© 2006 by Gang Ren. All rights reserved.

COMPILING VECTOR PROGRAMS FOR SIMD DEVICES

BY

GANG REN

B.E., Zhejiang University, 1998
M.S., Chinese Academy of Sciences, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

As an effective way of utilizing data parallelism in applications, SIMD architecture has been adopted by most today's microprocessors. Using intrinsic functions and automatic compilation are common programming methods for today's SIMD devices. However, neither methods can provide enough programmability and performance at the same time. Many issues must be addressed to generate efficient SIMD code. For example, most SIMD devices only support memory accesses on contiguous and aligned sections. Additional permutation instructions are needed for non-contiguous and/or misaligned references. Such overhead can cancel all performance benefits from SIMD computation.

VINCI, or **V**ector **I**-code **N**ovel **C**ompilation **I**nfrastructure, is proposed in this thesis. VINCI focuses on translating vector programs into efficient code for SIMD devices. Vectors in input programs can have arbitrary length, strides, and alignment settings. However, vectors required by SIMD devices must have the same fixed length, unit strides, and aligned addresses. VINCI employs a sequence of program transformations to convert all vectors into such specific format.

VINCI also includes several optimization algorithms. The optimization algorithm on data permutations is of great importance. By unifying all forms of data permutations into the explicit representation, the optimization algorithm can reduce the number of data permutations in vector programs by propagating them across statements and merging them whenever possible. In addition, an efficient code generation algorithm is included to generate native permutation instructions from vector permutation operations.

Besides, any common compiler analysis and optimizations were also extended for vector

representation and included in VINCI. Two examples are def-use analysis and copy propagation. In addition, two domain-specific optimization techniques for DSP programs are also extended for vector programs. These optimizations are necessary to delivery the final performance on SIMD devices.

VINCI was implemented on the HiLO compiler, an internal compiler used in SPIRAL. Experiments were conducted on two platforms, VMX and SSE2. Testing applications include both automatically-generated programs and manually-written kernels. The results show that up to 77% of the permutation instructions are eliminated and, as a result, the average performance improvement is 48% on VMX and 68% on SSE2. For several applications, near perfect speedups have been achieved on both platforms.

To My Family

Acknowledgements

First, I would like to express my deep and sincere gratitude to my advisor, Professor David Padua, for guiding and supporting me throughout the whole Ph.D. study. His advices, insights, and encouragement helped me in all the time of this five-year research. I benefited tremendously from his wide knowledge and logical way of thinking. The completion of this thesis would not have been possible without his guidance.

I would like to sincerely thank Dr. Peng Wu for her detailed and constructive suggestions. From the beginning of my graduate study, I have learned a lot from all the interesting and insightful discussions with her. It was such a great experience working with her as an intern at IBM.

I owe my warm gratitude to the members of my thesis committee, Professor Vikram Adve, Professor Williams Harrison and Professor Wen-Mei Hwu. Their insightful advices are invaluable for my thesis research. I am indebted to them for reading my dissertation and providing penetrating feedback.

I am also indebted to Dr. Aart Bik and Dr. Chris Lattner for donating their time to read draft versions of the thesis and giving me helpful comments.

I would like to thank Professor Maria Garzaran for her comments on my early work on automatic performance tuning and for her selfless help on my thesis research.

I would like to extend my thanks to Professor Josep Torrellas for his interesting classes and for his extremely helpful recommendation letter.

I want to thank Dr. Alexandre Eichenberger and the other members of the Cell compiler group at IBM Research. It was a wonderful experience working with these smart and friendly

people.

My thank also goes to all current and previous members of Polaris group, especially Jianxin Xiong, Jiajing Zhu, Zehra Sura, David Wong, Xiaoming Li and Nick Rizzolo. I am also grateful to Sheila Clark for her great administrative support during my graduate study.

I owe my loving thanks to my parents and grandparents. Without their encouragement it would have been absolutely impossible for me to finish my thesis work. My special thank goes to my grandfather, Dingyuan Chen, for the 30-year endless love and support. Last but not least, my warmest thank is due to my wife, Shengnan, for making my life complete.

Table of Contents

List of Figures xi

Chapter 1 Introduction 1

Chapter 2 Overview of SIMD Devices 8
- 2.1 Architecture of SIMD Devices 8
- 2.2 Characteristics of Today's SIMD Devices 11
- 2.3 Permutation Instruction Set 13
- 2.4 Programming SIMD Devices 16

Chapter 3 Compiling for SIMD Devices 20
- 3.1 An Overview of SIMD Compilation 20
- 3.2 Experiments with SIMD Compilers 22
  - 3.2.1 Berkeley Multimedia Workload 22
  - 3.2.2 Experimental Results 23
- 3.3 VINCI: Vector I-code Novel Compilation Infrastructure 25

Chapter 4 Vector and Vector Operations 26
- 4.1 Vector Representation 26
  - 4.1.1 Vector and Vector Expression 26
  - 4.1.2 Vector Operations 29
- 4.2 Data Permutation on Vectors 30
  - 4.2.1 Permutation Operation 30
  - 4.2.2 Properties of the Permutation Operation 32
  - 4.2.3 Reduction and Replication 34
  - 4.2.4 Permutation Operations in Vector I-code Programs 35

Chapter 5 Normalizing Vector Programs 37
- 5.1 Normalizing Vectors with Non-unit Strides 37
  - 5.1.1 Strided load 38
  - 5.1.2 Strided store 39
- 5.2 Coalescing Partially Utilized Permutations 40
- 5.3 Normalizing Vectors cross Loop Boundaries 41
- 5.4 Normalizing Vectors with Misalignment 42
- 5.5 Other Sources of Data Permutations 43

Chapter 6 Loop Unrolling and Vector Coalescing . . . 45
6.1 Array Expansion . . . 46
6.2 Coalescing Vector Statements . . . 50
6.3 Unrolling Loops with Data Permutations . . . 51

Chapter 7 Data Permutation Optimization . . . 53
7.1 An Overview of the Optimization Algorithm . . . 53
7.2 Propagation along Def-use Chains . . . 55
7.2.1 Reshaping Permutations . . . 57
7.2.2 Permutation Decomposition . . . 58
7.3 Propagation within a Statement . . . 63
7.4 Optimality Analysis . . . 65
7.4.1 NP-Completeness of the Problem . . . 65
7.4.2 Heuristic Solutions . . . 69
7.5 The Global Optimization Algorithm . . . 70
7.5.1 Extension on If-Statements . . . 70
7.5.2 Extension on Loops without Backward Dependences . . . 72
7.5.3 Extension on Loops with Backward Dependences . . . 74
7.6 Optimizing Special Permutations . . . 74
7.7 Related Work on Data Permutation Optimization . . . 75

Chapter 8 SIMD Code Generation . . . 77
8.1 The Code Generation Algorithm . . . 77
8.2 Generating SIMD Permutation Instructions . . . 78
8.2.1 Translating *Permute* into *vperm* Instructions . . . 79
8.2.2 Mapping *vperm* to Native Instructions . . . 83
8.2.3 Generating Reduction and Replication . . . 83

Chapter 9 Mixed-Mode SIMD Compilation . . . 85
9.1 A Motivating Example . . . 85
9.2 Mixed-Mode Extension on VINCI . . . 86
9.3 Discussion . . . 88

Chapter 10 Other Compiler Routines in VINCI . . . 90
10.1 Def-use Analysis on Vector Program . . . 90
10.2 Vector Copy Propagation . . . 91

Chapter 11 Implementation on the HiLO Compiler . . . 94
11.1 The SPL Compiler . . . 94
11.2 The I-Code Language . . . 96
11.3 The HiLO Compiler . . . 96
11.4 Implementation on HiLO . . . 97

Chapter 12 Domain Specific Optimization for DSP Programs . . . 100
12.1 Simplification of Arithmetic Operations . . . 100
12.2 Instruction Clustering . . . 101

Chapter 13 Experimental Results . . . 104
13.1 The SPIRAL System . . . 104
13.2 Experiment Setups . . . 105
13.3 Static Evaluation . . . 108
13.4 Runtime Performance Evaluation . . . 111
13.4.1 Evaluating the Optimization Algorithm . . . 111
13.4.2 Evaluating the Domain-specific Optimizations . . . 115
13.4.3 Evaluating the Code Generation Algorithm . . . 116
13.4.4 Overall Run-time Performance . . . 117

Chapter 14 Other Issues in SIMD Compilation . . . 119
14.1 Programming Styles of Multimedia Applications . . . 119
14.1.1 Pervasive Use of Pointers and Pointer Arithmetics . . . 119
14.1.2 User-Conducted Optimizations . . . 120
14.2 Vectorizing Outer Loops . . . 121
14.3 Mismatches Between Application and Language . . . 122
14.3.1 Subword Optimizations . . . 122
14.3.2 Identifying Saturated Operations . . . 124
14.4 Experimental Results of Manual Vectorization . . . 125
14.4.1 Core Procedures Vectorized . . . 125
14.4.2 Non-Vectorizable Core Procedures . . . 129

Chapter 15 Conclusion . . . 131
15.1 Future Work . . . 132

References . . . 134

Author's Biography . . . 140

List of Figures

1.1 8-point FFT codes with stride-2 accesses. 3
1.2 A naïve SIMD implementation of the FFT code in Figure 1.3. 4
1.3 An optimized SIMD implementation of the FFT code in Figure 1.3. 5
1.4 VINCI: A unified compilation framework for SIMD devices. 6

2.1 An overview of SIMD architecture. 8
2.2 $R_3 \leftarrow \text{vperm}(R_1, R_2, R_{pattern})$ 14
2.3 $R_3 \leftarrow \text{shufps}(R_1, R_2, I_{pattern})$ 15

3.1 Major components of VINCI. 25

5.1 An example of converting strided loads. 39

7.1 The algorithm of optimizing data permutations in a basic block. 54
7.2 The algorithm of reshaping permutations. 59
7.3 The algorithm of register-wise permutation decomposition. 62
7.4 The algorithm of platform-dependent permutation decomposition. 63
7.5 Three NP-Complete problems. 66
7.6 Normalized version of the 8-point FFT code in Figure 1.1. 72

8.1 Different code generation patterns for `v[0:3]` in (2). 80
8.2 The algorithm of translating *Permute* to *vperm* operations. 82

9.1 A simplified loop from GSM. 86
9.2 Vectorized loop from Figure 9.1. 88
9.3 Mixed-mode vectorization of `Recur(d[0], vrp*vu,+)`. 88
9.4 Vectorization of `Recur(d[0], vrp*vu,+)` using parallel prefix. 89

10.1 Vectorized loop from Figure 9.1. 92
10.2 After vector copy propagation. 93

11.1 Implementation on the HiLO Compiler 99

13.1 An overview of the SPIRAL system. 105
13.2 Performance of Group I programs. 113
13.3 Performance of 64-point FFT programs on SSE2. 114
13.4 Performance evaluation of domain-specific optimizations. 115

13.5 Performance of matrix transpose and bit-reversal ordering. 116
13.6 Performance of all applications on VMX and SSE2. 118

14.1 User-optimized code from LAME. 120
14.2 Another simplified loop from GSM. 121
14.3 Saturated add in GSM. 124
14.4 Saturated Add in MPEG2. 125
14.5 Dependence graph of ADPCM encoder. 130

Chapter 1

Introduction

Single-**I**nstruction-**M**ultiple-**D**ata (SIMD) devices are present in most today's microprocessors. In general-purpose processors, SIMD devices are typically called *multimedia extensions* [1]. Examples of multimedia extensions include MAX for PA-RISC [44], VIS for SPARC [36], the SSE family for Pentium [30], 3DNow! for Althon [2] and VMX/AltiVec for PowerPC [22]. Similar SIMD units can also be found in special-purpose processors for media processing such as game processors [34, 39], graphic processing units (GPUs) [48], and DSP (Digital Signal Processing) processors [24, 66].

To exploit the computing power provided by SIMD devices, various programming methods have been developed. Today, using intrinsic functions is one of the most common programming methods for SIMD devices. Although intrinsic functions in C programs can be used to program SIMD devices, this is a low-level programming method similar to assembly language. In fact, there is a one-to-one mapping of most intrinsic functions into hardware instructions. Hence, it is usually difficult to write, debug, and maintain SIMD programs based on intrinsic functions. In addition, intrinsic functions are oftentimes not portable between different devices.

As SIMD devices become more powerful and popular, more efficient programming methods are needed to access those devices. *SIMD compilation*, which relies on compilers to translate standard C programs into SIMD instructions, has attracted much attention in recent years [7, 5, 10, 13, 37, 41, 45, 53, 55, 64, 69]. However, as it has been learned from

[1] Depending on the context, multimedia extensions may refer to SIMD ISAs or SIMD units of general-purpose microprocessors.

the experiments on multimedia applications, which is discussed later in this thesis, SIMD compilation fails to generate efficient SIMD code in a number of important cases. There are many issues arising in generating efficient code for SIMD devices and they must be addressed to achieve a performance comparable with that obtained by using intrinsic functions [57].

The process of SIMD compilation can be roughly divided into two phases. First, the compiler needs to extract data parallelism from sequential programs. *Vectorization*, which is based on data dependence analysis, is one of most effective compiler techniques to explore data parallelism from loop structures. In the past, vectorizing compilers were able to generate efficient code for conventional vector processors [25].

Second, the compiler must translate vectorized programs into SIMD instructions. Current SIMD devices have idiosyncratic ISAs which differentiate them from conventional vector processors and complicate their programming. It is not a trivial task to translate vector programs into efficient code for SIMD devices. For example, most devices only support references to contiguous and aligned memory addresses. Additional permutation instructions are needed to access non-contiguous or misaligned memory sections. The overhead of those instructions might cancel the performance benefits from SIMD computation. It is therefore of great importance to minimize the number of permutation instructions when generating code for SIMD devices.

In this thesis, a unified compilation framework, named as VINCI (**V**ector **I**-code **N**ovel **C**ompilation **I**nfrastructure), is proposed to address the issues arising in the generation of efficient code for SIMD devices. Instead of starting from raw sequential programs, the framework focuses on the second phase. The input is a generic vector program, where data parallelism is explicitly expressed and the output code contains intrinsic functions for SIMD devices. VINCI consists of a set of program transformation and optimization techniques, which were developed to translate vector programs into efficient SIMD instructions.

The input vector program can come from programmers directly by using the vector-extended I-code language. Generic vector representation provides an explicit, compact, and

```
1.  t0[0:6:2] = x[0:3:1] + x[4:7:1];
2.  t0[1:7:2] = x[0:3:1] - x[4:7:1];
3.  t1[0:7:1] = T8[0:7:1] * t0[0:7:1];
4.  for (i = 0; i < 2; i++) {
5.    t2[0:2:2] = t1[i:i+2:2] + t1[i+4:i+6:2];
6.    t2[1:3:2] = t1[i:i+2:2] - t1[i+4:i+6:2];
7.    t3[0:3:1] = T4[0:3:1] * t2[0:3:1];
8.    y[i+0:i+2:2] = t3[0:1:1] + t3[2:3:1];
9.    y[i+4:i+6:2] = t3[0:1:1] - t3[2:3:1];
10. }
```

Figure 1.1: 8-point FFT codes with stride-2 accesses.
(Note: `t0`, `t1`, `t2`, `t3` are temporary arrays and `T4` and `T8` are constant arrays.)

beautiful way of expressing data parallelism in programs. Most vector I-code programs used in the experiment discussed later in the thesis are generated by library generators.

The input can also result from other program transformations, such as vectorization. VINCI can be incorporated in conventional vectorizing compilers to translate sequential programs into efficient SIMD instructions.

In vector programs, vectors can have arbitrary length, non-unit strides, and alignment settings[2]. Figure 1.1 shows such a generic vector program, where vector variables resemble those of Fortran 90 [1]. For example, `a[0:6:2]` represents the 4-element vector $\langle a[0], a[2], a[4], a[6] \rangle$. On the other hand, SIMD devices require all vectors to have the same short, fixed length, unit stride, and their addresses to be aligned with natural boundary of vectors, typically 8- or 16-byte. Thus, a sequence of transformation algorithms were developed during the course of this thesis to convert generic vectors into such format.

First, generic vectors are normalized on stride and alignment. All vectors with non-unit strides are converted into ones with unit strides. And all misaligned vectors are replaced with aligned ones. During normalization, explicit data permutation operations must be inserted into the vector program to preserve the semantics of original vector programs. In

[2]In the implementation used for the experiments described below, length and stride must be compile-time constants. But such constraints can be partially relaxed by strip-ming variable-length vectors into a variable-length loop with constant-length vectors.

```
1.  v1[0:3:1] = x[0:3:1] + x[4:7:1];
2.  v1[4:7:1] = x[0:3:1] - x[4:7:1];
3.  t0[0:7:1] = Permute(v1[0:7:1], P1);
4.  t1[0:7:1] = T8[0:7:1] * t0[0:7:1];
5.  v2[0:7:1] = Permute(t1[0:7:1], P2);
6.  u1[0:7:1] = Permute(v2[0:7:1], P3);
7.  u2[0:3:1] = u1[0:3:1] + u1[4:7:1];
8.  u2[4:7:1] = u1[0:3:1] - u1[4:7:1];
9.  v3[0:7:1] = Permute(u2[0:7:1], P4);
10. t2[0:7:1] = Permute(v3[0:7:1], P5);
11. t3[0:7:1] = T4_2[0:7:1] * t2[0:7:1];
12. u3[0:7:1] = Permute(t3[0:7:1], P6);
13. u4[0:3:1] = u3[0:3:1] + u3[4:7:1];
14. u4[4:7:1] = u3[0:3:1] - u3[4:7:1];
15. v4[0:7:1] = Permute(u4[0:7:1], P7);
16. y[0:7:1] = Permute(v4[0:7:1], P8);
```

Figure 1.2: A naïve SIMD implementation of the FFT code in Figure 1.3. (Note: After converting the stride-2 vectors to unit strides and unrolling the loop, temporary arrays `v1`, `v2`, `v3`, `v4` are introduced to convert stride-2 vectors and `u1`, `u2`, `u3`, `u4` are from loop unrolling. `P1` to `P8` specify the permutation patterns in those operations. `T4_2` is resulted from concatenating two `T4` arrays.)

other words, the normalization algorithm translates all implicit data permutations implied by non-unit strides and misaligned addresses into explicit permutations.

Second, inner loops are unrolled and vector statements from different iterations are coalesced together. Unrolling loop to produce long vectors is useful because long vectors can utilize the computing bandwidth of SIMD devices more efficiently than short ones. Besides, loop unrolling also helps performance by improving ILP (Instruction Level Parallelism) and reducing loop overhead.

After these two transformations, the vector program in Figure 1.1 is translated into the version shown in Figure 1.2. In Figure 1.2, data permutation is represented by a generic operation, $Permute(v, P)$, which reorders vector v according to the index vector P (*i.e.*, $v(P(:))$ in Fortran 90 notation).

Finally, after conducting various compiler optimizations on normalized vector programs, vector statements are translated into SIMD instructions (represented by intrinsic functions).

```
1.  v1[0:3:1] = x[0:3:1] + x[4:7:1];
2.  v1[4:7:1] = x[0:3:1] - x[4:7:1];
3.  t1[0:7:1] = T8[0:7:1] * v1[0:7:1];
4.  u1[0:7:1] = Permute(t1[0:7:1], Q1);
5.  u2[0:3:1] = u1[0:3:1] + u1[4:7:1];
6.  u2[4:7:1] = u1[0:3:1] - u1[4:7:1];
7.  t3[0:7:1] = T4_2[0:7:1] * u2[0:7:1];
8.  u3[0:7:1] = Permute(t3[0:7:1], Q2);
9.  y[0:3:1] = u3[0:3:1] + u3[4:7:1];
10. y[4:7:1] = u3[0:3:1] - u3[4:7:1];
```

Figure 1.3: An optimized SIMD implementation of the FFT code in Figure 1.3.

During code generation, long vectors are strip-mined into short ones with the same fixed length and then those short vectors are replaced by virtual SIMD register variables. Accordingly, most element-wise vector operations are translated into corresponding intrinsic functions supported by SIMD devices. An interesting problem is now to generate the minimum number of permutation instructions for a given *Permute* operation. An efficient algorithm was developed and implemented for the experiments reported below.

Besides these program transformations, a few other compiler optimization techniques are also included the framework. These optimizations are necessary to delivery maximum performance on SIMD devices. For example, in SIMD compilation, the overhead of permutation instructions is one of the biggest hurdles for achieving speedups over scalar code. Hence, an optimization algorithm is developed to minimize permutation instructions in vector programs. Representing all forms of data permutations in the same way during normalization facilitate the task of the optimization algorithm that propagates the permutation across statements and merges them together whenever possible. Figure 1.3 shows the optimized version of the vector program in Figure 1.2. As shown in the figures, the number of data permutations is reduced from 8 to 2 after conducting the optimization algorithm.

In addition to the newly designed algorithms, many common compiler analysis and optimization routines were also extended for vector programs. Most extensions are not trivial. For example, the def-use analysis on scalars only needs to handle two relationships, *same*

or *different*, between two variables, however, the def-use analysis on vectors needs to handle the third relationship, *overlap*, between two vector variables.

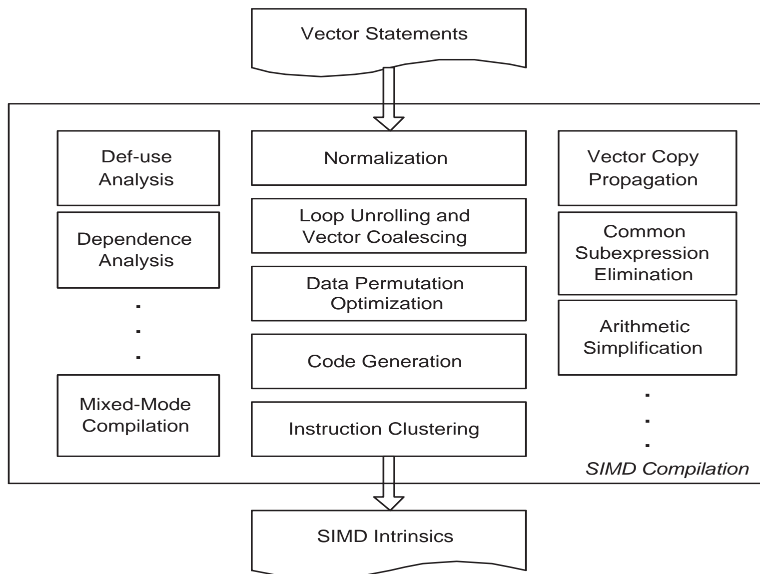

Figure 1.4: VINCI: A unified compilation framework for SIMD devices.

The framework was implemented on the HiLO compiler, a source-to-source compiler for automatic library generators [59]. Currently, it is used as the internal compiler of SPIRAL, an automatic DSP library generator [56]. When generating optimal DSP programs, several domain-specific optimizations are necessary to obtain the best performance [20]. These optimization were also extended for vector programs and included in the framework.

To evaluate the effectiveness of the proposed compilation strategy, experiments were conducted on two platforms, VMX and SSE2. Besides DSP programs generated by SPIRAL, a few other applications from other domains were tested. The experimental result indicates that the framework is able to generate efficient SIMD code on both platforms and achieves near-peak speedups over the highly optimized scalar code. Besides, the optimization algorithm of data permutations significantly reduces data permutations in vector programs.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of SIMD devices, including their architectures and programming paradigms. Chapter 3 describes SIMD compilation in detail and gives an overview of VINCI. After several important terms

are defined in Chapter 4, the major components of VINCI are discussed in Chapter 5-10.

Chapter 5 describes the normalization algorithm to convert all vectors in a vector program into stride-one and aligned ones and insert data permutations into the program. Chapter 6 introduces the algorithm to unroll small inner loops and coalesce vector statements from different iterations to generate long vectors. Chapter 7 illustrates the optimization algorithm on data permutations. Several related issues are also discussed in Chapter 7. Chapter 8 describes the algorithm to generate SIMD instructions. An algorithm of translating general permutations into native permutation instructions is also introduced in Chapter 8. Chapter 9 discusses the mixed-mode compilation strategy, which provides an efficient way of compiling partially vectorizable loops for SIMD devices. Chapter 10 shows how to extend common compiler routines, such as def-use analysis and copy propagation, on vector I-code programs.

The HiLO-based implementation is described in Chapter 11. Two domain-specific optimization techniques for DSP programs and their extensions for SIMD devices are introduced in Chapter 12. Chapter 13 summarizes the results of the experiment conducted on VMX and SSE2. Chapter 14 briefly describes the other issues in SIMD compilation that are not addressed by VINCI yet. Finally, Chapter 15 concludes the thesis and discusses the future work.

Chapter 2

Overview of SIMD Devices

This chapter gives an overview of SIMD devices. Section 2.1 describes the SIMD architecture and introduces typical SIMD devices. Then Section 2.2 discusses the specific architectural features of current SIMD devices. In Section 2.3, a special type of SIMD instructions, permutation instructions, are described in details. Finally, Section 2.4 summarizes several typical programming methods used for SIMD devices.

2.1 Architecture of SIMD Devices

Figure 2.1 illustrates an abstract architecture of SIMD devices. As shown in the figure, each register can hold multiple data elements and each instruction can conducts the same operation on all elements at the same time. That is what SIMD (Single Instruction, Multiple Data) stands for. In general, SIMD architecture provides an effective way of efficiently utilizing data parallelism.

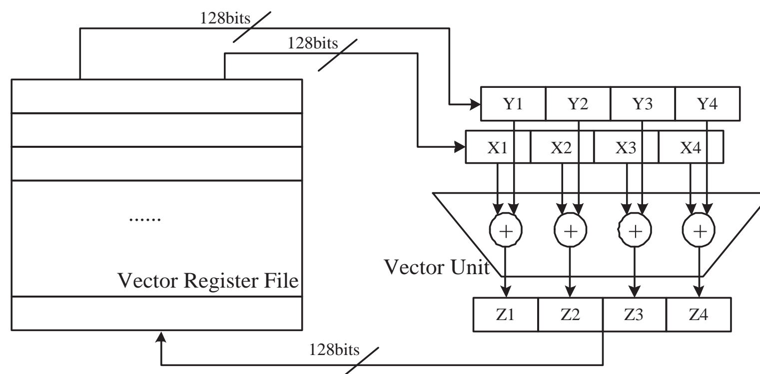

Figure 2.1: An overview of SIMD architecture.

The history of SIMD devices can be tracked back to the 1970s. A typical example are the Cray vector machines [60]. Although vector operations were typically executed in a pipeline fashion, vector machines can be considered as SIMD devices from the prospective of the instruction set architecture (ISA). Connection Machine 1(2) from Thinking Machines is another early example of SIMD machines [67]. At that time, SIMD architecture was mainly used in supercomputers.

In 1994, HP introduced SIMD architecture for general-purpose microprocessors. By packing several subword integers, such as 4 8-bit chars or 2 16-bit shorts, together to a 32-bit data, a set of SIMD instructions, called *MAX*, was added to PA-RISC's ISA to manipulate the packed subword data [44]. SIMD architectures provide an effective way to fully utilize the 32-bit bandwidth of the processor on subword integers. It improves the performance of multimedia applications in which operations on subword integers are very common.

Since 1994, SIMD units, oftentimes called *multimedia extensions*, have become standard architectural features for most general-purpose processors. During ten-year evolution, multimedia extensions have become more and more powerful. Two of most recent multimedia extensions are Intel's SSE3 [30][1] and IBM's VMX [22]. Compared with MAX, both SSE3 and VMX have larger bandwidth (128 bits) and support more data types and operations (see Table 2.1).

Not surprisingly, SIMD architecture has also been adopted for special-purpose processors, especially in the domain of media processing. The microprocessors used in video game consoles are a typical example. Two vector units are included in the Emotion Engine used in Sony's Play Station 2 [39] and a total of 9 SIMD units, including 8 SPUs and 1 VMX, can be found in the Cell Broadband Engine used in Play Station 3 [34]. These SIMD units share many architectural similarities with multimedia extensions.

DSP processors are processors specially designed for digital signal processing. One of the

[1] Intel has added a set of new SIMD instructions in its new processors, named as SSE4 or MNI (Merom New Instructions). But the details of those instructions have not been released.

Processor/SIMD	Vendor	Year	Width	Data Type	Insts	Regs
Cray-1	Cray	1976	64x64	Ints, FPs	128	8
CM-1	Thinking Mach.	1985	1x64K	Bit	N/A	8
MAX (PA-RISC)	HP	1994	32	Ints	9	32
VIS (SPARC v9)	Sun	1995	64	Ints, FPs	121	32
MMX (Pentium)	Intel	1997	128	Ints, Single	57	8
SSE3 (Pentium 4)	Intel	2004	128	Ints, FPs	157	8
VMX (PowerPC G5)	IBM/Motorola	1998	128	Ints, Single	162	32
VU (Emotion Engine)	Sony	1999	128	Single	164	32
SPU (Cell B. Engine)	STI	2005	128	Ints, FPs	194	128

Table 2.1: Several examples of SIMD devices.

most important uses of DSP processors is media processing. Recently, DSP processors also began to support SIMD instructions that operate on packed subword data [24, 66]. However, SIMD units in DSP processors are more like earlier multimedia extensions, such as MAX, than current ones. In addition, DSP processors typically combine SIMD and VLIW (Very Long Instruction Word) features.

Originally, graphics is one major application of multimedia extensions. However, in most computer systems today, a large fraction of graphics computation is offloaded from main processors to graphics processing units (GPU). To exploit data parallelism in graphics applications, SIMD architecture is also used in GPU [48]. The processing elements, such as vertex processing element and texture processing element, are duplicated and assigned to different data portions. Although lack of general-purpose programmability, SIMD architecture in GPUs has a larger scale than multimedia extensions in general-purpose processors [48].

Interestingly, SIMD architecture reappears in supercomputers recently, after vector machines were outpaced by multiprocessor systems built on fast-evolving microprocessors in early 90s. However, SIMD architecture is small-scale and used as extensions this time. SIMD units that are similar to multimedia extensions can be found in Itanium and BlueGene/L processors [61, 4].

For the rest of this thesis, *SIMD devices* mainly refer to today's SIMD devices and *multimedia extensions* only refer to SIMD devices in general-purpose microprocessors.

	Vector Processor	SIMD Devices
Example	Cray-1	VMX
Vector Length (K)	>64	2-16
Vector Bit Width	64x64	64, 128
Parallel Execution	Yes*	Yes
Conditional Execution	Yes	No
Chaining Execution	Yes	No
Nonunit Strides	Yes	No
Indexed References	Yes	No
Cache Support	Yes*	Yes
Cache Passing Support	Yes*	Yes
Subword Operations	Yes*	Yes
Saturated Operations	No	Yes
Uniform Operation Support	Yes	No
Application Domain	Scientific	Multimedia

Table 2.2: Comparing conventional vector processors and today's SIMD devices. (* These features were not supported initially.)

2.2 Characteristics of Today's SIMD Devices

In spite of sharing the same architecture, there are several major differences between conventional vector processors and today's SIMD devices. These differences are mainly because of different targeting application domains and cost constraints. Table 2.2 compares some major characteristics of conventional vector processors, like Cray-1, and today's SIMD devices, like VMX.

First, the scale of current devices is smaller than that of vector processors. Most vector processors have more than 64 (up to 1024) elements per vector register [25]. However, most SIMD devices have a fixed bit width, which is usually 128. Hence, the degree of parallelism of SIMD devices varies from 16 (for 8-bit chars) to 2 (for 64-bit double floating points) for different data types.

In this thesis, N is used to represent the bit width of a SIMD device and K_{type} is used to represent the degree of parallelism for a specific data type, *type*, on this device. Thus, $K_{single} = N/32$, $K_{double} = N/64$, $K_{int} = N/32$, $K_{short} = N/16$, and $K_{char} = N/8$. For 32-bit single floating points, a 128-bit device is a 4-way SIMD unit. In other words, if N is 128,

K_{single} will be 4. Similarly, K_{char}, K_{short}, K_{int}, and K_{double} are 16, 8, 4, and 2 respectively.

Besides the degree of parallelism, the theoretical maximum speedup of SIMD computations also depends on the host processor. For example, since the PowerPC G5 has two scalar FP units, the maximum speedup when executing FP computations on VMX is only 2 [27]. Another illustration comes from the P4 processor. The integer ALU in early P4 processors runs at 2x clock rate while SSE2 unit can only process 64-bit packed integer operations (i.e., half of the vector length) per cycle. Thus, the maximum speedup is only 1 for full-size integers, although the 128-bit SSE2 is a 4-way SIMD device for 32-bit integers.

Since ideal speedups are relative small, any additional overhead in SIMD execution might consume a large fraction of performance benefits and eventually make it unprofitable to execute on SIMD units. That is one of reasons why low-level programming methods is still commonly used for SIMD devices.

On the other hand, with using longer vectors, conventional vector processors was able to achieve much higher performance than scalar processors. In fact, vector processors were the fastest processors in the world for many years [25]. Some advanced techniques, such as chaining and conditional execution, were developed to delivery maximum performance on conventional vector processors [25]. However, neither of these techniques have been used in today's SIMD devices.

Another major difference is that the memory units in SIMD devices are more constrained than the those of vector processors. Unlike vector processors which support gathering or scattering memory operations, most of today's SIMD devices can only load/store data from/to contiguous memory locations. In addition, most devices require memory addresses to be aligned at typically 128-bit boundaries. For some devices, such as VMX, alignment must be enforced to ensure correctness. For others, such as SSE2, alignment is preferred to avoid the performance penalty of unaligned load/store instructions.

To overcome the limits on the memory units, SIMD devices provides instructions to reorganize data within registers. The details of the permutation instructions are introduced

in Section 2.3. By loading more data into registers and using permutation instructions to reorganize them within registers, misaligned and/or non-contiguous memory references can be handled with unignorable overhead. In fact, since most data permutations cost nothing in scalar code (after copy propagation), these permutation instructions are pure overhead introduced in SIMD code. Hence, reducing the number of permutation instructions is an important issue in generating efficient code for SIMD devices.

Finally, the ISA (Instruction Set Architecture) of current SIMD devices is less regular. On one hand, some common operations in media processing, such as saturated arithmetics, are natively supported by many devices. With the help of special hardware instructions, these operations can be more efficiently executed on SIMD devices and sometimes result in super-linear speedups over scalar execution. On the other hand, many operations, including those special operations, are oftentimes supported for only a few data types. For example, SSE2 includes multiplication instructions for 16-bit shorts but none for 8-bit chars or 32-bit integers [30].

2.3 Permutation Instruction Set

To help software overcome the constraints on memory units, the hardware provides basic support for reorganizing data in SIMD registers with various data permutation instructions. Most SIMD units provide variations of the general permutation instructions as well as customized permute instructions with built-in permutation patterns. It is important to understand these variations for efficient generation of permutation instructions. This section surveys the data-movement instructions supported by VMX and the SSE family [29, 49].

First, VMX supports a general-purpose permutation instruction [49],

$$R_3 \leftarrow \text{vperm}(R_1, R_2, R_{pattern}).$$

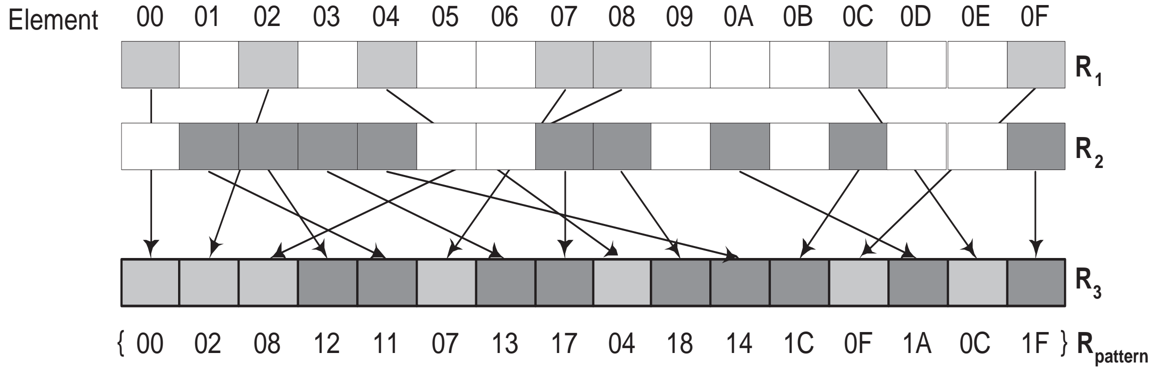

Figure 2.2: $R_3 \leftarrow \text{vperm}(R_1, R_2, R_{pattern})$

As shown in Figure 2.2, the vperm instruction selects an arbitrary set of 16 bytes from the two input registers, R_1 and R_2 (each 16-byte long), according to the permutation pattern specified in register $R_{pattern}$.

The SSE family (SSE/SSE2/SSE3) supports a more restricted form of the general permute,

$$R_3 \leftarrow \text{shufps}(R_1, R_2, I_{pattern}).$$

where the permutation is not done at byte level but at the 4-byte element (single floating point) level. In addition, elements from R_1 (R_2) can only go to the low-half (high-half) of the output register R_3. In `shufps`, the permutation pattern is specified by an intermediate constant, instead of another register like `vperm`. Besides, the SSE family provides instructions, `pshuhw` and `pshulw`, to shuffle 2-byte elements in the low-half and high-half of a register respectively. In the next version of SSE, SSE4 or MNI (Merom New Instructions), two more instructions, `pshufb` and `palignr`, will be added to provide more general support for data permutations.

Both VMX and the SSE family support customized permutation instructions where the permutation pattern is built into the instruction. Such permutation instructions include:

- *interleave* which interleaves data elements from the low-halves or high-halves of two inputs registers. It can be used to scatter stride-2 accesses. In fact, a special version

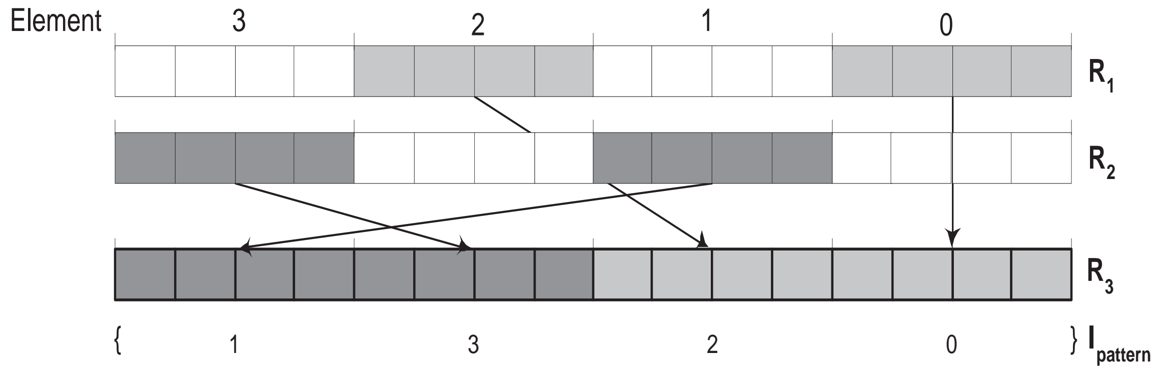

Figure 2.3: $R_3 \leftarrow \text{shufps}(R_1, R_2, I_{pattern})$

of interleave is *unpack*, which was originally designed for type conversion (e.g., from `short` to `int`) with possible sign- or zero-extensions [2].

- *pack* which packs two vectors into one. It is originally provided for type conversion (e.g., from `int` to `short`), but its unsaturated version (truncation only) can be used to gather stride-2 accesses.

- *shift* that rotates across elements within a register. In addition, VMX provides an instruction `vsldoi` to shift elements between two registers.

- *select* which selects data elements from either input registers and places it to the same location in the output register.

- *partial move* which inserts or extracts some elements from the input into the output.

- *splat* which replicates an element (usually the first one) in the input to create the full output register.

- *reduction* which conducts reductions on elements within a register. Only some special forms of reductions are supported by VMX and the SSE family

[2]Sign-extension may require additional instructions to generate sign mask for interleave.

Note that all customized permutation instructions can be implemented by general permutation instructions. However, since they have permutation patterns built into the instructions, they require fewer registers and/or no computation of the permutation mask.

Table 2.3 summarizes the data-movement instructions supported by VMX and the SSE family.

Operations	VMX	SSE Family
Permute	vperm	pshud, pshuhw*, shufps*
Interleave	vmrglw*	unpckhps*, punpckhdq*
Pack (Unsat.)	vpkuhum*	
Shift	vsldoi, vsl, vslo*	pslldq
Select	vsel	maskmovdqu
Partial Move		movhps*, movhlps, movss*, movq*
Splat	vspltw*, vspltisw*	movddup, movshdup*
Reduction	vmsumsbm*, vsum4shs*, vsumsws*	psadbw, pmaddwd, haddps*

Table 2.3: Data reorganization instructions supported VMX and the SSE family. (Instructions with * in the table represent a group of similar instructions (with different data types or patterns).)

Besides these permutation instructions introduced above, there is another kind of SIMD instructions that combine arithmetic (or logical, bit-wise) operations and data permutations together. Such kind of instructions are more common in DSP and BlueGene/L processors which do not provide any permutation instructions [4].

2.4 Programming SIMD Devices

To exploit the computing power of SIMD devices, various programming methods have been developed. In-line assembly, intrinsic functions, native libraries, language extensions, and automatic compilation are typical methods used by C programmers.

Supported by all SIMD devices, assembly language is the programming method at the lowest level. It provides a very basic way for programmers to access SIMD devices. It is well-

known that assembly language is difficult to read, write, debug, and maintain. Assembly programs are not portable between different platforms.

To improve the programmability, C intrinsic functions were introduced and supported by many native compilers for SIMD devices. By using intrinsic functions, programmers are released from some low-level optimizations, such as register allocation and instruction scheduling, on which it is believed that the compiler is able to perform fairly well.

Most intrinsic functions are mapped onto native SIMD instructions in a one-to-one fashion. A few other functions are included for common C language features, such as constant variables and variable initialization. In general, comparing with assembly language, intrinsic functions are easier to read, write, debug, and maintain.

On the other hand, using intrinsic functions is still a low-level programming method. Expert knowledge of SIMD devices is required to use intrinsic functions, especially because many operations are not uniformly supported by all devices. In addition, intrinsic functions are oftentimes not portable between different SIMD devices and sometimes even not portable between different compilers for the same device. Despite these limitations, using intrinsic functions is still one of the most common programming methods today, since it enables programmers to obtain maximum performance on SIMD devices.

Besides intrinsic functions, there are a few other ways to extend C language for SIMD devices. To address the portability problem of intrinsic functions, *abstract* intrinsic functions were introduced for common operations among SIMD devices. Abstraction functions can be simply defined as macro-functions and be mapped into native intrinsic functions by preprocessors [18]. More complicated abstractions will rely on compilers to translate abstract functions into native intrinsic functions (or assembly instructions) for the target device [65, 16].

Nonetheless, such abstraction can only include common operations supported by most devices. Thus, its application is very limited. Besides, while the abstraction improves the portability between different platforms, it decreases the portability between different

compilers when the abstraction is not standardized.

An alternative extension is to introduce generic vector representation into sequential languages [11]. For example, in Fortran 90, data parallelism can be represented explicitly by vectors. Hence, the task of finding data parallelism in sequential programs is offloaded from compilers to programmers. Nevertheless, it is still a challenging task for compilers to translate generic vector programs into efficient code for SIMD devices today. As discussed in the thesis, there are many newly-arising issues that must be addressed in order to generate code that can compete with either scalar codes or manually written code based on intrinsic functions. The framework proposed in this thesis belongs to this category.

Compared with explicit SIMD programming by using intrinsic function, automatic compilation is obviously a better solution. If compiler transformations could automatically translate sequential programs into SIMD instructions, the computing power of SIMD devices would be available for most standard C programs. On the other hand, automatic compilation would not be widely accepted unless its performance is comparable to that of the other programming methods. More details about SIMD compilation are discussed in Chapter 3.

Besides the general programming methods discussed above, there are a few less flexible ways of exploiting SIMD devices. For example, native libraries are provided by many microprocessor vendors, such as Intel's MKL and IPP libraries [28, 31]. The libraries include important kernels in various application domains and have been manually optimized for SIMD devices. It is a common practice for programmers to call native libraries instead of writing and optimizing kernels by themselves.

Method	General Purpose	Programm-ability	Compiler Compatibility	Architecture Compatibility	Perfor-mance
Assembly	Yes	Lowest	No	No	High
Intrinsic	Yes	Low	Partial	No	High
Ab. Intrinsic	Yes	Medium	No	Yes	High
Library	No	Medium	Yes	Partial	High
V. Compilation	Yes	High	Yes	Yes	Medium
S. Compilation	Yes	High	Yes	Yes	Low

Table 2.4: Comparison of common programming methods for SIMD devices. (Abbr.: Ab. Instrinsic - Abstract Instrinsic; V. Compilation - Compilation of Vector Programs; S. Compilation - Compilation of Sequential Programs;)

Chapter 3

Compiling for SIMD Devices

Automatic compilation for SIMD devices, or *SIMD compilation*, is described detailedly in this chapter. Section 3.1 describes the general SIMD compilation process by dividing it into two steps. Some related work and current status of SIMD compilation is also summarized in Section 3.1. Some experiments were conducted to evaluate the effectiveness of SIMD compilers. The results are presented in Section 3.2. Finally, Section 3.3 gives an overview of the compilation framework proposed in the thesis.

3.1 An Overview of SIMD Compilation

In general, the process of SIMD compilation can be divided into two phases. In the first phase, data parallelism is extracted from sequential programs. Built on top of data dependence graphs, vectorization is a compiler technique that extracts data parallelism from loop structures [3, 35, 72]. With the help of other compiler techniques, such as loop transformations, vectorization was very successful in generating efficient code for traditional vector processors.

Sometimes data parallelism can be found inside basic blocks, especially when the basic blocks are actually the loops that have been unrolled by programmers. Thus, a specific technique, called *loop rerolling*, was developed to translate the unrolled loops back to loop structures to apply vectorization to them. A more general technique packs scalar statements into vector statements [41, 45]. After data parallelism is extracted by vectorization and instruction packing, sequential programs can be translated into vector programs.

In the second phase, vector programs are translated into SIMD instruction sequences unless they have already been organized in this form by the vectorization pass. Also, vector variables are transformed into variables with the same fixed length, unit stride, and aligned addresses unless they are already in this form.

However, such translation does not guarantee the efficiency of SIMD code. There are many other issues arising in generating efficient codes for SIMD devices. Many post-vectorization optimizations are needed to address these issues to achieve speedups on SIMD devices. For example, due to the constraints on memory units, data reorganization may introduce significant overhead in SIMD code by adding additional permutation instructions. It is a performance-critical problem to minimize permutation instructions.

Some of the post-vectorization optimizations can be conducted directly on SIMD code, however, is usually more efficient to apply them on vector programs where the compiler has more information about the whole program.

SIMD compilation has been studied for several years. In [10], Cheong and Lam developed an optimizer for VIS, the SIMD extension of SPARC. Krall and Lelait applied traditional vectorization to generate VIS code [37]. Sreraman and Govindarajan developed a vectorizer for Intel's MMX [64]. In [41], an instruction packing technique, called as *superword level parallelism*, was proposed by Larsen and Amarasinghe to translate sequential basic blocks into SIMD instructions. In [45], similar techniques were also introduced to combine isomorphic instructions together to SIMD instructions. In [7, 5], Bik introduced the SIMD compilation techniques developed for the Intel compiler in detail. In [70], Wu et al. describes the SIMD compilation framework used in IBM XL compilers.

More recently, commercial compilers also started to support SIMD compilation. The Crescent Bay Software extends VAST, a vectorizing compiler, to generate codes for AltiVec [13]. The Portland Group offers the PGI Workstation Fortran/C/C++ compilers that support automatic usage of SSE/SSE2 [55]. The Codeplay announces the VectorC compiler for all x86 extensions and the Emotion Engine used in the Play Station 2 [12]. Also, Intel

extended its own product compiler to vectorize for MMX and the SSE family [7, 5] and IBM included automatic SIMD compilation in its XL compilers [15, 69, 70]. Starting from version 4.0, the GNU compiler also supports SIMD compilation for VMX and the SSE family [53]. Table 3.1 lists several on-shelf SIMD compilers that support today's SIMD devices.

Compilers	Vendor	Language	Intr	Vec	SIMD Devices
VAST-AltiVec	Crescent Bay	C/C++/For	No	Yes	VMX
PGI Workstation	Portland Group	C/C++/For	No	Yes	SSE Family
VectorC	Codeplay	C/C++	Yes	Yes	SSE Family, PS2
XL Compilers	IBM	C/C++/For	Yes	Yes	VMX, BG/L, Cell
Intel Compilers	Intel	C/C++/For	Yes	Yes	SSE Family
GCC	GNU	C/C++/For	Yes	Yes	VMX, SSE, 3DNow

Table 3.1: Several examples of SIMD compilers.
(Abbr.: Intr-Intrinsic Function; Vec-Automatic Vectorization; For-Fortran)

In addition to general compilation frameworks for SIMD devices, there have been various compiler techniques developed to address new issues arising in SIMD compilation, mainly due to special architectural features of today's SIMD devices. Most of them are introduced in Chapter 14. The others are distributed in the related chapters and discussed in detail.

3.2 Experiments with SIMD Compilers

To evaluate the effectiveness of SIMD compilation, some experiments were conducted on multimedia applications, which are supposed to gain the most performance benefits from SIMD devices.

3.2.1 Berkeley Multimedia Workload

Berkeley Multimedia Workload (BMW) benchmark [62] is an extension of MediaBench [43]. Table 3.2.1 lists the 12 applications from the BMW benchmark that considered in this paper. Some of the applications may consist of several standalone programs that share a common code base. For example, there are four different programs in `Mesa` and some applications

include two programs, one for encoding and the other for decoding.

Name	Description	#Proc	%Ex.
ADPCM	Audio compression (Encoder)	1	100
	Audio compression (Decoder)	1	100
GSM	Speech compression (Encoder)	2	73.8
	Speech compression (Decoder)	1	74.0
LAME	MPEG audio encoder	2	30.7
mpg123	MPEG audio decoder	1	57.9
DVJU	Image compression (Encoder)	2	80.9
	Image compression (Decoder)	1	84.0
JPEG	Image compression (Encoder)	2	51.6
	Image compression (Decoder)	3	77.8
MPEG2	Video compression (E/D)	2	69.2
	Video compression (Decoder)	3	59.2
POVray	Ray tracer	1	15.5
Mesa	3D Graphics (Gear)	1	81.3
	3D Graphics (Morph3D)	2	23.8
	3D Graphics (Reflect)	1	48.7
	3D Graphics (Pointblast)	3	70.0
Doom	FPS video game	1	25.9
Rsynth	Speech synthesizer	1	70.5
Timidity	MIDI music rendering	3	52.2
Average		2	62.3

Table 3.2: BMW multimedia applications.

In the experiment, we only study procedures that consume more than 10% of total execution time. We call them *core procedures*. As shown in Table 3.2.1, although core procedures include less than 5% of the total lines, they consume a large portion, 62% on average, of the total execution time of an application. Thus, speedups on these core procedures will lead to a significant whole-program performance improvement.

3.2.2 Experimental Results

We first investigate the effectiveness of state-of-the-art MME vectorizing compilers. The experiment uses the Intel compiler v8.0 (ICC) and is conducted on a Pentium 4 machine with SSE2. Among all the commercial vectorizing compilers for SSE2, the Intel compiler is

one of the most widely used and best documented (both by user manuals [5, 29] and research reports [6, 7]). In addition, the Intel compiler can successfully vectorize 73 out of 135 loops from the Callahan-Dongarra-Levine Fortran test suite [46]. This number is comparable with those of the vectorizers reported in [46]. It shows that the Intel compiler is a competent vectorizer according to traditional vectorization standards.

Table 3.3 summarizes the number of loops in the core procedures that are vectorized by the Intel compiler. Out of 160 loops in the core procedures, 114 are innermost loops, 14 of them are fully vectorized, 3 are reported by the Intel compiler as "vectorizable but (vectorization) seems inefficient". For loops that are not vectorized, Table 3.3 further classifies them according to the reasons why vectorization fails as reported by the Intel compiler.

Reasons Reported by ICC	# of Loops
Vectorizable	
Vectorized	14
Vectorizable but inefficient	3
Not vectorizable due to	
Outer Loops	46
Irregular Loop Structure	30
Data Dependence	19
Unsupported Instructions	18
Unsupported Data Types	9
Conditions	18
Others	3
Total Number of Loops	160

Table 3.3: Loops vectorized by ICC v8.0.

To understand why the compiler fails so often, an empirical study was conducted on these multimedia applications. Most of the findings from the study are summarized in Chapter 14. In addition, some core procedures were tentatively translated into SIMD code by hand. The manual translation successfully vectorizes 23 of total 34 core procedures and achieve up to 3.39 speedups on 16 procedures. It significantly outperforms automatic compilation on most core procedures, except one from the MPEG2 encoder.

In a summary, the experiment indicates that SIMD compilation has difficulties to achieve

a comparable performance as using intrinsic functions. There are many issues that must be addressed to generate efficient code for SIMD devices.

3.3 VINCI: Vector I-code Novel Compilation Infrastructure

In this thesis, a unified compilation framework is proposed to generate efficient code for SIMD devices. As shown in Figure 3.1, the components of the framework include various program analysis, transformation and optimization techniques. The following chapters will discuss them in details.

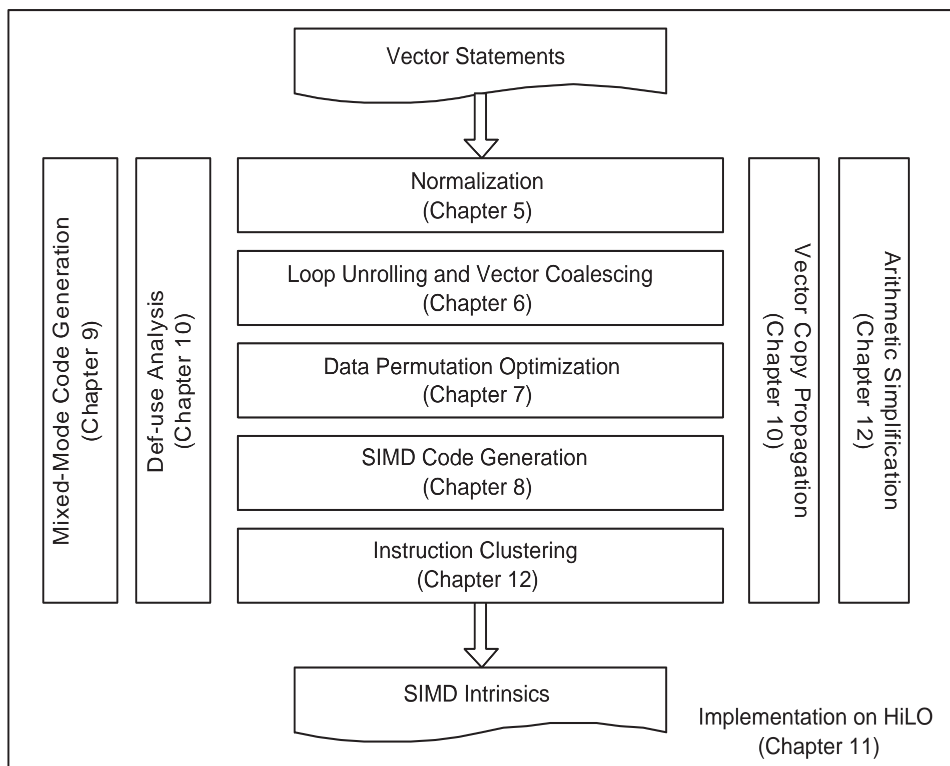

Figure 3.1: Major components of VINCI.

Chapter 4

Vector and Vector Operations

The input of the compilation framework is a vector I-code program. During translation of vector statements into *SIMD instructions*, most of the program analysis and transformations are applied to vector statements.

Thus, before introducing the framework, we first define several terms, such as vector and vector operations. Section 4.1 defines vector, vector sections, and vector operations. Then, Section 4.2 discusses data permutations, a special type of vector operations in detail. Detailed information about I-code language is given in Chapter 11.

4.1 Vector Representation

The VINCI compiler uses a vector representation similar to the one used by Fortran 90. Vectors are used in both input programs and internal representations. Vector variables are represented as array sections, using triplet notation (or section subscripts). Most vector operations are natural extensions of scalar arithmetic, logical, or bit-wise operations on vectors. A special class of vector operations replicate or permute data elements in vectors.

4.1.1 Vector and Vector Expression

A *vector* is a sequence of data elements of the same type. In this thesis, vectors are usually represented with a subscripted capital letter of the form, V_n. The subscript, n, is the number of elements in the vector. Sometimes a vector can also be presented by enumerating its data

elements, such as $\langle v_0, v_1, v_2, ..., v_{n-1}\rangle$. Both representations are equivalent.

$$V_n \equiv \langle v_0, v_1, v_2, ..., v_{n-1}\rangle$$

The *length* of vector V is defined as the number of data elements in V, represented as $|V|$. It is an important property of vectors. From the definition, we have

$$|V_n| \equiv n$$

On the other hand, the *bit length* of vector V is the number of bits needed to store V, represented as $|V|_b$. In essence, the bit length of V equals the number of bits needed to store each element of V, v_i, times the length of V, $|V|$. Thus,

$$|V_n|_b \equiv |V_n| \times |v_i|_b \equiv n \times |v_i|_b$$

$|v_i|_b$ is the bit length of element v_i, which is determined by its data type [1]. For example, for an integer element, $|v_i|_b$ is 32.

In the implementation used for the experiments of this thesis, the length of vectors must be compile-time constant. Such constraint could be partially relaxed through strip-mining variable-length vectors into a variable-length loop with constant-length vectors.

As a specific type of vectors, *vector section* (of an array) is common in vector programs. A vector section is defined as a sequence of selected elements of an array. For example, if a is declared to be a one-dimensional array,

$$float\ a[10];$$

[1] Here we assume that all data elements in a vector have the same type.

then the vector section $a[0:6:2]$ refers to a four-element vector,

$$a[0:6:2] \equiv \langle a[0], a[2], a[4], a[6] \rangle$$

The triplet notation of Fortran 90 is used to specify a vector section. Thus, in

$$a[b:e:s]$$

a is an array variable and b, e and s are three integer expressions. The first and second subscripts, b and e, are the *begin* offset and the *end* offset respectively. They specify the first and the last array elements referenced by $a[b:e:s]$. In other words, both b and e are inclusive boundaries. The third subscript s, called as *stride*, specifies the increment between successive array elements in the sequence. The vector section $a[b:e:s]$ defines the following sequence of array elements.

$$a[b:e:s] \equiv \langle a[i] \mid i = b + k * s, b \leq i \leq e \rangle$$

The length of this vector expression can also be calculated from the subscript triplet as follows.

$$|a[b:e:s]| = \lfloor (e-b)/s \rfloor + 1$$

In the implementation used for the experiment, s must be a compile-time constant.

Besides vector sections, constant vectors and temporary vectors are other common types of vectors. A *vector literal* is a sequence of constant numbers. It is usually specified by enumerating all the elements: $\langle c_0, c_1, ..., c_{n-1} \rangle$. The *index vector* used in a data permutation operation (see Section 4.2) is a typical example of constant integer vectors.

Different from the other vectors, *temporary vectors* are mainly created by the compiler during program transformations. Essentially, a temporary vector is a vector section where

the variable is a compiler-created temporary array. Unlike vector sections, temporary vectors have no memory space allocated and associated with them. They will be stored into registers.

4.1.2 Vector Operations

Most vector operations belong to one of three different categories: scalar extraction or insertion, element-wise operations, and data permutations.

First, for scalar operations, it is possible to extract or insert data elements from (to) vectors. For example, $V(i)$ refers the i-th element of V. In essence, element reference operations provide a way of exchanging data elements between scalars and vectors. For vector sections, such element references are actually referencing array elements. On the other hand, for literal vectors, such references can be replaced with constant numbers directly. Finally, for temporary vectors, element references will eventually require extra instructions to extract (insert) data from SIMD registers to scalar registers. In the worst case, for those SIMD devices that do not support direct communication between scalar units and SIMD units, such element references are very expensive since the data have to go through the memory system.

However, since we expect vectors in vector programs to be vector sections, such element references are actually referencing array elements.

In order to conduct arithmetic, logical, and other operations on vectors, most scalar operations are naturally extended to element-wise vector operations. An element-wise operation on vectors is essentially a sequence of same scalar operations on the corresponding data elements in vector operands.

$$X_n \; op \; Y_n \equiv \{X(i) \; op \; Y(i),\ 0 \leq i \leq n-1\}$$

For example,

$$\langle 1, 2, 3, 4 \rangle + \langle 2, 2, 3, 3 \rangle \rightarrow \langle 3, 4, 6, 7 \rangle$$

Thus, a valid binary operation requires two vector operands to have the same length. In general, *op*, can be assignment, arithmetic operations, comparisons, logical operations, or bit-wise operations. Since these operations are conducted on vector elements independently, they can be executed in any arbitrary order.

The third category are data permutations, which reorganize data elements within vectors. The detail about data permutation is described in the next section.

4.2 Data Permutation on Vectors

In this section, we define the three data movement operations, *Permute*, *Reduct*, and *Spread*, used in generic vector programs.

4.2.1 Permutation Operation

The operation $Y \leftarrow Permute(X_n, P)$ performs a permutation P on a vector X_n of n elements to produce a vector Y_n of the same length. P can be specified as a *permutation matrix*, $P_{n\times n}$, which is an identity square matrix with its rows reordered. A permutation operation can be viewed as a matrix-vector multiplication, i.e.,

$$Permute(X_n, P_{n\times n}) \equiv P_{n\times n} \cdot X_n.$$

For example, assume the input vector is $X_4 = \langle x_0, x_1, x_2, x_3 \rangle$, the permutation matrix $P_{4\times 4}$ is

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

then the output vector will be

$$Permute(X_4, P_{4\times4}) = \langle x_0, x_2, x_1, x_3\rangle = P_{4\times4} \cdot X_4$$

A more compact representation specifies the permutation pattern as an *index vector* whose i-th element is the index of the input vector element that is to be moved to the i-th element of the output vector. For example,

$$Permute(\langle x_0, x_1, x_2, x_3\rangle, \langle 0, 2, 1, 3\rangle) = \langle x_0, x_2, x_1, x_3\rangle.$$

Maintaining a square permutation matrix greatly simplifies our optimization algorithm. However, it also requires the input and output vectors be of the same length. To express data permutations with mismatched input and output lengths, we use two special values:

- The first special value is a *star*, denotated as $\star$. If the i-th element of an index vector is $\star$, the i-th element of the output vector is undefined (i.e., we do not care what its value is). It can be used to specify data permutations where the output vector is shorter than the input. For example, the following permutation gathers the odd elements of the input vector,

$$Permute(\langle x_0, x_1, x_2, x_3\rangle, \langle 1, 3, \star, \star\rangle) = \langle x_1, x_3, \star, \star\rangle.$$

  Using $\star$ elements, we can explicit track "unused" elements during a permutation. This information can be used by the optimization algorithm. For example, $\star$ indicates that the permutation bandwidth is not fully utilized and that the permutation may be combined with other permutations applied to the same input vector.

- The second special value is a *diamond*, denotated as $\diamond$. If the i-th element of an index vector is $\diamond$, the i-th element of the output vector will remain unchanged. It is used to

specify data permutations where the output vector also implicitly serves as input. For example, suppose $Y_4 = \langle y_0, y_1, y_2, y_3 \rangle$. The outcome of the permutation,

$$Y_4 \leftarrow Permute(\langle x_0, x_1, x_2, x_3 \rangle, \langle \diamond, 1, \diamond, 2 \rangle)$$

is $Y_4 = \langle y_0, x_1, y_2, x_2 \rangle$.

As discussed in Chapter 5, $\diamond$ elements are generated when normalizing memory stores into strided vectors. Such conversion involves a special read-modify-write sequence that can be conveniently represented by the $\diamond$s.

Using $\diamond$ elements, we can explicitly track "unchanged" elements during a permutation. This information can be used by the optimization algorithm. For example, one may combine two permutations over the same output vectors into a permutation without $\diamond$ elements.

Alternatively, rectangular permutation can be introduced to avoid using $\star$ elements in square permutations. Similarly, square permutations with $\diamond$ can be represented by using rectangular permutations and vector catenation. However, the implementation with rectangular permutations and catenation is more complicated than the one with square permutations.

4.2.2 Properties of the Permutation Operation

There are two rules that are the foundation of our optimization algorithm on data permutations.

Composition Rule. This rule states that two consecutive permutations can be composed into one.

$$\begin{aligned} & Permute(Permute(X_n, P_{n\times n}), Q_{n\times n}) \\ \equiv\ & Permute(X_n, Q_{n\times n} \cdot P_{n\times n}) \end{aligned}$$

This composition rule is the basis of our optimization algorithm. Each time the rule is applied, we reduce by one the number of permutations.

It is safe to apply composition rule in the presence of $\star$ since these values will be discarded eventually. On the other hand, since permutation containing $\diamond$ elements combines data from both input and output, the composition rule cannot be applied when there is $\diamond$ in P. However, the composition rule is still applicable when Q contains $\diamond$ elements.

Assume that the i-th element of the index vectors of P and Q are p_i and q_i respectively. Then, the i-th element of the index vector of $R = Q \cdot P$ can be computed as follows:

$$r_i = \begin{cases} p_{q_i} & \text{if } q_i \neq \star, \diamond \text{ and } p_i \neq \diamond \\ \star & \text{if } q_i = \star \text{ and } p_i \neq \diamond \\ \diamond & \text{if } q_i = \diamond \text{ and } p_i \neq \diamond \\ invalid & \text{if } p_i = \diamond \end{cases}$$

The composition rule also says that a permutation can be decomposed into two permutations. Permutation decomposition is also very important during permutation propagation, especially when different patterns may have different costs. However, there may be numerous ways to decompose a permutation. Thus, in practice, we only attempt to decompose an expensive but non-propagatable permutation, which needs more native permutation instructions to implement, into two permutations. One is propagatable to partial uses and the other is less expensive than the original one (see Section 7.2.2).

Distributive Rule. This rule says that a permutation operation can be distributed over element-by-element vector operations. Let *op* be an element-by-element operation, we have

$$\begin{aligned} & Permute(X_n, P_{n\times n}) \ op \ Permute(Y_n, P_{n\times n}) \\ \equiv \ & Permute(X_n \ op \ Y_n, P_{n\times n}) \end{aligned}$$

The distributive rule allows us to move permutations over operations. By moving common permutations toward the root of an expression, we can reduce the number of permutations. More importantly, it also creates more consecutive permutations to enable the application of the composition rule.

It is usually safe to apply the distributive rule in the presence of $\star$, unless it will introduce exceptions for some specific operations, like division, on some SIMD devices. While doing so can save permutation operations, it may also result in additional arithmetic operations. Depending on the relative cost of permutation and arithmetic operations, one may choose to distribute or un-distribute in the presence of $\star$.

On the other hand, it is generally unsafe to distribute in the presence of $\diamond$.

4.2.3 Reduction and Replication

Reduction operations involve a type of data movement that cannot be expressed with general permutations. Therefore, we introduce the notation, $Y_n \leftarrow Reduct(X_n, op_r)$, to specify a reduction with operation op_r on all n elements of X_n and stores the result in the first element of Y_n. For example, $Y_n \leftarrow Reduct(X_n, +)$ represents

$$y_0 = x_0 + x_1 + ... + x_{n-1}$$

From the view of data permutation, replication is the reverse operation of reduction. A replication (*Spread*) operation expands a scalar into a vector. $Y_n \leftarrow Spread(X_n, i)$ specifies a replication of the i-th element of X_n to fill Y_n, i.e.,

$$Spread(\langle \star, \star, x_2, \star \rangle, 2) = \langle x_2, x_2, x_2, x_2 \rangle$$

Notice that both the input of the replication and the output of the reduction are scalars. However, vectors are used in both representations mainly because most SIMD devices do

not support reduction (or spread) instructions that directly target scalar registers as output (or as input). Using vectors also would facilitate the optimizations on *Reduct* and *Spread* operations.

Composition Rule This rule says that consecutive *Reduct-Permute* and *Permute-Spread* can be composed into one, assuming that op_r is associative.

$$\begin{aligned} Reduct(Permute(X_n, P_{n\times n}), op_r) &\equiv Reduct(X_n, op_r) \\ Permute(Spread(X_n, i), P_{n\times n}) &\equiv Spread(X_n, i) \end{aligned}$$

Distributive Rule This rule says that *Spread* can be distributed over element-by-element vector operations.

$$Spread(X_n \; op \; Y_n, i) \equiv Spread(X_n, i) \; op \; Spread(Y_n, i)$$

4.2.4 Permutation Operations in Vector I-code Programs

Permute, *Reduct* and *Spread* are all supported by the vector-extended version of the I-code language. However, even using index mapping vectors, the representation of *Permute* can still be too lengthy to be used in input programs. Thus, several other forms of data permutations are also defined in VINCI to represent those permutation with regular patterns. For example, `Transpose(v, l, s, u)` treats the input vector, `v`, as a flattened four-dimensional matrix and then transposes the two middle dimensions. Thus, the index mapping vector can be calculated as follows,

$$\langle p_i | p_i = \lfloor i/l \rfloor * l + \lfloor i \bmod l/s \rfloor * u + \lfloor i \bmod s/u \rfloor * s + i \bmod u \rangle$$

where we use the notation, $\lfloor x \rfloor$, to represent the truncation of x to the closest smaller integer value.

In the words, `Transpose` can be represented as `Permute` by using the index mapping vector above. However, `Transpose` is more compact. Like `Transpose`, some other compact representation of data permutations are used in input programs and can also be represented by `Permute` by expanding index mapping vectors.

Another advantage of using such compact representations for data permutations is to provide an effective way to expand the VINCI system to deal with variable strides and misalignment settings. It would be an interesting future work to do.

On the other hand, not all permutation patterns can be represented by those alternatives. That also limits the application of the composition rule on them.

Chapter 5

Normalizing Vector Programs

The first step of our compilation framework is to normalize all vector sections. That is, to transform the input program so that all vector sections are stride-one, aligned and have a length that is a multiple of L_r, the length of the SIMD register. This is accomplished by inserting *Permute* operations to pack and/or align data. By normalizing vector sections, all implicit permutation operations in non-contiguous or misaligned references are explicitly expressed.

During this transformations, the algorithm may introduce temporary arrays (vectors) to hold intermediate results. The life range of these temporary arrays is limited to the basic block being translated. They will be allocated to (virtual) vector registers and therefore will not occupy any memory space.

Section 5.1 describes how to normalize vectors with non-unit strides. To avoid introducing redundant data permutations, an optimization of coalescing vectors is conducted after the normalization. The details of vector coalescing are introduced in Section 5.2. Finally, Section 5.4 shows how to normalized misaligned vectors.

5.1 Normalizing Vectors with Non-unit Strides

Normalization transforms a strided vector load into a load of a stride-one vector and a *pack* operation and a strided vector store into a store to a stride-one vector and an *unpack-merge* sequence. Both pack and unpack-merge can be expressed using *Permute* operations. Normalizing misaligned loads and stores are discussed separately next.

5.1.1 Strided load

Consider a vector load,

```
S:  ... = ... v[b:e:s] ... ;
```

The first step is to allocate a temporary vector `t[0:L*s-1]` where `L=(e-b)/s+1` is the length of `v[b:e:s]`. Vector `t[0:L*s-1]` is defined by a permutation operation on `v[b:b+L*s-1]`, which is inserted at `S`. A contiguous portion of the vector `t[0:L-1]` is then used to replace `v[b:e:s]` in statement `S`. The final version of the code is:

```
S: t[0:L*s-1] = Permute(v[b:b+L*s-1], P);
     ... = ... t[0:L-1] ... ;
```

where `P` is ⟨`0,s,2s,...,L*s-s,`⋆`,...,`⋆⟩. Figure 5.1 (Scheme I) shows an example where stride-2 vector loads are normalized using this scheme.

However, this conversion scheme has a major drawback when merging permutations. Consider the example in Figure 5.1. Scheme I generates two permutations on two largely overlapping vectors `b[0:99]` and `b[1:100]`. As discussed in Section 5.2, our merging algorithm tries to combine permutations that operate on the same input vectors.

To facilitate the merging of permutations, we modify the original instruction sequence by truncating generated vector addresses to the closest s boundaries. When s is a multiple of L_r, the length of SIMD registers, such modification also helps avoid unnecessary realignment.

We use the notation, $\lfloor x \rfloor_y$, to represent the truncation of x to the closest smaller value that is a multiple of y. Assume that the base addresses of arrays are L_r-byte aligned, and the elements are L_e-byte long, the enhanced stride conversion results in

```
S: t[0:L*s-1] = Permute(v[b':b'+L*s-1], P);
     ... = ... t[q*L:q*L+L-1] ... ;
```

Original statement

```
a[0:49] = b[0:98:2] + b[1:99:2];
```

After stride conversion (Scheme I)

```
t1[0:99] = Permute(b[0:99],⟨0, 2, 4, ···, 98, ⋆, ···, ⋆⟩);
t2[0:99] = Permute(b[1:100],⟨0, 2, 4, ···, 98, ⋆, ···, ⋆⟩);
a[0:49]  = t1[0:49] + t2[0:49];
```

After stride conversion (Scheme II)

```
t1[0:99] = Permute(b[0:99],⟨0, 2, 4, ···, 98, ⋆, ···, ⋆⟩);
t2[0:99] = Permute(b[0:99],⟨⋆, ···, ⋆, 1, 3, 5, ···, 99⟩);
a[0:49]  = t1[0:49] + t2[50:99];
```

Figure 5.1: An example of converting strided loads.

where $b' = \lfloor b \rfloor_s$, $q = b - b' = b \bmod s$, and

$$P = \langle \underbrace{\star, ..., \star}_{q*L}, q, q+s, q+2s, ..., q+(L-1)*s, \star, ..., \star \rangle.$$

Figure 5.1 (Scheme II) shows an example of converting stride-2 vector loads using this scheme.

5.1.2 Strided store

Consider a vector store,

```
S: v[b:e:s] = ...;
```

To normalize the store, a temporary vector `t[0:L*s-1]` is allocated and `v[b:e:s]` is replaced by `t[q*L:q*L+L-1]`. A permutation statement from `t` to `v` is inserted immediately after `S`. Using the same definition of `o` and `b'` as above, the final version of the code is:

```
S: t[q*L:q*L+L-1] = ...;
   v[b':b'+L*s-1] = Permute(t[0:L*s-1], P);
```

where

$$P = \langle \underbrace{\diamond, ..., \diamond}_{q}, 0, \underbrace{\diamond, ..., \diamond}_{s-1}, 1, \underbrace{\diamond, ..., \diamond}_{s-1}, 2, \diamond, ..., \diamond, L-1, \underbrace{\diamond, ..., \diamond}_{s-q-1} \rangle$$

5.2 Coalescing Partially Utilized Permutations

Stride conversion often generates permutations with many $\star$ and $\diamond$ elements. The presence of these elements indicates opportunities to merge permutations.

Two permutations with $\star$ elements that operate on the same input vectors can be merged if their index vectors can be merged by merging corresponding indices. Similarly, two permutations with $\diamond$ elements that have the same input and output vectors respectively can be merged together. We define a commutative operator, $\wedge$, to merge two permutation indices, as follows:

$$a \wedge a = a,\ a \wedge \star = a,\ a \wedge \diamond = a, \star \wedge \star = \star, \diamond \wedge \diamond = \diamond, \diamond \wedge \star = \diamond$$

Otherwise, we say the two permutation indices cannot be merged. We say two index vectors, A and B, can be merged if and only if all of their corresponding indices can be merged. In that case, the index vector of the merged permutation is $A \wedge B$, where $\wedge$ is applied element-by-element.

Merging opportunities are common when several strided references in the region jointly access a contiguous chunk of memory. This situation arises frequently in many applications.

Consider our previous example in Figure 5.1. The permutations produced by Scheme I cannot be directly merged because they operate on two slightly different vectors (`b[0:99]` and `b[1:100]`), whereas merging the permutations generated by Scheme II is straightforward:

```
t[0:99] = Permute(b[0:99],⟨0, 2, 4, ..., 98, 1, 3, ..., 99⟩);
a[0:49] = t[0:49] + t[50:99];
```

5.3 Normalizing Vectors cross Loop Boundaries

However, it is not always possible to coalesce these partially utilized permutations generated during normalization, especially when those permutations are distributed in the different iterations of the same loop. Considering the following example,

```
for(int i=0; i<4; i++)
  ... = ... a[i:i+12:4] ...;
```

where `a` is defined outside the loop. As shown in the example, `a[i:i+16:4]` is a 4-element vector referencing a 16-element section of the array `a`. Each iteration of the loop accesses one element for each four elements. Although all 16 data elements are referenced in the loop, the references are distributed into four different iterations. Thus, after normalization with Scheme I, we have,

```
for(int i=0; i<4; i++)
  t[0:15] = Permute(a[i:i+12], ⟨ 0,4,8,12,⋆, ..., ⋆ ⟩);
  ... = ... t[0:3] ...;
```

However, in order to facilitate coalescing, Scheme II should be used.

```
for(int i=0; i<4; i++)
  t[0:15] = Permute(a[i:i+12], P_i);
  ... = ... t[i:i+3] ...;
```

where P_i is defined as

$$P_i = \langle \underbrace{\star, ..., \star}_{i*4}, i, i+4, i+8, i+12, \underbrace{\star, ..., \star}_{12-4*i} \rangle$$

It makes the situation more complicated since the begin offset of `a[i:i+16:4]` is the loop index, which should be constant as required by the normalization algorithm to insert *Permute* operations.

For this problem, one solution is to insert a *Permute* operation immediately before the loop to reorganize the whole section of `a` referenced in the loop. By doing that, the stride-two vector, `a[i:i+12:4]`, can be replaced by a temporary vector with unit stride, `t[i:i+3]`.

```
t[0:15] = Permute(a[0:15], ⟨ 0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15 ⟩);
for(int i=0; i<4; i++)
  ... = ... t[i:i+3] ...;
```

First, the strided vector is normalized with Scheme II as shown above. Then, the *Permute* statement is instantiated and moved out of the loop. After that, there will be four partially-utilized *Permute* statements before the loop. Finally, those *Permute* statements are coalesced together into one *Permute*. In order to coalesce different instances of the *Permute* statement, it is required that the original strided vector references different data elements in each iterations, which is satisfied in all programs used in our experiments.

However, as described in Chapter 7, it is oftentimes performance-efficient to have an isolated permutation between loops. Although a complete permutation can fully utilize the bandwidth of permutation instructions, it might introduce additional overhead from memory loads and stores. Hence, an alternative solution is to keep such *Permute* operation inside the loop.

5.4 Normalizing Vectors with Misalignment

This normalization transforms accesses to vectors starting at addresses that are not a multiple of L_r into accesses to aligned vectors. If we assume the base address of `v` is aligned, vector `v[b:e]` is misaligned if $b \bmod (L_r/L_e)$ is not zero. Consider a misaligned load,

```
S: ... = ... v[b:e] ...;
```

As in the case of stride conversion, a temporary vector, `t[0:L'-1]`, is allocated for the replacement, where `L'` is the length of the aligned section `v[b':e']`, where `b'=`$\lfloor b \rfloor_{L_r/L_e}$ and `e'=`$\lfloor e+1 \rfloor_{L_r/L_e}$`-1`. For simplicity, let us assume L to be a multiple of L_r/L_e and $b' \neq b$. Then, `L'=L+`L_r/L_e, where `L` is the length of `v[b:e]`. Thus, there are L_r/L_e more elements in the aligned vector than the original one. In fact, the new vector could need at most $L + 2 * L_r/L_e$ elements if we had not assume that L is a multiple of L_r/L_e. At the same time, a permutation from $v[b' : e']$ to $t[0 : L' - 1]$ must be inserted at S.

```
S: t[0:L'-1] = Permute(v[b':e'], P)
   ... = ... t[0:L-1] ...;
```

where $P = \langle 0, 1, 2, ..., L-1, \underbrace{\star, ..., \star}_{L_r/L_e} \rangle$.

Similarly, consider a misaligned store,

```
S: v[b:e] = ...;
```

It is converted to

```
S: t[0:L-1] = ...;
   v[b':e'] = Permute(t[0:L'-1], P)
```

where $P = \langle \underbrace{\diamond, ..., \diamond}_{b'-b}, 0, 1, 2, ..., L-1, \diamond, ..., \diamond \rangle$.

5.5 Other Sources of Data Permutations

Besides data permutations introduced by converting strided and misaligned memory references, store-load forwarding between partial def and use of vectors may also introduce data permutations. For example, to propagate the right hand side of `t[b:e] = x[b:e]` to `t[b+1:e+1]`, it is necessary to insert an explicit permutation (and an element-insert operation). More details about vector copy propagation are described in Section 10.2.

In addition, data permutations inherent to programs, such as matrix transpose and bit-reverse reordering, are also very common. Sometimes it is difficult for compilers to recognize those data permutations from the standard C implementation. It would be easier to use these permutation operations as intrinsics to specify data permutations directly in input programs.

Chapter 6

Loop Unrolling and Vector Coalescing

In vector programs, vectors can have arbitrary lengths, whereas SIMD devices requires all vectors to have the same fixed bit-length. For example, on a 128-bit device, all vectors of 32-bit floating points must have the same length of 4. This number is also called as the degree of parallelism (Section 2.1).

As a result, if all vectors in a vector statement have the same length of 2 but the degree of parallelism of a target SIMD device is 4, only half of the computing bandwidth of the target device can be utilized by translating the vector statement to SIMD instructions directly. All vectors with irregular lengths, which are not multiple of the degree of parallelism, will lose some performance benefits from SIMD devices. For example, a statement with 5-element vectors can only utilize up to 67.5% of the computing power provided by a 4-way SIMD device.

One way to generate longer vectors is to unroll loops and coalesce vector statements from different iterations. For example, by unrolling an inner loop twice, each statement in the loop is copied twice. Then if two copies of a vector statement can be merged, the length of vectors in the resulting statement will be duplicated.

Furthermore, loop unrolling helps generate large basic blocks (of final SIMD instructions) for better ILP (Instruction Level Parallelism) and less loop overhead. More importantly, when a loop is fully unrolled, the indices of some array references might become constant so that these array references can be replaced by scalars. Such scalar replacement plays an important role in optimizing DSP programs.

6.1 Array Expansion

In general, it is relatively straightforward to unroll a loop, especially when the trip count is constant. The loop body is duplicated and put together in the new loop. Loop unrolling is always valid as long as the order of the statements is not changed.

However, in order to coalesce vector statements from different iterations together, the compiler needs to reorder those statements to be next to each other. Thus, in the case of vector coalescing, data dependence analysis is necessary to ensure that the reordering is valid.

In an unrolled loop, it is typical to see the anti- and/or output data dependence introduced by private variables which are defined and used only within the loop. Those dependences might prevent the compiler from reordering statements. For example, consider the following loop, where `s` is a private variable,

```
1. for(i=0; i<m; i++) {
2.   float s;
3.   s = ... ;
4.   ... = ... s ... ;
5. }
```

After unrolling the loop twice (assume `m` can be divided by 2), it becomes

```
1. for(i=0; i<m; i+=2) {
2.   float s;
3.   s = ... ;          /* 3 */
4.   ... = ... s ... ;  /* 4 */
5.   s = ... ;          /* 3 */
6.   ... = ... s ... ;  /* 4 */
7. }
```

In the original loop, there is a true dependence from 3 to 4. However, in the unrolled loop, besides the duplicated true dependences, an anti-dependence from 4 to 5 and an output dependence from 3 to 5 are introduced by `s`.

Scalar expansion is a common privatization technique to eliminate such false dependences. It expands each scalar variable into an array so that each iteration has its own copy of the private variable and no anti- or output dependence will be introduced after unrolling. Still using the previous example, after applying scalar expansion, the original loop becomes,

```
1. float s[m];
2. for(i=0; i<m; i++) {
3.   s[i] = ... ;
4.   ... = ... s[i] ... ;
5. }
```

Then it is unrolled twice as follows,

```
1. float s[m];
2. for(i=0; i<m; i+=2) {
3.   s[i] = ... ;
4.   ... = ... s[i] ... ;
5.   s[i+1] = ... ;
6.   ... = ... s[i+1] ... ;
7. }
```

where there is no dependence between 4 and 5 anymore.

Besides scalar expansion, there are other well-known methods can be used to eliminate the anti- or output dependence introduced by private variables. For example, one solution is rename scalar variables from different iterations to different variables. In essence, the idea behind variable renaming and scalar expansion is same. The only difference is that variable renaming results in a set of unrelated scalar variables, instead of a private array. Since the

next step after unrolling is to coalesce vector statements to produce longer vectors, it is more efficient to keep all copies of the same original variable together. Hence, scalar expansion and its extension on arrays, *array expansion*, is used in VINCI.

Like private scalars, private arrays will also introduce additional data dependence after loop unrolling. When there is an array locally defined in the loop, any accesses of this array will introduce anti- and/or output dependence in the unrolled loop. Again like scalar variables, array variables can also be expanded by applying array expansion. Similarly, it duplicates a private array as many times as the unrolling factor and allocates different sections for each iterations.

As an example, assume there is an array reference, $a[k]$, of a private array a, in a m-iteration loop, and the index k is a constant integer.

```
for(i=0; i<m; i++) {
  float a[n];
  ... a[k] ...
}
```

As described in Chapter 11, multi-dimensional array is not supported by the HiLO compiler. Thus, after array expansion, the result array is still a one-dimensional array but with its length increased.

Like expanding scalar variables, each element in the private array are duplicated by r times, where r is the unrolling factor. On the other hand, different from scalar expansion, array expansion can have different ways of organizing the duplicated elements in the expanded array. For simplicity, let us assume that the loop is fully unrolled, thus, the unrolling factor is the loop trip count.

- **Catenating** Expands the private array so that the k-th element of an n-element private array becomes the $(i * n + k)$-th element of an $n * m$-element global array, where i is the loop index and the m is the trip count. Thus, the distance between two duplicated

elements is n. In other words, the array is expanded by row.

```
float a[n*m];
for(i=0; i<m; i++) {
  ... a[i*n+k] ...
}
```

- **Interleaving** the elements from different copies one by one. The k-th element of an n-element private array becomes the $(k*m+i)$-th element of an $n*m$-element global array. Thus, the distance between two duplicated elements is 1. In other words, the array is expanded by column.

```
float a[n*m];
for(i=0; i<m; i++) {
  ... a[k*m+i] ...
}
```

Similarly, for a vector expression, $a[b:e]$, in such a loop, array expansion will translate it to,

1. $a[i*n+b:i*n+e]$ (Catenating)
2. $a[b*m+i:e*m+i:m]$ (Interleaving)

When the unrolling factor, u, is not the same as the trip count, m, the array will be duplicated by u times. Correspondingly, $a[b:e]$ will be expanded to

1. $a[i*n+b:i*n+e]$ (Catenating)
2. $a[b*u+i:e*u+i:u]$ (Interleaving)

Notice that the resulting vector from catenating expansion is the same as the fully unrolled version. This is an important advantage of catenating expansion over interleaving expansion.

There are many other ways to expand private arrays but require more complex indices. The distance between two duplicated elements can vary from 1 to n and each number corresponds to a different type of array expansion. However, as discussed in Section 6.2 and 12.1, they have different effects on other optimizations. For performance reasons, we always use *interleaving expansion*. For the same reason, we usually apply the same type of array expansion on all private arrays in a given loop to avoid introducing additional data permutations.

6.2 Coalescing Vector Statements

Besides the constraints from data dependences, there are other requirements on coalescing vector statements. Currently, we require all vectors to be exclusive. In other words, they cannot share any elements, if they are to be coalesced together. Because of array expansion, private vectors naturally satisfy this condition. The algorithm needs only to check vectors referencing global arrays that are visible outside.

In DSP programs generated by SPIRAL, those vectors referencing global arrays takes one of the following two forms[1].

1. $a[i*n+b : i*n+e]$ (F1)

2. $a[b*u+i : e*u+i : u]$ (F2)

Based on this observation, it is relatively straightforward to check whether the instances of a vector expression from different iterations can be coalesced together. As long as the following conditions are satisfied, there is no overlapping between vectors from different iterations.

1. $b-e+1 \leq n$ for F1

[1] If normalization is conducted before loop unrolling and vector coalescing, only $F1$ form will be possible in vector programs.

2. $u \leq m$ for F2

where n is the number of elements referenced in the loop body and m is the loop trip count.

Note that the catenating and interleaving expansion (on vector expressions) result in vector expressions in $F1$ and $F2$ format, respectively. As a result, after array expansion, all vector expressions are unified into these two forms, which greatly simplifies the job of vector coalescing.

In essence, this pass needs to first unroll the loop, reorder the statements from different iterations, and then coalesce them. In implementation, the algorithm combines all 3 steps together and directly expands vector statements in the loop. Depending on the format of vector expressions, the expansion is conducted as follows,

- If all vector expressions in a vector statement are in $a[i*n+b : i*n+e]$ ($F1$) format, they will be translated into $a[b*l : e*l+l]$. If $(e-b+1) \neq n$, data permutations will be inserted to keep the triplet representation.

- If all vector expressions in a vector statement use in $a[b*u+i : e*u+i : u]$ ($F2$) format. It will also be translated to $a[b*u : e*u+u]$. However, if the unrolling factor u is not m, data permutations will be inserted to keep unit strides (if this pass is conducted after the normalization);

- If vector expressions in a vector statement are in different formats, data permutations will be inserted.

6.3 Unrolling Loops with Data Permutations

When conducting loop unrolling and vector coalescing, it is not uncommon to see data permutations in loops. Those permutations can come from normalization, if it is conducted first, from an early pass of loop unrolling and vector coalescing on inner loops, or directly

from the input program. Data permutations are handled in a very similar way as other vector statements.

First, both input and output vectors are expanded by using array expansion. Like other vectors, interleaving scheme is used for the performance reason. Second, the permutation pattern is also expanded correspondingly. This is a new issue arising only during coalescing permutation operations.

If the data permutation is represented as a *Permute* operation with an index mapping vector, $\langle p_i \rangle$, the index mapping vector is expanded as follows,

$$\langle p'_i | p'_i = p_{\lfloor i/u \rfloor} + i \bmod u \rangle$$

where u is the unrolling factor.

If the data permutation is represented as a regular *Transpose* operation, the expansion is more straightforward. We only need to multiply all integer parameters by u. For example,

$$Transpose(a[b:e], l, s, t) \Rightarrow Transpose(a[b*u : e*u+u-1], l*u, s*u, t*u)$$

Chapter 7

Data Permutation Optimization

In this section, we introduce an algorithm to reduce the number of permutation operations within a basic block. The input to the algorithm is a straight-line vector program that has been normalized as discussed in the preceding chapters.

7.1 An Overview of the Optimization Algorithm

The optimization algorithm propagates permutations along the def-use graph built from the input program. It then applies the composition and distributive rules to reduce the number of permutations. The def-use graph is an extension of conventional def-use graph to accommodate vectors (see Section 10.1).

The algorithm is shown in Figure 7.1. The worklist `W` contains all statements of the form:

```
v = Permute(x, P);
```

where `P` contains no $\diamond$ element[1]. The algorithm propagates along def-use chains of each statement in `W`. It addresses cases such as partial definition and partial use by conducting a check at line 3 in `propagate_and_merge` (see Section 7.2 for more details about partial use boundaries). Once a definition of the form `v = Permute(x, P)` is propagated to a permutation of the form `Permute (v, Q)`, the algorithm merges `P` and `Q` applying composition rule and adds the resulting statement to `W`. The algorithm also tries to reorganize other

[1] Permutations containing $\diamond$ cannot be propagated because they use data from both input and output vectors.

```
W ← ∅;
optimize_permutation(basicblock bb)
  W ← {S| S:"v = Permute (..., P)" at top of DU graph}
  WHILE W ≠  ∅ DO
    remove a statement, S:"v=Permute(...,P)", from W
    U ← set of all use statements of v, the lhs of S
    propagate_and_merge(U, S)
  END

propagate_and_merge(set U, stmt "v=Permute(...,P)")
1.  FOR each statement T in U DO
2.    Let v' be the use of v in T
3.    IF possible to propagate Permute(x,P) to v' THEN
4.      IF T is of the form "w = Permute(v', P')" THEN
5.        T ← "w = Permute(x, P'*P)"
6.        add T to W
7.      ELSE
8.        tentatively replace v' with "Permute(x, P)"
9.        IF possible to convert T.rhs into "Permute(<expression>,P)" THEN
10.         carry out the transformation and make the change permanently
11.         add T to W
12.       ELSE
13.         undo the replacement at 8
14.       ENDIF
15.     ENDIF
16.   ENDIF
17. END
```

Figure 7.1: The algorithm of optimizing data permutations in a basic block.

statements into the form `v = Permute(..., P)` (see Section 7.3). The algorithm repeats this process until `W` becomes the empty set.

When the use is not an input to a permute (e.g., `... = v + Permute(...)`), the algorithm may still tentatively propagate `v`. If at the end of the algorithm, these tentative replacements do not lead to a consolidated right hand side of the form `Permute(...)` then the replacement is undone (line 13 in `propagate_and_merge`).

If a permutation is propagated to all its uses, the permutation itself will be deleted by dead-code elimination.

7.2 Propagation along Def-use Chains

The algorithm in Figure 7.1 assumes that the def-use information of program is already available before optimization. Unlike the traditional def-use analysis on scalars, the def-use analysis on vectors needs to deal with the partial relationship between two vectors[2]. For example,

```
t[0:7] = Permute(a[0:7], P);                                    (1)
b[0:3] = t[0:3] + t[4:7];
```

The use of `t[0:3]` (or `t[4:7]`) is a *partial use* of `t[0:7]`.

Our algorithm must handle propagation of definitions to multiple partial uses. In code segment (1), unless P can be partitioned into permutations on 4-element vectors, it cannot be propagated to the partial uses of `t[0:7]`. For example, if P is $P_a : \langle 0, 2, 1, 3, 4, 6, 5, 7 \rangle$ and can also be represented as an 8×8 permutation matrix,

$$P_a : \left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

If we evenly divide the permutation matrix above into four 4×4 sub-matrices, element 1 only occurs in two diagonal sub-matrices. Thus the permutation can be bisected into two independent but smaller permutations and then propagated to the partial uses as follows,

[2]More details about the def-use analysis on vector programs is discussed in Section 10.1

```
t[0:3] = Permute(a[0:3], ⟨0,2,1,3⟩);
t[4:7] = Permute(a[4:7], ⟨0,2,1,3⟩);
```

On the other hand, if P is $P_b : \langle 0, 4, 1, 5, 2, 6, 3, 7\rangle$, whose matrix representation is,

$$P_b : \left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

In this permutation matrix, 1 occurs in all four sub-matrices. The matrix would not be block-diagonal and cannot be partitioned into two sub-matrices. Thus the permutation cannot be propagated to the partial uses.

Although it is straightforward to determine whether a large permutation can be partitioned into smaller permutations from its matrix representation, the algorithm needs to make such a decision directly for its index mapping vector.

Let $y[b_y : e_y] = Permute(x[b_x : e_x], P)$ be the permutation to be propagated and $y[b'_y : e'_y]$ be a partial use. That is, we have that $b'_y \geq b_y$, $e'_y \leq e_y$. Assume P is represented as the index vector $\langle p_0, p_1, ..., p_{n-1}\rangle$.

The following condition must be satisfied so that the permutation can be partitioned and propagated to the use.

$$\forall i \in [b_y, e_y], p_i \in [b'_x, e'_x]$$

where $[b'_x, e'_x] \subseteq [b_x, e_x]$, $e'_x - b'_x = e'_y - b'_y$ and b'_x is a multiple of L_r/L_e, the number of elements in each physical vector register.

The condition specifies that all elements in the partial use must come from a contiguous (and aligned) section of the source vector. If the condition is not satisfied, we say this permutation crosses the *partial use boundary*. Partial use boundary is one of the major reasons preventing propagation in many applications.

To enable propagation across partial use boundaries, we developed several techniques, including *permutation reshaping* and *permutation decomposition* which are discussed in the next two subsections.

In real applications, it is possible for a vector definition and the corresponding use to overlap partially but neither of them contains the other. Our algorithm conservatively does not propagate the permutation in such situations.

If the definition is a subset of the use, we can propagate such partial permutation to use and continues the algorithm like other permutations. However, it would be better to collect all partial definitions that have the same (or joinable) source vectors and merge them together before propagating them to the uses.

In straight-line programs, any use has only one definition. However, one definition can be linked to multiple different uses. Thus, when we propagate data permutations from uses to definitions, the algorithm has to check whether all permutations propagated from the uses are consistent or not. If this is not the case, the propagation ends.

7.2.1 Reshaping Permutations

When a pair of partial uses of a permutation are operands of a commutative operation, it may be possible to reshape the permutation.

```
t[0:7] = Permute(a[0:7], ⟨0,5,2,7,4,1,6,3⟩);
b[0:3] = t[0:3] + t[4:7];
```

For example, consider the code sequence above, which is same as (1). The permutation, $P : \langle 0, 5, 2, 7, 4, 1, 6, 3\rangle$, cannot be propagated to its partial uses. However, because the two

partial uses are operands of a commutative operation (add), we can reshape the permutation pattern in the definition to $\langle 0, 1, 2, 3, 4, 5, 6, 7\rangle$ by exchanging corresponding elements ($a[5] \leftrightarrow a[1]$ and $a[7] \leftrightarrow a[3]$) of the two operands.

```
t[0:7] = Permute(a[0:7], ⟨0,1,2,3,4,5,6,7⟩);
b[0:3] = t[0:3] + t[4:7];
```

The reshaped permutation does not affect the value of `b[0:3]`, which is still $\langle a[0] + a[4], a[1]+a[5], a[2]+a[6], a[3]+a[7]\rangle$, and does not cross the partial use boundaries. In fact, in this example, P is reshaped to an identity permutation, thus can be eliminated completely as follows,

```
b[0:3] = a[0:3] + a[4:7];
```

Figure 7.2 gives the general algorithm for reshaping permutations. Vectors `v1` and `v2` are operands of a commutative operator as well as partial uses of vector `v` defined by permutation `P`. In the algorithm, the symbol "||" represents vector concatenation. After sorting all elements in `v1` and `v2` based on their original position in `x`, the loop at line 7 checks whether the elements in `v1` and `v2` come from two contiguous aligned sections of `x` of the size of `v1`(`v2`). This is a necessary condition for reshaping so that by switching elements one may move all elements from one contiguous aligned section of `x` to `v1` and the others to `v2`. The second loop reshapes the permutations. If a pair of corresponding elements from `v1` and `v2` comes from the same section of `x`, such as `b < Q[v1.begin+i] AND b < Q[v2.begin+i]`, there must be another pair of corresponding elements from the same section that is different from this pair. Thus, it becomes impossible to put all elements from the same section together in `v1` or `v2` and reshaping fails.

7.2.2 Permutation Decomposition

This section introduces two techniques to decompose a costly non-propagatable permutation into two permutations. One fast and the other propagatable.

```
reshape_permutation(use v1, use v2, stmt "v=Permute(x,P)")
1.   v1 ≡ v[v1.begin:v1.end]
2.   v2 ≡ v[v2.begin:v2.end]
3.   a[0:2*v1.length-1] ← P[v1.begin:v1.end]||P[v2.begin:v2.end]
4.   sort elements in a;
5.   e ← a[0] + v1.length;
6.   b ← a[2*v1.length-1] - v1.length
7.   FOR i = 0 TO 2*v1.length-1 DO
8.     IF e <= a[i] <= b THEN RETURN P;
9.   END
10.  Q ← P;
11.  FOR i = 0 TO v1.length-1 DO
12.    IF (b < Q[v1.begin+i] AND b < Q[v2.begin+i]) OR
          (Q[v1.begin+i] < e AND Q[v2.begin+i] < e) THEN
13.      RETURN P;
14.    ENDIF
15.    IF b < Q[v1.begin+i] AND Q[v2.begin+i] < e THEN
16.      Q[v1.begin+i] ↔ Q[v2.begin+i];
17.    ENDIF
18.  END
19.  RETURN Q;
```

Figure 7.2: The algorithm of reshaping permutations.

Register-wise Permutations

Certain permutations can be translated into register assignments, which in turn may be folded by copy propagation, thus do not incur any permutation overhead. We referred to such permutations as *register-wise permutations*. For example, $\langle 0,1,4,5,2,3,6,7\rangle$ is a register-wise permutation for two-element SIMD registers.

More specifically, considering the following *Permute* statement,

```
y[0:7] = Permute(x[0:7], ⟨0, 1, 4, 5, 2, 3, 6, 7⟩);
```

After code generation, if we assume that registers `X0` to `X3` are allocated for `x[0:7]` and `Y0` to `Y3` are allocated for `y[0:7]`, the SIMD code will be

```
Y0 = X0; Y1 = X2; Y2 = X1; Y3 = X3;
```

which includes only register-copy instructions. With the help of copy propagation, those assignments can be eliminated eventually.

Therefore, if a permutation cannot be propagated through the partial use boundaries, we can attempt to decompose it into two permutations. One is a register-wise permutation. The other can be partitioned and then propagated to the partial uses.

```
t[0:7] = Permute(a[0:7], ⟨1,0,5,4,3,2,7,6⟩);
b[0:3] = t[0:3] + t[4:7];
```

Consider the above code sequence, which is again same as (1). The permutation, P : $\langle 1,0,5,4,3,2,7,6\rangle$, crosses partial use boundaries and cannot be propagated. However, we can decompose P into,

$$P_2 \cdot P_1 = \langle 1,0,3,2,5,4,7,6\rangle \cdot \langle 0,1,4,5,2,3,6,7\rangle$$

where P_1 is a register-wise permutation (for 2-element SIMD registers), and P_2 is bisectable and therefore can be propagated into partial use. The final code after decomposition is,

```
t1[0:7] = Permute(a[0:7], ⟨1,0,5,4,3,2,7,6⟩);
t2[0:3] = t1[0:3] + t1[4:7];
b[0:3] = Permute(t2[0:3], ⟨1,0,3,2⟩);
```

Function `decompose_permutation_reg` in Figure 7.3 gives the permutation decomposition algorithm. Vector `v1` is a partial use of vector `v` defined by permutation `P`. In the function, vector `a` records the original positions in vector `x` for all elements of vector `v`. The first loop at line 4 checks whether a permutation is decomposable so that one of the resulting permutation is a register-wise permutation and the other is propagatable. To be specific, it checks whether all elements in each r-element section in vector `v` come from the same r-element section in vector `x`. If the permutation passes the check, an identity permutation, `P'`, is created to record the register-wise permutation. Then the second loop at line 10 does the decomposition by switching elements between two r-element sections. After that, `P'` becomes a register-wise permutation and is used to replace the original permutation, `P`. The propagatable permutation, `P"`, is computed in line 15, based on `P = P"·P'`, and returned by the function.

Platform-dependent decomposition

For some SIMD devices, a general permutation instruction must be translated into multiple native instructions. Consider a permutation $\langle 0,4,2,6\rangle$ assuming that the physical vector register is 4-way, which means that each register can hold 4 data elements. On SSE, this permutation must be implemented by two shuffle instructions (see Section 8.2). One moves elements from the first input register to the low-half of the output, and elements from the second to the high-half. In other words, the first SSE shuffle instruction performs the permutation $\langle 0,2,4,6\rangle$. To complete the translation, a second SSE shuffle instruction implementing the permutation $\langle 0,2,1,3\rangle$ must be generated.

Notice that the second instruction is an *intra-register* shuffle, which only reorganize data elements within a register and thus is always propagatable. Therefore, it is helpful to de-

```
decompose_permutation_reg(use v1, stmt "v=Permute(x,P)")
1.   a[0:v.length-1] ← P[v.begin:v.end]
2.   sort elements in a
3.   Let r be the element size of each vector register
4.   FOR i = 0 TO v.length/r-1 THEN
5.     IF NOT exist j, j*r <= a[i*r:i*r+r-1] < j*r+r THEN
6.       RETURN (P,I);                  // Not decomposable
7.     ENDIF
8.   END
9.   P' ← I, an identity permutation;
10.  FOR i = 0 TO v.length/r-1 THEN
11.    IF a[i*r] ≠ v.begin+i*r THEN
12.      P'[i*r:i*r+r-1] ↔ P'[v.begin+i*r:v.begin+i*r+r-1]
13.    ENDIF
14.  END
15.  RETURN (P*INV(P'), P')     // INV computes the inverse
```

Figure 7.3: The algorithm of register-wise permutation decomposition.

compose those permutations, which results in shuffles not supported natively, into two permutations so that the second permutation includes only intra-register data movements.

$P = \langle 0, 4, 2, 6, 1, 5, 3, 7 \rangle$ will be translated to 4 SSE shuffle instructions, two of which are intra-register shuffles (Section 8.2). Considering the following *Permute* operation,

```
y[0:7] = Permute(x[0:7], ⟨0, 4, 2, 6, 1, 5, 3, 7⟩);
```

Assume that two 4-element registers, `X0` and `X1`, are allocated for `x[0:7` and similarly `Y0` and `Y1` for `y[0:7]`. The SSE2 instructions needed to implement this statement are,

```
1. T0 = shufps(X0, X1, ⟨0, 2, 0, 2⟩);
2. Y0 = shufps(T0, T0, ⟨0, 2, 1, 3⟩);
3. T1 = shufps(X0, X1, ⟨1, 3, 1, 3⟩);
4. Y1 = shufps(T1, T1, ⟨0, 2, 1, 3⟩);
```

where `T0` and `T1` are two temporary register variables allocated for intermediate results and `shufps` is a shuffle instruction supported by SSE2 (see Section 2.3). Statement 2 and 4 are intra-register shuffles.

```
decompose_permutation_shuf(use v1, stmt "v=Permute(x,P)")
1.   P’ ← I, an identity permutation
2.   FOR i = 0 TO v.length/r-1 THEN
3.     insts ← code_gen(P[v.begin+i*r:v.begin+i*r+r-1])
4.     IF last shuffle in insts is an intra-register one THEN
5.       P’[v.begin+i*r:v.begin+i*r+r-1] = shuffle pattern
6.     END
7.   END
8.   RETURN (P’, INV(P’)*P)    // INV computes the inverse
```

Figure 7.4: The algorithm of platform-dependent permutation decomposition.

Thus, we can decompose P into $P_1 \cdot P_2$, with $P1 = \langle 0,2,4,6,1,3,5,7 \rangle$ and $P_2 = \langle 0,2,1,3,4,6,5,7 \rangle$. P_1 can be translated into 2 shuffle instructions and P_2 can be propagated to the uses and further merged with the other permutations.

```
t1[0:7] = Permute(a[0:7], ⟨0,2,4,6,1,3,5,7⟩);
t2[0:3] = t1[0:3] + t1[4:7];
b[0:3] = Permute(t2[0:3], ⟨0, 2, 1, 3⟩);
```

Function `decompose_permutation_shuf` in Figure 7.4 shows the algorithm for this transformation. The algorithm makes use of the code generation algorithm discussed in Section 8.2 to compute the `shuffle pattern`. Unlike `decompose_permutation_reg`, this function computes `P’` to propagate and then use it to calculate `P"` to replace `P`.

7.3 Propagation within a Statement

After a permutation is propagated, the algorithm consolidates consecutive permutations in the statement (composition rule), and, if needed, transforms the statement into the form `v = Permute(...)` (distributive rule) to enable further propagation.

Without loss of generality, we limit our focus to three-address statements with vector expressions. Let $v = Permute(x, P)$ be the permutation to be propagated. The expression containing the use of `v` will be transformed bottom up. Let n be an expression node and *op*

any regular element-wise operation.

The following transfer functions specify how to move *Permute* in an expression tree depending on the type of node n:

- $n \equiv Permute(v, Q)$. Apply the composition rule to obtain

$$n \equiv Permute(x, P \cdot Q).$$

- $n \equiv op(v)$. Move the permutation out side op to obtain

$$n \equiv Permute(op(x), P).$$

- $n \equiv op(v, e)$. Consider the following cases:

    1. If $e \equiv Permute(r, P)$ we move the permutation outside op to obtain

    $$n \equiv Permute(op(x, r), P)$$

    2. If $e \equiv C$ where C is a constant vector move the permutation outside op to obtain

    $$n \equiv Permute(op(x, C'), P)$$

    where C' is a constant vector obtained by applying the reverse permutation on C, as $C = Permute(C', P)$. Constant vectors can be permuted at compile time.

    3. If e is neither of these values, we wait for the propagation to e to finish. If this changes e to $Permute(r, P)$, we will be able to continue the propagation.

If the algorithm finds a node that does not match one the these cases, it stops and returns.

7.4 Optimality Analysis

As shown in Figure 7.1, there are three key operations in the optimization algorithm. The composition (at line 5 in Figure 7.1) always reduces permutations. The propagation within a statement (at line 8 in Figure 7.1), or the application of the distributive rule, sometimes reduces permutations. However, the propagation statement at line 3 might introduce more permutations if there are multiple uses. Thus, does the algorithm always produce the minimum number of data permutations?

Unfortunately, there is no deterministic algorithm that can find the global optimal for an arbitrary basic block in polynomial time, unless P = NP. The problem of finding the minimum number of permutations for statements in a basic block can be mapped to a multi-terminal cut problem introduced in [14], if we assume there is no partial use boundary and all data permutations cost same. The multi-terminal cut problem is an NP-complete problem unless the graph is a tree or there are only two terminals in the graph [14].

7.4.1 NP-Completeness of the Problem

The problem of optimizing data permutations in a straight-line vector code, called *MIN-Permute* problem, can be formally defined as follows,

Definition 1 *Given a straight-line vector program, where some of the statements are data permutations, find the minimum number of data permutations needed for the program.*

To simplify the problem, a few assumptions are made on the vector statements in the straight-line program.

1. There is no partial use or partial definition. All data permutations can be freely propagated along all def-use links within the basic block.

2. There is no data permutation in the middle of the basic block. In the other words, all data permutations are either source or sink nodes in the data-flow graph representation

Figure 7.5: Three NP-Complete problems.

of the basic block.

3. There is no vertex in the data-flow graph that has the degree larger than 3. That means any node in the graph is connected to at most two other nodes.

Notice that the optimization problem with these assumptions is actually a special case of the general optimization problem. However, as to be proved as following, this simplified problem is already NP-complete.

The NP-completeness can be proofed by creating a mapping from the MIN-Permute problem to the *multi-terminal cut* problem, which is a known NP-complete problem [14].

Definition 2 *Given a graph $G = (V, E)$, a set of $S = s_1, s_2, ..., s_k$ of k specified vertices (or terminals), and a positive weight $w(e)$ for each edge $e \in E$, find a minimum weight set of edges $E' \subseteq E$ such that the removal of E' from E disconnects each terminals from all the others.*

Instead of creating a direct mapping between these two problems, we define a graph-coloring problem and use it to map the MIN-Permute problem and the multi-terminal cut problem. The graph-coloring problem, called *Min-Color-Edge* problem, is defined as,

Definition 3 *Given a graph $G = (V, E)$, a set of $S = s_1, s_2, ..., s_k$ of k specified vertices (or virtual nodes), and a partial coloring function $c(v)$ for each $v \in S$, find an complete coloring function c' so that $c(v) = c'(v)$ for each $v \in S$ and the number of edges that connect nodes with different colors is minimum.*

By creating a mapping from the MIN-Permute problem to the Min-Color-Edge problem and then a mapping from the Min-Color-Edge problem to the multi-terminal problem, it can

be proofed that three problems are equivalent so that the MIN-Permute is a NP-complete problem (Figure 7.5).

First, the vector program is represented by an extended data-flow graph. Like regular data-flow graphs, leaf nodes in the extended data-flow graph represents vector variables that are defined or used outside and internal nodes are used to represent different vector operations (or the local variables produced by the operations). The edges show the relationship between variables and operations. However, since data permutations can be propagated along both directions of def-use links (Assumption 1), the edges are undirected. That is different from most regular data-flow graphs.

In addition, each vertex includes an important property to record its permutation pattern. Except for those variables that are defined by data permutations, all variables are marked as *identity permutation*. Therefore, in the extended data-flow graph, data permutations are defined as the transactions between two nodes with different permutation patterns, instead of being represented as internal nodes like other vector operations.

In addition, a few virtual nodes are added into the graph. Each virtual node represents a specific permutation pattern in the program. Additional edges are also added in the graph to connect the virtual node to the variable nodes marked with the same permutation pattern. Also, a special virtual node is introduced to represent the identity permutation and connected to those leaf nodes marked with the identity permutation pattern. After introducing these virtual nodes, all nodes in the original graph are remarked with the identity permutation pattern.

After representing the vector program as such an extended data-flow graph, the next task is to create a one-to-one mapping between data permutation operations and edges connecting two nodes with different patterns. Based on Assumption 3, which limits the degree of any vertex (except for virtual nodes) to be no larger than 3, such one-to-one mapping can be proofed as follows,

Proof 1 *Assume a node p connects to three nodes, q, r and s. In an optimal solution, p*

must have the same permutation pattern as the dominant one among q, r and s. Otherwise, the permutation pattern of p can be changed to the dominant one and the less number of data permutations are needed. Thus,

1. *If there are three different permutation patterns among q, r and s, p can choose any one of three patterns. Assume p uses the same pattern as q. Then two data permutations are needed between p and r and between p and s. On the other hand, there are two edges, $p - r$ and $p - s$, connecting nodes with different permutation patterns. Thus, two numbers are same.*

2. *If there are two different permutation patterns among q, r and s, p will use the dominant one. Assume q and r share the same pattern so that p uses this pattern too. Then, only one data permutation is needed between p and s. There is also only one edge, $p-s$, that connects nodes with different permutation patterns. Again, two numbers are same.*

3. *If there is only one permutation pattern associated with q, r and s, p will choose the same pattern. Finally, there is no data permutation needed and no edge connecting two nodes with different patterns. Again, two numbers are same.*

For the vertice whose degree is less than 3, similar proofs can be derived as the above one. Notice that the limit on the vertex degree is necessary to create such a mapping. Considering the following code example,

```
......
v3 = v1 + v2;
v4 = ... v3 ...;
v5 = ... v3 ...;
......
```

In the data-flow graph, the vertex degree of vector `v3` (or add operation) is 4. Assume vector `v1` and vector `v2` are marked with permutation pattern `p1`, vector `v4` and vector `v5`

are marked with pattern `p2`. Then no matter whether vector `v3` is marked as `p1` or `p2`, there will be two edges connecting nodes with different permutation patterns in the graph. However, the program needs only one permutation operation to convert vector `v3` from `p1` to `p2`. These two numbers are not same any more.

The last step of mapping is straightforward. We can use different colors to represent different permutations patterns. After that, the MIN-Permute problem is mapped to the Min-Color-Trans problem.

The mapping between the Min-Color-Trans problem and the multi-terminal cut problem is more straightforward. If we treat these k specified nodes as terminals, the Min-Color-Trans problem is actually a special case of multi-terminal cut problem where $w(e) = 1$ for all $e \in E$, which is also an NP-complete problem [14].

By creating a mapping to the multi-terminal cut problem, we have proofed that a simplified MIN-Permute problem is NP-complete. The general MIN-Permute problem must be at least NP-complete.

7.4.2 Heuristic Solutions

Since the problem of finding the minimum number of data permutations in a straight-line vector program is NP-complete, several heuristics are included in the optimization algorithm.

First, only if the propagated permutation to a use can be merged with another permutation, the permutation will be propagated to the use. Otherwise, no propagation will be conducted. By proceeding conservatively in this way, we guarantee that the number of permutations decreases monotonically.

Second, the above algorithm uses a top-down propagation strategy. An alternative way is to propagate the permutations bottom-up by following use-def chains. The reverse direction of propagation can sometimes create more opportunities for composition. In fact, propagating in different directions may produce entirely different permutations for the same program. In our algorithm, we propagate in both directions to obtain the final permutation

results, which is always better than one-direction propagation.

7.5 The Global Optimization Algorithm

The optimization algorithm proposed in the previous sections limits its scope to straight-line code only. In this section, we discuss how to extend it to deal with if-statements (Section 7.5.1) and loop structures (Section 7.5.2 and 7.5.3).

7.5.1 Extension on If-Statements

In SIMD compilation, there are two different ways of handling if-statements in vector programs. First, if-statement can be converted into conditional statements by applying *if-conversion*, a common compiler technique that was developed for vector processors [35]. With the help of if-conversion, several basic blocks that are separated by a if-statement can be merged into a larger basic block. At the same time, a condition mask will be generated and used to merge results from different branches together. Considering a simple example as follows,

```
if( cond )
  b[0:15] = Permute(a[0:15], P1);
else
  b[0:15] = Permute(a[0:15], P2);
```

After if-conversion, the code becomes,

```
mask[0:15] = Gen_Mask(cond);
t1[0:15] = Permute(a[0:15], P1);
t2[0:15] = Permute(a[0:15], P2);
b[0:15] = Merge(t1[0:15], t2[0:15], mask[0:15]);
```

where `Gen_Mask` is a simple function to generate the vector mask from the condition, which is supported by most SIMD devices, and `Merge` is a vector operation to merge two vectors together with the guidance of the mask. In essence, `Merge` is a data reorganization operation itself. There must be some opportunities to optimize `Merge` operations with other data permutations, if we could extend the representation of data permutations.

However, as shown in this example, both permutation operations will be executed after if-conversion. It is one of major drawbacks of if-conversion. Thus, for small-scale SIMD devices, if-conversion works only on those if-statements whose branches are small enough. It is oftentimes more efficient to keep if-statement unchanged.

In SSA (Static Single Assignment) representation, if-statements will introduce phi-node to merge data from different branches [50]. By treating phi-operation as an element-wise vector operations, the optimization algorithm can be directly extended. Data permutations can also be propagated through phi-statements by applying the distribution rule on phi-operations. Unlike most other element-wise operations, the application of the distribution rule on phi-statements will not reduce data permutations. However, it can still create more opportunities to apply the composition rule. Considering the following example, which is already in SSA form,

```
if( cond ) {
  ...
  b1[0:15] = Permute(a[0:15], P1);
  c1[0:15] = b1[0:15] * T[0:15];
}
else {
  ...
  b2[0:15] = Permute(a[0:15], P1);
  c2[0:15] = b2[0:15] + T[0:15];
}
```

```
c[0:15] = phi(c1[0:15], c2[0:15]);
```

The global optimization algorithm will first propagate two permutations across the add and the multiplication respectively. If two permutations have the same permutation pattern, which means $P1 = P2$, the permutation can be propagated from the right-hand side of the phi-statement to the left-hand side. The propagation will continue along the def-use links of `c[0:15]`.

7.5.2 Extension on Loops without Backward Dependences

In the DSP programs generated by the SPIRAL systems, all loops have no backward dependence. In addition, each iteration of the loop is independent from the others. Thus, the extension on such loops is different from the one on general loops with backward dependences. The following example is the normalized version of the 8-point FFT program shown in Figure 1.1.

```
1.  t1[0:3] = x[0:3:1] + x[4:7];
2.  t1[4:7] = x[0:3:1] - x[4:7];
3.  v0[0:7] = Permute(t1[0:7], P1);
4.  v1[0:7] = T8[0:7] * v0[0:7];
5.  t2[0:7] = Permute(v1[0:7], P2);
6.  for (i0 = 0; i0 < 2; i0++) {
7.    t3[0:1] = t2[4*i0:4*i0+1] + t2[4*i0+2:4*i0+3];
8.    t3[2:3] = t2[4*i0:4*i0+1] - t2[4*i0+2:4*i0+3];
9.    v2[0:3] = Permute(t3[0:3], P3);
10.   v3[0:3] = T4[0:3] * v2[0:3];
11.   t4[4*i0+0:4*i0+1] = v3[0:1] + v3[2:3];
12.   t4[4*i0+2:4*i0+3] = v3[0:1] - v3[2:3];
13. }
14. y[0:7] = Permute(t4[0:7:1], P4);
```

Figure 7.6: Normalized version of the 8-point FFT code in Figure 1.1.

First, the algorithm is extended to propagate data permutations across the loop boundary. Instead of stopping propagation when the data permutation is used in a different loop, the global algorithm continues propagating permutations out to the outer loop or into the

inner loop. In general, it is easier to propagate a data permutation from the inner loop to the outer loop. Such propagation can be conducting by applying loop unrolling and vector coalescing on the data permutation being propagated. For the example shown in Figure 7.6, if the data permutation at 9 can be propagated across statements 10, 11 and 12, the algorithm will propagate it out of the loop and then merge with the data permutation at 14.

On the other hand, it is more difficult to propagate a data permutation from the outer loop to the inner loop. The problem of partial use boundary must be addressed when propagating data permutation from the outer loop to the inner loop. For example, if we want to propagate the data permutation at 5 into the loop, we have to check whether it can be propagated through the partial use boundaries of `t2[4*i0:4*i0+1]` and `t2[4*i0+2:4*i0+3]`. In fact, the situation is even worse since some optimization techniques introduced in Section 7.2 might not be applicable when loop exists. For example, it might not be profitable any more to decompose a non-propagatable permutation into a propagatable permutation and a register-wise permutation, since an isolated register-wise permutation still costs a lot of memory operations.

In addition, since the optimization algorithm is based on the def-use information of vector programs, the def-use analysis must also be extended to create def-use and use-def links across loop boundaries. More details are discussed in Section 10.1.

We have implemented the extended algorithm on the HiLO compiler. However, the experimental results indicates that it might not be efficient to propagate data permutation across loop boundaries for the overall performance of the program (see Section 13.4). It can result in more memory loads and stores, which might cancel the performance benefits from reduced permutation instructions.

Hence, in the current implementation, we limits the propagation of data permutations within loop boundaries. It preserves the data flow structure of the original vector programs. As the cost of doing that, more permutation instructions and probably some partially-utilized loads (stores) are needed in the program.

7.5.3 Extension on Loops with Backward Dependences

Like if-statements, loops with backward dependences also introduces phi-statements in the SSA representation. A straightforward extension of the optimization algorithm on such loops is to propagate data permutation along backward def-use links. For simplicity, we assume that the use of a backward def-use link is always a phi-node. When the algorithm first propagates a data permutation to a phi-node, it assumes that the backward link will returns the same permutation so that the permutation can be propagated through. When the algorithm visits the same phi-node again from the backward links, it simply checks whether the propagated permutation is same as the one being propagated through in the first visit. If that is not the case, the propagation (of the second visit) ends.

A more complicated solution is to use the data-flow analysis framework [50]. By recording all possible permutation patterns for each vector and propagating such information along the data-flow graph, it might result in less data permutations than the first method.

In the current implementation of VINCI, neither if-statements nor loops with backward dependences is supported. Thus, one of the next steps is to introduce if-statements in I-code and handle loops with backward dependences (on vectors) in HiLO.

7.6 Optimizing Special Permutations

In the optimization algorithm, the special permutations, such as reduction and replication, need to be handled differently. For example, a reduction can be merged with a permutation propagated to its input. However, a reduction cannot be used as the starting point of the propagation. Similarly, a replication operation can absorb permutations propagated along use-def links but will not be propagated.

Although it is feasible to propagate permutations with $\star$ elements as other permutations, it might introduce more computation. For example, if half or more elements in a permutation are $\star$, it might not be beneficial to propagate it.

On the other hand, since permutations with $\diamond$ elements essentially combine two vectors together, it becomes extremely complicated to propagate them. Thus, in our algorithm, any permutation with $\diamond$ elements will not be propagated. But like reduction and replication, permutations including $\diamond$ elements can still be merged with other permutations propagated to them.

7.7 Related Work on Data Permutation Optimization

Memory alignment is a common source of data permutations and has been studied extensively. Early work on alignment handling [5, 10, 42] apply loop peeling and versioning to translate computations where misaligned references are relatively aligned to each other. Therefore, those schemes do not generate any permutations. Recent work in [15, 69] handle arbitrary memory alignment while minimizing permutation overhead. When dealing with misalignment within a statement, our conversion algorithm is equivalent to the zero-shift policy in [15], and the lazy-shift and dominant-shift policies in [15] can be derived by applying distributive and composition rules on permute. Compared to [15, 69], our algorithm is more powerful in terms of minimizing alignment overhead across statements thanks to our propagation algorithm. On the other hand, [15, 69] handles arbitrary alignment, whereas ours handles only compile-time alignment. Another major difference is that [15, 69] target loops while our algorithm is for straight-line code. Thus, the code generation for permutation is quite different.

There are a few recent studies on generating efficient permutation instructions for SIMD devices [38, 52, 54]. In [38], an algorithm was introduced to generate permutation instructions for SIMD devices. Despite having similar workflow, their algorithm and ours work at different levels of intermediate representation. In [52], an algorithm was proposed to automatically generate permutation instructions for a new language, StreamIt, and output platform, VIRAM. In [54], an extension on GCC vectorizer was introduced to represent

data references with non-unit strides and generate efficient permutation instructions for them. Both [52] and [54] focus on permutation representation and code generation, instead of optimizing data permutations. The optimization algorithm introduced in this chapter is discussed with less detail in [58].

In the domain of signal processing, an optimization algorithm on permutation matrices at the formula level was introduced as a key step of vectorization for SIMD devices [17]. In [17, 21], domain-specific techniques are proposed to generate efficient SIMD code for complex computation in DSP programs. With the support of indirect register accesses in a DSP processor, a different scheme of handling data permutations was discussed in [51].

There were several other interesting studies on more general definition of data permutations. In [63], permutation matrices are used to optimize bit-wise operations in StreamIt programs. Our strategy to optimize element-wise permutations is similar to theirs. Despite targeting distributed memory systems, the algorithm presented in [9] for array alignment can also be extended as an alternative of our optimizing algorithm. Our composition rule for data permutations is similar to the idea of synthesizing array operations in [26].

Chapter 8

SIMD Code Generation

The last step of the compilation framework is to translate vector statements into SIMD instructions. The code generation algorithm is introduced in this chapter. Section 8.1 describes the general algorithm of generating SIMD instructions for vector programs. Section 8.2 introduces an efficient algorithm to generate native permutation instructions for SIMD devices.

8.1 The Code Generation Algorithm

The translation from vector statements into SIMD instructions can be divided into two steps. First, long vectors are strip-mined into short vectors with the same bit-length, which is determined by the target device. For example, vectors of 32-bit single floating point numbers are divided into 4-element vectors for 128-bit SIMD devices. Since all vectors have been normalized by the preceding passes, such kind of strip-mining is straightforward.

Second, these fixed-length vectors are replaced with virtual register variables for SIMD devices. Most native compilers of SIMD devices allow to declare SIMD variable directly. Such transformation is similar to scalar replacement, a common compiler optimization to replace subscript variables with scalars. In this step, all vectors referencing temporary arrays are completely replaced with SIMD variables. The other vectors, which reference globally visible arrays, are also replaced with SIMD variable but additional SIMD load/store instructions are inserted to the program.

As short vectors are replaced with SIMD variables, vector operations are also trans-

lated into SIMD intrinsic functions. Most element-wise vector operations can be mapped into native instructions directly supported by SIMD devices. A few other operations are implemented with several instructions. For example, *negative* operation on floating point numbers can be translated into a pair of constant-load instruction and subtract instruction. In fact, the major difficulty in this step is to generate efficient SIMD instructions from data permutation operations.

When the length of long vectors are not multiple of the degree of parallelism of SIMD devices, there are two alternative solutions for the elements left after strip-mining. One is to generate SIMD instructions to partially utilize data parallelism. The other is to generate scalar code directly. Although the first solution might need fewer instructions for the computation, it might require more instructions to maintain correct data. For example, if the device does not support partial store, additional load and permutation instructions are needed to write partial results into memory. In addition, as discussed in Chapter 9, scalar instructions can be executed simultaneously with SIMD instructions on some devices so that more scalar instructions might not cost more cycles. In general, it depends specific features of the target device to choose which solution.

8.2 Generating SIMD Permutation Instructions

This section describes the algorithm that translates *Permute* operations into target machine instructions (see Section 2.3). The algorithm has two steps:

1. Translates each *Permute* operation into a sequence of `vperm` operations (Section 8.2.1).

2. Maps `vperm` operations into native data-movement instructions (Section 8.2.2).

8.2.1 Translating *Permute* into *vperm* Instructions

We first translate generic permutation operations to an internal abstract instruction similar to the `vperm` instruction (Figure 2.2) of VMX. We use *vperm* because it closely resembles the format of actual data permutation instructions. The format of our *vperm* operation is:

$$R_3 = vperm(R_1, R_2, P)$$

where, R_1, R_2 and R_3 are three *virtual vector registers* of length L_r, the size of physical register of the target SIMD device, and P is a vector literal of the same size.

When translating normalized vector programs, each vector will be mapped to a set of virtual vector registers. Since all vectors are stride-one, aligned and of length that is a multiple of L_r, the mapping is straightforward. It is achieved by strip-mining a long vector into chunks and allocating a virtual vector register for each chunk. For example, assuming that a vector register is 4 element long, `u[0:15]` will be strip-mined into 4 chunks and mapped to 4 virtual vector registers. Since the mapping between vector chunks and virtual registers is one to one, we use vector expressions to represent mapped virtual vector registers. For example, we use `u[0:3]` to represent the first virtual register allocated for `u[0:15]`.

For a generic permutation, $y = Permute(x, P)$, the goal of our algorithm is to generate as few *vperm* operations as possible to implement the permutation specified by P. No fast optimal algorithm is known for this problem.

Consider a single output register whose elements come from N different input registers. Since each *vperm* operation can only collect elements from two registers to one, at least $N - 1$ instructions are needed to generate the output. However, the bandwidth of those *vperm* operations may not be fully utilized (when $N > 2$), thus there might be some *unused* slots in the virtual registers used to hold intermediate results. In the hope that those slots can be used in the construction of other output registers, our algorithm always maximizes the number of unused slots during the process of building one output register.

To illustrate these ideas, consider the following statement:

$$\texttt{v[0:15]=Permute(u[0:15], P)} \tag{2}$$

where P=$\langle 0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15 \rangle$. Assume that a vector register is 4 data elements long. During code generation, `v[0:15]` will be mapped to 4 virtual vector registers. To compute the first output vector, i.e., `v[0:3]`, three `vperm` operations are needed to collect elements `u[0]`, `u[4]`, `u[8]`, `u[12]`, as shown in Figure 8.1.

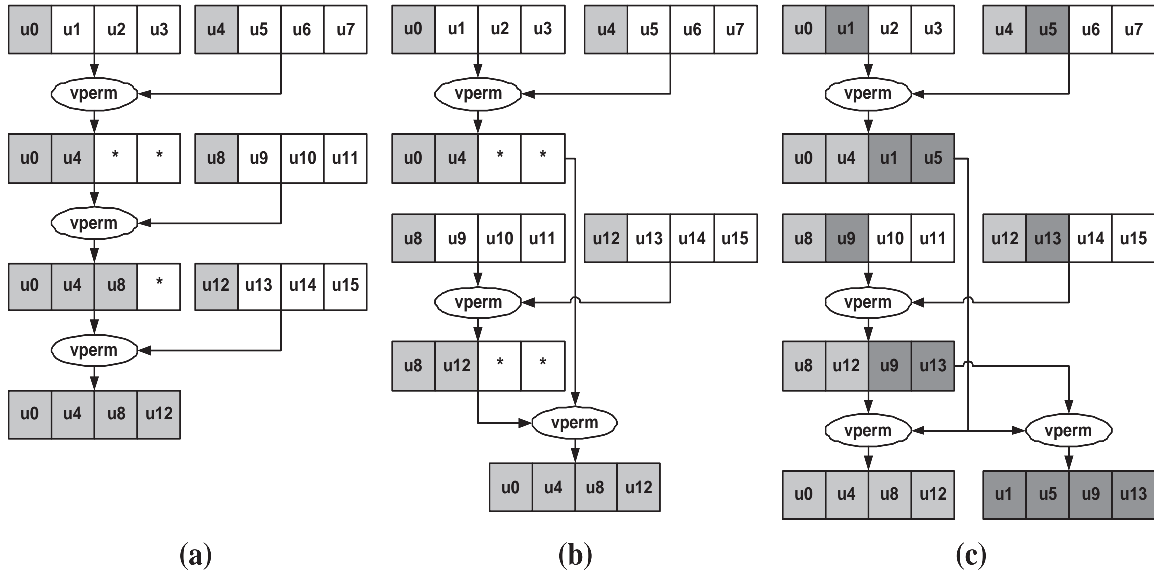

Figure 8.1: Different code generation patterns for `v[0:3]` in (2).

Both (a) and (b) in Figure 8.1 are feasible solutions to generate the result, `v[0:3]`, using three *vperm* operations. However, in (a), `u[0]` and `u[4]` are moved three times, `u[8]` twice and `u[12]` once. Thus there are total 9 element movements in (a). On the other hand, in (b), each element is moved twice and the total number is 8. Thus (b) requires fewer data movements than (a). With the same number of *vperm* operations, (b) has more unused slots than (a).

In order to minimize the number of element movements for one output register, $\hat{y}$, of a permutation, $y = Permute(x, P)$, the algorithm proceeds as follows:

1. Create a set `W` containing all the virtual registers containing elements that will be moved to $\hat{y}$.

2. Select two registers R_1 and R_2 with the fewest elements that will go into $\hat{y}$

3. Generate a *vperm* operation, $R_0 = vperm(R_1, R_2, P)$ to collect elements from these two registers, and replace R_1 and R_2 in `W` with R_0.

4. Repeat the last two steps until `W` contains a single virtual register, which happens to be the final output register.

The number of data movements is guaranteed to be minimum for the same reason that the Huffman encoding guarantees the minimum weighted path length in the resulting coding tree.

Once the sequence of *vperm* operations is determined, the next task is to maximize the utilization of each *vperm* instruction generated. As shown in Figure 8.1, (b) has four unused slots in the output of the first two *vperm* operations. If we use those slots, we need only one more *vperm* to produce the second output virtual register `v[4:7]`, as shown in Figure 8.1(c). Thus the algorithm always tries to find other pairs from two inputs that will finally go to the same output and use them to fill the unused slots in a *vperm* operation.

The general algorithm for translating $y = Permute(x, P)$ to *vperm* operations is shown in Figure 8.2. The algorithm also handles special elements, $\star$ and $\diamond$, in generic permutation. Firstly, if all elements in a register are $\star$ or $\diamond$, no permutation instruction is needed. Otherwise, $\star$ elements can be ignored completely during generating *vperm* operations. On the other hand, $\diamond$ elements will be replaced with the corresponding elements in the output register, when calculating `VR` in Figure 8.2.

```
generate_vperm(stmt "y = Permute(x, P)")
  Map x to the virtual registers $X_0$, $X_1$, ...
  Map y to the virtual registers $Y_0$, $Y_1$, ...
  Let VR[i] be $X_n$ containing element y[i](=x[P[i]])
            or $Y_m$ if P[i] is ◇ element
  Let Loc[i] be y[i]'s location in VR[i]
  Let r be the element size of the vector register
  FOR i = 0 TO y.length/r-1 DO
    W ← set of different values in VR[i*r:i*r+r-1]
    WHILE |W| > 1 DO
      n[R] ← |{j|VR[j]=R, i*r <= j < i*r+r}|
      Find $R_1$, $R_2$ from W that minimizes n[$R_1$]+n[$R_2$]
      IF n[$R_1$] + n[$R_2$] < r THEN
        Find pairs in $R_1$ and $R_2$ to the same output, $Y_k$
        Select pairs to fill the unused slots
      ENDIF
      Generate "$R_0$ = vperm($R_1$, $R_2$, P')"
      Update VR and Loc for the elements moved to $R_0$
      W ← set of different values in VR[i*r:i*r+r-1]
   END
  END
```

Figure 8.2: The algorithm of translating *Permute* to *vperm* operations.

8.2.2 Mapping *vperm* to Native Instructions

The last step in the translation process is to map the *vperm* operations into native data movement instructions. This is just a pattern matching problem. In the case of VMX, we first attempt to match the permute pattern to restricted shuffle instructions, such as *interleave* or *element-wise shift*. The reason to try these instructions first is that they are more efficient than `vperm` on VMX. If no such match is found, a `vperm` instruction is generated. For the SSE family, the process is the same except that several `shufps` instructions may be needed to implement a *vperm* operation.

To increase the likelihood that special permutations are generated, the `generate_vperm` function tries to select the appropriate permutations whenever an intermediate register is produced. Instead of designating fixed positions for elements in an intermediate result, the algorithm carefully places data elements so that the *vperm* operation for the intermediate result will be mapped to a native instruction directly. However, in the case of the *vperm* operation that generates the final output virtual register, there is in our strategy no room left to choose.

By following this strategy, the overhead introduced by the two-step code generation is minimized, if not removed completely. In our experiments, every *vperm* operation used to generate intermediate results are mapped to a single `shufps` or `interleave` instruction for SSE2. In addition, the permutation decomposition discussed in Section 7.2.2 helps to reduce the number of native instructions to generate final results.

8.2.3 Generating Reduction and Replication

Assume that reduction is associative, a vector reduction will be translated to two sequences of SIMD instructions. The first sequence conducts inter-register reduction by applying reduction operations on all input vector registers. The second sequence implements the reduction operation across elements of the result vector register. Unless the targeted SIMD

device provides intra-register reduction instruction for the reduction operation, a sequence of permutations and arithmetic instructions will be generated for intra-register reduction.

The code generation of *Spread* operation is straightforward, since most SIMD devices provide hardware instructions to replicate scalar data.

Chapter 9

Mixed-Mode SIMD Compilation

Due to inherent data dependences in sequential programs and/or architectural constraints of SIMD devices, it is not always possible to translate all scalar statements into SIMD instructions. Thus, it is an unavoidable problem for SIMD compilers to generate efficient code mixed with both scalar and SIMD instructions. This chapter proposes an extension of VINCI to deal with mixed code. Section 9.1 uses an example to motivate the problem of mixed code. Section 9.2 describes the mixed-mode extension on VINCI. Finally, several related issues are discussed in Section 9.3.

9.1 A Motivating Example

It is not always feasible to translate a scalar statement into SIMD instructions. Sometimes it is because of the inherent data dependence cycle on the statement. Sometimes it is because the operation in the statement is not supported by SIMD devices. It is common in real-world applications to see a loop that includes both some statements that can be translated into SIMD instructions and some statements that cannot be translated into SIMD instructions. Such loop is called as a *mixed* loop.

Consider the loop given in Figure 9.1. It is a simplified and scalar-expanded version of the kernel loop from `GSM` encoder. In the innermost i-loop, there is a dependence cycle from `d[i+1]` to `d[i]` at statement 4. But the other two statements (5 and 6) are both vectorizable. Thus, this loop is a mixed loop.

The state-of-the-art solution for mixed loops is to distribute statements into different

```
1.  for (; k--; s++) {
2.     d[0] = t[0] = *s;
3.     for (i = 0; i < 8; i++) {
4.        d[i+1] = d[i] + (rp[i] * u[i]);
5.        t[i+1] = u[i] + (rp[i] * d[i]);
6.        u[i]   = t[i];
7.     }
8.     *s = d[8];
9.  }
```

Figure 9.1: A simplified loop from GSM.

loops, called as *loop distribution* [35]. Although loop distribution was suitable for traditional vector machines, it faces new difficulties in today's SIMD compilation. If two statements are distributed into two loops, the reuse distance of the shared variables in two statements will increase dramatically. Using the above example, if 4 and 5 are distributed into two loops, the reuse distance of "d[i]" increases from 1 to 24 (=3*8). The situation can be even worse when the loop trip count is larger.

Therefore, loop distribution might result in poor data locality. Because of the memory hierarchy, it can finally result in more memory operations (fewer register reuses) or higher latencies for memory operations (fewer cache hits). In addition, smaller basic blocks is another drawback of loop distribution. In the experiment, it was observed that the performance benefits from SIMD computation oftentimes cannot comprise the performance loss due to poor memory performance.

Alternatively, a better solution is to generate scalar and SIMD code in a loop directly, instead of distributing them into different loops. An extension of VINCI was proposed to enable mixed-mode code generation.

9.2 Mixed-Mode Extension on VINCI

In general, there are two different kinds of statements that cannot be translated into SIMD instructions. The first kind is vectorizable but its operation is not supported by SIMD

devices, or not profitable to be executed on SIMD devices. For such kind of statements, since they can still naturally be represented as vector statements, only the code generation algorithm needs to be extended to generate mixed code. During code generation, those operations that are not supported by the target device are identified and then translated back to scalar instructions. Besides code generation, other optimizations on vectors can be applied on those statements without any modifications.

The other kind is completely non-vectorizable due to dependence cycles. For such kind of statements, the compilation framework is extended by introducing *pseudo vectors* to represent these non-vectorizable statements.

With the help of pseudo vectors, the code in Figure 9.1 can be vectorized, as shown in Figure 9.2. In Figure 9.2, the numbers in the comments to the right show the corresponding statements in the original loop. Note that the inner loop is completely replaced with straight line vector statements. With using function `Recur`, statement 3 is actually a pseudo vector statement. `Recur`(s, v_{in}, op) computes a vector v_{out} from vector v_{in}, scalar s, and operation op, as following,

$$v_{out}(0) = v_{in}(0)\ op\ s;$$
$$v_{out}(i) = v_{out}(i-1)\ op\ v_{in}(i), \forall i > 0$$

Like those vector statements whose operations are not supported by SIMD devices, these pseudo vector statements will be translated back to scalar statements during code generation. However, different from the other vector statements, the order of scalar statements generated from a pseudo vector statement is fixed. Thus, the organization of data elements in vectors cannot be changed by any program transformations and optimizations and data permutations cannot be propagated through these pseudo statements. Nonetheless, some other vector optimizations, such as copy propagation, is still applicable on pseudo vectors.

Finally, scalar code is generated from pseudo vector statements and those vector state-

```
1. for (; k--; s++) {
2.   d[0] = t[0] = *s;                       /* 2 */
3.   d[1:8] = Recur(d[0], rp[0:7] * u[0:7], +); /* 4 */
4.   t[1:8] = u[0:7] + rp[0:7] * d[0:7]; /* 5 */
5.   u[0:7] = t[0:7];                        /* 6 */
6.   *s = d[8];                              /* 8 */
7. }
```

Figure 9.2: Vectorized loop from Figure 9.1.

ments with unsupported operations and SIMD code is generated for the other vector statements. For some processors, addition instruction are inserted for explicit data communication between scalar variables and SIMD variables.

```
1.  t1[0:7] = rp[0:7] * u[0:7];
2.  d[1] = d[0] + t1[0];
3.  d[2] = d[1] + t1[1];
4.  d[3] = d[2] + t1[2];
5.  d[4] = d[3] + t1[3];
6.  d[5] = d[4] + t1[4];
7.  d[6] = d[5] + t1[5];
8.  d[7] = d[6] + t1[6];
9.  d[8] = d[7] + t1[7];
```

Figure 9.3: Mixed-mode vectorization of `Recur(d[0], vrp*vu,+)`.

Furthermore, notice that the statement 4 in Figure 9.1 is actually partially vectorizable. The multiply of $rp[i]$ and $u[i]$ can be vectorized. Thus, instead of translating the whole pseudo vector statement into scalar statements, the compiler translates this multiplication into SIMD instructions, as shown in Figure 9.3.

9.3 Discussion

Due to specific architectural features of SIMD devices and their host processors, there are other issues arising in generating mixed-mode code. First, many microprocessors have separated scalar units and SIMD units. Thus, it is more beneficial for these devices to mix scalar and SIMD instructions within a single loop so that all units can be fully utilized at the same

time. Second, there are some SIMD devices, such as SPU in Cell and SSE2 in Pentium 4, on which scalar operations are also executed on SIMD units. For these devices, it is beneficial to combine scalar and SIMD instructions in a single loop. Third, a few processors, such as PowerPC, do not support direct communication between scalar units and SIMD units. It is very expensive to exchange data between two units, since all data has to go through the memory system. Thus, the situation becomes more complicated for these processors. The compiler may choose to map vector code onto scalar units or vice versa, depending on the performance.

On the other hand, even there is a dependence cycle on the statement, sometimes it is still possible to partially vectorize it. For example, the dependence cycle between $d[i+1]$ and $d[i]$ in Figure 9.1 is actually a *linear recurrence*. There are known techniques to vectorize linear recurrences of this type, such as *parallel prefix* [40]. Figure 9.3 shows 8 scalar statements needed for sequential execution of the linear recurrence, where `LeftShift(`v, i`)` shifts left vector v by i elements. However, by using parallel prefix, only 4 SIMD statements are needed.

```
1.  t1[0:7] = rp[0:7] * u[0:7];
2.  t2[0:7] = LeftShift(t1[0:7], 1) + t1[0:7];
3.  t3[0:7] = LeftShift(t2[0:7], 2) + t2[0:7];
4.  t4[0:7] = LeftShift(t3[0:7], 4) + t3[0:7];
5.  d[1:8]  = Spread(d[7:0], 0) + t4[0:7];
```

Figure 9.4: Vectorization of `Recur(d[0], vrp*vu,+)` using parallel prefix.

However, we would like to point out that, in `GSM`, saturated add instead of modulo add is used in the linear recurrence. Although saturated add on signed integers is not associative in general, we can loosen the restriction based on programmer-provided directives and/or compiler-collected information. For two core procedures in `GSM`, parallel prefix adds 50% to the performance of the vectorized code and produces correct results for the given input data set.

Chapter 10

Other Compiler Routines in VINCI

In the HiLO compiler, where VINCI was implemented, there are many supportive routines provided for common program analyses and optimizations. Most of these routines must be extended to deal with vectors properly, which is usually nontrivial. Several such extensions are discussed in this chapter. Section 10.1 describes how to extend def-use analysis for vector programs. Section 10.2 shows new issues arising on vector copy propagations.

10.1 Def-use Analysis on Vector Program

In general, the def-use analysis of vector programs is an important issue. Unlike two scalars, which are either *same* or *different*, two vectors can be identical, completely different or *overlapped*. In the following example, each vector uses in the second statement, `t[0:3]` or `t[4:7]`, is only half of `t[0:7]`, the vector defined in the first statement. Thus, the def-use analysis has to deal with the partial relation between vectors.

```
1.  t[0:7] = ...;
2.  b[0:3] = t[0:3] + t[4:7];
```

Assume there are two vector expressions, $v_1 : a[b_1 : e_1]$ and $v_2 : a[b_2 : e_2]$, referencing the same array a. The relationship between two vectors can be either,

1. **Same**: $v_1 = v_2$, if $b_1 = b_2$ and $e_1 = e_2$;

2. **Different**: $v_1 \neq v_2$, if $b_1 > e_2$ or $b_2 > e_1$;

3. **Larger**: $v_1 \geq v_2$, if $b_1 \leq b_2$ and $e_1 \geq e_2$;

4. **Smaller**: $v_1 \leq v_2$, if $b_1 \geq b_2$ and $e_1 \leq e_2$;

5. **Overlapped**: $v_1 \approx v_2$, the others;

Although it is straightforward to compare two subscripts when they are both constant integers, more complicated methods, such as symbolic execution, are needed for those subscripts with loop-index variables. Thus, it is not always possible for compiler to determine the relationship between two vector expressions. In the current implementation, the sixth relationship, **Unknown**, is introduced and treated as same as **Overlapped**.

When traversing the program to build def-use links, the compiler has to keep the record of all live definitions and correctly update it whenever a new definition kills old definitions. Such task is relatively easy for scalar programs. However, in vector programs, it become more complicated to record partial kills. Considering the following code example,

```
1.  t[0:7] = ...;
2.  t[0:3] = ...;
3.  b[0:3] = t[0:3] + t[4:7];
4.  t[4:7] = ...;
```

Notice that the definition at 2 only partially kills the definition at 1. Thus, the two vector uses in 3 are defined by 1 and 2 respectively. Only after 4, the definition at 1 is completely killed by two definitions at 2 and 4.

10.2 Vector Copy Propagation

Not surprisingly, many compiler optimizations face the similar challenges as the def-use analysis, when they are extended to deal with vector programs. Copy propagation is an example discussed in this section. Considering the vectorized code in Figure 9.2. To make

the problem clearer, the code example is further translated into SIMD-like code, as shown in Figure 10.1. In the code example, vector variable names have a prefix of "v", and, unless otherwise specified, variables used in the examples are either short, arrays of short, or vectors of short. Also, we introduce the following notation,

- `v(i)` represents the i-th element of a vector v.
- `VLoad(`$addr$`)` represents a vector load from $addr$.
- `VStore(`$v, addr$`)` stores a vector v to $addr$.
- `LeftShift(`v, i`)` shifts left vector v by i elements.

In addition, it is also slightly optimized by moving the loop-invariant load out of the loop.

```
1. vrp = VLoad(&rp[0]);
2. for (; k--; s++) {
3.   d[0] = t[0] = *s;                /* 2 */
4.   vu = VLoad(&u[0]);               /* 4 */
5.   VStore(Recur(d[0], vrp*vu, +), &d[1]);
6.   vt = vu + (vrp * VLoad(&d[0])); /* 5 */
7.   VStore(vt, &t[1]);
8.   VStore(VLoad(&t[0]), &u[0]);    /* 6 */
9.   *s = d[8];                       /* 8 */
10.}
```

Figure 10.1: Vectorized loop from Figure 9.1.

Notice that the vector loop shown in Figure 10.1 can be further optimized by vector copy propagation to eliminate redundant memory accesses. For example, there is a copy propagation opportunity between statements 5 and 6 as one stores to memory `d[1]` to `d[8]`, and the other loads from memory `d[0]` to `d[7]`. Similar pattern also occurs between statements 7 and 8, and between 8 and 4.

In the first two cases, memory accessed by the pair of store and load do not completely overlap, i.e., with one element difference. To propagate from stores to partially overlapped loads, additional data reorganization operations are needed. For example, one can construct

the value of $\widehat{\text{d}}[0]$ to `d[7]` by combining scalar value `d[0]` with the vector that contains value of `d[1]` to `d[8]`. Figure 10.2 shows the vectorized loop after applying vector copy propagation. Since Pentium 4 is a "little endian" processor, `LeftShift` is used in statement 8 and 12.

```
1. vrp = VLoad(&rp[0]);
2. vu = VLoad(&u[0]);
3. for (; k--; s++) {
4.   d[0] = t[0] = *s;
5.   vu = vt2;
6.   vd = Recur(d[0], vrp*vu, +);
7.   VStore(vd, &d[1]);
8.   vd2 = LeftShift(vd, 1);
9.   vd2(0) = d[0];
10.  vt = vu + (vrp * vd2);
11.  VStore(vt, &t[1]);
12.  vt2 = LeftShift(vt, 1);
13.  vt2(0) = t[0];
14.  VStore(vt2, &u[0]);
15.  *s = d[8];
16.}
```

Figure 10.2: After vector copy propagation.

After all vector loads in Figure 9.2 are eliminated, the remaining vector stores may be moved out of the loop or eliminated as dead code, depending on whether the memory accessed by the store is used by later computation.

In `GSM`, vector copy propagation combined with dead store elimination improves the vectorization speedups from 0.88-1.27 to 2.08-2.97 on SSE2.

Chapter 11

Implementation on the HiLO Compiler

The compilation framework was implemented on the HiLO compiler, an internal compiler used in SPIRAL [59]. The HiLO compiler evolves from the SPL compiler [71]. The details of SPL and HiLO are introduced in Section 11.1 and 11.3 respectively. The I-code language, which serves as the interface between SPL and HiLO, is described in Section 11.2. Section 11.4 describes the implementation of VINCI on HiLO.

11.1 The SPL Compiler

The SPL (**S**ingal **P**rocessing **L**anguage) compiler was developed to translate DSP formulas into C/Fortran programs that can be compiled by native compilers. It is a key component of the SPIRAL system [56].

The input of the SPL compiler is SPL formulas. Each formula represents a specific DSP transform from an input vector to an output vector. In essence, the transform can be defined as a matrix-vector multiplication.

The following is an example of SPL formula.

```
(tensor (I 2) (F 2))
```

where `(I 2)` and `(F 2)` are two constant transforms and `tensor` is an operation on transforms to combine small ones to large one. There are other constant transforms and transform operations defined in SPL [71].

The SPL formula is then explained in a functional style: the result of any transforms is a new array and the input array is not notified. Thus, transforms do not have any side-

effects. On the other hand, such functional style introduces many temporary arrays in DSP programs. Thus, instead of simply translating SPL formulas into DSP programs, the SPL compiler includes some optimization techniques to improve the quality of code generated.

The compilation of a SPL formula can be roughly divided into two steps. First, the formula is translated into an I-code program. I-code is an internal intermediate representation used in the SPL compiler. Several formula-level optimizations are also conducted during the translation. The SPL formula in the previous example will be translated into the following program.

```
for (i0 = 0; i0 < 2; i0++) {
  y[2*i0+1] = x[2*i0] - x[2*i0+1];
  y[2*i0] = x[2*i0] + x[2*i0+1];
}
```

After that, a sequence of compiler optimizations is conducted on the I-code programs. The optimizations include loop unrolling, arithmetic simplification, constant propagation, scalar replacement, and many other common compiler optimizations. Although native compilers usually provide most optimizations, the SPL compiler can apply them more efficiently on I-code programs. After all optimizations being conducted by the SPL compiler, the example program becomes,

```
y[1] = x[0] - x[1];
y[0] = x[0] + x[1];
y[3] = x[2] - x[3];
y[2] = x[2] + x[3];
```

The SPL compiler actually includes two different components, a translator from SPL formulas to I-code programs and an optimizer on I-code programs. Hence, the recent version of the SPIRAL system divides it to two separated passes. The optimization tasks are moved from the SPL compiler to the HiLO compiler, an internal optimizing compiler on I-code

programs. The SPL compiler only does translation from SPL formulas to I-code programs. However, several formula-level optimizations, which are mainly loop optimizations, are still applied by the SPL compiler.

11.2 The I-Code Language

As the original SPL compiler was divided into two different compilers, the I-code language was also extended to an intermediate language and used as an interface connecting these two compilers. In essence, I-code is a subset of C with following simplifications.

First, in terms of control flow structures, I-code only allows regular loop structures. No early breaks or goto statements are supported. Besides, if-statements and switch statements are not supported either.

Second, for data types, both single and double floating point numbers are supported. For integers, only full-size integers are supported and neither shorts nor chars are allowed. In addition, only one dimensional array can be used in I-code programs. Currently, I-code does not allow either multi-dimensional arrays or structures and unions.

Finally, no pointer and pointer arithmetic is supported by I-code, although most common arithmetic operations and comparison operations are already included in I-code languages.

With these simplifications, I-code makes it easier for the HiLO compiler to obtain precise information from the program and apply more aggressive optimization in the more efficient way.

11.3 The HiLO Compiler

The HiLO (**Hi**gh **L**evel **O**ptimization) compiler was designed as a back-end compiler for different library generators. The input of HiLO is I-code programs and the output is standard C programs that can be compiled by standard C compilers.

I-code programs are represented as AST (Abstract Syntax Tree) in the HiLO compiler. In the HiLO compiler, either program analysis or compiler optimization is defined as a program-traversal pass. The operation is associated with each AST node. The advantage of such design is to provide a clear structure for compiler transformations and make it easier to implement new transformations. However, it introduces additional overhead and lacks of global view of the whole program. The HiLO compiler is written in Java.

Most common compiler analysis and optimization routines have been implemented in the HiLO compiler. The analysis routines includes semantics check, def-use analysis, induction variable analysis, and even a simple version of dependence analysis. The optimization routines include loop unrolling, copy propagation, dead code elimination, common subexpression elimination, arithmetic simplification, strength reduction, and so on. In addition, some low-level optimizations, such as instruction scheduling and software pipelining, are also implemented in HiLO [59].

For most library generators, it is oftentimes necessary to search for optimal or sub-optimal optimizations parameters. Hence, the HiLO compiler is designed as a fully parameterized compilers. Not only the order of transformations but also the parameters of a specific transformation, such as the loop unrolling factor, can be specified from outside such as search engine.

More important, because of its simple input language and clear structure, the HiLO compiler is an ideal platform to experiment new compiler optimizations. Hence, VINCI was implemented on the HiLO compiler.

11.4 Implementation on HiLO

The implementation of VINCI needs to extend all three components described above. First, I-code language is also extended by adding vector expressions. The syntax of vector expressions is similar to Fortran 90, as described in Section 4.1. Since all arrays are one-dimensional

in I-code program, all vector expressions are one-dimensional too.

Correspondingly, both the SPL compiler and the HiLO compiler are extended to generate and accept vector-extended I-code programs respectively. Since vectors are already internal representation used in the SPL compiler, no vectorization is needed for such extension. Instead, the task of the SPL compiler is simplified by generating vector programs directly.

After the parser being extended to accept vector representation, new passes were also added into the HiLO compiler. The newly-added major passes include vector normalization (Chapter 5), loop unrolling and vector coalescing (Chapter 6), data permutation optimization (Chapter 7), and code generation (Chapter 8). Besides, several existing modules in HiLO are also significantly extended, such as def-use analysis (Chapter 10).

Finally, the output of the HiLO compiler is extended to include the SIMD intrinsics supported by the native compilers. In fact, instead of generating intrinsic functions that can be understood directly by native compilers, the compiler generates *abstract* intrinsic functions that are mapped to native intrinsic functions through macro definitions. This additional level is introduced mainly to address the incompatibility introduced by different compilers. For different SIMD devices, macro-definition mapping is not enough to address the difference on ISAs.

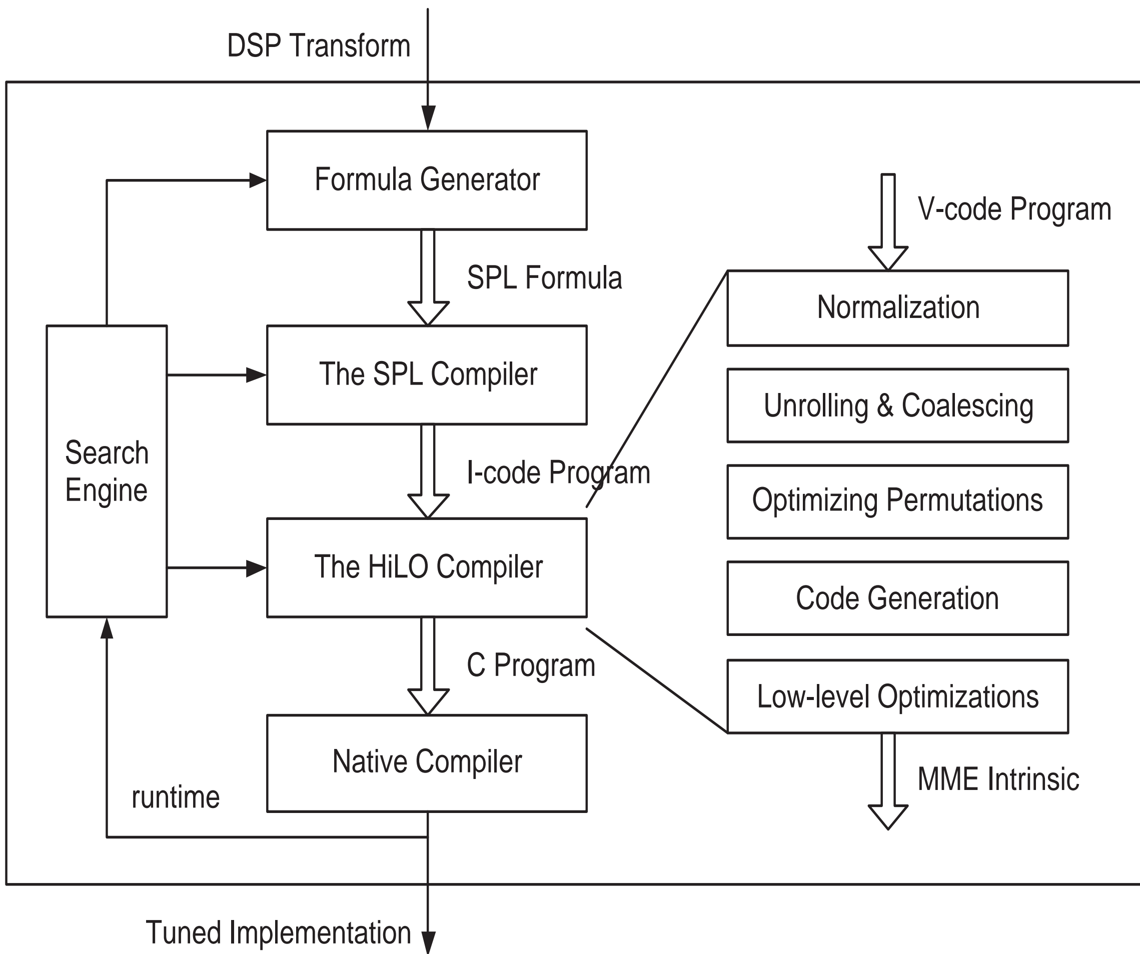

Figure 11.1: Implementation on the HiLO Compiler

Chapter 12

Domain Specific Optimization for DSP Programs

When generating DSP programs, several domain-specific optimizations are critical to achieve the optimal performance. Two of such optimizations, called *simplifier* and *scheduler*, were introduced in [20]. Like other compiler optimizations, the extension of these domain-specific optimization on vector programs are not straightforward. Section 12.1 describes how to simplify SIMD arithmetic operations by identifying special constant values. Section 12.2 introduces an instruction scheduling algorithm to improve register allocation.

12.1 Simplification of Arithmetic Operations

In FFT programs, multiplication is one of most expensive computation tasks. In fact, the number of multiplication operations was used to measure the performance of FFT programs. Thus, it is critical to reduce the number of multiplication in programs.

In FFT programs, the multiplication is conducted with constant numbers. Some constants have special values, including 0, 1, and -1. If the compiler can recognize such special constants, the multiplication with these constants can be greatly simplified. Furthermore, if the constant is 0, the following add/subtract operations can also be simplified. On scalar programs, it is relatively straightforward to implement such optimization by using constant folding and constant propagation [50].

However, such arithmetic simplification becomes more complicated when being applied on vector programs. For SIMD computation, unless all elements in a register variable have the same special value (0, 1 or -1), the multiplication with this SIMD constant variable

cannot be simplified. On the other hand, when there are special values in a SIMD constant, the computing bandwidth is not fully utilized by the multiplication with this constant.

Therefore, additional optimization is needed to reorganize constant vectors into the specific formats so that same special values are located in the same register variable. It could be done by conducting a compiler pass to reorganize constant arrays and insert data permutation explicitly if necessary. Alternatively, a more direct solution is to use *interleaving expansion* to expand constant arrays during loop unrolling (see Section 6.1). By doing this, all constant vectors are naturally extended in the way that all same special values are located together. Hence, arithmetic simplification is also an important reason why interleaving expansion is used in loop unrolling and vector coalescing.

However, another issue associated with arithmetic simplification is how to interact with the algorithm of data permutation optimization. In Section 7.3, it is simply assumed that constant vectors can be reorganized for free so that data permutations can be propagated through them. However, when considering arithmetic simplification on constant vectors, such assumption is not valid any more. It is possible that the propagation through constant vectors may result in more multiplications and other arithmetic operations.

Therefore, an extension was added into the optimization algorithm on data permutations to check whether the propagated permutation changes the format of constant vectors. If so, the propagation is not allowed. Other permutations, such as those intra-register permutations, can still be propagated through constant vectors.

12.2 Instruction Clustering

In the current implementation on HiLO, output programs are standard C programs with SIMD intrinsics. It relies on native compilers for register allocation and instruction scheduling. However, in the experiment, a lot of unnecessary memory spills are found in the assembly code generated by native compilers, due to poor register allocation. Actually, it is a common

problem for automatic library generators that many compilers cannot generate efficient code for large basic blocks [20].

To address this problem, an algorithm was developed to reorder statements in source code to help the register allocator in back-end compilers. The key idea of the algorithm is to minimize the lifespan of register variables. Due to the special data flow of FFT programs, reducing the lifespan of a variable oftentimes results in increasing the lifespan of another variable. Thus, a simple greedy algorithm that attempts to minimize the lifespan of each encountered variable is not feasible for FFT programs.

On the other hand, there is an important observation about the lifespan of variables. If the lifespan of a variable is larger than a specific number, which is usually the number of register in SIMD devices, it most likely will be spilled out[1]. As long as the lifespan is larger than this threshold, it does not matter how large it is. Thus, it is more important to have more variables whose lifespans are less than the threshold than to minimize the overall lifespan of all variables.

Based on this observation, an algorithm, called as *instruction clustering*, was developed to reorder instructions in SIMD programs. Starting from creating a cluster for a randomly picked instruction, the algorithm tries to add as many instructions as possible to the cluster, as long as the number of registers needed for the statements in the cluster is still less than a preset number, which is usually slightly smaller than the number of registers in the target SIMD device. After a cluster is full, the algorithm creates another one by picking another instruction. Finally, all instructions are clustered and then the algorithm reorders the instructions by putting all instructions in the same cluster together.

Notice that it is an NP-complete problem to find the maximum number of registers needed for a given program [23]. Thus, the algorithm uses the heuristics to decide which instruction is picked when creating a new cluster. It usually picks the instruction on the top of the def-use web of all instructions not being clustered yet. In addition, another heuristic

[1] Of course, it depends the register allocation algorithm employed by the native compiler.

is used to schedule clusters by following the original order of the first instructions in the clusters.

Chapter 13

Experimental Results

The implementation of VINCI on the HiLO compiler was evaluated on different applications and SIMD devices. The experimental results are summarized in this chapter. Section 13.1 briefly introduces the overall framework of the SPIRAL system, which generates most testing programs used in the experiment. Section 13.2 describes the experiment setups. The result of static evaluation on assembly programs is shown in Section 13.3 and the run-time performance result is illustrated in Section 13.4.

13.1 The SPIRAL System

In multimedia applications, digital signal processing (DSP) plays a critical role. Many kernels from multimedia applications, such as DCT (Discrete Cosine Transform), WHT (Walsh-Hadamard Transform) and FFT (Fast Fourier Transform), are typical DSP transforms. Thus, many microprocessor vendors provide natively hand-tuned DSP libraries for their platforms. However, due to the complex architecture of current microprocessors, even the performance tuning of small kernels can be extremely time-consuming and require expert knowledge on both application and micro-architecture. Alternatively, automatic generation of high-performance native library has been tried out in both industry and academia. FFTW and SPIRAL are two typical DSP library generators [21, 56].

Different from FFTW [21], which targets FFT transforms only, the SPIRAL system provides a more general solution for different DSP transforms. Given a specific transform in the SPL formula representation, the formula generator produces different SPL formulas by

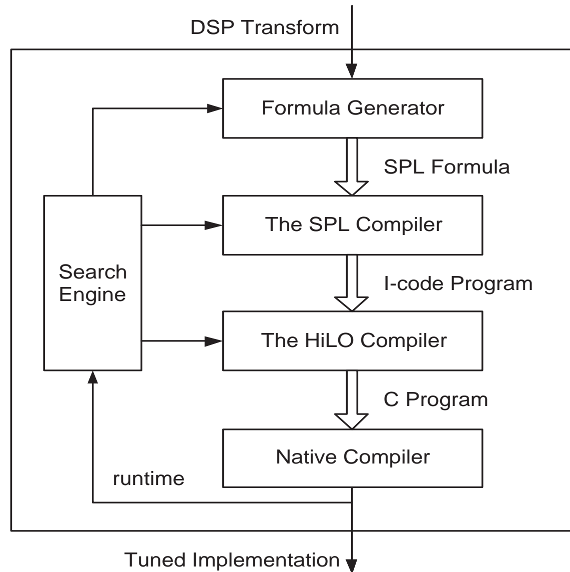

Figure 13.1: An overview of the SPIRAL system.

applying the rewriting rules specified by the user. Then each SPL formula is translated to an I-code program by the SPL compiler (see Section 11.1). I-code is an internal language used to represent intermediate programs (see Section 11.2). After that, the HiLO compiler applies various compiler optimizations on the I-code program and translates it to C code for the native compiler. By searching different formulas(algorithms), and compiler optimizations, the SPIRAL system is able to generate very efficient DSP programs on most platforms [56]. The whole workflow of the SPIRAL system is shown in Figure 13.1.

13.2 Experiment Setups

The experiments were conducted on two platforms, a Pentium-4 with SSE2 and a PowerPC G5 with VMX. On both systems, the native compilers, the Intel compiler (icc) and IBM XL C compiler (xlc), were used to compile SIMD codes generated by the HiLO compiler with VINCI implementation. Table 13.1 summarizes the experimental setup for the two platforms.

If not specified explicitly, the default data type used in all applications is 32-bit single-

precision floating point. Both VMX and SSE2 are 128-bit SIMD devices that can process 4 single-precision FP operations in one instruction. Considering that there are two scalar FP units on the Power G5 processor [49], the ideal speedups on VMX and SSE2 are 2 and 4 respectively if no data permutation overhead were introduced.

Processor	2.0G Pentium 4	1.8G PowerPC G5
SIMD Unit	SSE2	VMX
Main Memory	1024 MB	2048 MB
Operating System	Linux v2.4	Mac OS 10.4
Compiler	icc v9.0	xlc v6.0
Compiler Option	-fast(-O3)	-O3 -qaltivec

Table 13.1: Experiment platforms.

The source vectors programs used in the experiment were produced by library generators. A few kernels written by hand were also used in the experiment. The scalar codes used for comparison were also produced by library generators. These scalar codes were fully optimized and achieved a performance comparable with that of the hand-tuned vendor's libraries [47, 56, 71].

The vector programs used in the experiment can be classified into three groups:

- **Group I.** Programs with complicated data permutation patterns. Unlike the codes in previous group, these programs interleave computation and data permutations across the whole program. The data permutations are an inherent part of the algorithm.
  - `FFT`: Fast fourier transform programs generated by a vector-extended version of the SPIRAL system [56]. An example of the output of the extended SPIRAL is shown in Figure 1.3. We evaluate a set of FFT routines where `fft.`n represents a 2^n-point FFT.
  - `WHT`: Walsh-Hadamard transform routines generated by our vector-extended SPIRAL. The permutation patterns inherent to WHTs are usually simpler than those

of FFT. We evaluate a set of WHT routines where `wht.`n represents a 2^n-point WHT.

- `Bitonic sort`: One of the fastest sorting network algorithms. When the data can be stored in processor registers, bitonic sorting is one of the fastest sorting algorithms [47]. We evaluate a set of bitonic sorting programs where `bitonic.`n represents the 2^n-point bitonic sorting.

* **Group II.** Routines involving data permutations exclusively. These routines are mainly used to evaluate the code generation algorithm. Two common examples of such routines are:

  - `Matrix transpose`: Turn rows of a matrix into columns.
  - `Bit-reversal order`: Reorder elements in an array with the bits of index flipped left-for-right, mainly used in hand-written FFT programs (but not used in FFT codes generated by SPIRAL).

* **Group III.** Programs with relatively simple permutation patterns. In those programs, the input (and output) data are not in the format required by the SIMD operations. Thus, data permutations must be introduced to pack, unpack, or realigned data.

  - `C-Saxpy`: Multiply a complex vector by a constant complex vector and adds it to another complex vector.

```
for(int i=0; i<128; i++) {
  y[2*i]   += x[2*i]*C[2*i]   - x[2*i+1]*C[2*i+1];
  y[2*i+1] += x[2*i]*C[2*i+1] + x[2*i+1]*C[2*i];
}
```

  - `R-Color`: Convert input from the RGB color space to the YC_bC_r color space as specified by the CCIR recommendation [8].

```
for(int i=0; i<196; i++) {
  y[3*i]   = .299*x[3*i]+.578*x[3*i+1]+.114*x[3*i+2];
  y[3*i+1]= .500*x[3*i]-.419*x[3*i+1]-.081*x[3*i+2];
  y[3*i+2]=-.169*x[3*i]-.331*x[3*i+1]+.500*x[3*i+2];
}
```

– `C-Dot`: Compute the dot-product of two complex vectors.

```
re = im = 0;
for(int i=0; i<128; i++) {
  re += x[2*i]*y[2*i]   - x[2*i+1]*y[2*i+1];
  im += x[2*i]*y[2*i+1] + x[2*i+1]*y[2*i];
}
```

– `R-FIR`: Apply a real finite impulse response (FIR) filter on a real vector, with a small number of taps.

```
for(int i=3; i<128; i++) {
  y[i] =   x[i]*C[0] + x[i-1]*C[1]
       + x[i-2]*C[2] + x[i-3]*C[3];
}
```

In addition, on the applications listed above, different alignment settings have been tried out to evaluate the permutation optimization algorithm.

13.3 Static Evaluation

The optimization algorithm on data permutations is first evaluated in terms of the number of permutation instructions generated. Table 13.2 summarizes the number of permutation instructions generated for different applications on VMX and SSE2. The baseline (*Base* in Table 13.2) generates data permutations (and other SIMD instructions) without performing

		VMX			SSE2		
Program	Size	Base	Opt	Reduced	Base	Opt	Reduced
fft.4	16	96	24	75.0%	96	24	75.0%
fft.5	32	208	48	76.9%	208	48	76.9%
fft.6	64	352	96	72.7%	352	96	72.7%
wht.4	16	48	12	75.0%	48	12	75.0%
wht.5	32	96	24	75.0%	96	24	75.0%
wht.6	64	192	48	75.0%	192	48	75.0%
bitonic.4	16	52	34	34.6%	56	34	39.3 %
bitonic.5	32	136	92	32.3%	144	92	36.1 %
bitonic.6	64	336	232	31.0%	352	232	34.1 %

Table 13.2: The number of permutation instructions on VMX and SSE2. (Abbr: Opt - Optimized)

any propagation or permutation composition. Column *Opt* lists the number of permutation instructions after applying all optimization techniques described in Chapter 7 and *Reduced* shows the percentage of permutation instructions being eliminated.

In summary, the optimization algorithm can significantly reduce the number of permutation instructions by a factor of 3 to 5 on both VMX and SSE2. For most applications, the number of permutation instructions are same on both platforms, which is an indication of the efficiency of the code generation algorithm. For the others, even though the baseline number on SSE2 is slightly larger, it becomes the same after optimization because of permutation decomposition for special shuffle instructions (Section 7.2.2).

As shown in Table 13.3, four different optimization strategies were evaluated on SSE2, including combination of two consecutive permutations without applying the distributive rule (*Comb*), combination of consecutive permutations with propagation along def-use chains (*Top*), with propagation along use-def chains (*Bot*), with two-direction propagation (*T&B*). In these four cases no other optimizations introduced in Section 7.2 were applied.

As shown in the table, the combination of consecutive permutations by itself can eliminate many permutations on FFT and WHT programs and a few on bitonic sorting. Notice that the propagation from definitions to uses is still needed to combine two consecutive permutations. When the optimization techniques, including permutation reshaping and

Program	Base	Opt	Comb.	Top	Bot	T&B	Reshape	Reg	Shuf	Top+	Bot+	T&B+
fft.4	96	24	40	24	40	24	24	24	24	24	40	24
fft.5	208	48	128	128	112	112	112	64	48	48	80	48
fft.6	352	96	160	160	128	128	128	128	96	96	128	96
wht.4	48	12	16	16	16	16	16	16	12	12	16	12
wht.5	96	24	48	48	48	48	48	32	24	40	32	24
wht.6	192	48	64	64	64	64	64	64	48	48	64	48
bitonic.4	56	34	52	52	52	52	46	46	34	42	44	34
bitonic.5	144	92	136	136	136	136	46	46	34	116	112	92
bitonic.6	352	232	336	336	336	336	312	312	232	296	272	232

Table 13.3: Comparison of different optimization strategies.
(Abbr: Opt - Optimized; Comb. - Combination Optimization; Top - Top-down Propagation; Bot - Bottom-up Propagation; T&B - Top-town and Bottom-up Propagations; Reshape - Permutation Reshaping; Reg - Register-wise Permutation Decomposition; Shuf - Shuffle-Instruction Permutation Decomposition)

decomposition (Section 7.2), are not applied, the propagation can only further eliminate one-fifth to one-third of the rest on FFT programs. However, after applying all these techniques, the propagation can reduce many permutation instructions, especially on bitonic sorting programs, as shown in the last three columns, *Top+*, *Bot+* and *T&B+*, which correspond to *Top*, *Bot* and *T&B* respectively.

The optimization techniques introduced in Section 7.2.1 and 7.2.2 were also evaluated in our experiments. On the "base" algorithm with two-direction propagation, we applied permutation reshaping (*Reshape*) and then permutation decompositions for either inter-register permutation (*Reg*) and special shuffle instructions on SSE2 (*Shuf*).

On bitonic sorting programs, the base algorithm (with two-pass propagation) can only remove 3% to 10% of all permutation instructions. Applying permutation reshaping technique removes additional 5% to 15%. Finally, with all techniques applied, the algorithm removes 34% to 40% of all permutation instructions. For FFT and WHT, those techniques can further eliminate more permutation instructions (up to 57% of the rest).

For large-size bitonic sorting programs, the optimization algorithm eliminates a lower percent of permutation instructions. The reason is that in large-size bitonic sorting pro-

Program	Size	Aligned				Misaligned			
		VMX		SSE2		VMX		SSE2	
		Base	Opt	Base	Opt	Base	Opt	Base	Opt
fft.5	32	208	48	208	48	224	64	224	64
wht.5	32	96	24	96	24	104	28	104	32
bitonic.5	32	136	92	144	92	144	100	152	100
transpose	256	128	128	128	128	192	129	192	129
bit-reversal	256	128	128	128	128	192	129	192	129
c-saxpy	128	128	128	128	128	192	160	192	192
c-dot	128	132	132	132	132	196	160	196	196
r-fir	256	252	252	252	252	316	252	316	252
r-color	128	320	320	416	416	511	352	607	448

Table 13.4: Comparison of different alignment settings in terms of number of data permutation instructions generated.

grams, more data permutations are register-wise permutations, which are not mapped to any permutation instruction. In fact, the ratio between data permutation and computation becomes smaller as the size increases.

Finally, we evaluate the optimization algorithm by using different alignment settings. As listed in Table 13.4, although the mis-alignment of input array introduces more data-movement instructions, many of them can be eliminated by the optimization algorithm.

13.4 Runtime Performance Evaluation

We now discuss the runtime performance of the programs listed in Section 13.2. Time is measured using system call *gettimeofday*. The overall performance depends on many aspects other than the number of data permutation instructions.

13.4.1 Evaluating the Optimization Algorithm

In Figure 13.2, (a) and (b) illustrate the performance of 32-point FFT programs on VMX and SSE2. The horizontal axis represents 45 different algorithms (routines) for the same

size FFT. The optimized SIMD code can achieve speedups between 1.52 and 2.63 on VMX and between 3.80 and 5.68 on SSE2 over highly-optimized scalar codes. On the average, the optimization algorithm improves performance by 42% on VMX and 70% on SSE2 over unoptimized SIMD codes.

Library generators typically choose the program with the best performance. In terms of the maximum performance among 45 programs, the optimized SIMD code achieves a speedup of 1.73 on VMX and 5.20 on SSE2 over the scalar code. On SSE2, the best optimized SIMD code performs 45% better than the vendor-provided library.

Pentium-4 provides two exclusive execution modes for FP operations. When the "-fast" option is specified, the Intel compiler will generate scalar SSE2 instructions, whereas it generates conventional x87 FP instructions with the "-O3" option. Usually, SSE2 FP instructions run faster than x87 FP instructions on Pentium-4 processor. However, for those FFT programs, the "-O3" option achieves better performance than "-fast". For comparison, we present the performance of scalar codes compiled with the "-O3" option in the figure.

The run-time performance of 32-point WHT programs is shown in (c) and (d) in Figure 13.2. Similarly, the horizontal axis represents different algorithms. The speedups over scalar code are between 1.23 and 2.67 on VMX and between 2.02 and 2.60 on SSE2, which is lower than those FFT programs because WHT requires fewer arithmetic operations. The optimization algorithm improves the performance on the average by 47% on VMX and 88% on SSE2 among all 45 programs.

Graphs (e) and (f) show the performance of bitonic sort programs on the two platforms. Each graph includes five programs whose input sizes are 16 to 256. The XLC compiler generates less efficient code than GCC for the sorting routines. The opposite is true for all other routines. We present the performance of scalar sorting code under GCC in graph (e).

As one of the fastest sorting algorithms for small-size data, bitonic sorting programs can be integrated into sorting algorithms for large data sets. Either quick sort, merge sort or radix sort can call bitonic sorting at the end of their recursion, if the (sub) data set can

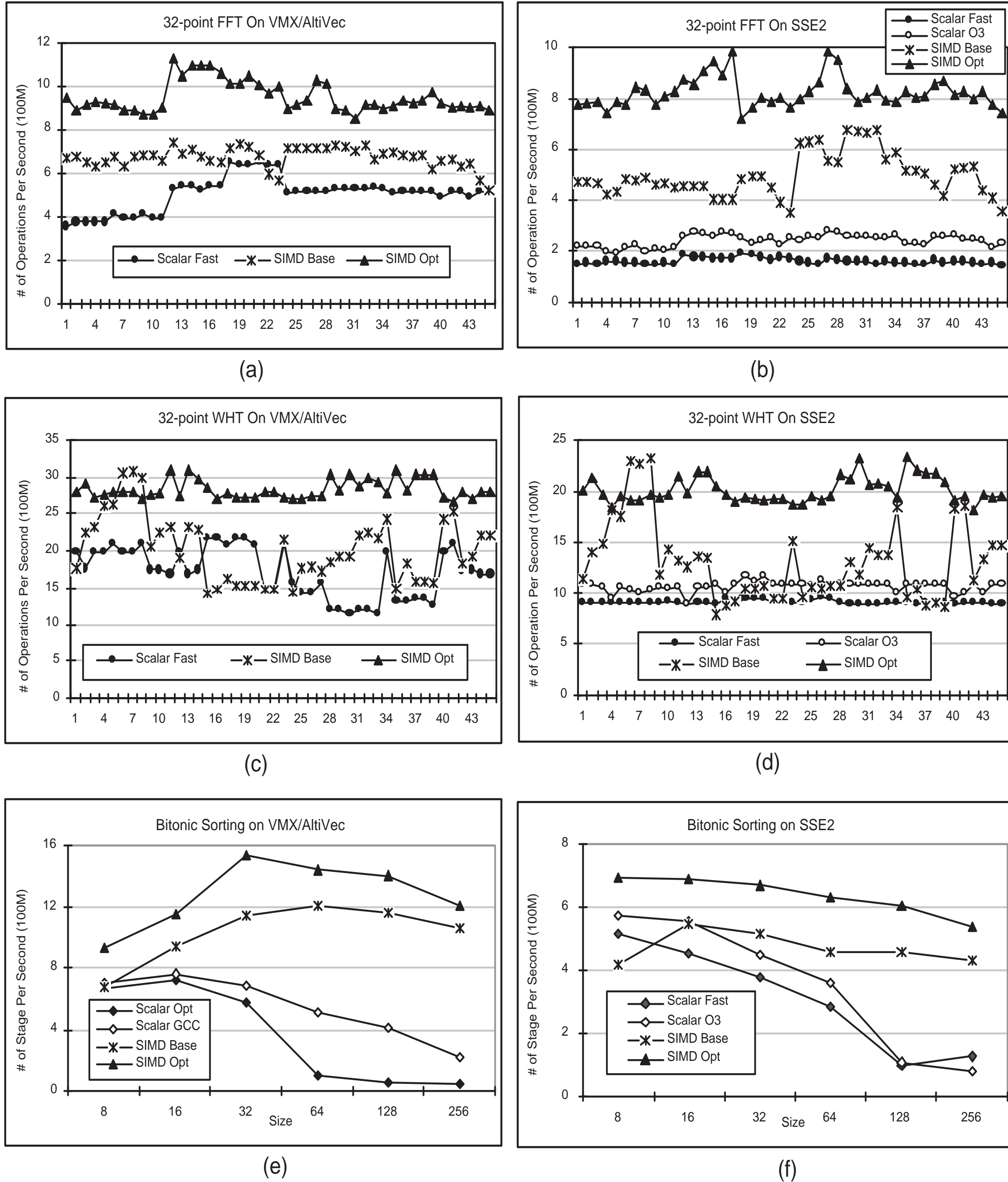

Figure 13.2: Performance of Group I programs.
(Note: “Scalar Opt”, “Scalar Fast”, “Scalar O3” are optimized scalar codes generated by SPIRAL, compiled with -O3 -qaltivec, -fast, and -O3, respectively; “SIMD base” are SIMD codes with “Base” permutation generation as in Table 13.2; “SIMD opt” are SIMD codes with the permutation optimization.)

fit in registers. We combined bitonic sort and merge sort. The resulting hybrid algorithm improves the performance by 10% on VMX and 15% on SSE2 and achieves comparable or better performance than the vendor-provided libraries.

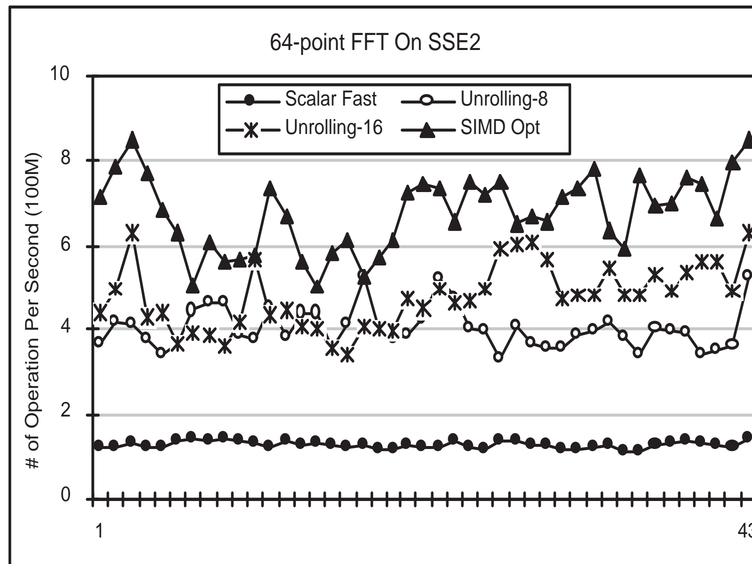

Figure 13.3: Performance of 64-point FFT programs on SSE2.

Some experiments were also conducted to measure the performance of those programs with loops. Instead of fully unrolling all loops in FFT programs, the algorithm decides the unrolling factor based on the number of register supported by the target SIMD device. In Figure 13.3, three different unrolling factors are evaluated. One results in fully unrolling, which is "SIMD Opt". "Unrolling-8" results more loops than "Unrolling-16" so that "Unrolling-16" has better performance than "Unrolling-8" most time. In addition to the overhead of loop structures, there are several other reasons why small unrolling factors result in worse performance. As discussed in Section 7.5, even the optimization algorithm is able to propagate data permutations across loop boundaries, it might result in isolated register-wise permutations between loops, which will cause lots of memory load and store instructions. In addition, it usually results in more data permutations if we keep loop structures in programs.

13.4.2 Evaluating the Domain-specific Optimizations

As discussed in Chapter 12, two important domain-specific optimizations have been extended for vector programs. The results in Figure 13.4 show the effectiveness of these optimizations. In the figure, two sub-graphs illustrate the performance of 32-point FFT programs on VMX and SSE2, similar to (a) and (b) in Figure 13.2. Each graph in Figure 13.4 has two lines to show the performance achieved without either *arithmetic simplification* or *instruction clustering*.

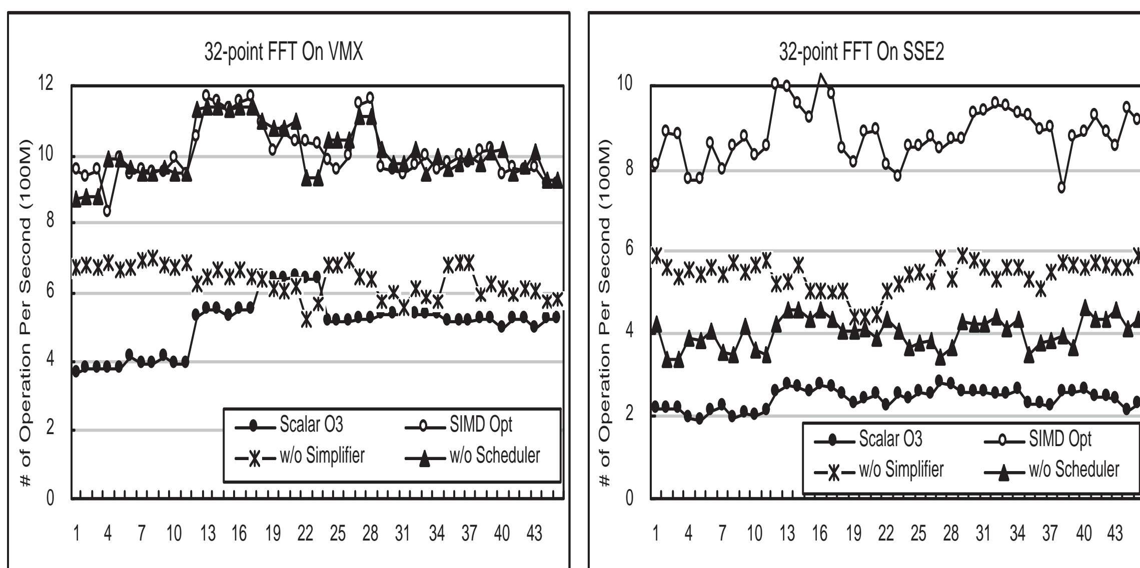

Figure 13.4: Performance evaluation of domain-specific optimizations.

On VMX, arithmetic simplification plays an important role to achieve speedups on SIMD computation. Sometimes, the performance of SIMD code without arithmetic simplification is even worse than the one of scalar code with arithmetic simplification. However, instruction clustering does not affect performance in an obvious way. Two possible reasons are the large register space and the superior register allocator in the XL C compiler. On SSE2, both optimizations are important for the performance. Arithmetic simplification and instruction clustering improves the performance by 70% and 120% respectively.

13.4.3 Evaluating the Code Generation Algorithm

The results shown in Figure 13.5 demonstrate the efficiency of the code generation algorithm. For matrix transposes, it achieves speedups up to 1.62 on VMX and 1.72 on SSE2 over the scalar code. Interestingly, the performance of matrix transpose is completely different on the two platforms. On VMX, except for 4x4 matrix, speedups are reported on all other matrices. However, on SSE2, the SIMD code can only achieve 61% of the performance of the scalar code in the worst case. It is mainly because SSE2 needs more permutation instructions than VMX, especially when the permutation pattern is less regular.

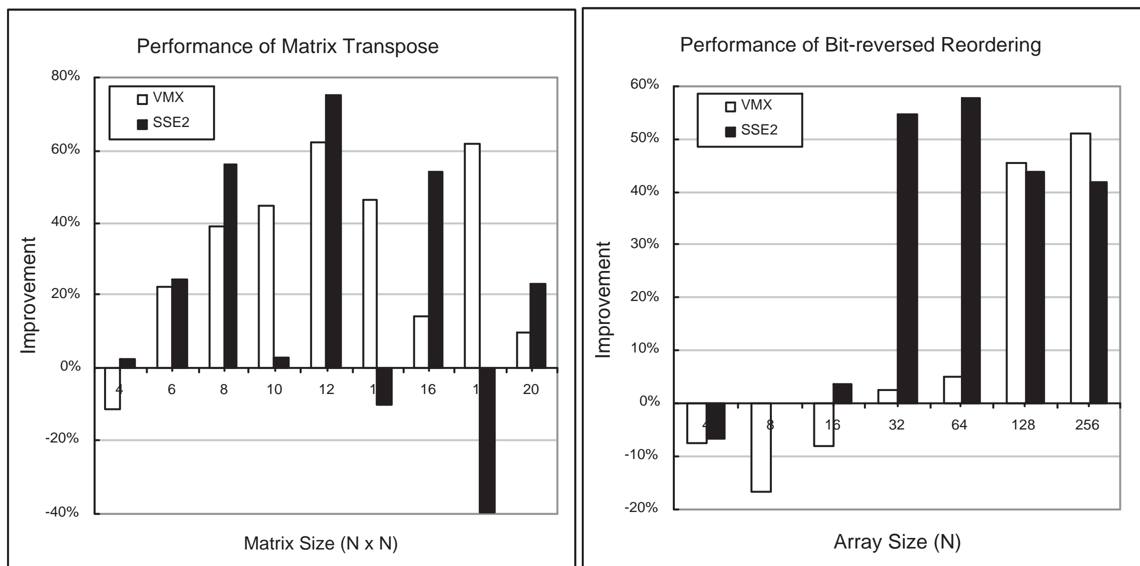

Figure 13.5: Performance of matrix transpose and bit-reversal ordering.

Figure 13.5 also shows the performance of bit-reversal array reordering. On both platforms, the SIMD code only shows performance improvement for large array sizes. That means that the overhead of data permutations nullifies the performance benefit of SIMD load/stores. But the aggregated register space helps the performance for large sizes.

13.4.4 Overall Run-time Performance

Finally, the speedups over scalar code on all applications are shown in Figure 13.6. The speedups ranges from 1.14 to 2.58 on VMX and 1.51 to 3.77 on SSE2. Since VMX has two scalar FP units, the speedups on VMX are lower than those on SSE2 for most applications. The above-2 speedup on VMX is obtained on the bitonic sorting program, where expensive comparison and swap operations are replaced by native SIMD max and min operation.

Figure 13.6 also shows that the speedups of all applications when the input data is misaligned. On the average, the performance drops 3.2% and 8.0% on VMX and SSE2 respectively, because of misalignment. For Group II applications, the permutation optimization algorithm improves the performance by 60% and 140% on VMX and SSE2 respectively, since the data permutations introduced by the misalignment are almost completely eliminated by the optimization algorithm (see Table 13.4). However, for Group III application, the improvement obtained by the optimization algorithm is only 6% and 10% on VMX and SSE2.

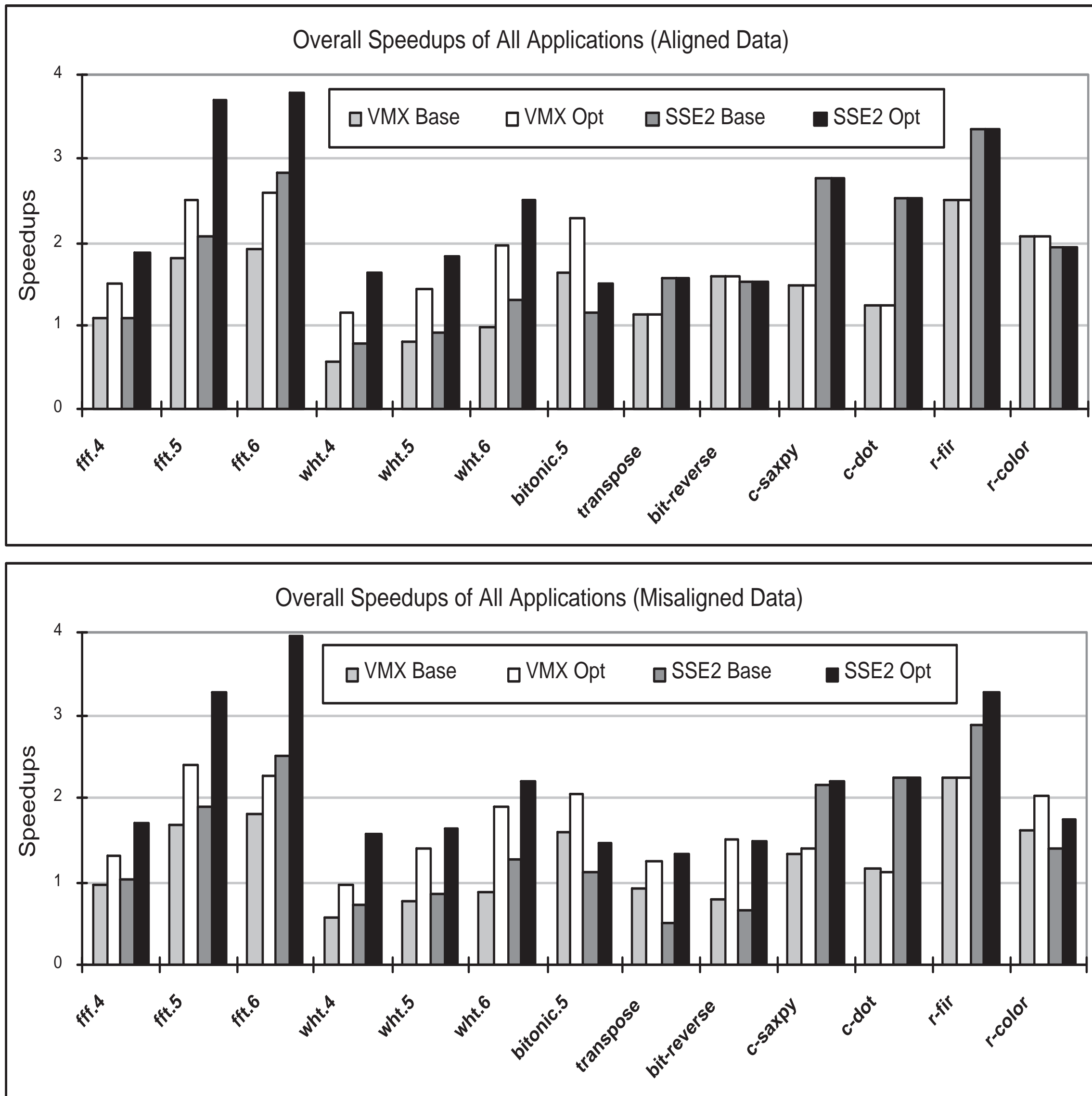

Figure 13.6: Performance of all applications on VMX and SSE2.
(Note: "VMX-Base" and "SSE2-Base" are speedups of SIMD codes without the permutation optimization on VMX and SSE2 respectively. "VMX-Opt" and "SSE2-Opt", on the other hand, are speedups with the permutation optimization.)

Chapter 14

Other Issues in SIMD Compilation

This chapter summarizes several other issues identified by the empirical study conducted on multimedia applications. Section 14.1 lists the important characteristics of multimedia applications and shows how these characteristics effect SIMD compilation. Section 14.3 discusses several issues introduced by the mismatches between applications, C language, and SIMD devices. Finally, some experimental results of manual vectorization are summarized in Section 14.4.

14.1 Programming Styles of Multimedia Applications

14.1.1 Pervasive Use of Pointers and Pointer Arithmetics

Traditional vectorization is very effective for programs that spend most of their time on tight loops involving explicit array accesses. Multimedia programs, however, often use pointers and pointer arithmetic to access data in computationally intensive loops. Of the twelve programs we studied (see Section 3.2.1), all of them use pointers in their core procedures, and six of them use pointer arithmetic in addition.

Pervasive use of pointers and pointer arithmetic has a great impact on both memory disambiguation and dependence testing for vectorization. One commonly used technique is to translates pointer accesses and pointer arithmetics onto array accesses. This technique is called array recovery [19]. Basically, it treats pointers as induction variables and expresses pointer accesses in terms of a base address and an closed form expression of the surrounding

loop counters. We also observed loops which contain pointers that have no closed-forms. In this case, one can exploit the monotonicity of the pointers to estimate the access region as well as conducting dependence analysis [68].

14.1.2 User-Conducted Optimizations

Many multimedia programs are hand optimized. Some of the manual optimizations can completely change the appearance of original algorithms and oftentimes make it more difficult for the compiler to vectorize. Furthermore, user optimizations make it more difficult to make vectorization profitable, especially when the optimization on scalar codes is not applicable to vectorized codes.

One of most common user optimizations is loop unrolling. One technique to vectorize unrolled loops is to first reroll the loop then apply loop-level vectorization [46]. Another technique is to pack isomorphic instructions within the unrolled loop into SIMD instructions [41].

```
1.  if (init==0)
2.    for (i=0;i<LUTABSIZE;i++)
3.      lutab[i]=nint(pow((float)i/10.0, 0.75)-0.0946);
    ...
4.  for (i=0;i<l_end;i++) {
5.    temp=istep*fabs(xr[i]);
6.    if (temp<0.499996) ix[i]=0;
7.    else if (temp<1.862955) ix[i]=1;
      ...
8.    else if (temp<1000.0)
9.      ix[i]=lutab[(INT32)(temp*10.0)];
10.   else
11.     ix[i]=(INT32)(sqrt(sqrt(temp)*temp)+0.4054);
12. }
```

Figure 14.1: User-optimized code from LAME.

Another common optimization is to use lookup tables and/or if-statements to shortcut expensive computation for a set of frequently encountered inputs. Figure 14.1 gives an

example of using both lookup table and if-statement to shortcut the expensive computation, `pow(x,0.75)-0.0946`. In this example, array `lutab` stores the precomputed value for `x < 1000` in the initialization (at statement 3). If an input value falls within the interval, the result is directly retrieved from `lutab` (statement 9) instead of being computed (statement 11). The code is further optimized by using if-statement shortcuts, where statements 6 and 7 return 0 or 1 as the result of `pow(x,0.75)-0.0946` when `x` is less than `1.862955`.

Since most multimedia extensions do not support indexed memory access, it is extremely difficult to vectorize loops with lookup table accesses. If we vectorize other computation in the loop and leave lookup tables in scalar forms, the overhead introduced by transferring data between scalar and vector registers could result in a significant slowdown (2-3 times in our experiments).

14.2 Vectorizing Outer Loops

Figure 14.2 gives another core loop extracted from GSM that implements a FIR(Finite Impulse Response) filter. In the original GSM code, the *i*-loop is completely unrolled by programmers. In this example, we rerolled the unrolled statements back to an inner loop. Both unrolled and rerolled implementations of FIR computation are very common in DSP and media processing domain.

```
1.  for(lambda=40; lambda<=120; lambda++) {
2.    sum = 0;
3.    for(int i=0; i<40; i++)
4.      sum += wt[i] * dp[i - lambda];
5.    L_result[lambda] = sum;
6.  }
```

Figure 14.2: Another simplified loop from GSM.

Let us first consider the vectorization of the inner *i*-loop. The *i*-loop involves reduction and conversion from short to int (as *sum* is 32-bit, *wt* and *dp* are arrays of 16-bit

integers) Vectorization of reduction is straightforward. Conversion can be vectorized into packing/unpacking operations.

After vectorizing the inner loop, we would still need to store *sum* into *L_result* in each iteration of the outer loop. It would be better to combine stores to contiguous locations in the outer loop into a vector store. This can be accomplished by vectorizing the outer loop. This transformation introduces data movement operations. For example, to store a 4-element vector to `L_result`, at least three data movement instructions are needed to pack the four (scalar) results. Therefore, the benefit of this optimization depends on the memory access latency, data movement overhead, and register pressure.

In our experiment, this optimization can up to achieve an additional 66%performance improvement on core procedures.

14.3 Mismatches Between Application and Language

14.3.1 Subword Optimizations

Multimedia programs often use 8-bit or 16-bit integers (referred to as *subword* integers) to represent media data, such as colors or pixels. As shown in Table 14.1, 9 out of 12 applications we studied use subword integers as their primary data types. The rest use floating points.

Type	Applications
char	MPEG2, Doom, Mesa
short	ADPCM, GSM, DVJU, JPEG, Timidity, mpg123
single	Rsynth
double	LAME, POVray

Table 14.1: Major data types in BMW.

According to ANSI C semantics, all subword integers are automatically promoted to register-length integers before conducting any arithmetic operations. This is known as integral promotion [32] and is implemented in most commercial compilers. In terms of vectoriza-

tion, integral promotion of subword types can waste more than half of the total computation bandwidth as well as incur the additional overhead of integer extension. Furthermore, multimedia extensions often provide better support for subword operations than for 32-bit integer operations. For example, SSE2 supports `max` and `min` for 8- and 16-bit integers, but not for 32-bit integers. Therefore, any efficient MME vectorizer needs to be able to eliminate redundant integral promotion without affecting the program semantics.

Besides eliminating unnecessary integral promotions, there are other opportunities to optimize SIMD instructions on subword computation. Consider the following operation from GSM,

$$vc = ((va * vb) + 16384) \;>>\; 15$$

where `va`, `vb` and `vc` are vectors of shorts.

In SSE2, a 16-bit multiply is implemented by two native operations, `MultiplyLow` and `MultiplyHigh`, to produce the low and high halves of the 32-bit result as follows,

$$\begin{aligned} vx &= MultiplyLow(va, vb) \\ vy &= MultiplyHigh(va, vb) \\ va * vb &= vy << 16 + vx \end{aligned}$$

where vx and vy are both vectors of shorts [30].

By exploiting the arithmetic properties of ">>", "<<", and "+", we can simplify the original computation as follows,

$$\begin{aligned} &((va * vb) + 16384) >> 15 \\ &\equiv ((vy << 16 + vx) + 16384) >> 15 \\ &\equiv (vy << 2 + vx >> 14 + 16384 >> 14) >> 1 \\ &\equiv (vy << 1) + (vx >> 14 + 1) >> 1 \end{aligned}$$

The key to this simplification is the distribution of right shift over add. In general, a right shift can be distributed into an add only if at least one operand of the add has more trailing zeros than the number of bits to be shifted. Note that the constant 16384 has 14 trailing zeros. That is why we decompose the original 15-bit shift into a 14-bit shift and a 1-bit shift and distribute the 14-bit shift into the add.

In our experiment, such subword arithmetic optimization can achieve additional 16-18% performance improvements on the core procedures.

14.3.2 Identifying Saturated Operations

Saturated arithmetic is widely used in multimedia programs, especially in audio and image processing applications. Since C does not support saturated arithmetic as native operation, programmers must express saturated operations using other operations.

Figure 14.3 gives one implementation of saturated add in C. Using if-conversion, the code fragment in Figure 14.3 can be vectorized into a sequence of compare, mask, subtract and add. However, for multimedia extensions with native support for saturated arithmetic, the best performance can only be achieved by recognizing the sequence and transforming it into a saturated add instruction. Idiom recognition can be extended to identify these saturated operations [6, 33].

```
/* MAX_WORD and MIN_WORD are constants */
/* short a, b; int ltmp; */
#define GSM_ADD(a, b)  ((unsigned)((ltmp=(int)(a)+(int)(b))
   - MIN_WORD) > (MAX_WORD - MIN_WORD) ?
   (ltmp > 0 ? MAX_WORD : MIN_WORD) : ltmp )
```

Figure 14.3: Saturated add in GSM.

Within the BMW benchmark, there are also other implementations of saturated operations. Figure 14.4 gives such an example. There array `Clip` is a constant array generated during initialization that maps a subscript to its corresponding saturated 8-bit value. It seems more difficult for a compiler to recognize this type of saturated operation.

```
/* short *bp; char *rfp; */
1.  for (i=0; i<8; i++)
2.    for (j=0; j<8; j++) {
3.      *rfp = Clip[*bp++ + *rfp];
4.      rfp++;
5.    }
```

Figure 14.4: Saturated Add in MPEG2.

14.4 Experimental Results of Manual Vectorization

To evaluate the effectiveness of our techniques, we selected 34 core procedures from the BMW benchmarks as candidates for applying manual vectorization. The characteristics of these core procedures are summarized in Tables 14.2 and 14.4. The vectorized codes invoke SSE2 operations via intrinsic functions and are subsequently compiled by the Intel compiler (v8.0).

During the manual transformation, we made the following assumptions. First, global pointer information is available for the procedures to be transformed. This information could be obtained either through compiler analysis or provided by programmers via pragmas. Second, a powerful idiom recognizer is employed to identify code patterns such as `min`, `max`, `average`, and saturated arithmetic. In essence, the manual transformation is designed to evaluate the effectiveness of vectorization transformations without being limited by the analysis-related issues discussed in Section 14.1.

14.4.1 Core Procedures Vectorized

Of the 34 core procedures, we are able to vectorize the 23 procedures listed in Table 14.2. Performance (measured in speedups) of the vectorized procedures is given in Table 14.3. The table is divided into 3 sections. The first section contains procedures that can be vectorized without algorithm change and achieves 10% to 239% of improvement after vectorization. The middle section contains procedures that require algorithm changes to be vectorized. The last section contains procedures that show little performance speedups or even slowdowns as the

result of vectorization.

In Table 14.3, column "Best" presents the best speedups we have achieved by applying all transformations to the procedure. The next columns, "-VCP", "-VOL", and "-SAC", show the speedups achieved if individual vectorization optimizations are excluded. For some procedures, we can achieve much better performance by replacing floating point algorithm with a fixed point algorithm that is more suitable for vectorization. Column "Alg." shows the speedups resulting from such algorithm substitution. For example, in `fdct`, we achieved an additional speedup of 3.23(=9.41/2.91). Column "ParP" shows the speedup resulting from vectorization of linear recurrences using parallel prefix. This transformation assumes associativity of saturated arithmetic.

We also measured the performance of the Intel compiler (v8.0) vectorization. The last column, "ICC", shows the performance achieved by vectorizing the procedure using the option listed in Table 13.1. This column only shows the speedups of those procedures that have at least one loop vectorized by the Intel compiler. The Intel compiler successfully vectorizes all innermost loops in `dist1` from `MPEG.Encoder` and reaches near-optimal speedup on this procedure. However, it has little or negative impact on other core procedures. [1]

Some user optimizations, such as if-shortcut and lookup tables, preclude vectorization on otherwise vectorizable programs. In general, it is very difficult for the compiler to reverse engineer such optimizations. Therefore we segregated procedures that require manual "de-optimization" to be vectorized from the others to the middle section of Table 14.3. For example, `quantize` uses both lookup table and if-statement shortcuts, and `ycc_rgb_convert` uses a lookup table to implement saturated operations.

About one third of the vectorized procedures show little speedups over the original scalar codes as shown at the bottom section of Table 14.3. This is partly due to the inefficient implementation of SSE2. As discussed in Section 2.2, the theoretical maximum speedups

[1] Some core procedures, such as `synth_1to1` from `mpg123`, benefit from SSE2 extension without vectorization.

Application	Core Procedures	%Ex	Loops	Ptr	SatOp	MT	Lookup	ManOpt
GSM.E	`Calculation_of...`	36.51	1x2	P		*		Unroll
GSM.E	`Short_Term_Ana...`	37.30	1x2	P	*			
GSM.D	`Short_Term_Syn...`	74.00	1x2	P	*			
LAME	`quantize`	17.38	3x1	P	*	*	Math	Short
LAME	`calc_noise2`	13.30	2x2	P	*	*	Math	Short
mpg123	`synth_1to1`	57.92	2x1	P	*	*		Unroll
MPEG2.E	`dist1`	43.80	4x2	P		*		Unroll
MPEG2.E	`fdct`	25.38	2x3	P		*		
MPEG2.D	`form_component...`	12.14	4x2	P		*		
MPEG2.D	`idct`	35.49	2x0	P		*	*	Unroll
MPEG2.D	`Dither_Frame`	11.57	4x2	PA		*	SatOp	
JPEG.D	`ycc_rgb_convert`	16.67	1x2	PA	*	*		
JPEG.D	`jpeg_idct_islow`	38.89	2x1	PA		*	SatOp	Unroll
DJVU.E	`IWPixmap::init`	15.24	3x2	P	*	*		
DJVU.E	`forward_filter`	65.65	3x1	P		*		
DJVU.D	`backward_filter`	83.99	3x1	P		*		
Morph3D	`gl_shade_rgba_fast`	13.77	1x2	P		*		
Reflect	`persp_textured...`	48.70	2x3	P		*		
Pointblast	`gl_depth_test_s...`	27.22	12x1	PA				
POVray	`Dnoise`	15.49	1x0	PA		*	*	
Timidity	`rs_vib_loop`	29.31	1x2	PA		*		
Timidity	`mix_mystery_signal`	10.18	2x2	PA		*		
Timidity	`mix_single_signal`	12.75	2x2	PA	*	*		

Table 14.2: Characteristics of core procedures successfully vectorized.

(Abbr: %Ex-Percentage of Execution time; Loops-Important loops in the procedure, where mxn means there are m n-nest important loops in the procedure; E-Encoder; D-Decoder; Ptr-Pointer reference; P-Pointers as parameters; PA-Pointer Arithmetic; SatOp-Saturated Operations; MT-Mixed Types; Lookup-Table Lookups; ManOpt-Manual Optimizations; Unroll-Loop Unrolling; Short-Short cuts;)

Application	Core Procedures	%Ex	Best	-VCP	-VOL	-SAO	Alg.	Par	ICC
GSM.E	`Calculation_of...`	36.51	2.46		1.85				0.69
GSM.E	`Short_Term_Ana...`	37.30	2.08	0.88		1.79		2.59	
GSM.D	`Short_Term_Syn...`	74.00	2.97	1.27		2.51		3.46	
LAME	`calc_noise2`	13.30	1.52						0.95
mpg123	`synth_1to1`	57.92	1.75	1.28					
JPEG.D	`jpeg_idct_islow`	38.89	1.10				1.44		
MPEG2.E	`dist1`	43.83	3.39	3.37		3.20			3.05
MPEG2.E	`fdct`	25.38	2.91				9.41		0.87
MPEG2.D	`form_component...`	12.14	2.12						0.90
MPEG2.D	`idct`	35.49	1.18				3.68		
DJVU.E	`IWPixmap::init`	15.24	1.95		1.66				1.04
Reflect	`persp_textured...`	48.70	1.40						
Pointblast	`gl_depth_test_s...`	27.22	1.45						0.99
Timidity	`mix_mystery_signal`	10.18	1.15						
LAME	`quantize`	17.38	2.00						1.17
JPEG.D	`ycc_rgb_convert`	16.67	1.20		1.07				
MPEG2.D	`Dither_Frame`	11.57	0.71	0.71					
DJVU.E	`forward_filter`	65.65	1.00						
DJVU.D	`backward_filter`	83.99	1.00						
Morph3D	`gl_shade_rgba_fast`	13.77	0.89						
POVray	`Dnoise`	15.49	0.99						
Timidity	`rs_vib_loop`	29.31	1.01						
Timidity	`mix_single_signal`	12.75	0.45						

Table 14.3: Performance summary of manual vectorization on core procedures.

(Abbr: Best-The best speedup; -VCP-exclude Vector Copy Propagation; -VOL-exclude Vectorizing Outer Loops; -SAC-exclude Subword Arithmetic Optimization; Alg.-Algorithm Changes; Par-Parallel Prefix (Assuming the associativity of saturated arithmetic);)

achievable on vectorizing int, short and char operations, except for multiplication, are 1, 2 and 4, respectively. Furthermore, many core procedures have been optimized by hand, such as using lookup tables, and need to be "de-optimized" to be vectorized. However, sometimes the performance gains of vectorizing un-optimized algorithm may not make up the performance loss of "de-optimization". Finally, the overhead of data reorganization, necessary to vectorize reduction and non unit-stride memory accesses (for example, `mix_single_signal`), can also offset the performance benefit of vectorization.

14.4.2 Non-Vectorizable Core Procedures

There are 11 core procedures (as listed in Table 14.4) that cannot be vectorized. Among them, 6 procedures have dependence cycles that cannot be expressed as either reduction or linear recurrences. For example, Figure 14.5 gives the simplified dependence graph of the ADPCM encoder. In the figure, the circles presents the different assignment to variables and the links shows the dependences between them. And the dotted lines represent complex paths involving operations such as table lookups and multiple if-statements.

Application	Core Procedures	%Ex	Loop	Ptr	SatOp	MT	Lookup	Reason
ADPCM.E	`adpcm_coder`	100.0	1x1	P	*	*	*	D-Cycles
ADPCM.D	`adpcm_decoder`	100.0	1x1	P	*	*	*	D-Cycles
JPEG.E	`encode_mcu_AC_refine`	37.38	1x2	P		*		D-Cycles
JPEG.E	`encode_mcu_AC_first`	14.20	1x2	P		*		D-Cycles
JPEG.D	`decode_mcu_AC_refine`	22.22	1x2	P		*	SatOp	D-Cycles
Rsynth	`parwave`	70.49	1x2	P				D-Cycles
Gears	`flat_TRUECOLOR_...`	81.34	1x3	PA				I/O
Morph3D	`smooth_TRUECOLOR_...`	10.05	1x3	PA				I/O
Pointblast	`write_span_mono_x...`	29.44	1x1	PA				I/O
Pointblast	`dist_atten_anti...`	18.33	2x3	PA		*		Nonclose
DOOM	`R_DrawColumn`	25.88	1x1	P			*	Lookup

Table 14.4: Characteristics of non-vectorizable core procedures.

In addition, three procedures from `Mesa` frequently invoke the I/O function in their hot loops. Due to the overhead of transferring between scalar and vector registers, it is not

beneficial to vectorize the other parts of the loop while leaving the function call in scalar form.

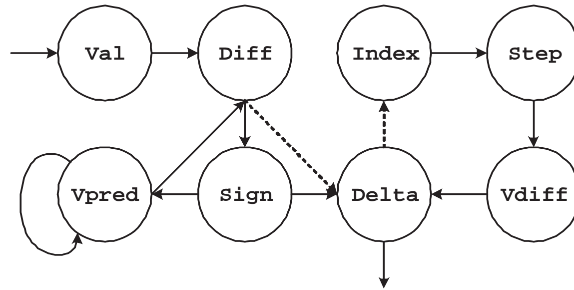

Figure 14.5: Dependence graph of ADPCM encoder.

Chapter 15

Conclusion

Due to the special architectural features of today's SIMD devices, it is extremely difficult to automatically generate efficient code for these devices, even when data parallelism has been extracted from sequential programs and expressed in vector representation. There are many new issues arising in translating vector statements into SIMD instructions and they must be addressed to achieve a performance comparable with other low-level programming methods, like using intrinsic functions.

VINCI, or Vector I-code Novel Compilation Infrastructure, is proposed in this thesis to translate vector I-code programs into efficient SIMD instructions. VINCI consists of many program transformations. Those transformations can be roughly classified into several groups. The first group is to normalize generic vectors with arbitrary length, strides and alignment settings into the specific format required by SIMD devices. The second group is to optimize either vector programs or SIMD instructions to improve the performance. In this group, some transformations are newly designed algorithms and the others are extended from well-known general or domain-specific optimization techniques. The last group includes all other supportive compiler routines, which conduct various analysis on vector programs to collect enough information for other transformations.

Among all program transformations and optimizations, the optimization algorithm on data permutations is the main focus of this thesis. Due to the constraints on memory units, the overhead of data permutations makes it extremely difficult to achieve peak performance on SIMD devices. The strategy used in VINCI includes three steps to optimize all forms of data permutations in a vector program. First, all data permutations are converted into

an explicit unified representation. Second, the optimization algorithm tries to reduce data permutations in the vector program by propagating them across statements and merging consecutive ones. Finally, the code generation algorithm translates general permutation operations into native permutation instructions supported by the target SIMD device. The experimental results show that the performance of SIMD computation can be significantly improved by our optimization strategy.

VINCI was implemented on the HiLO compiler and then evaluated on two platforms, VMX and SSE2. Applications from various domains are used in the experiment. For all applications tested, SIMD code outperforms scalar version after all optimizations in VINCI being applied. Near-peak speedups have been achieved on some applications, including those complicated ones such as FFT and bitonic sorting.

15.1 Future Work

As mentioned in the thesis, VINCI is only the first step to provide an easy, simple programming method to utilize SIMD devices efficiently. There are still a lot of other issues that need to be addressed to fully unleash the computing power of SIMD devices. Some of them are discussed in Chapter 14. Hence, an intermediate next step is to generalize these techniques and include them in VINCI.

On the other hand, there are still some limitations in VINCI. For example, the current presentation of data permutations requires that strides, alignment, and vector length (essentially permutation pattern) must be known at compile-time. Relaxing this restriction is an important next step. Besides, the optimization algorithm is conservative on data permutations with special elements $\star$ and $\diamond$. It would be interesting to explore more aggressive optimization strategies on this type of data permutations.

In addition, limited by the input language, VINCI was only evaluated on a limited number of applications. It would also be interesting to migrate VINCI into an existing vectorizing

compiler to improve the quality of SIMD code.

In the experiment, we have seen in many applications that the data permutation optimization often interacts with other compiler transformations. It would also be valuable to explore such interaction.

References

[1] *Fortran, Part 1, Base Language, ISO/IEC 1539-1*, 2004.

[2] Advanced Micro Devices. *3DNow! Technology Manual*, 2000.

[3] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[4] Leonardo Bachega, Siddhartha Chatterjee, Kenneth A. Dockser, John A. Gunnels, Manish Gupta, Fred G. Gustavson, Christopher A. Lapkowski, Gary K. Liu, Mark P. Mendell, Charles D. Wait, and T. J. Chris Ward. A high-performance SIMD floating point unit for BlueGene/L: Architecture, compilation, and algorithm design. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 85–96, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Aart J. C. Bik. *The Software Vectorization Handbook : Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[6] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic detection of saturation and clipping idioms. In *LCPC '02: Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computers*, 2002.

[7] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the Intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[8] CCIR Recommendation 601-2. *Encoding Parameters of Digital Television for Studios*, 1990.

[9] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28. ACM Press, 1993.

[10] Gerald Cheong and Monica Lam. An optimizer for multimedia instruction sets. In *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[11] Paul Cockshott. Vector pascal an array language for multimedia code. In *APL '02: Proceedings of the 2002 conference on APL*, pages 83–91, New York, NY, USA, 2002. ACM Press.

[12] CodePlay. *VectorC PC Overview*, 2004. CPV.

[13] Crescent Bay Software. *VAST-C/AltiVec: Automatic C Vectorizer for Motorola AltiVec*, 2004. VAST.

[14] E. Dahlhaus, D. S. Johson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM J. Computing*, 23:864–894, 1994.

[15] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 82–93. ACM Press, 2004.

[16] Randall James Fisher. *General-purpose SIMD within a Register: Parallel Procesing on Consumer Microprocessors*. PhD thesis, Purdue University, 2003.

[17] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W. Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93(2):409–425, 2005.

[18] Franz Franchetti and Markus Puschel. A SIMD vectorizing compiler for digital signal processing algorithms. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS) 2002*, 2002.

[19] Bjorn Franke and Michael O'boyle. Array recovery and high-level transformations for DSP applications. *Trans. on Embedded Computing Sys.*, 2(2):132–162, 2003.

[20] Matteo Frigo. A fast Fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 169–180. ACM Press, 1999.

[21] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[22] Sam Fuller. Motorola's AltiVec technology. Technical report, Motorola Inc., 1998.

[23] R. Govindarajan, Hongbo Yang, Jose Nelson Amaral, Chihong Zhang, and Guang R. Gao. Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. *IEEE Trans. Comput.*, 52(1):4–20, 2003.

[24] Stefan Hacker. Static superscalar design: A new architecture for the TigerSHARC DSP processor. Technical report, Analog Devices, 2006.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[26] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. An array operation synthesis scheme to optimize FORTRAN 90 programs. In *PPOPP '95: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 112–122. ACM Press, 1995.

[27] IBM. *IBM PowerPC 970FX RISC Microprocessor User's Manual*, 2005.

[28] Intel Corporation. *Intel Math Kernel Library*, 2006 edition.

[29] Intel Corporation. *IA32 Intel Architecture Optimization*, 2004.

[30] Intel Corporation. *IA32 Intel Architecture Software Developer's Manual (Volume 1: Basic Architecture)*, 2004.

[31] Intel Corporation. *Intel Integrated Performance Primitives for Intel Architecture*, 2006.

[32] International Standard Organization. *Programming Languages - C, ISO/IEC 9899*, 1999. ISO.

[33] Weihua Jiang, Chao Mei, Bo Huang, Jianhui Li, Jiahua Zhu, Binyu Zang, and Chuanqi Zhu. Boosting the performance of multimedia applications using SIMD instructions. In *CC '05: Proceedings of the 14th International Conference on Compiler Construction*, 2005.

[34] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49:589–604, 2005.

[35] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures*. Morgan-Kaufmann Publishers, 2002.

[36] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The visual instruction set VIS in UltraSPARC. In *COMPCON '95: Proceedings of the 40th IEEE Computer Society International Conference*, page 462, Washington, DC, USA, 1995. IEEE Computer Society.

[37] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[38] Alexei Kudriavtsev and Peter Kogge. Generation of permutations for SIMD processors. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 147–156. ACM Press, 2005.

[39] Atsushi Kunimatsu, Nobuhiro Ide, Toshinori Sato, Yukio Endo, Hiroaki Murakami, Takayuki Kamei, Masashi Hirano, Fujio Ishihara, Haruyuki Tago, Masaaki Oka, Akio Ohba, Teiji Yutaka, Toyoshi Okada, and Masakazu Suzuoki. Vector unit architecture for emotion synthesis. *IEEE Micro*, 20(2):40–47, 2000.

[40] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[41] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 145–156. ACM Press, 2000.

[42] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and detecting memory address congruence. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29. IEEE Computer Society, 2002.

[43] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[44] Rudy Lee and Larry McMahan. Mapping of application software to the multimedia instructions of general-purpose microprocessors. In *Proceedings of IS&T/SPIE Symposium on Electric Imaging: Multimedia Hardware Architectures 1997*, pages 122–133, 1997.

[45] Rainer Leupers. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools.* Kluwer Academic Publishers, 2000.

[46] David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17(10-11):1223–1244, 1991.

[47] Xiaoming Li, Maria Jesus Garzaran, and David Padua. Optimizing sorting with genetic algorithms. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 99–110. IEEE Computer Society, 2005.

[48] John Montrym and Henry Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, 2005.

[49] Motorola Inc. *AltiVec Technology Programming Environments Manual*, 1998.

[50] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[51] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP architecture. In *CASES '03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 2–11. ACM Press, 2003.

[52] Manikandan Narayanan and Katherine A. Yelick. Generating permutation instructions from a high-level description. In *MSP '04: Proceedings of the 6th Workshop on Media and Streaming Processors*, 2004.

[53] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[54] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, 2006. to appear.

[55] The Portland Group Compiler Technology. *PGI User's Guide : Parallel Fortran, C and C++ for Scientists and Engineers*, 2004. PGI.

[56] Markus Puschel, Jose M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[57] Gang Ren, Peng Wu, and David Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *IPDPS '05: Proceedings of the 19th International Parallel & Distributed Processing Symposium*, 2005.

[58] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for simd devices. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 118–131, New York, NY, USA, 2006. ACM Press.

[59] Nicholas Rizzolo and David Padua. HiLO: High level optimization of FFTs. In *LCPC '04: Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.

[60] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[61] Harsh Sharangpani and Ken Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, 2000.

[62] Nathan T. Slingerland and Alan Jay Smith. Design and characterization of the berkeley multimedia workload. *Multimedia Systems*, 8(4):315–327, 2002.

[63] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–294. ACM Press, 2005.

[64] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–300, 2000.

[65] Richard M. Stallman. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, 2003.

[66] Texas Instruments. *TMS320C64x Technical Overview*, 2001.

[67] Lewis W. Tucker and George G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26–38, 1988.

[68] Peng Wu, Albert Cohen, Jay Hoeflinger, and David Padua. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 78–91. ACM Press, 2001.

[69] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 153–164. IEEE Computer Society, 2005.

[70] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 169–178, New York, NY, USA, 2005. ACM Press.

[71] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. SPL: a language and compiler for dsp algorithms. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 298–308. ACM Press, 2001.

[72] Hans Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

Author's Biography

Gang Ren was born in Hangzhou, China. He received his Bachelor of Engineering degree in Computer Science from Zhejiang University, Hangzhou, China in May 1998. In August 1998, Gang started his graduate study in the Institute of Computing Technology of the Chinese Academy of Sciences, Beijing, China, under the direction of Prof. Zhaohua Feng. He earned his Master of Science degree in Computer Science in May 2001. The title of his master thesis is *Evolutionary Multiprocessor Task Schedulers*. In August 2001, Gang continued his graduate study in the Department of Computer Science at the University of Illinois at Urbana-Champaign, under the direction of Prof. David Padua. During his five-year Ph.D. study, Gang worked as an intern in IBM T.J. Watson Research Center with Peng Wu and Alexandre Eichenberger on SIMD compilation in 2002 and 2005 respectively. Gang will join Google after graduation.