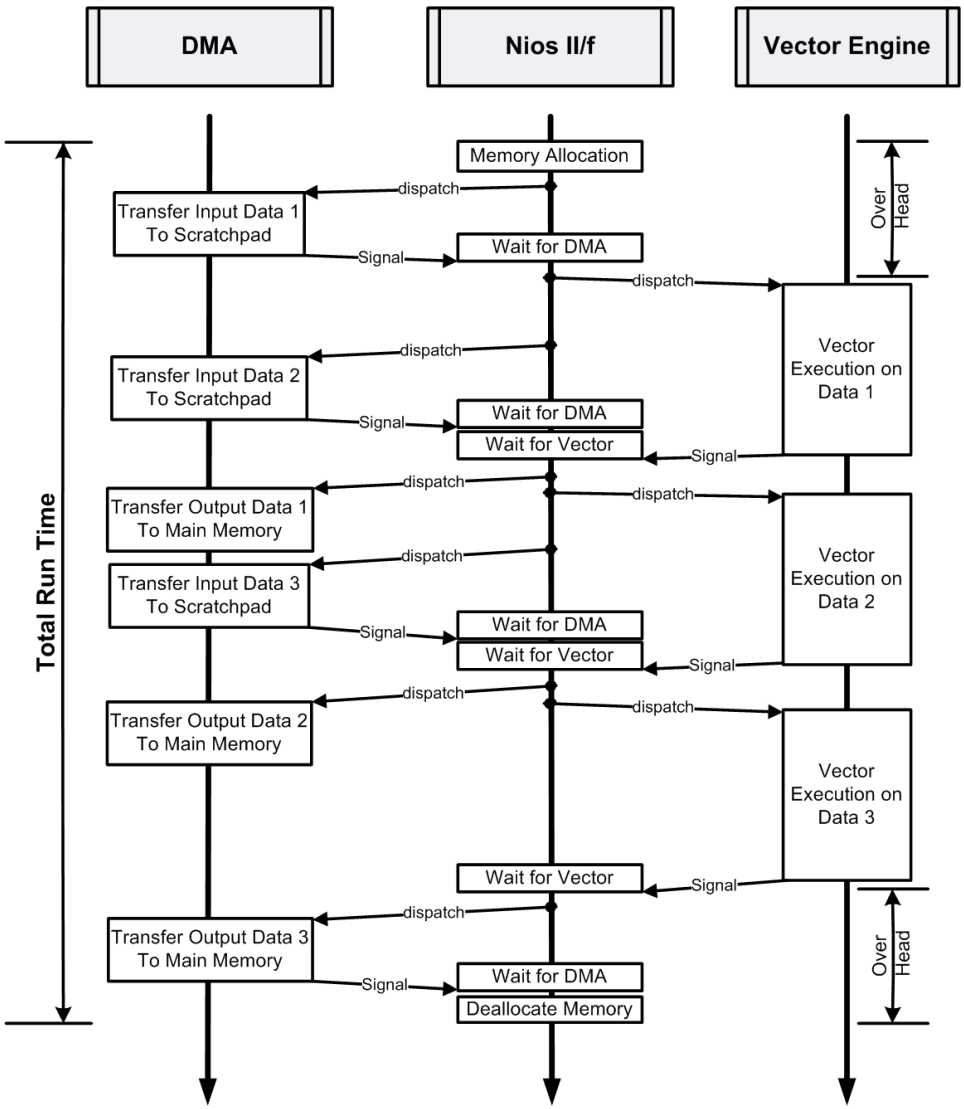

Figure 2.4: VENICE execution flow

are computed on the first piece of data, the second piece of input data is ready to be processed. By alternating the two buffers for computation and data transfer, most of the memory latency can be successfully hidden.

Figure 2.4 portrays how the three parts of VENICE work together concurrently

by applying double-buffering. The input data is partitioned into three pieces. With vector execution and data transfer happening concurrently, most of the overhead comes from transferring the first piece of input data and transferring back the last piece of result.

On the other hand, VENICE has a pipelined architecture as indicated in figure 2.1. Figure 2.5 is a simplified pipeline structure of VENICE. There are 7 pipeline stages in total. Since the scratchpad memory is multi-ported, memory read and write can be completed in the same cycle. Therefore, for the first vector instruction entering the pipeline, there will be a 6 cycle latency. Any instruction that has dependency on a previous one has to wait for 6 cycles as well to enter the pipeline. Within one instruction, all elements of a specified vector will stream through the pipeline with no stalls. Therefore, in order to keep high throughput of the vector engine, an as-long-as possible vector length is desired. However, this may expose the long data transfer overhead of the first and last pieces.

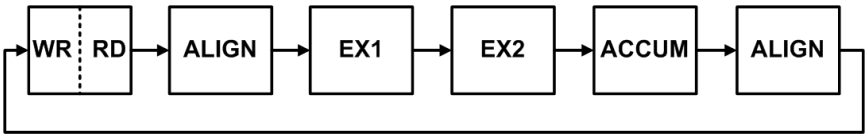

Figure 2.5: VENICE pipeline structure

2.3 Vectorizing Compilers

This section will introduce a few works similar to the Microsoft Accelerator system and compiler work in this thesis.

VIRAM was the pioneer of applying the vector processor in the embedded domain. The VIRAM vcc compiler was based on Cray's compiler and was developed simultaneously with the VIRAM hardware. It is claimed to be able to automatically

vectorize C/C++ directly without any special pragma or language annotations in most cases. However, it still needed pragmas to help with analysis on applications with complex data dependencies [8]. A deep analysis of an incomplete version of the vcc compiler was performed in [11]. The results showed that the compiler could not detect vectorizable loops consistently. All results were derived from the Dongarra Loops instead of real world benchmarks. Results from using the VIRAM simulator were reported in [12]. However, there wasn't enough evidence to prove the completeness and efficiency of the compiler.

Intel's ARBB [13] consists of a virtual machine and a C++ API that defines new parallel types such as collections to avoid direct data dependencies. These collections are treated like values and the Just-In-Time (JIT) compilation engine optimizes these and extract thread and data parallelism. It targets multi-core processors, GPU and Intel Many Integrated Core Architecture processors.

IBM's Lime [14] is a Java-based language capable of generating multiple low-level programming languages such as OpenCL for GPUs and Verilog for FPGAs. It relies on a user-built task graph and explicit data flow specification to identify the parallelizable kernel. It requires special value keywords to achieve data immutability and gets hints from user-declared private, local or global data types to manage complicated device memory systems.

For the OpenCL target, Lime generates a mix of Java ByteCode and OpenCL code for GPU kernel computation. For the FPGA target, it generates a mix of Java ByteCode and Verilog code. The advantage of Lime is less restriction to applications since it uses standard array types. However, there is a substantial overhead caused by Java to C and C back to Java conversion [15]. It is reported that it could achieve equivalent performance of hand-tuned native OpenCL code

on GPUs. However, it only measures the computation kernel runtime without all the data transfer and other overheads introduced by the framework. Results for the FPGA target have not been published yet.

MATLAB's parallel computing toolbox provides a flexibility of combining hardware independent and hardware dependent code together to achieve maximum performance, which gives the user the choice of whether to get involved in hardware manipulation. It is available for multi-core processors, GPUs, and computer clusters. The built-in libraries could be a time saver for developers. It also provides task parallelism among multiple applications and Single Program, Multiple Data (SPMD) mode. The disadvantage of the MATLAB PCT is that programs are not portable between GPU and multi-core or computer clusters.

There is a vast amount of other works for new languages and its associated compilation and run-time techniques as well such as StreamIt [16], hiCUDA [17], X10 [18], Sponge [19], etc. They will not be introduced one by one in detail since they do not offer both high-level abstraction and diverse platform compatibility.

2.4 Microsoft Accelerator System

The Accelerator system developed by Microsoft Research [20, 21] is a domain-specific language aimed at manipulating arrays. It presents to the user a set of high-level data parallel operations and object types that can be embedded in multiple primitive languages such as C/C++, C# and F#. The key assets of the Accelerator system is its ability to target entirely different devices with a single source language description. Accelerator provides a level of abstraction that completely hides hardware details from the programmer, and it auto-parallelizes the code for each target. It currently supports three different back-end targets: Graphics Pro-

cessing Units (GPUs) using DX9 and CUDA, Multi-core using Streaming SIMD Extensions (SSE), and FPGAs using VHDL [22]. The language and some compiler front-end basics will be described in this section.

2.4.1 Accelerator Language Fundamentals

```
1    #include "Accelerator.h"
2    #include "MulticoreTarget.h"
3
4    using namespace ParallelArrays;
5    using namespace MicrosoftTargets;
6
7    int main()
8    {
9         Target *tgtMC = CreateMultiCoreTarget();
10
11        const int length = 8;
12        int A[length] = {1,2,3,4,5,6,7,8};
13        int B[length] = {10,20,30,40,50,60,70,80};
14        int C[length] = {100,200,300,400,500,600,700,800};
15        int D[length];
16
17        // constructors copy user data to PA objects
18        IPA a = IPA( A, length );
19        IPA b = IPA( B, length );
20        IPA c = IPA( C, length );
21
22        // assignment statements build an expression tree
23        IPA d = a + b + c;
24
25        // the ToArray() call evaluates the expression tree
26        tgtMC->ToArray( d, D, length );
27
28        tgtMC->Delete();
29    }
```

Figure 2.6: Accelerator code to add 3 vectors

This section serves as a brief introduction on how to program using Accelerator. Figure 2.6 shows sample code for Accelerator adding three vectors together.

Accelerator declares and stores data arrays as Parallel Array (PA) objects. The PA objects are largely opaque to the programmer and restrict them from manipulating the array by index. Five primitive parallel array data types are supported by Accelerator which are boolean, integer, float, double, and Float4 (a set of 4 floats). Note there is no native support for byte or halfword (short) data types.

When an Accelerator PA object is instantiated, it is automatically initialized by making a copy of the original user array. Accelerator does a lazy functional evaluation of operations with PA objects. That is, expressions and assignments involving PA objects are not evaluated instantly, instead they are used to build up the expression graph. Loops and conditionals must be analyzable (i.e., not data-dependent on the parallel computation) so they can be fully unrolled, if necessary.

At the end of a series of operations, the Accelerator `ToArray()` function must be called. This results in the expression graph being optimized, translated into native code using a JIT compilation process, and evaluated.

At its core, Accelerator allows easy manipulation of arrays using a rich variety of element-wise operations, including both binary and unary arithmetic, comparisons, logical operations, and type-conversions. It also supports reductions (sum, product, max, min, or, and) that can be applied to the entire array or just rows of a 2D array, as well as linear algebra operations (inner product and outer product). Finally, it also provides a number of transforms which can shift, expand, stretch, transpose, or otherwise modify the shape and relative positions of the vector/matrix entries.

As shown in figure 2.6, with user data arrays declared and initialized (line 12-14), programming in Accelerator is straightforward as follows:

1. Create a target (Line 9)

2. Create parallel array objects for each input data array (Line 18-20)

3. Write expressions with parallel array objects and operations from the Accelerator library (Line 23)

4. Call `ToArray()` function to evaluate the result and copy it back to a regular C array (line 26)

In figure 2.6, the `CreateMCTarget()` function indicates that a subsequent `ToArray()` call will be evaluated on a Multi-Core platform. The `IPA` type represents an integer parallel array object. Similarly, the `FPA` type represents a floating point parallel array object. Here, A, B, and C are declared as input PA objects that are initialized by user arrays a, b, and c. D is the output PA which is converted to user array d in the end. The `ToArray()` triggers the compiler to start the evaluation of D. As stated previously, except for creating the proper target, the program is unaware of all hardware-related details. Targeting a different device can be done simply by changing the `CreateMCTarget()` function.

2.4.2 The Accelerator Front-end

The Accelerator front-end [23] is a built-in process common to all targets of Accelerator. Accelerator uses deferred evaluation. It builds a Directed Acyclic Graph (DAG) that consists of operations and associated data but it does not immediately perform any computations until the evaluation method – `ToArray()` is called. The DAG expression graph is composed of singly-linked expression node objects. All of the loops will be analyzed and unrolled during this process

which might produce a large number of duplicated sub-expressions in the expression graph. The expression graph will be handed over to the compiler front-end by a root operation node. After obtaining the full graph, the expression graph will be converted to an IR graph which is also a DAG. The IR graph is usually a near-duplicate of the expression graph but represented by a different object which serves as a working copy of the expression graph. The IR graph is easier to further analyze and optimize.

After the IR graph is validated, the system performs initial optimizations to the graph including constant folding and Common Subexpression Elimination (CSE). The CSE process will detect all of the common sub-expressions, including those introduced by loop unrolling when building the initial expression graph, and mark necessary breaks on relevant IR nodes.

One of the most important optimizations performed by the Accelerator front-end is analyzing all of the memory-transforming operations, combining them together and binding them to a leaf node (input array) in the IR graph in the form of index bound objects. Memory-transforming operations here refer to operations that rearrange the elements of a PA object or change the array dimensions by adding or deleting elements. An index bound is an object that stores a certain access pattern of a PA object in the form of a start point, an array length, a stride value, and a boundary condition for each dimension of the data array. The start point could be a negative value if there is an out-of-bounds access. The array length could also exceed the original array boundary as well. All these 'out-of-bounds' situations will be handled according to the boundary condition, which can be 'Wrap' - assign values from the opposite edge of the array to the new elements, 'Clamp' - assign values from the nearest element to the new elements, or 'DefaultValue' - assign

a specified value to the new elements. The goal of this optimization is to put the transform information at the point that data is retrieved from memory, rather than performing the transform later as a separate operation.

Figure 2.7 explains how the Accelerator front-end converts memory transform operations into index bound objects of leaf data nodes. Here, the shift operation performed on intermediate data X will be detected as a memory transform operation. The front-end optimizer will perform a series steps upon this observation. First, the `Shift` and `'+'` operations are swapped, which results in the `Shift` operating on the two child data nodes. These `Shift` operations then can be pushed to leaf data nodes as attached bounds objects. Therefore, the `Shift` operations can be completely removed from the IR graph. This process is called a lowering process of memory transform operations.

Multiple composable transforms can be combined together during the lowering process. In-composable transforms will be marked as forced breaks of the original IR graph. Memory transforms with the same boundary condition are considered to be composable. With all memory transforming operations pushed down to leaf nodes, no IR graph should contain any such operations.

In the end, the original IR graph are subdivided into a collection of small graphs. Break points can be caused by the CSE process as common sub-expression, or by memory transformation analysis process for in-composable memory transforms, or by user-specified Evaluation points using the Evaluate() method.

The Accelerator framework provides the target developer enough flexibility here to do any target-specific optimizations on the IR graphs from this point. After all the optimizations, the IR graph is ready for conversion to a Linear Intermediate Representation (LIR). The LIR is a linearized format that consists of operation

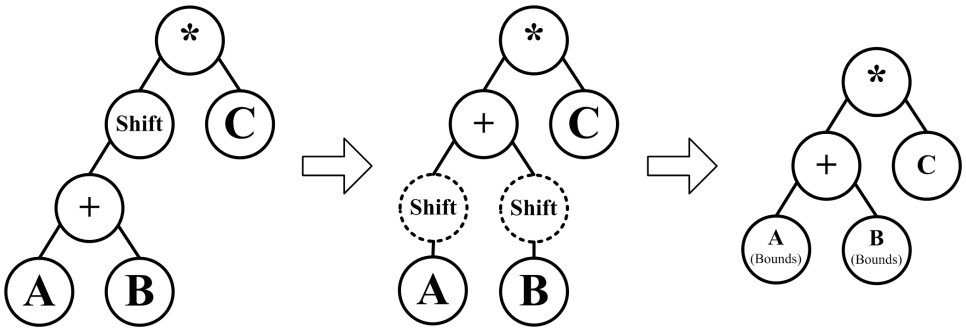

Figure 2.7: Lowering memory transform operations

and data nodes in stack-machine order. The primary purpose of the LIR is to represent an optimized IR graph in a format that can be readily converted to processor-specific code. The built-in common interface for IR to LIR conversion provides a convenient mechanism for generating processor-specific code. Figure 2.8 shows a conceptional flow from IR graph to final processor-specific code generation that is usually common to all targets.

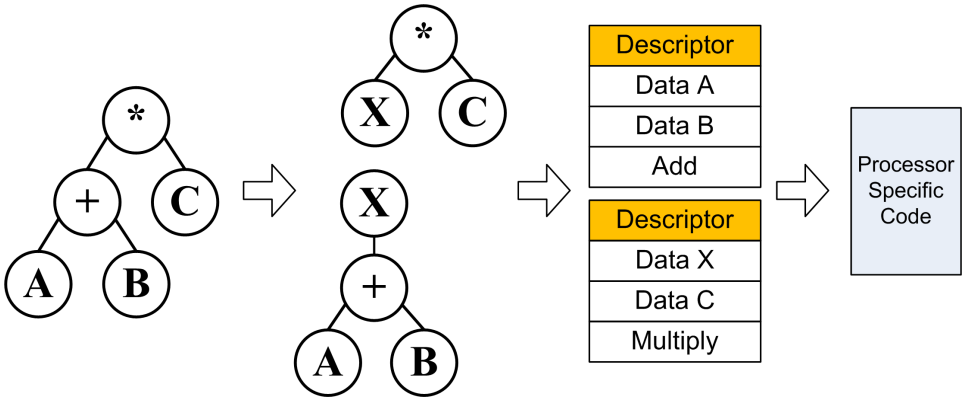

Figure 2.8: From IR to code generation

All targets have their own back-end which starts from the semi-optimized IR. Each back-end compiler can perform its own target-specific optimizations on the

IR graphs before serialization to LIR. The code generation process will also be different for different devices.

2.5 The Sethi-Ullman Algorithm and Appel's Generalization

As introduced in previous section, the Accelerator front-end produces a set of IR graphs representing the whole computation. Each target needs its own back-end compiler to perform target-specific optimizations, code generation, and execution. The pipeline structure of VENICE, introduced in section 2.2, indicates that short vector lengths cause idle cycles for ALUs and negatively affect performance. Therefore, a vector length beyond a certain value is usually favored for avoiding pipeline bubbles. In this thesis, the concept of 'virtual vector registers' is used to hold vector data in the scratchpad. Due to the limited on-chip scratchpad memory, a minimum number of virtual vector registers is desired.

The Sethi-Ullman algorithm [24] is a common technique used in compilers to obtain an optimal ordering of a computation tree, which results in the fewest number of registers required for storing intermediate results. The algorithm can be applied to any binary computation tree.

The algorithm works bottom-up on the tree. Each node is labeled with a 'Sethi-Ullman number', which is the number of registers required during its computation. This number is equal to 1 if the node is a leaf node, or the maximum of the numbers of its children if they are unequal, or to the number of either child plus one if two children have equal numbers. During the code generation stage, each node will first recursively generate code for the child with a bigger number, then recursively generate code for the other child then generate its own operation instruction.

Figure 2.9 shows an example of how the labeling process is done.

This is important because registers are usually a scarce resource in most architectures. In this work, the entire scratchpad memory is treated as a set of fixed-size virtual vector registers. The fewer vector registers needed, the more memory can be assigned to each one. However, the IR produced by the Accelerator front-end is not always a binary tree. Therefore, a modified version of Sethi-Ullman algorithm is adopted.

Appel and Supowit generalized the Sethi-Ullman algorithm [25] to accommodate operations with three or more operands. In the case that a node has three or more children, it will be labeled the maximum of all its children's numbers if they differ from each other. If there are two or more children with the same number that are equals to the maximum number of all children, each additional child node that has a number equal to the maximum number will add one to the parent node number. For example, a parent node with three children numbered 2, 2, and 2 respectively will be labeled 4.

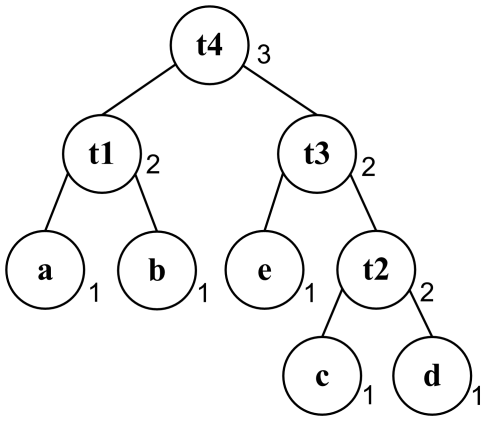

Figure 2.9: Sethi-Ullman labeling algorithm

Chapter 3

Compiler Implementation

This chapter describes the end-to-end flow of the VENICE compiler, which serves as a source-to-source translator. It compiles Accelerator code into VENICE instructions and stores them in a C file. The Accelerator built-in front-end is used for building the expression graph and initial conversion from expression graph to Intermediate Representation (IR). It also applies basic common optimizations to the IR. The VENICE back-end implementation, which is the main topic of this thesis, starts from target-specific optimizations focusing mainly on the efficient usage of the scratchpad memory. Then it converts the IR to a stack-machine-style LIR. Code generation from LIR follows the steps for programming VENICE introduced in chapter 2. This chapter also talks about some limitations of the compiler.

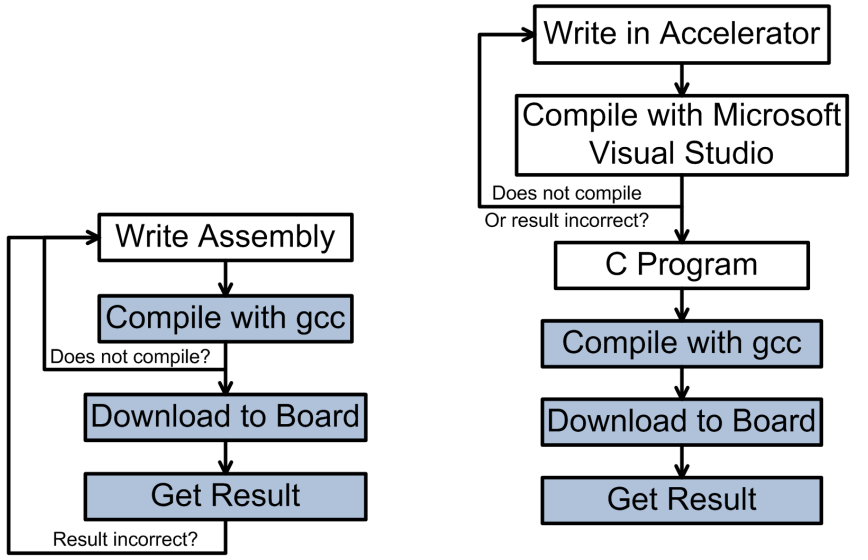

Figure 3.1: Development flows with native VENICE and Accelerator

3.1 Compiler Flow

3.1.1 Compiler Overview

Figure 3.1 shows the user flow for (a) programming VENICE directly, and (b) programming in Accelerator. Unlike Multi-core, CUDA and DX9 targets, which use a JIT compilation process, the VENICE back-end for Accelerator adopts an off-line approach similar to the FPGA target. It serves as a source-to-source compiler by writing out another C program that uses the VENICE APIs.

When writing native VENICE code and debugging, the user has to download the compiled assembly to an FPGA board to check the correctness of results. In contrast, the existing JIT mode targets of Accelerator allow fast debugging within the Visual Studio debugger. Once the Accelerator code is complete, it only requires a one-line change of the code to switch to the VENICE target and generate code

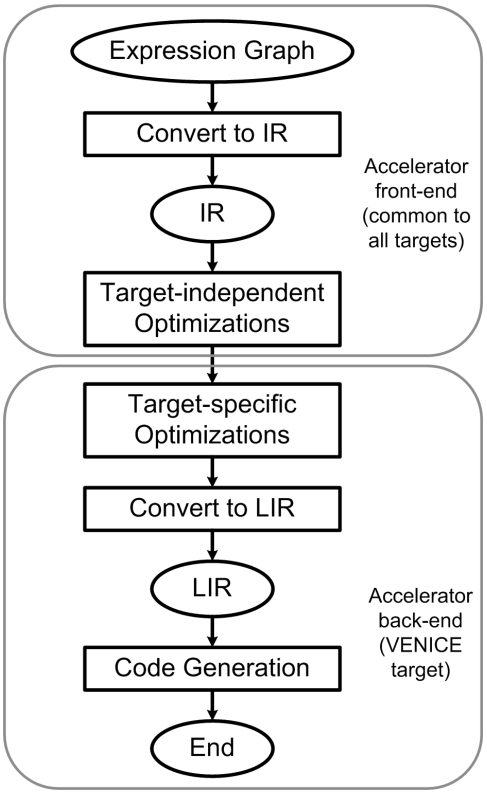

Figure 3.2: Accelerator compiler flow

for VENICE. The output is written into a C file, which can be compiled with gcc and downloaded to an FPGA board for execution.

Figure 3.2 illustrates the high-level flow of Accelerator compilers. The Accelerator front-end (upper box in figure 3.2) is common to all targets. It builds an expression graph from user-written Parallel Array (PA) expressions, and produces an Intermediate Representation (IR) that serves as a working copy of the expression graph. Optimizations can be performed on the IR. After the IR graph is produced, each back-end compiler (lower box in figure 3.2) performs target-specific optimizations on the IR before code generation. The remainder of this

section will walk through the steps in the back-end compiler needed to generate native VENICE code that efficiently uses the available scratchpad memory. The back-end first sets an evaluation order of each IR graph. Then, it counts the number of vector registers needed across all the IR graphs. Next, the IR graphs are converted to a LIR format before final code generation. Steps in code generation include data allocation and initialization, data transfers, and mapping Accelerator APIs to VENICE instructions.

3.1.2 Accelerator Front-end

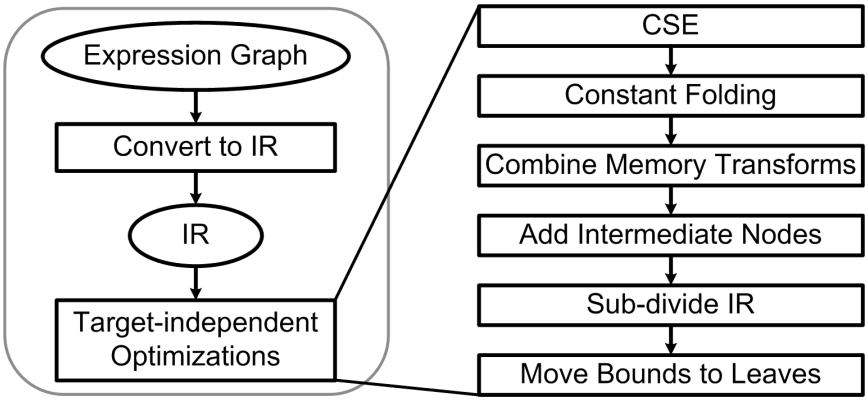

Figure 3.3: Accelerator front-end flow

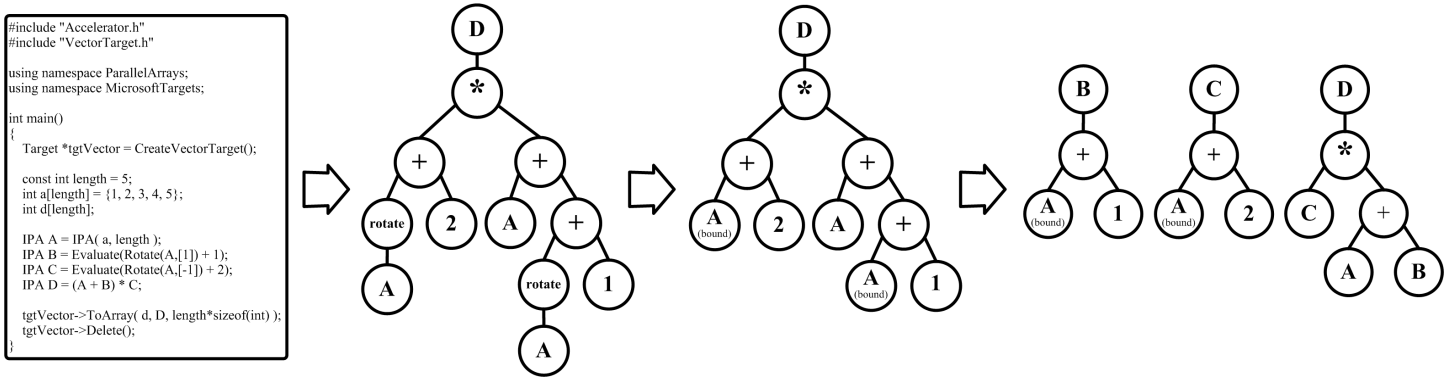

Figure 3.4: Example on conversion from user-written code to IR graphs

The Accelerator front-end completes several tasks shown in figure 3.3. It converts the user-written Accelerator code into an expression graph. The expression graph is copied to an IR graph for initial validations and optimizations. It performs constant folding, Common Subexpression Elimination (CSE), memory transform combination, and memory transform lowering. In the end, it sub-divides the IR graph into smaller IRs. Details on these processes of the front-end were introduced in chapter 2, and will not be repeated here. Figure 3.4 shows an example of how the Accelerator front-end produces the resulting IR graphs. In the example, two rotations of parallel array A are added by scalar values. The use of Evaluate() on these two additions forms two break points in the IR. The final resulting parallel array D is an expression of user input array A, and intermediate results B and C. The rotate operations in the original IR are combined into leaf node A. The IR is then sub-divided into three smaller IRs due to the user-specified early evaluation points. This example will continue to be used throughout this chapter to demonstrate the steps in the back-end compiler.

3.1.3 Target-specific Optimizations

As the first step in the Accelerator back-end compiler, this sub-section describes some target-specific optimizations on the IR graphs in preparation for conversion to the LIR. It includes three steps shown in figure 3.5. The main goal of target-specific optimizations is the efficient use of scratchpad memory. The back-end follows the intuitive approach of treating the scratchpad space as a pseudo-vector-registerfile. Similar approaches can be found in [26, 27] for scalar architectures. It first uses an algorithm based on Appel's generalization of the Sethi-Ullman algorithm introduced in section 2.5 to determine the evaluation order of each IR graph.

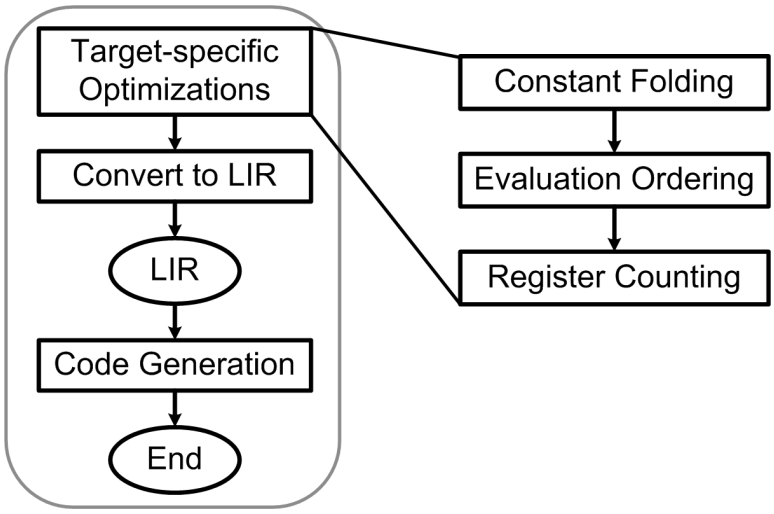

Figure 3.5: Target-specific optimizations

Then it uses a reference-counting algorithm that walks through all the IR graphs in the established evaluation order and calculates the exact number of vector registers needed across all IRs. Since the entire scratchpad memory can be partitioned into arbitrary number of vector registers with arbitrary size, this number is crucial for optimized scratchpad memory partitioning.

Constant Folding

It is found beneficial to perform another round of constant folding in the back-end in addition to the existing pass done in the front-end. This is necessary because the Accelerator front-end is provided in pre-compiled form by Microsoft, so this part of the flow could not be modified due to limited access to the source code.

Evaluation Order

To simplify the management of scratchpad memory and code generation, the scratchpad is divided into the fewest equal-size 'virtual-vector-registers' that are

needed. Several techniques are employed to limit the number of vector registers used, so as to maximize their size.

Chapter 2 introduced Appel's generalization of Sethi-Ullman algorithm for determining the evaluation order of computation trees. The algorithm can handle operations that have three or more operands, where the Sethi-Ullman algorithm only handles binary computation trees. Since Accelerator contains functions that have more than two operands, Appel's algorithm is adopted to determine the evaluation order of sub-expressions in each IR graph.

However, modifications have to be made to accommodate the scratchpad architecture. The Sethi-Ullman labels are used to represent the number of vector registers in scratchpad memory. For common processor architectures, variables and immediate values usually require a register to temporarily store the value. However, in the VENICE architecture, all input data arrays are loaded to scratchpad memory. Immediate values (scalar values) come directly from the instruction queue, and do not take any space in the scratchpad. Therefore, all the scalar nodes are labeled 0.

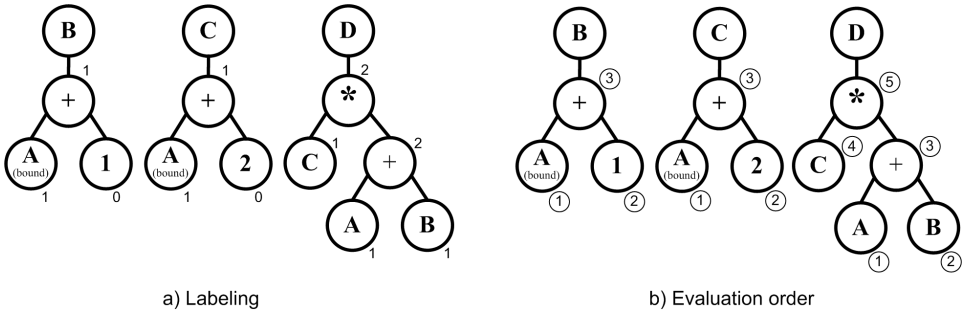

Figure 3.6: Establish evaluation order

Figure 3.6 shows an example on how evaluation order is set for the IR graphs produced at the end of figure 3.4. In part a) of figure 3.6, all nodes are labeled

with the Sethi-Ullman number. The circled numbers in part b) indicate the final evaluation order of each IR graph.

Register Counting

In order to partition the scratchpad, the compiler must know the exact number of vector registers needed by the computation. The labeling is done individually on each IR graph. The compiler must combine all the IRs together to obtain a total number of registers. Furthermore, the labeling does not take vector register re-use into account. To reduce unnecessary data transfers, it is desirable to re-use vector registers as much as possible. A reference-counting method is adopted to achieve this goal.

Initially, each leaf data node is associated with a vector register. Each vector resister may be re-used several times during the whole computation. To re-use a vector register, the back-end keeps a list of leaf data nodes, plus the number of remaining references to each of them. Since each IR graph produces an intermediate result that will become the leaf node of other IR graphs later in the computation, not only are these nodes kept in the reference-counting list, but the back-end also keeps a list that records when an intermediate data node becomes available and when a data node dies after its reference count becomes 0.

The back-end calculates the exact total number of vector registers needed using these two lists. This process follows the evaluation order established in the previous pass, and mimics the code generation process.

Figure 3.7 demonstrates this register-counting process. The computation consists three IR graphs, which are results of figure 3.6. Node A is a user input array. Node B and C are intermediate results. Node D is the final output. When an IR

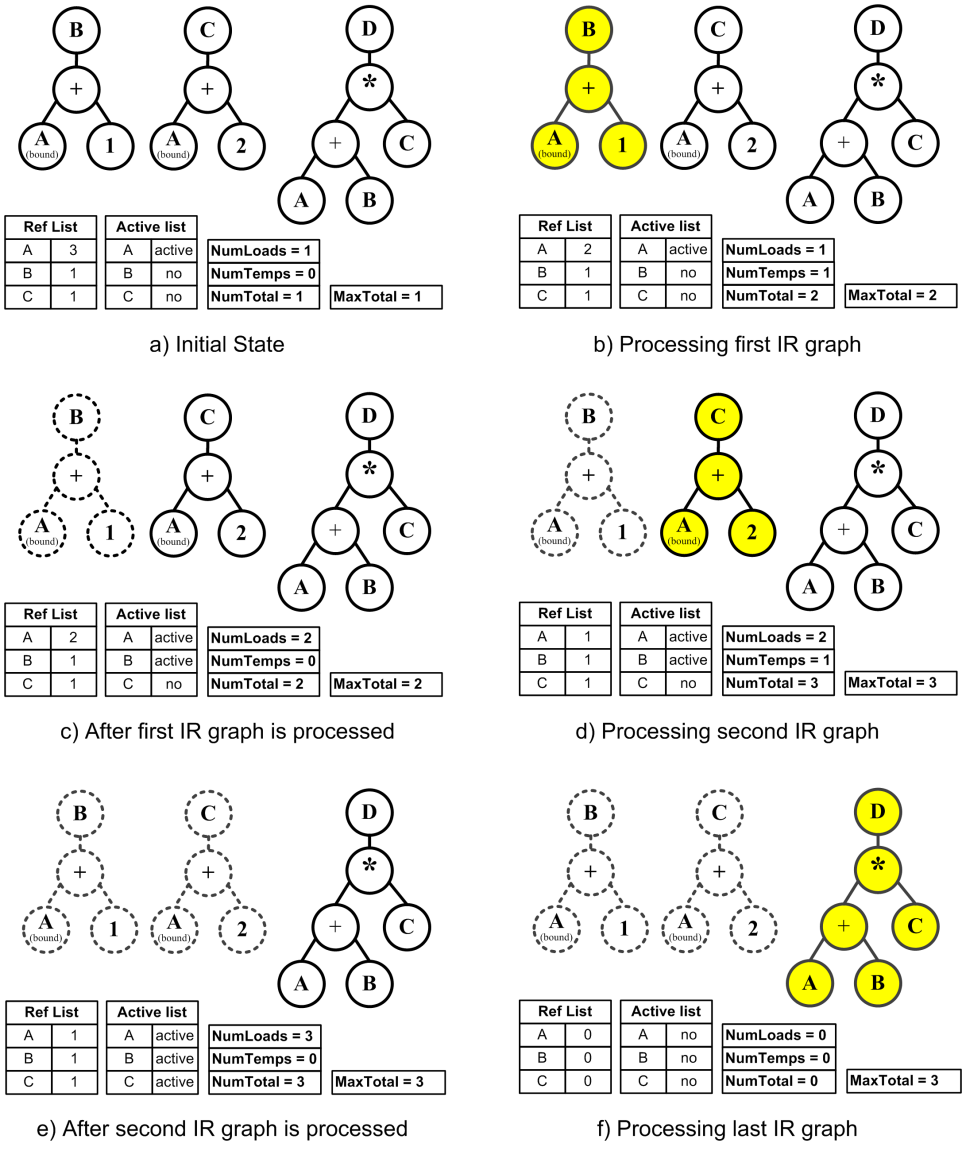

Figure 3.7: Reference-counting process

is highlighted, it means the IR is being processed, and when an IR is in dashed outline, it has finished being processed. All of the IR graphs are already organized in a desired evaluation order. The value 'active' in the active list indicates a leaf

data node is reserving a vector register. The value 'no' means the opposite. Two variables are used to keep track of the number of registers used by each IR graph. NumLoads is the number of active leaf nodes that are reserving a vector register. NumTemps is the number of temporary registers requested by operation IR nodes that cannot re-use any of its children's registers.

The number of registers needed for leaf nodes (numLoads) is always the sum of total active data nodes in the active list. Part a) shows initially only user input array A is active. Therefore, numLoads is 1. Whenever an IR graph is completed, the temporary result will activate the corresponding intermediate leaf data node in the active list and update the numLoads. Part c) shows B becoming active after the first IR is completed, and numLoads becomes 2. Each time an input data node is consumed, its reference count will be reduced by 1. Whenever a reference count becomes 0, that vector register is no longer needed to hold an input array or intermediate result and is available for re-use immediately. When encountering an operation node, the compiler will check if it can re-use any of its children's registers. If one of the children is an operation node, that register is always considered reusable. If all children are leaf data nodes, the reference count of each leaf node is checked to see if it has become 0. If no re-usable register is found, numTemps will be increased. In the example shown in figure 3.7, when the first `'+'` node in the first IR graph is being processed, it finds that its left child `A` has a non-zero reference count, and its right child is a scalar node. Therefore, it adds 1 to numTemps. The numTemps variable will be reset to 0 after an IR graph is processed.

The total number of registers needed (numTotal) is the sum of registers used to hold leaf data nodes (numLoads) plus temporary registers needed by operation nodes that cannot re-use its children's registers (numTemps). The maxTotal vari-

able keeps track of the maximum value of numTotal. At the end of processing all of the IR graphs, the maxTotal variable indicates the number of virtual vector registers needed to perform the computation.

3.1.4 Convert to LIR

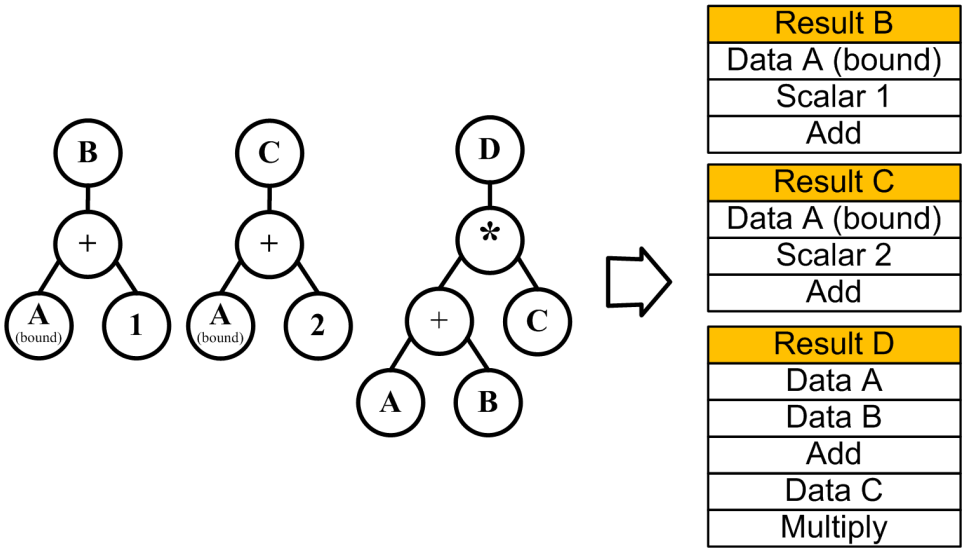

Figure 3.8: Convert IR to LIR

The Accelerator framework recommends use of the LIR for the convenience of code generation. The LIR consists of node objects similar to IR nodes, but in a stack-machine style. Figure 3.8 shows how the optimized IR graphs are converted to LIR. The compiler creates LIR nodes in the established evaluation order. The bound objects associated with each input data, the total number of registers computed from previous section, and all other necessary information that is useful for code generation, are passed over to the LIR during the conversion. When the LIR is complete, it is ready for code generation.

3.1.5 Code Generation

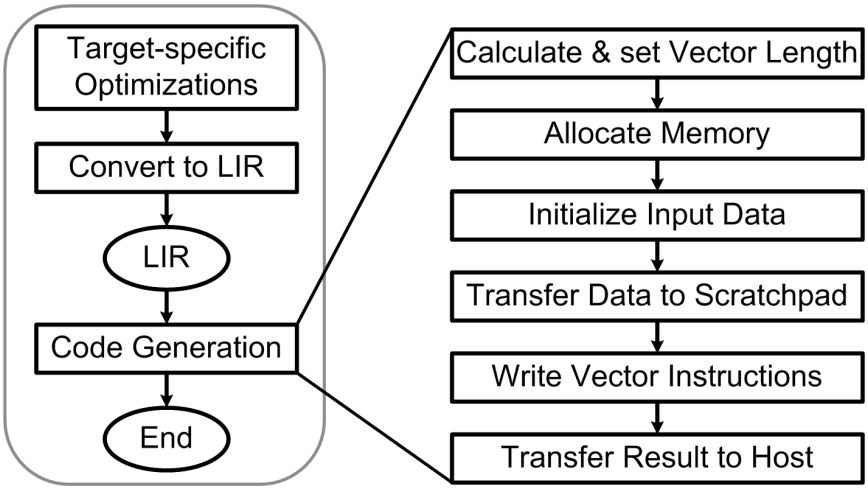

Figure 3.9: Code generation flow

The code generation follows the steps required for VENICE assembly programming. This flow is depicted in figure 3.9. This sub-section will go through these steps one by one.

Memory Allocation and Data Initialization

The compiler calculates the size of each vector register and allocates memory in the scratchpad according to the number of vector registers computed from section 3.1.3. As discussed in chapter 2, one convenience feature of Accelerator is the efficient handling of out-of-bounds array indices that comes from memory-transforming operations such as `Shift()` and `Rotate()`. In the front-end, Accelerator combines memory transforms and propagates the array bounds to leaf data nodes, so the maximum extents are known. The back-end takes this information into account, and allocates extra memory in the scratchpad for these out-of-bound cases. All remaining scratchpad memory is allocated equally among the

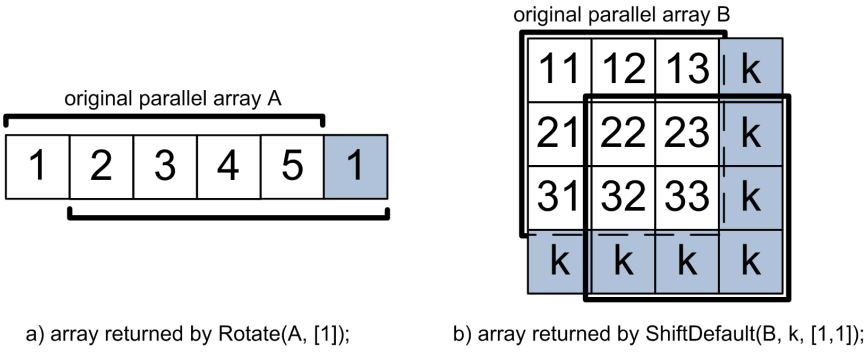

Figure 3.10: Memory transform examples

number of vector registers.

The initialization stage copies user data to the output C file, and prepares for memory transforms by padding the input array with proper values for out-of-bounds accesses.

Figure 3.10 demonstrates how input data padding is done. Part a) shows a rotation performed on a 1D array. The original array is white, with the out-of-bounds elements shaded. In this case, the last element `1` appears padded after the last element. The new 1×5 array formed by the `Rotate()` is indicated by the lower bold black bar. Therefore, a vector register size of 6 words will be allocated for `A`. Part b) shows a shift by 1 row and 1 column on a 2D array. Values past the bounds are initialized with the specified default value of k. The new 3×3 array is highlighted by a bold black box in the foreground. When allocating memory in scratchpad, a total of 4×4 words will be assigned to `B`.

Each input data array might be referenced multiple times with different bound conditions. For example, if the 1D array A in figure 3.10 is rotated to the left and to the right by two different data nodes, not only will `1` be padded after the last element, but the element `5` is also padded before the first element. This is shown in

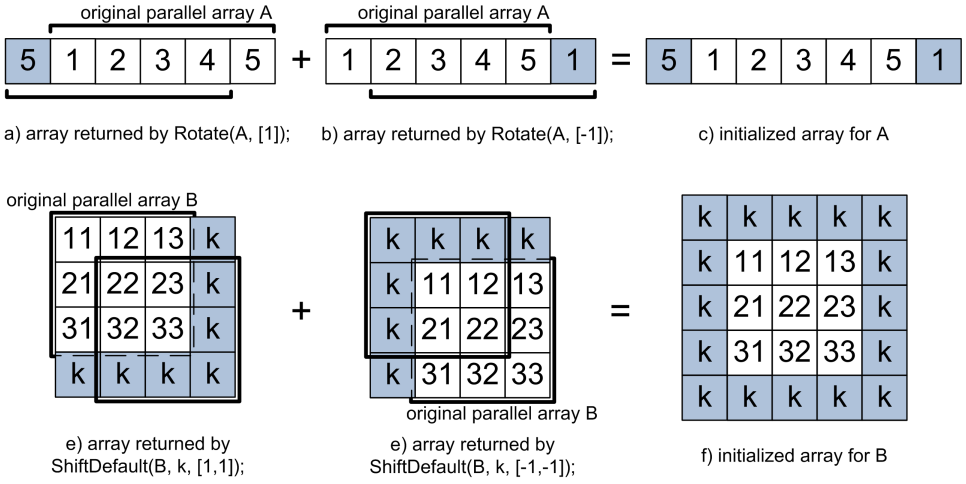

Figure 3.11: Data initialization for multiple memory transforms combined together

the upper half of figure 3.11. Similarly, if the 2D array B in figure 3.10 is shifted up and left by 1, and later it is shifted down and right by 1, the compiler must prepare for both cases during data initialization. Therefore, the initialization of this 2D array will look like the lower half of figure 3.11.

However, if two shift operations on the 2D array in figure 3.11 need to be padded with different values, the in-composable memory-bound conditions will result in creation of a new 2D array.

All scratchpad memory is freed at the end of the entire program. This is mainly because VENICE does not support partial de-allocation of the scratchpad memory.

Data Transfer

DMA transfer instructions are generated after memory allocation and data initialization. Input data arrays are transferred to the scratchpad memory all at once. A wait-for-DMA instruction is generated before vector instructions. All the results

will be transferred back to the main memory after vector instructions are completed.

Generate Vector Instructions

The ability to manipulate arrays is intrinsic to both Accelerator and VENICE. Therefore, in many cases, a direct mapping of an Accelerator operation to a VENICE instruction for basic element-wise operations is possible. For example, the simple `'+'` operator in Accelerator is mapped to `VADD` instruction in VENICE. In a few cases that Accelerator operators are not directly supported by VENICE, such as divide, modulo, power, and matrix multiply, pre-written library code is required.

Memory transforms were discussed in the previous sub-section, and they were handled by properly initializing the input data array. Extra elements might be padded outside of the normal array bounds. In the code generation stage, the compiler must extract desired data from the original array based on the bounds object associated with the individual data node.

Take the arrays in figure 3.10, for example. With all data properly initialized, extracting partial data from the 1D array is simply done by adding an offset to the starting address in the scratchpad memory. The Accelerator expression:

```
D = Rotate(A, [1]) + 2;
```

will be translated into the following VENICE assembly:

```
vector_set_vl(5);
vector(SVW, VADD, VD, 2, VA+1);
```

For 2D arrays, the row stride amounts can be adjusted to step over any padding elements at both ends. The Accelerator expression:

```
D = Shift(B, {1,1}) + 2;
```

will be translated into the following VENICE assembly:

```
vector_set_vl(3);
vector_setup_2D(3, 0, 0, 1);
vector_2D(SVW, VADD, VD, 2, VB+1);
```

Accelerator provides APIs to do element-wise comparisons between arrays such as Max() and Min(). There is also a Select() API that does a conditional move operation. Chapter 2 introduced that VENICE supports conditional moves by using flag bits. This feature is used to perform Accelerator comparison operations. For example, the Accelerator expression:

```
D = Select(A-B, A, B);
```

which moves an element in A to D if an element in A-B is greater than or equal to 0 and moves the element in B to D otherwise, will be converted into VENICE assembly as follows:

```
vector(VVW, VSUB, VTemp, VA, VB);
vector(VVW, VCMV_GTEZ, VD, VA, VTemp);
vector(VVW, VCMV_LTZ, VD, VB, VTemp);
```

The subtract instruction sets the flag bits associated with each element in VTemp according to whether VTemp is positive or negative. The first conditional move instruction VCMV_GTEZ moves elements in VA to VD if flag bits of VTemp are not set. The last line of code moves elements in VB to VD if flag bits of vector VTemp are set.

Figure 3.12 shows an example of the final stage of compiler that generates a human-readable C program with VENICE APIs.

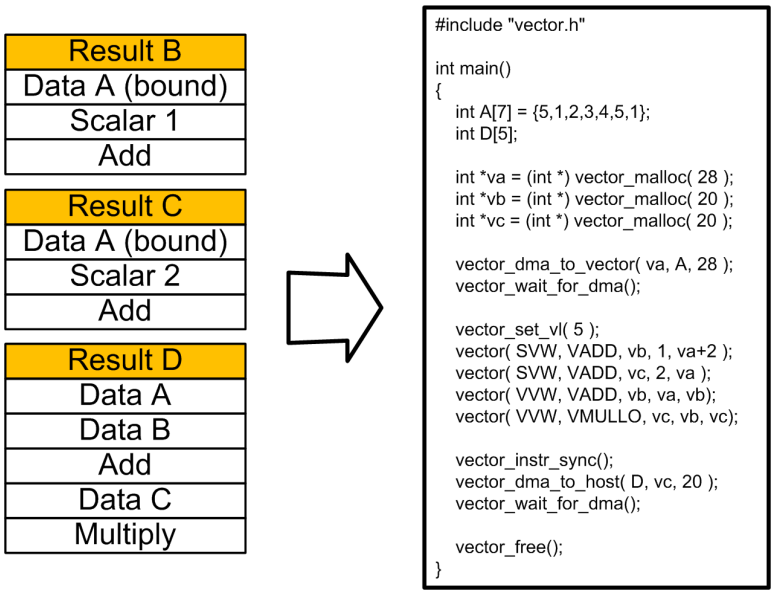

Figure 3.12: Example of generating code from LIR

3.2 Implementation Limitations

The compiler does not support all of the Accelerator APIs. One of the reasons is that VENICE cannot perform floating-point arithmetic. Therefore, float and double data types are not supported by the compiler. All APIs that require these data types, such as Log() and Sin(), are hence omitted. Some rarely used operations, such as `OuterProduct()`, `Replicate()` and `Pad()`, are also not supported due to the low priority of the task during compiler development.

On the other hand, there are several VENICE instructions that cannot be expressed by Accelerator APIs, such as the multiply-high instruction which returns the upper N bits of an $N \times N$ multiply. However, the generated C program is human-readable, and can be easily modified to utilize such features of VENICE that Accelerator cannot exploit.

As described in chapter 2, the Accelerator front-end has a built-in CSE algorithm to eliminate duplicated evaluation of sub-expressions that are considered to be expensive. However, VENICE is unlike other targets where sometimes computing a sub-expression locally may be faster than storing intermediate data into memory and reading from memory again. In VENICE, reading from and writing to the scratchpad memory requires no extra overhead. Any duplicated computation is an overhead. Therefore, the built-in CSE algorithm is not suitable for the VENICE target. Attempts have been made to replace the built-in CSE algorithm with a more aggressive version, but it is not yet integrated into the Accelerator front-end. Instead, CSE must be performed manually by the user using the `Evaluate()` operation.

The ToArray() call triggers generation of a C file with VENICE APIs. Applications that require multiple ToArray() calls require the manual effort of the user to merge multiple C files and the associated memory management.

The scratchpad memory is treated as a medium that can be sub-divided into as many pieces as possible. Demand for a large number of vector registers will only result in a small size for each one. Demand for an exceptionally large number of registers might result in a register size that degrades performance. For extreme cases that require thousands of vector registers resulting in a register size that is smaller than 1 word, the compiler reports an error.

3.3 Summary

The back-end compiler starts with an intermediate representation produced by the Accelerator front-end, and follows a flow recommended by the Accelerator framework. The back-end compiler performs a series of device-specific optimizations.

The key optimizations are the optimized evaluation order specific to VENICE resulting in a minimal number of vector registers required by the whole computation, and the precise calculation of the exact number of vector registers needed. This number is used to partition the scratchpad memory. These optimizations provide a maximum vector length for vector execution. The resulting C program with in-line VENICE API calls is generated from a linearized intermediate representation.

The entire compiler implementation required approximately 1 year of work and approximately 6000 lines of code.

This chapter has listed some of the limitations of the compiler. More will be discussed in detail in chapter 6, where possible future work is presented.

Chapter 4

Performance Enhancement

This chapter describes two improvements made to enhance the performance of the VENICE back-end compiler. First, the compiler handles the case when data arrays cannot fit into the scratchpad memory by sub-dividing data arrays and performing data transfers in a double-buffered fashion. Based on the input sizes and number of operations in the application, the compiler references a look-up table that stores the near-optimal vector lengths to partition the data arrays. These near-optimal vector lengths are obtained by profiling on synthetic benchmarks.

Second, the back-end compiler is able to reduce vector-instruction count by combining multiple Accelerator operations into one VENICE instruction. Experimental results are provided to demonstrate the impacts of these optimizations.

4.1 Dynamic Partitioning

As mentioned in chapter 3, when the VENICE back-end performs target-specific optimizations, it always precisely calculates a minimal number of vector registers needed by the computation, so as to maximize register sizes. This is because the

vector-register size directly becomes the vector length for vector execution, which affects the pipeline utilization and end-to-end runtime performance. In the case where the user array size exceeds this maximized register size, the back-end needs to sub-divide data array into smaller segments. In order to hide data-transfer overhead, data movement can be done in a double-buffered fashion by pre-fetching the next-needed pieces. This fully exploits the capability that DMA transactions and vector computation can take place at the same time. Overlapping the two can almost completely hides the overhead of the data transfers. However, even when the entire computation can be performed on VENICE without being segmented, applying double-buffering may still be beneficial.

Unlike a GPU or multi-core architecture, memory latency could dominate VENICE's performance. The Multi-core target relies upon caching to hide memory latency. GPUs rely upon switching between thousands of threads to tolerate memory latency. However, VENICE uses a DMA engine to transfer data between scratchpad memory and main memory. Although the DMA engine owns a dedicated 4-byte read/write port to the entire width of the scratchpad memory, the main memory doesn't support multiple memory banks. This constrains the memory bandwidth to be at most 32 bytes, which is the maximum bandwidth of the main memory at the board-level in the DE3 development system used by this work. For smaller VENICE configurations using less than 8 lanes (4-byte wide each), the bandwidth will be further limited by connections between DMA and scratchpad memory. For example, a VENICE V4 only has a memory bandwidth of 16 bytes. Furthermore, in addition to the bulk data transfer time, there is the additional latency from the DMA instruction being dispatched until the data actually arrives the destination. Taken together, these factors strongly impact VENICE's performance.

Trade-offs have to be considered when performing double-buffering. Treating the entire input data as a whole will result in no overlap for data transfer and vector operations. With all the memory transfer overhead being exposed, performance will be significantly degraded, especially for small benchmarks. On the other hand, dividing data array into too small segments will result in a short vector length that cannot maintain high throughput in the VENICE pipeline. The initiation of memory transfer will also introduce additional delays.

Taking another look at the VENICE execution example in figure 2.4, vector execution time is shown taking longer than the memory transfers. Most of the overhead comes from transferring the first segment of the input data to scratchpad and transferring the last segment of the result back to main memory. A smaller vector length might actually reduce these overheads and improve performance.

In contrast, applications can also be memory-bound as demonstrated in figure 4.1. Memory transfers become dominant, and vector executions are hidden by the memory latency. The total runtime is basically the time for transferring all the inputs to scratchpad and transferring the results back. Due to the additional latency introduced by each data transfer instruction, the less data transfer instructions initiated, the less time it takes to complete transferring the entire input and output. In these cases, a larger vector length is preferred to reduce the total overhead of initiating all data transfers.

Based on these observations, an assumption can be made that for each configuration of VENICE, there should always be an optimal vector length for each application that will achieve the best balance of the data transfer and vector execution and yields a minimal total runtime.

As mentioned in chapter 2, VENICE has 7 pipeline stages in total, which causes

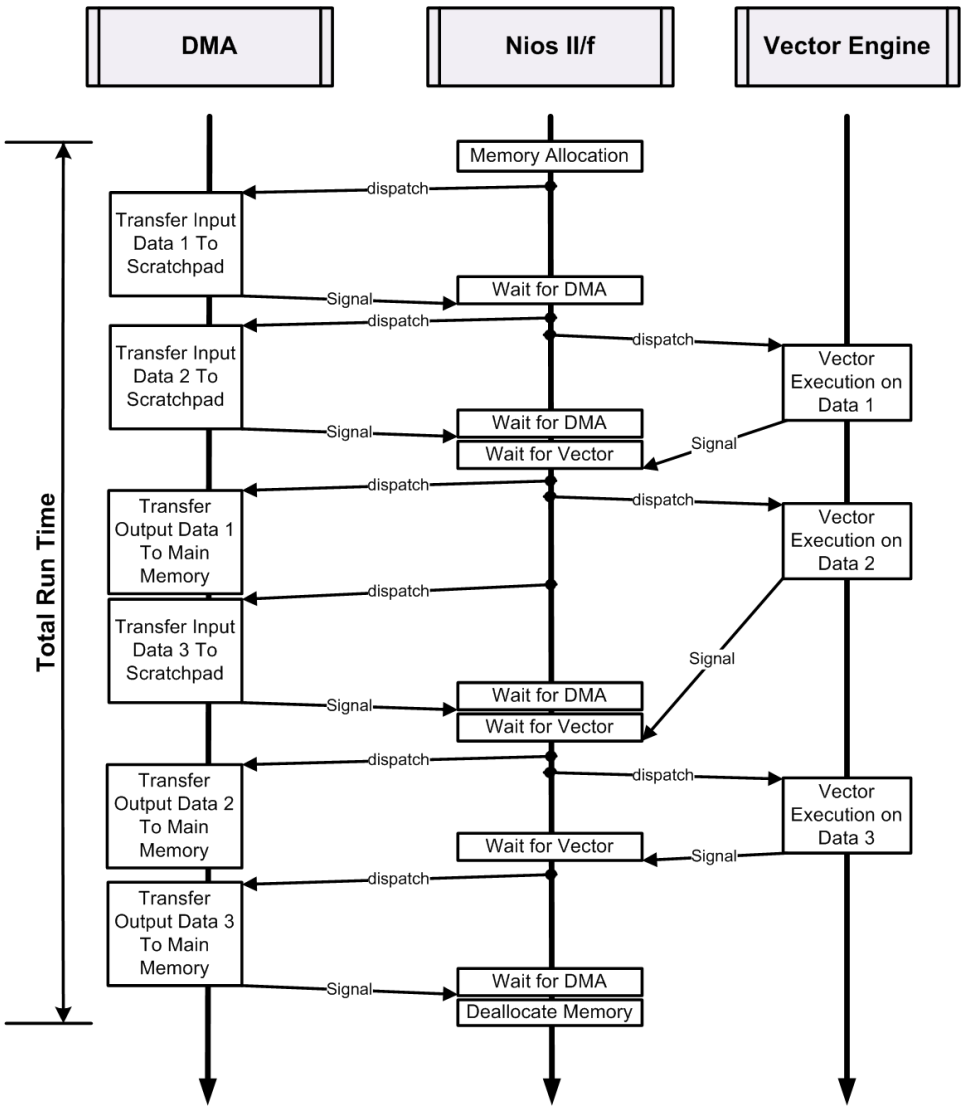

Figure 4.1: VENICE execution flow for memory-bound applications

a 6 cycle delay between any consecutive instructions with a data dependency. Since Accelerator does not require the programmer to be aware of any hardware details, which may affect parallelism, data dependencies are prevalent in many bench-

marks. [1] The VENICE back-end does not perform deep analysis on such data dependencies as this would radically change the structure of a given computation tree. Instead, this work focuses on the workload balancing optimization by carefully choosing a vector length that is large enough to limit the impact of pipeline bubbles caused by data dependencies and also small enough to avoid long latencies caused by the first and last piece of data transfers.

For each application, data-transfer time and vector-execution time can be expressed by complex equations based on the input data sizes, the number of operations performed on each data element, the number of vector lanes available, memory bandwidth, memory latency, function call overhead, and the to-be-determined vector length. Each application can be further categorized into three different situations based on the data-transfer and vector-execution time – compute-bound, memory-bound, and balanced. Each situation would use a different equation to calculate a vector length that achieves the shortest total runtime. The calculation might involve deciding clear boundaries of variables, calculating derivatives of equations, and computing a vector length that minimizes the equation. However, the sensitivity of this approach to the accuracy of the equations and detailed modeling of the the VENICE architecture and application characteristics could easily make this method imprecise or incorrect.

Therefore, instead of designing a complicated algorithm to compute the optimal vector length for each program, a simple look-up table is created containing selected vector lengths for each pair of input size and operations per data element (instruction count). In order to achieve this goal, a synthetic benchmark is used

[1] Accelerator assumes there is sufficient data -level parallelism in each operation. Hence, it does not focus on parallelism between operations.

with controllable input sizes and number of vector instructions. By sweeping on each parameter, a performance table with runtime for each given input size and number of instructions is derived. The vector length that results in the least runtime is stored in the final look-up table.

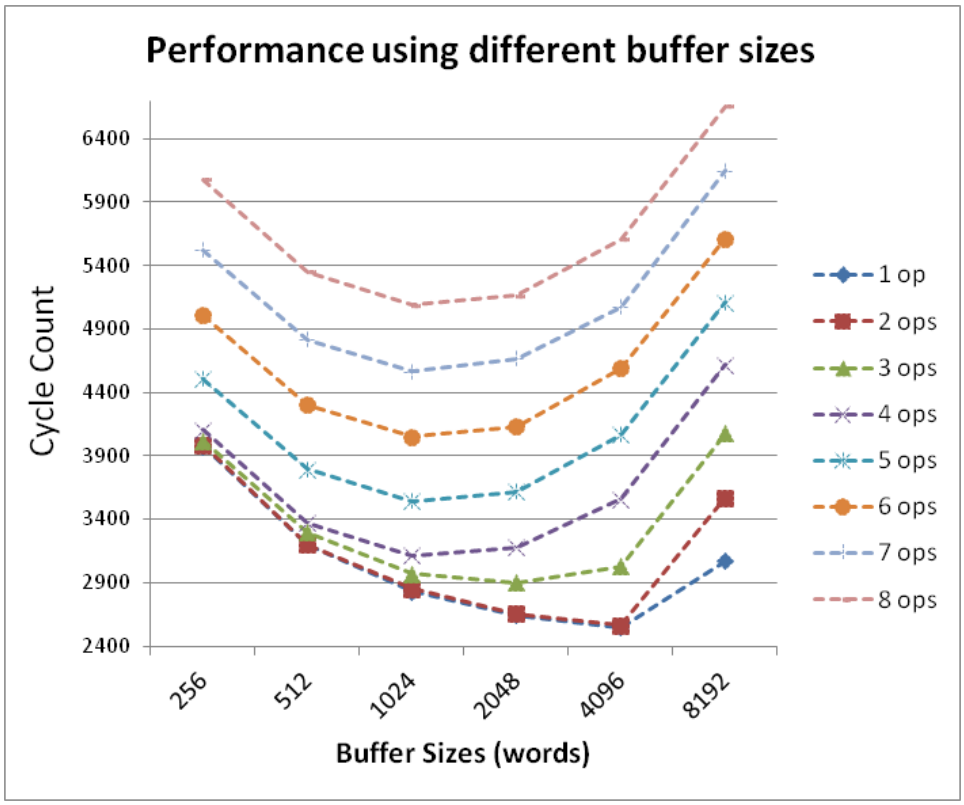

Figure 4.2: Impact of vector length selection for input size of 8192 (words) with different instruction counts

Figure 4.2 shows the number of clock cycles required when performing a different number of operations on each input array element with a fixed size (8192 words) of input data. The Y axis is the total cycle count; a smaller number indicates better performance. The X axis is the vector length, aka the vector register

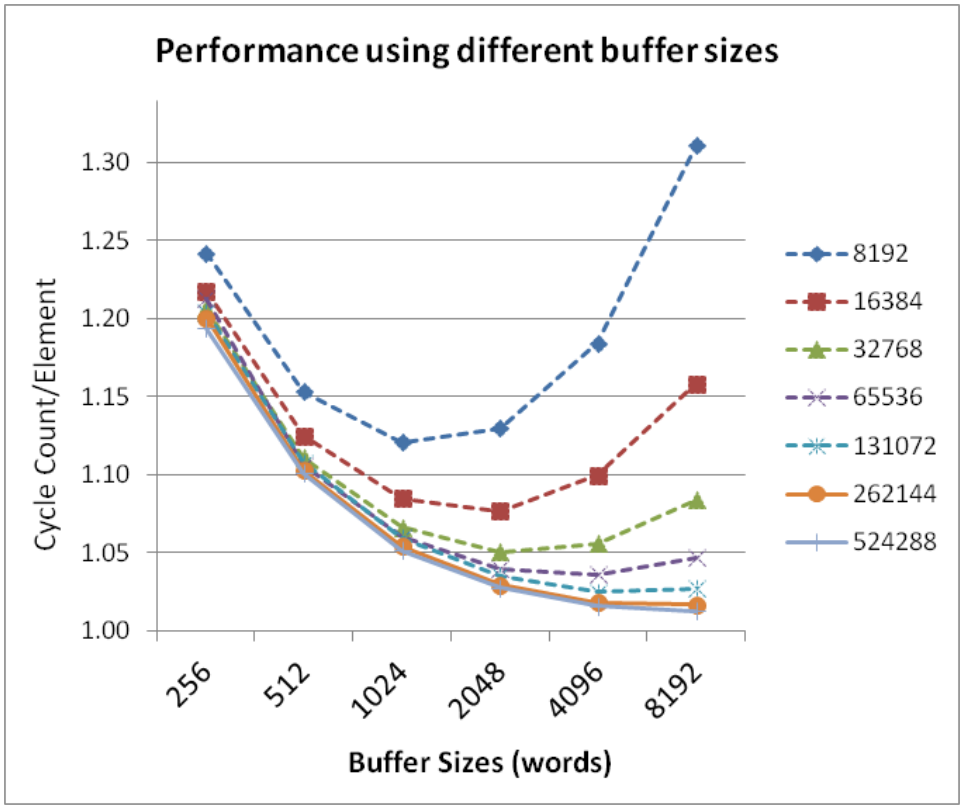

Figure 4.3: Impact of vector length selection for instruction count of 16 with different input data sizes

size used. When there are only 1-2 vector instructions, the application is memory-bound, and a vector length of 4096 (words) is preferred since it is the biggest size that can be used with double-buffering. When more instructions are involved, for example 5-8, the application moves towards compute-bound, and a vector length of 1024 (words) has the best performance. This choice provides good balance between memory transfer and vector execution. When the input size changes, the optimal vector length also changes.

Figure 4.3 shows cycle count per element when a fixed number of vector op-

erations (16) are performed with different input sizes, and different vector lengths. Y axis is switched to cycle count per element in order to compensate the large total cycle count differences. In figure 4.3, almost all cases are compute-bound. It shows that the time for transferring the first and last piece of data becomes less noticeable as input size increases to a large value. A larger vector length is again preferred for fewer pipeline bubbles.

Optimal Buffer Sizes for V16		Input Data Sizes (words)							
		8192	16384	32768	65536	131072	262144	524288	1048576
Instru -ction Count	1	4096	8192	8192	8192	8192	8192	8192	8192
	2	4096	8192	8192	8192	8192	8192	8192	8192
	3	2048	2048	4096	4096	8192	8192	8192	8192
	4	1024	2048	2048	4096	4096	8192	8192	8192
	5	1024	2048	2048	4096	4096	8192	8192	8192
	6	1024	2048	2048	4096	4096	8192	8192	8192
	7	1024	2048	2048	4096	4096	8192	8192	8192
	8	1024	2048	2048	4096	4096	8192	8192	8192
	9	1024	2048	2048	4096	4096	8192	8192	8192
	10	1024	2048	2048	4096	4096	8192	8192	8192
	11	1024	2048	2048	4096	4096	8192	8192	8192
	12	1024	2048	2048	4096	4096	8192	8192	8192
	13	1024	2048	2048	4096	4096	8192	8192	8192
	14	1024	2048	2048	4096	4096	8192	8192	8192
	15	1024	2048	2048	4096	4096	8192	8192	8192
	16	1024	2048	2048	4096	4096	8192	8192	8192

Figure 4.4: Vector length look-up table for V16

Several tables similar to figure 4.2 and figure 4.3 were collected for each VENICE configuration. A set of these tables forms a final look-up table like the one shown in figure 4.4 for each VENICE configuration. The best vector length (in number of words) that gives the best performance is listed in the table. The

compiler hashes the computation tree into an entry in the table based on its input size and number of operation nodes.

As can be seen from the table, when the instruction count exceeds a certain number, the vector length will saturate. Since it is impossible to have a table with an infinite number of rows, the table stops at 16 as a point where most of the vector lengths are saturated. Similarly, when the input array exceeds a certain size, the optimal vector length stops changing. Therefore, a 16×8 table like figure 4.4 is used for each VENICE configuration. For values that fall outside of the boundaries in the look-up table, they are simply rounded to the closest value in the table.

Vector lengths are chosen to be power of 2 because these values used most often in hand-written code. A more fine-grained vector lengths might be beneficial. However, the look-up tables only serve as references when choosing a vector length for real applications. Pursuing a perfect optimal point from a synthetic program is meaningless. In addition, curves in figure 4.3 and figure 4.2 become nearly flat when they get close to the lowest point, which leaves a wide margin for a vector length that leads to the near-optimal performance.

Using double-buffering will increase the total number of vector registers because two vector registers will be assigned to each user input data array.

4.2 Combining Operations

Although Accelerator and VENICE are both designed for operating on data arrays, the two instruction sets do not match perfectly.

As mentioned in chapter 3, the back-end sometimes has to write a sequence of code for certain Accelerator APIs. There are also opportunities where VENICE benefits from combining certain short sequences of simple operators into a single

compound VENICE operation. For example, VENICE has an absolute-value unit following each ALU within the pipeline. Taking an absolute value on the result of most vector instructions is free. This does not apply to multiplication and shift operations, which take place in the two-cycle multiply ALU. Therefore, the compiler can take advantage of this feature, and reduce instruction count in order to increase performance. For example, an Accelerator expression that takes an absolute difference of parallel array A and scalar value of 1 like:

```
D = Abs(A - 1);
```

can be translated into one vector instruction:

```
vector_abs_2D(SVW, VSUB, VD, 1, VA);
```

This also applies to accumulate operations as well, since VENICE has a dedicated accumulator inside the pipeline. The compiler will walk through all the IR graphs, find these sequences of operations, and replace them with new VENICE operations.

Out of the 6 selected benchmarks, motion estimation benefits the most from this optimization, because it calculates the absolute differences of two image frames. A total instruction count reduction of 30% is achieved, which saves the execution time by 32% (a 1.47× speedup) on average across different VENICE configurations. Table 4.1 shows the performance improvement for motion estimation in detail.

	V4	V16	V64
runtime before combine operators (ms)	2.03	0.55	0.30
runtime after combine operators (ms)	1.36	0.37	0.21
speedups	1.49×	1.48×	1.43×

Table 4.1: Performance before and after combining operations for motion estimation

4.3 Summary

This chapter walks through a few target-specific optimizations made upon the basic implementation described in chapter 3. Experimental results showed solid evidence on how the selection of vector length affects runtime. It demonstrated that, for each application, there exists a vector length that achieves optimal performance based on its input array sizes and the number of operations performed on each array element. In addition, to take advantage of special function units in the VENICE architecture, such as the absolute value unit, the compiler makes an additional pass to combine certain sequences of Accelerator operations into a single VENICE instruction. This further improved performance on some of benchmarks. Figure 4.5 summarizes the final compiler flow with all the optimization stages.

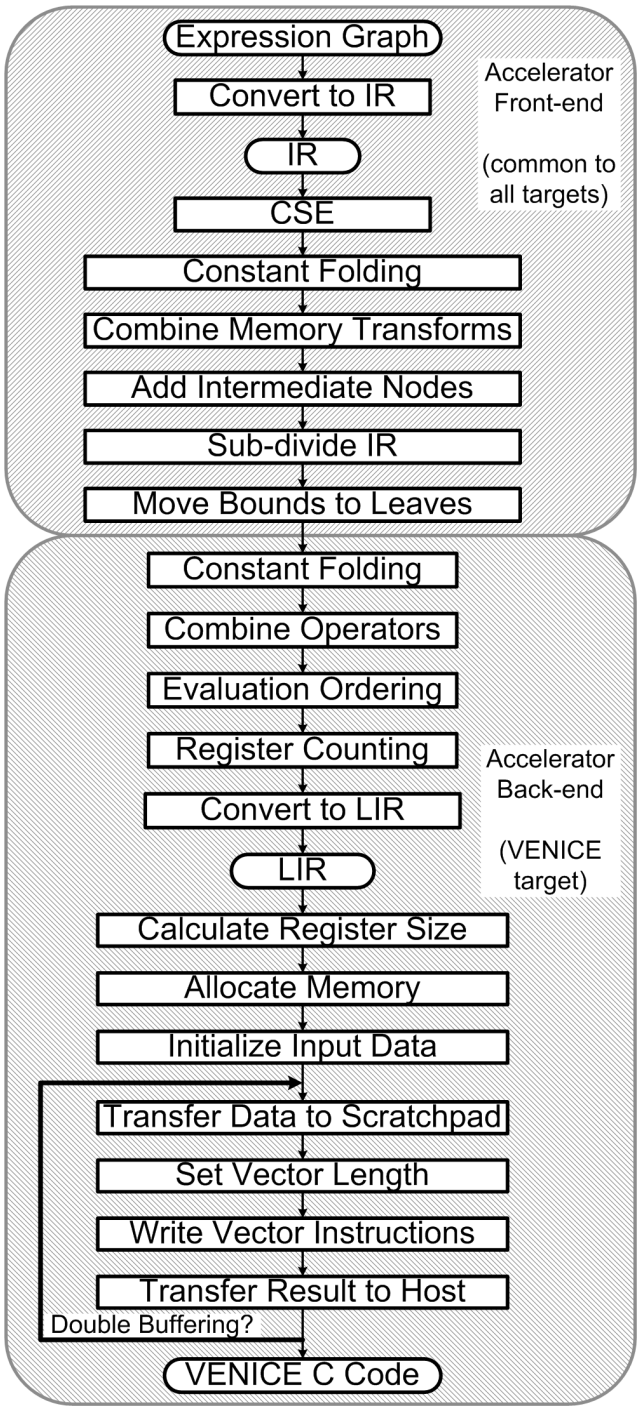

Figure 4.5: Complete VENICE compiler flow

Chapter 5

Experimental Results

This chapter will introduce the benchmarks selected to test the quality of the compiler design followed by experimental strategy. Runtime results will be presented showing that the compiler-generated code achieves performance close-to or better-than human-optimized assembly. By comparing to execution time on a single Intel core, VENICE is shown to be competitive with modern processors. Finally, evidence is given to show that the VENICE target delivers better scalability than the Multi-core target of Accelerator.

5.1 Benchmarks

To measure the quality of the compiler, a set of 6 highly data-parallel benchmarks are selected to test how well it performs on a variety of applications on both small and large size SVPs.

Benchmark Name	Description	Input Size (words)	Output Size (words)	Instruction Count	Additional Information
fir	Finite Impulse Response filter	8192	8176	32	Fixed point arithmetic. Consists of multiply and add operations on 16 shifted variants of a 1D input array.
2D fir	Two-dimensional fir filter	320×240	318×238	18	Fixed point arithmetic. Consists of multiply and add operations with tap size of 3x3.
life	Conway's game of life	256×256	256×256	11	Integer arithmetic. Consists of sum of values of neighbor grids and comparison operations. Performs one iteration on the initial state of grid.
imgblend	Image blend	$320 \times 240 \times 2$	320×240	4	Fixed point arithmetic. Combines two images into one by multiplying different coefficients to the pixels of two input images.
median	Median filter	128×128	124×124	678	Integer arithmetic. Replaces each pixel of input image with the median of a 5×5 nearby window. The median value is calculated using a bubble sort algorithm.
motest	Motion estimation	48×48	32×32	512	Integer arithmetic. Sum of absolute differences between the input image and a sample image within a 16×16 2D window.

Table 5.1: Benchmark descriptions

Table 5.1 lists the set of benchmarks used for performance evaluation. These highly data-parallel applications are shown to scale fairly well. The selection of benchmarks was also influenced by the ease of expressing the computation using a subset of Accelerator APIs supported by the VENICE target. Other applications are expected to benefit in the same way.

The benchmarks cover both 1D applications, such as fir, and 2D applications, such as motest. Some of the benchmarks are memory-bound, such as imgblend, where others are compute-intensive, such as median. The instruction count column indicates the number of operations performed on each element of the input array. Medium input sizes are used for all benchmarks, which provides enough data parallelism and fast collection of results.

5.2 Experimental Strategy

In order to demonstrate the validity, effectiveness and efficiency of the compiler, experiments are designed in order to answer the following questions:

1. Is the generated code correct?
2. Is the generated code of good quality?
3. How well does the performance scale over SVPs with different sizes?
4. How much overhead does the compiler introduce?

To answer these questions, all the benchmarks are coded in both Accelerator and native VENICE assembly. The assembly code is carefully optimized for performance following the principles described in Chapter 2. During the result collecting stage, some of the hand-written benchmarks were found to be much slower

than the compiler-generated ones. In these cases, they were re-written to get better performance. Both compiler-generated code and human-optimized assembly are run on VENICE with 4 different configurations – V1, V4, V16, and V64. The number after V means the number of ALUs instantiated in the vector engine. This also affects the size of scratchpad memory, since a fixed 8KB of memory is assigned to each vector lane. V64 is an exception with only 4KB memory available for each vector lane. This is due to the limited on-chip memory in the Stratix III FPGA used by this work. Due to a shortage of DSP blocks, V64 is also modified to fit in a DE3 system by removing 8-bit and 16-bit multiply support; Accelerator does not support these modes either.

The execution time for human-optimized assembly and compiler-generated code are compared side by side to see if the compiler could intelligently translate Accelerator code into VENICE assembly. Performance of the same benchmark running on VENICE with a different number of ALUs also tests if the compiler can generate scalable code.

The scalability of the code generated by the VENICE back-end is also compared to performance of Accelerator's Multi-core target. Ideally, the experiments should show the scaling speedups from the Multi-core target when it uses a different number of CPUs on the same machine. However, since Accelerator is designed to completely hide hardware information from the user, it is impossible to directly control resource usage. The Multi-core target always uses all resources available in the hardware. Instead, the VirtualBox hypervisor is used. Varying hardware configurations for the multi-core device is accomplished by assigning different number of processors to the virtual machine.

Due to the inseparable JIT time and relatively high JIT overhead of the Multi-core target, larger input arrays are used for multi-core execution. Since only scalability is being compared, the change of input sizes should not affect the validity of the results. The Multi-core target always generates SSE3-compliant code, which mostly works with single-precision floating-point data. This lack of efficient integer arithmetic in SSE3 forces the Multi-core target to unpack values already loaded to the SSE registers and send them to general purpose registers. The unpacking occurred in almost every operation, which becomes a huge overhead. To avoid this drawback, the benchmarks are re-written to use float data type for the Multi-core target. This change of data type is considered to be harmless since float and integer types are of the same size (4 bytes), which puts the same burden on memory accesses.

However, floating-point arithmetic is not supported by VENICE. Also because Accelerator does not support the byte or short data types, integer is used as a unified data type for all benchmarks in VENICE assembly and the VENICE Accelerator back-end.

The best runtime results of VENICE produced by the compiler-generated code are selected to compare to an Intel CPU to show that even running at a low frequency, VENICE can be competitive with a modern CPU running at a much higher frequency.

Each benchmark self-verifies the results of vector execution against a simple, optimized sequential C solution. This sequential C code is used for both the single-core Intel CPU and the Nios II execution baseline.

All soft processor results are measured by running the benchmarks on an Altera DE3-150 development system with a Nios II/f processor running at 200MHz.

The Nios II/f processor is configured to use a 4kB instruction cache and a 4kB data cache. All vector engines run at a fixed clock rate of 100MHz. (Small size SVPs like V4 can run at a much higher frequency, but the maximum frequency drops quickly when the vector engine increases to 64 lanes.) Single-CPU results are derived from an Intel Xeon E5540 running at 2.53GHz. The Multi-core target is tested on two machines. One is the same Intel Xeon machine used for sequential execution. This Intel machine has 4 cores and 8MB cache; VirtualBox is configured to use 16GB of memory. The other one is an AMD Opteron 6128 machine with a total of 32 cores, each running at 2GHz. There is only a 512KB cache on the AMD machine. The 32 cores are distributed among 4 sockets with 8 cores per socket; 16GB of memory is assigned to the VirtualBox on AMD as well.

5.3 Results

5.3.1 Execution Time

Speedups for different VENICE configurations – V1, V4, V16, and V64, over the serial Nios II/f implementation are shown in figure 5.1. All VENICE results are a minimum runtime of 10 consecutive runs. For both hand-written and compiler-generated code, only the part related to vector execution from allocating memories in the scratchpad, until all scratchpad memory is freed, is timed. Time for code generation and initialization of input arrays is not included.

Table 5.2 is a detailed listing of speedups of compiler-generated code versus hand-written code on the 4 VENICE configurations. As shown, the compiler-generated code out-performs the manually-written code in most cases. This is because Accelerator puts more effort into the optimizing process than a human:

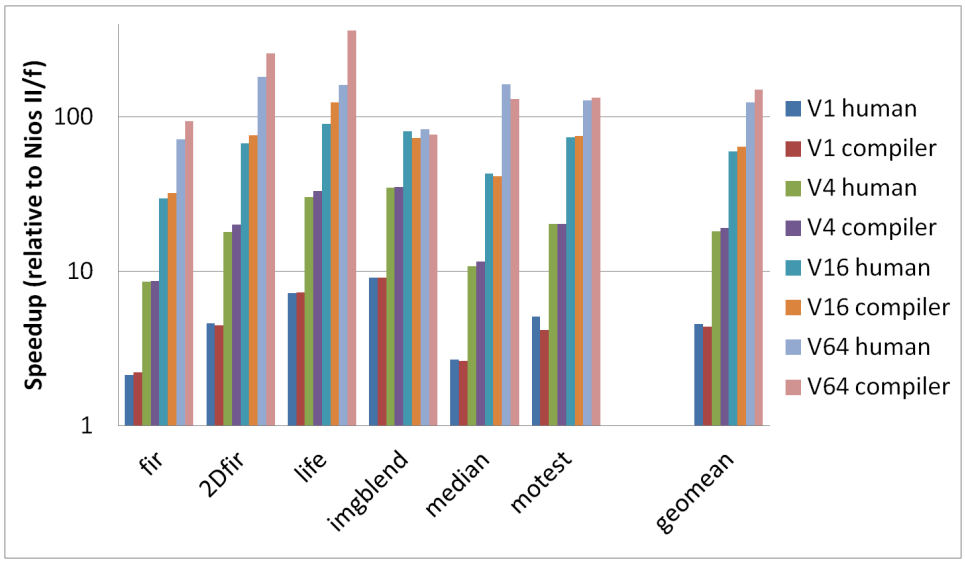

Figure 5.1: Speedups over Nios II implementation for compiler generated code and hand-written code

	fir	2Dfir	life	imgblend	median	motest	geomean
V1	1.04	0.97	1.01	1.00	0.99	0.82	0.97
V4	1.01	1.12	1.10	1.02	1.07	1.01	1.05
V16	1.09	1.12	1.38	0.90	0.96	1.01	1.07
V64	1.30	1.42	2.24	0.92	0.81	1.04	1.22

Table 5.2: Speedups of compiler generated over hand-written code

1. Accelerator fully unrolls inner loops to reduce loop overhead;

2. Accelerator carefully chooses the vector length by a profiled look-up table rather than conservatively rounding down or guessing;

3. Accelerator always double-buffers data transfers when necessary;

4. Accelerator inlines all function calls to avoid function call overhead.

The fastest, life, achieves 370× speedup compared to a Nios II/f, and a 2.24×

speedup compared to the hand-written code on V64. However, humans can sometimes do far better than the compiler: motion estimation can achieve another 1.5× speedup if the VENICE accumulator is used in a clever way, where the compiler cannot do so automatically.

```
1    input_type *v_min = v_input1;
2    input_type *v_max = v_input2;
3    // copy min values to tmp
4    vector( VVW, VOR, v_tmp, v_min, v_min );
5    // v_sub flags are set if max<min
6    vector( VVW, VSUB, v_sub, v_max, v_min );
7    // cond. move, min = (max<min)?max:min;
8    vector( VVW, VCMV_LTZ, v_min, v_max, v_sub );
9    // cond. move, max = (max<min)?min:max;
10   vector( VVW, VCMV_LTZ, v_max, v_tmp, v_sub );
```

Figure 5.2: Hand-written compare and swap

```
1    vector(VVW, VSUB, v_tmp, v_input1, v_input2);
2    vector(VVW, VCMV_GTEZ, v_min, v_input2, v_tmp);
3    vector(VVW, VCMV_LTZ, v_min, v_input1, v_tmp);
4    vector(VVW, VCMV_GTEZ, v_max, v_input1, v_tmp);
5    vector(VVW, VCMV_LTZ, v_max, v_input2, v_tmp);
```

Figure 5.3: Compiler generated compare and swap

The one benchmark that falls short of performance on compiler-generated code is median, which only yields in 0.81× performance of the hand-written code on V64. This is mainly because of the difference between hand-written code and generated code for the compare-and-swap operation sequence. Figure 5.2 and figure 5.3 present the hand-written code and compiler-generated code for the compare-and-swap, respectively. The hand-written code intelligently re-uses data buffers and results in just 4 instructions instead of 5 produced by the compiler. This difference also leads to a demand for 39 buffers for the compiler-generated code,

which are 12 more than the human-written program. Since median filter uses a bubble sort algorithm, almost all the vector instructions are doing compare-and-swap. This leads to an almost 25% instruction count increase in total for the compiler-generated code, which is almost exactly the amount of performance degradation seen in median on V64.

Table 5.3 compares runtime results of the VENICE compiler (not human) to a single-core 2.5GHz Intel Xeon E5540 processor compiled with Visual Studio 2010 with -O2 optimization. Each benchmark kernel is run 1000 times for the Intel execution, and average runtime is reported. Interestingly, VENICE is able to beat the Intel CPU on 4 of the 6 benchmarks. Intel beats VENICE only on imgblend, which is memory bandwidth limited.

CPU	fir	2Dfir	life	imgblend	median	motest
Xeon-E5540	0.07	0.44	0.53	0.12	9.97	0.24
VENICE V64	0.07	0.29	0.23	0.33	2.50	0.15
Speedup	1.0×	1.5×	2.3×	0.4×	4.0×	1.6×

Table 5.3: VENICE and single-CPU runtimes (ms) and speedups of VENICE vs. single CPU

Most of these benchmarks do not need to use 32-bit integers. Using a smaller data type in VENICE gives better performance, because each 32-bit ALU can be fractured into four 8-bit or two 16-bit ALUs. Table 5.4 listed the actual data type originally used for each benchmark. To demonstrate the performance impact of using smaller data types, the hand-written code is modified to express the same benchmark in possible smaller data types. Input sizes (number of elements) are kept the same with the ones in table 5.1. When using smaller data types, vector lengths can also scale to a larger value.

In table 5.5, the three benchmarks operate on bytes are listed, along with speedups over using word integer types on V1, V4, and V16.[1] An up to 4× speedup is achieved. In table 5.6, the two benchmarks that operate on half words are shown, along with speedups over word integer types on V1, V4, and V16. Speedups of up to 2× are shown. Unfortunately, these data types cannot be expressed in Accelerator to get better performance. It is desirable for Accelerator to support custom data types.

fir	2Dfir	life	imgblend	median	motest
byte	half word	byte	half word	byte	word

Table 5.4: Benchmarks' natural data types

	fir	life	median	geomean
V1	3.93	4.36	4.07	4.12
V4	3.54	3.83	4.03	3.79
V16	2.90	3.22	4.00	3.34

Table 5.5: Speedups for benchmarks operating on byte vs. word

	2d fir	imgblend	geomean
V1	1.96	1.54	1.74
V4	2.00	1.46	1.71
V16	1.97	1.83	1.90

Table 5.6: Speedups for benchmarks operating on half word vs. word

5.3.2 Scalability

Figure 5.4 and figure 5.5 show how the Accelerator Multi-core target scales across 32 CPUs on an AMD machine and 4 CPUs on an Intel machine, respectively.

[1] V64 is not used because support for 8-bit and 16-bit multiply was removed.

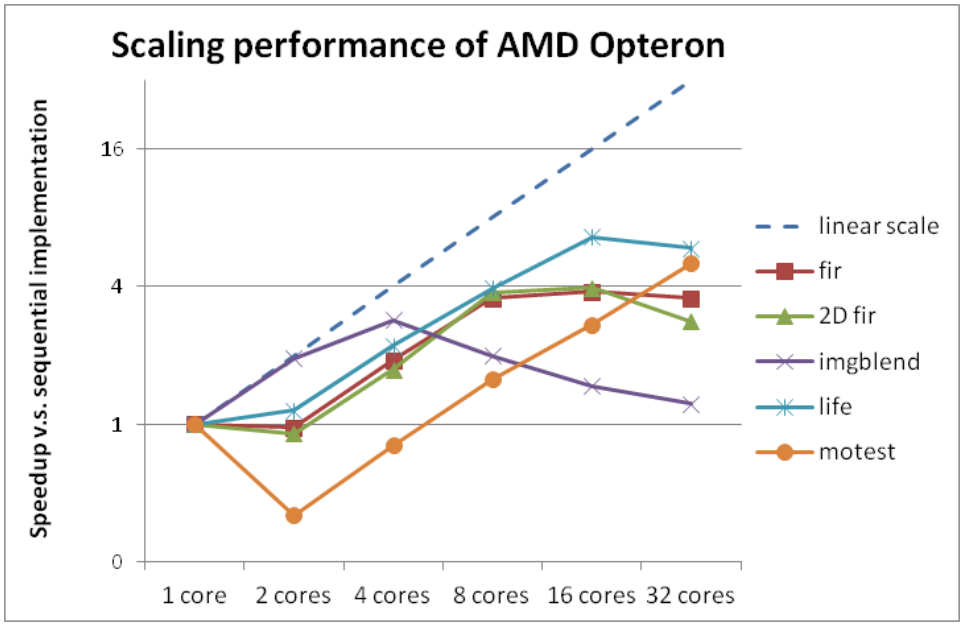

Figure 5.4: Scaling performance of the Accelerator multi-core target on AMD Opteron

Larger input sizes are used for multi-core execution due to processors' much higher frequency than VENICE. Table 5.7 lists the two sets of inputs used for multi-core and VENICE execution. Figure 5.6 shows how VENICE target scales over a set of VENICE configurations using 1, 4, 16, and 64 ALUs. In all figures, the dotted line shows linear scaling performance for all cases.

For VENICE, V1 is used as baseline in figure 5.6, because the scalar Nios II execution suffers from loop overhead and cache misses; VENICE directly operates on scratchpad memory with Nios II handling all loop bounds in parallel. Results in figure 5.1 indicate that V1 is up to 9× faster than Nios II alone.

On the AMD machine, there is no performance gain when using more than 4 cores for the benchmark imgblend, and performance stops scaling beyond 8 cores

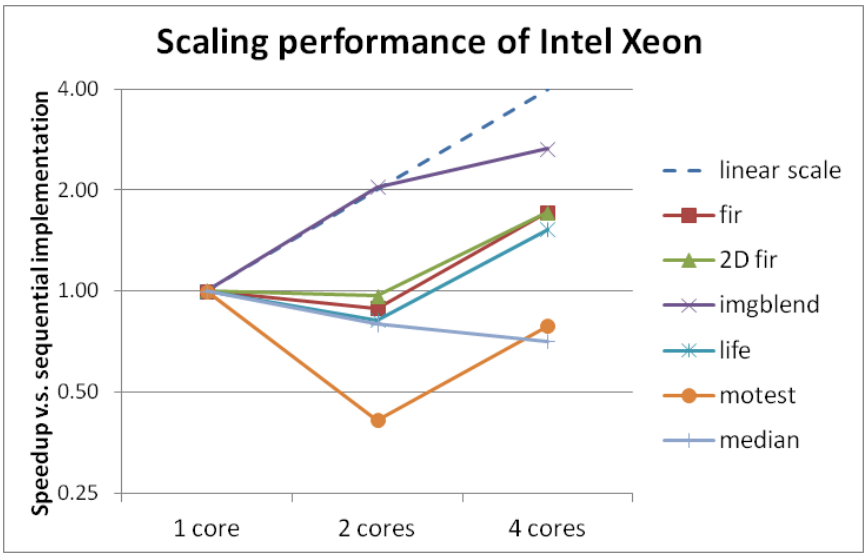

Figure 5.5: Scaling performance of the Accelerator multi-core target on Intel Xeon

	fir	2d fir	life	imgblend	median	motest
Input size for Multi-core (million words)	481.92	1105.92	67.11	768	67.11	1.92
Input size for VENICE (words)	8192	76800	65536	76800	16384	1024

Table 5.7: Input sizes used for Multi-core and VENICE execution

for benchmarks fir and 2D fir. The life benchmark also starts to slow down beyond 16 cores. The benchmark median suffers from significant slow-down on the AMD machine, and it is omitted from figure 5.4 due to being too far away from all the other curves. Only motest shows continuing speedups at 32 cores. However, there is significant overhead introduced by the Multi-core target. Curves for most of the benchmarks always keep a distance from the linear scaling line. Performance on the Intel machine delivers similar messages. Median has better performance on

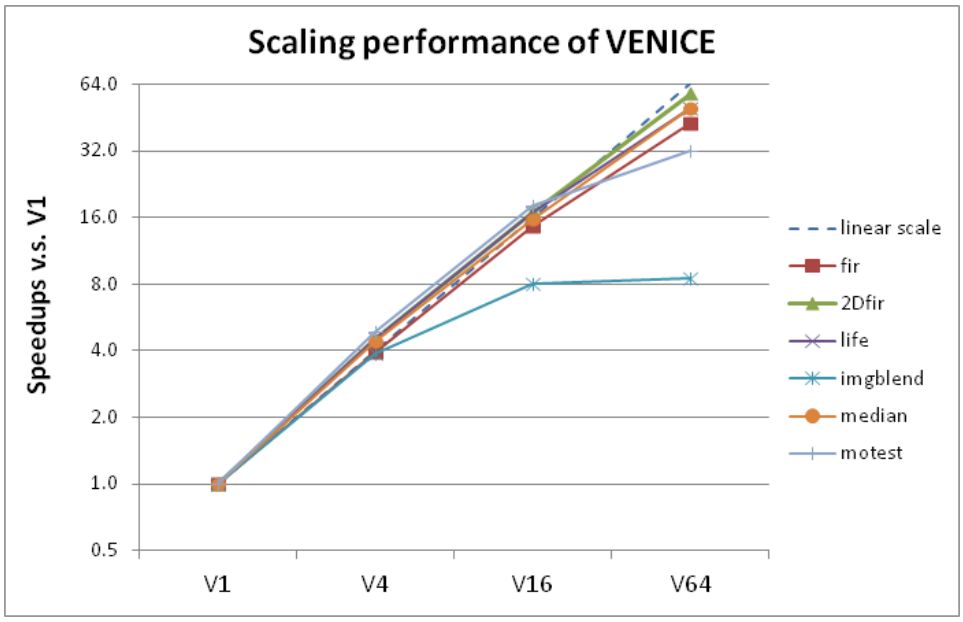

Figure 5.6: Scaling performance of the Accelerator VENICE target

the Intel machine, and is shown in figure 5.5. However, it keeps slowing down when using more cores. On the other hand, the VENICE scales near-perfectly up to 64 ALUs for most of the benchmarks. Only imgblend, which is memory-bound, saturates early. Some of the benchmarks show super-linear speedups in figure 5.6. This is because a larger VENICE configuration has a larger scratchpad memory capacity, allowing near-optimal vector length being used. For example, 2D-fir requires 4 vector registers; with a scratchpad capacity of 4KB on a V1, only 256 words can be assigned to each one. However, in the profiled look-up table, a vector length of 512 words is desired to get optimal performance. When executing on a V4, 1024 words are available for each vector register, which is the desired vector length.

5.3.3 Compile Time

All reported benchmark runtimes for VENICE in previous sections are kernel runtimes, while runtimes for the Multi-core target include the JIT time. This is an unfair comparison due to the omitted VENICE compile time. Compile time here refers to the time for running the Accelerator VENICE target to generate the output C file. Table 5.8 records the compile time of all benchmarks.

The compile time is mostly determined by the complexity of the expression graph, since the compiler walks the IR graphs several times to do conversions and optimizations. Most of the benchmarks have a relatively simple expression graph except for median, which consists of 678 operation nodes and 39 IR graphs in total.

For applications with small input sizes, the compile time could kill the performance gained by VENICE parallelism. However, for large input sizes, compile time will stay the same as it is independent from input data sizes. In this case, the compile time will have a much smaller impact on performance. When applying the same large input sizes used for multi-core execution, the compile times only take about 17% of the total execution times on average. Total execution time is a combination of compile time and kernel runtime. Furthermore, the compiler implementation could be optimized in many places to achieve a reduced compile time. This is listed as a future work in the next chapter.

fir	2Dfir	life	img	median	motest	geomean
4.7	5.1	4.5	4.4	92.7	24.3	10.1

Table 5.8: VENICE target compile time (ms)

Due to the lack of code access, compile time of the VENICE target cannot be compared to JIT time for the Accelerator Multi-core target. It is mentioned in [22]

that overhead of Accelerator system does not impose a noticeable cost in execution time.

5.4 Summary

This chapter reported some experimental results. Comparisons between runtimes of hand-crafted code and compiler-generated code on VENICE show that the compiler is able to generate code with high quality that delivers performance close-to or better-than hand-written code. The improved performance using a smaller integer data type for some of the benchmarks demonstrates the further potential benefits of supporting these data types in Accelerator. Speedups of the VENICE target versus a sequential implementation on an Intel single-core CPU show that VENICE is competitive with modern processor architectures. The VENICE compiler also deliver near-linear scaling up to 64 ALUs, where the performance of the Accelerator Multi-core target stops scaling at a much earlier point on 5 of 6 benchmarks. Compile time of the VENICE back-end is also reported. It could be larger than the vector execution time for small applications. However, it does not impose significant overhead on applications with much larger input sizes. In addition, the compile time can be reduced by optimizing the compiler implementation.

Chapter 6

Future Work

In this chapter, some notable limitations of this work are enumerated. They are caused by the Accelerator framework, or by VENICE architecture restrictions, or by prioritizing implementation tasks within limited time. Ideas on further improving the system are also presented.

6.1 Limitations

Some of the compiler design limitations were mentioned in chapter 3. More will be discussed in different aspects with more detail in this section.

6.1.1 Accelerator

Unlike a self-developed language and compiler co-design approach, the already-in-the-market Microsoft Accelerator is used as high-level model, and supports pluggable 3rd-party back-ends. It greatly saves time for development. However, during the process of the compiler implementation, the compiler design was held back by several restrictions introduced by Accelerator APIs and its built-in front-end.

First, the CSE algorithm inside the Accelerator front-end can not be modified. Benchmark code uses the Evaluate() API as a work-around to force breaking up the IR graph. The user should not be aware of this limitation and be forced to alter their programs.

Second, it is desirable for Accelerator to support extensible APIs and data types. Custom APIs were listed as an upcoming feature of Accelerator in [28]. Due to the mismatched instruction set design between Accelerator and VENICE, functionalities of VENICE are not fully utilized. All of the logic operations such as bit-level AND, OR, XOR, SHIFT and ROTATE are completely missing from the Accelerator APIs. These are very important for VENICE, which works mainly on integer and fixed-point arithmetic.

As shown in chapter 5, smaller data types that take advantage of the fracturable ALUs in VENICE can achieve another 2-4$\times$ performance boost. This is one of the key assets of the VENICE architecture. Unfortunately, the compiler could not offer these performance advantages. Also, there is no unsigned integer type in Accelerator. This might affect applications that can be expressed in smaller data types. Support for these new data types is mentioned in [28] as a potential future work for Accelerator.

Another issue with data types was also found during the result collecting process. The Accelerator Multi-core target produces SSE3-compliant instructions. However, due to lack of integer support in SSE3, all integers have to be unpacked and moved to General Purpose Registers (GPRs) from the packed 128-bit SSE registers to perform the operations. This unpacking and moving process hinders the Multi-core target from getting any speedups over a single-CPU implementation on integer applications. Floating point is used instead, resulting in a slightly longer

runtime than integer (less than 15% increase in runtime). The Accelerator Multi-core target should be able to detect existence of SSE4.1 or SSE4.2 support on the fly, and take the advantage of additional integer support in SSE4.1 and SSE4.2.

Third, due to the inseparability of JIT time from algorithm run-time in the Multi-core target, the Multi-core target performance could not be deeply analyzed. Supporting a pre-compilation mode, which separates the JIT-ing phase from execution, will be a nice feature for developers and researchers.

Fourth, since Accelerator is mainly used internally within Microsoft Research, only an external release of the CUDA target is available, and there were troubles collecting CUDA target results due to some technical issues. It will be interesting to see the CUDA target performance since it is more similar to the FPGA architecture with a separate device memory from the host memory.

Fifth, Accelerator is designed to completely hide hardware details from programmers, and target diverse platforms with portable code. This restricts the user from managing any hardware resources. With Accelerator always using all available resources, it is impossible to directly test the scalability of the Multi-core target. Instead, a virtual machine is used to control the resources that are exposed to Accelerator. However, the virtual machine adds overhead to the thread management, and degrades the Multi-core performance. This overhead can be avoided if Accelerator provides user-level control over the amount of hardware resources used. Since the number of threads used by the CUDA target is also not controllable, no further effort was made on getting the CUDA target to work.

6.1.2 VENICE

The VENICE architecture is still an on-going research project being developed at the SoC lab in The University of British Columbia. The limitations described below are under consideration to be improved for future generations.

Many compute-intensive applications require floating-point units for greater dynamic range, but VENICE was designed to be area efficient and could not afford a resource-expensive FPU for each vector lane. This prevented us from supporting any floating-point in the Accelerator APIs, e.g., Log(), Sin(), and Cos(). These operations can be found in many compute-intensive financial applications.

The support for scatter/gather operations in VENICE is still at the early experimental stage. Effective scatter/gather execution is usually needed, particularly for doing data conversion (e.g., converging bytes to words) and for rearranging data (e.g., de-interleaving r,g,b triplets). A lookup-table instruction is also needed to assist with gathering data. These special memory access operations combined together would enable VENICE to perform algorithms such as histograms and sorting.

6.1.3 Compiler Design

Currently, only a subset of the complete Accelerator Instruction Set Architecture (ISA) is supported. Part of this is due to lack of an FPU in VENICE. Also due to time constraints of the project, some rarely used APIs are omitted from this work. Therefore, supporting these APIs would make this work more complete.

Another drawback of the design is how memory transformations were handled. It is mentioned in chapter 3 that out-of-bound memory accesses are handled by extending the original array with appropriate values during initializaion. Extra

memory will also be allocated to these input arrays in the scratchpad. However, the padding could be expensive. For example, if an 4×4 input array is shifted up and down by 3 rows, the padding nearly triples the memory allocated to this input array as shown in figure 6.1. With double buffering, a total of $4 \times 10 \times 2$ words of memory needs to be assigned to this input. For a much larger array, this cost can increase dramatically. Therefore, a mechanism is needed to detect such cases when padding might hinder the performance by taking too much on-chip memory, and handle the memory transform differently.

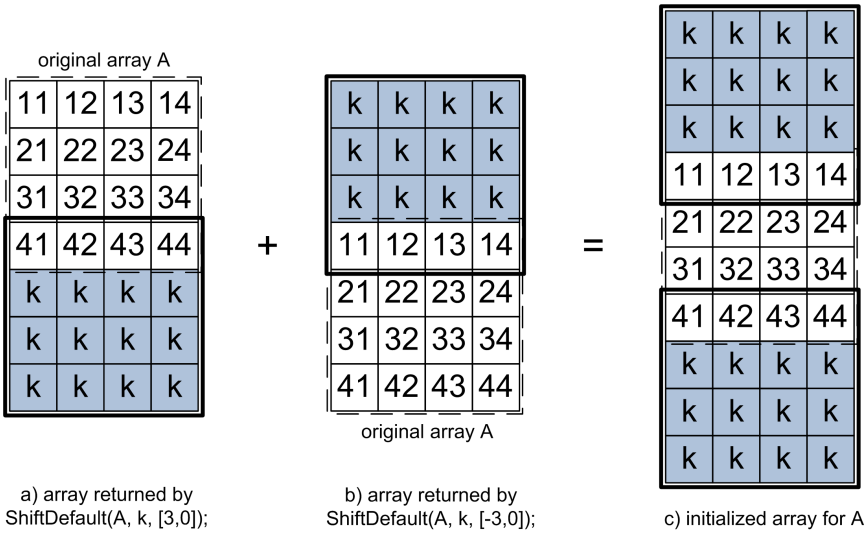

Figure 6.1: Accelerator compiler flow

As mentioned in chapter 3, when partitioning the scratchpad memory, the scratchpad is considered to have an infinite number of vector registers. As the register count goes up, the compiler simply reduces the size of each one. Therefore, the entire computation can be performed upon one segment of the input array, which produces a segment of the final result. This saves the overhead of transferring temporary results back to main memory and loading to the scratchpad again

later. However, if the number of vector registers required by an application is exceptionally high, resulting in a vector length even shorter than the total lane width of the vector engine, the situation should be handled by spilling partial results back to main memory to maintain a reasonable level of data parallelism.

Details on how to decide the vector length according to a profiled look-up table for each VENICE configuration are presented in chapter 4. Vector lengths selections are set to powers of 2, ranging from 512 words to the maximum value that allows double buffering within scratchpad capacity. This only gives a near-optimal solution; performance might be slightly improved by using sizes with finer granularity.

Due to the time limit of this work, the compile time reported in chapter 5 can take up to as much as the vector execution time on applications with small data sizes. The compile time may be further reduced to have minimal performance impact. Right now, the back-end does several walk-down passes to the IR and LIR graphs for different optimization purposes or different steps of code generation. Some of this can be eliminated if the Accelerator front-end code is accessable, e.g. there is another round of constant folding in the back-end, which is repeated work from the front-end. Some of this can be combined into one pass and thus save compile time.

Although implemented in C++, Accelerator targets usually support multiple primitive languages such as C++, C#, F#, and Fortran. One future work should also be adding a wrapper for other languages besides the native C++.

6.1.4 Evaluation Methodology

When selecting benchmarks for this work, [5] is used as a reference. These benchmarks provide an easy comparison of hand-written assembly between the old architecture and the new one. In addition, some of this code is ready to use with minor modifications. However, some of the benchmarks were omitted either due to poor scalability, the requirement for small integer data types, or complicated dimension changes and memory accesses that are hard to express in Accelerator. Benchmarks in [22] and [29] are also taken into consideration, and the ones that do not require floating-point operations are selected as well. However, the entire benchmarks suite still does not cover enough diversity.

Most restrictions come from the existing Accelerator APIs and the off-line approach taken by the VENICE back-end. With custom data types and custom API support by Accelerator, there will be more integer based or fixed-point compatible applications that can be efficiently expressed in Accelerator. An FPU and scatter/gather support by VENICE will open the door to an increased number of compute-intensive applications.

A set of configurations of VENICE is selected for this work, which are V1, V4, V16, and V64. In these configurations, a unified DMA queue length of 1 and a unified instruction queue length of 4 are used. However, these queues are configurable components. Although configuring these components differently should not have significant impact on performance, a detailed investigation should be conducted to verify this theory. The commercial version of VENICE, which has an improved hazard detection system, might also give more competitive results.

The VENICE system runs on a Stratix III FPGA on a DE3-150 board. The

limited on-chip memory and multiplier blocks forced us to use a modified version of V64 with all 8-bit and 16-bit multipliers removed, and a reduced memory per vector lane. Conducting experiments on a complete V64 with a larger FPGA might provide more precise performance. Furthermore, it would be exciting to see if the compiler can deliver continuous scaling performance beyond V64.

6.2 Ideas

6.2.1 JIT mode

The off-line approach taken by the VENICE target imposes some restrictions on the compiler. A JIT mode for VENICE target would be really useful for further extending the usability of the system. The JIT mode could enable analyzing loop bounds on the fly, which the VENICE target is unable to do. This strikes out the opportunity for implementing certain benchmarks. In addition, there is no direct access to the host memory space in the off-line mode. This forces the VENICE target to spend long time copying user data into the generated code.

Unlike the FPGA target, VENICE does not require a lengthy place-and-route process for each run. The generated code only needs to be compiled by gcc and downloaded to an FPGA board. This secondary compilation time and downloading time is acceptable for a JIT mode. However, challenges remain with the difficulty of reading and writing from and to the FPGA board. A standard for transferring data on and off FPGA would be an ideal solution. Basic support for a file system in Nios II will also be a solution.

6.2.2 Instruction Scheduling

It is mentioned in chapter 4 that data dependencies can produce pipeline bubbles between vector instructions. With a relatively long vector length, this effect can be minimized. However, when an application requires a large number of vector registers forcing a small vector length, the instruction order might be transformed by the compiler to greatly improve performance.

For example, when using a V64 with a VL of 64, each vector instruction could be completed within 1 cycle. If there is data dependency holding up the next instruction, a 5 cycle bubble will be introduced after each instruction, resulting in almost 6$\times$ more execution time.

6.3 Summary

Many possible limitations of this work were thoroughly considered in this chapter. The lack of support for custom APIs and data types of the Accelerator framework restricted some of the benchmarks from getting better performance. The benchmark suite could be enlarged with support of custom APIs. Some developing features of VENICE could further widen the application range. The compiler design suffers from some drawbacks as well, such as imcomplete support of entire Accelerator ISA and handling extreme cases. More aggressive ideas on the project, such as supporting a JIT mode, and instruction scheduling to remove data dependencies were discussed as well.

With all these limitations for Accelerator, VENICE architecture, and compiler design resolved, more applications will be able to benefit from the system.

Chapter 7

Conclusions

This work presents a compiler design for the VENICE soft vector processor. It translates Accelerator, which is a high-level abstract language, into the native assembly of VENICE. The entire design is embedded inside the Microsoft Accelerator system, serving as a new back-end target in addition to the existing Multi-core, DX9, CUDA, and FPGA targets.

The compiler design greatly improves the programming and debugging experience for VENICE. The user can write simple expressions in Accelerator without learning about architecture details of VENICE, and debug with Visual Studio debugger instead of writing print statements required by assembly debugging.

One of the major challenges in compiler design is to efficiently manage the scratchpad memory in VENICE, because the scratchpad can be divided into an arbitrary number of vectors with arbitrary sizes. To simplify this task, the scratchpad memory is treated as a 'virtual vector registerfile'. The number of vector registers and their sizes are dynamically decided by the compiler. Each application is presented as a computation tree internally. First, the back-end compiler uses

a modified version of Appel's generalization of Sethi-Ullman's algorithm to obtain an optimal evaluation order of computation trees. Then, a reference-counting method is adopted to precisely calculate the number of vector registers needed. To further improve performance, the compiler takes a profiling approach to obtain an optimal vector length for each application characterized according to its number of instructions and input sizes. These vector lengths form a look-up table, which is then referenced by the compiler to decide the vector register size to use. When generating code, the compiler also applies a double-buffering technique to hide memory latencies.

With all of these optimizations, the compiler can generate high-quality code with comparable performance to human-optimized programs. Experimental results show that compiler-generated code for VENICE can achieve speedups over 370× using Nios II/f as a baseline, and speedups up to 2.24× versus the human-optimized assembly. In addition, the compiler-generated code executing on VENICE operating at 100MHz is up to 3.s× faster than C code running on a 2.53GHz Intel Xeon E5540. Compared to limited scalability of the Multi-core target of Accelerator, the VENICE back-end delivers an almost linear scaling from 1 to 64 ALUs on 5 of 6 selected benchmarks.

Microsoft Accelerator is an excellent choice as a basis for developing a vectorizing back-end code generator. It provides an easy-to-use user interface and an easy-to-extend developer interface. This work has demonstrated that systems like Accelerator could be a sustainable solution for emerging architectures.

Limitations are discovered on both the software and hardware sides that would be good to address with future work. These are extensively discussed in chapter 6.

For VENICE, most benchmarks are done with fixed-point arithmetic; the addi-

tion of floating-point would greatly expand the application base. Also, support for irregular memory access is needed.

For Accelerator, the compiler implementation was held back by lack of support for custom APIs and data types. Limited code access to the front-end also forces repeated work in the back-end and increases the compiler runtime.

For the back-end implementation, future work includes finishing support for the rarely-used Accelerator APIs that were omitted, adding support for primitive languages other than C++, and reducing compiler runtime. Also, the benchmark suite and methodology need to be expanded. A JIT mode for the VENICE target can be a more aggressive goal to further improve the system.

Bibliography

[1] Z. Liu, A. Severance, S. Singh, and G. G. Lemieux, "Accelerator compiler for the venice vector processor," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 229–232. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145732

[2] A. Severance and G. G. F. Lemieux, "VENICE: a compact vector processor for FPGA applications," 2011. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.student-posters/HC23.19.p20-VENICE-Severence-UBC.pdf

[3] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core CPU accelerator," in *FPGA*, Monterey, California, USA, 2008, pp. 222–232. [Online]. Available: http://portal.acm.org/citation.cfm?id=1344704

[4] J. Yu, C. Eagleston, C. H. Chou, M. Perreault, and G. Lemieux, "Vector processing as a soft processor accelerator," *ACM TRETS*, vol. 2, no. 2, pp. 1–34, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1534916.1534922

[5] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, "VEGAS: soft vector processor with scratchpad memory," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950420

[6] R. M. Russell, "The cray-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978. [Online]. Available: http://doi.acm.org/10.1145/359327.359336

[7] T. Zeiser, G. Hager, and G. Wellein, "The world's fastest cpu and smp node: Some performance results from the nec sx-9," in *Proceedings of the 2009*

IEEE International Symposium on Parallel&Distributed Processing, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161089

[8] C. E. Kozyrakis and D. A. Patterson, "Scalable vector processors for embedded systems," *IEEE Micro*, vol. 23, no. 6, pp. 36–45, Nov. 2003. [Online]. Available: http://dx.doi.org/10.1109/MM.2003.1261385

[9] "VESPA: portable, scalable, and flexible FPGA-based vector processors." [Online]. Available: http://portal.acm.org/citation.cfm?id=1450107

[10] P. Yiannacouras, J. G. Steffan, and J. Rose, "Data parallel FPGA workloads: Software versus hardware," in *FPL*, Progue, Czech Republic, 2009, pp. 51–58.

[11] S. Williams, "Preliminary analysis of vcc c/c++ compiler for viram."

[12] D. Judd, K. A. Yelick, C. E. Kozyrakis, D. Martin, and D. A. Patterson, "Exploiting on-chip memory bandwidth in the viram compiler," in *Revised Papers from the Second International Workshop on Intelligent Memory Systems*, ser. IMS '00. London, UK, UK: Springer-Verlag, 2001, pp. 122–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=648002.743085

[13] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's array building blocks: A retargetable, dynamic compiler and embedded language," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 224–235. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190069

[14] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," *SIGPLAN Not.*, vol. 45, no. 10, pp. 89–108, Oct. 2010. [Online]. Available: http://doi.acm.org/10.1145/1932682.1869469

[15] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for gpus: (via language support for architectures and compilers)," *SIGPLAN Not.*, vol. 47, no. 6, pp. 1–12, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345156.2254066

[16] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 179–196. [Online]. Available: http://dl.acm.org/citation.cfm?id=647478.727935

[17] T. Han and T. Abdelrahman, "hicuda: High-level gpgpu programming," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 78 –90, jan. 2011.

[18] D. Cunningham, R. Bordawekar, and V. Saraswat, "Gpu programming in a high level language: compiling x10 to cuda," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, ser. X10 '11. New York, NY, USA: ACM, 2011, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/2212736.2212744

[19] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," *SIGPLAN Not.*, vol. 46, no. 3, pp. 381–392, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1961296.1950409

[20] "Accelerator," http://research.microsoft.org/en-us/projects/accelerator.

[21] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data parallelism to program gpus for general-purpose uses," in *ASPLOS*, San Jose, California, USA, 2006, pp. 325–355. [Online]. Available: http://portal.acm.org/citation.cfm?id=1168898

[22] B. Bond, K. Hammil, L. Litchev, and S. Singh, "FPGA circuit synthesis of accelerator data-parallel programs," in *FCCM*, Charlotte, North Carolina, USA, 2010, pp. 167–170. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5474053

[23] Accelerator Team, "Microsoft accelerator target implementors' guide," 2010.

[24] r. sethi and j. d. ullman, "The generation of optimal code for arithmetic expressions," *j. acm*, vol. 17, no. 4, pp. 715–728, Oct. 1970. [Online]. Available: http://doi.acm.org/10.1145/321607.321620

[25] A. Appel and K. J. Supowit, "Generalizations of the sethi-ullman algorithm for register allocation," *Software – Practice and Experience*, vol. 17, pp. 417–421, 1987.

[26] L. Li, L. Gao, and J. Xue, "Memory coloring: A compiler approach for scratchpad memory management," in *PACT*, Sydney, Australia, 2005, pp. 329–338. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1515604

[27] B. Egger, J. Lee, and H. Shin, "Scratchpad memory management for portable systems with a memory management unit," in *PACT*, Seoul, Korea, 2006, pp. 321–330. [Online]. Available: http://portal.acm.org/citation.cfm?id=1176933

[28] Accelerator Team, "An introduction to microsoft accelerator v2," 2011.

[29] B. Bond, A. Davidson, L. Litchev, and S. Singh, "From SMPs to FPGAs: multi-target data-parallel programming," 2010. [Online]. Available: http://www.cs.bham.ac.uk/~drg/cuda/satnam.pdf