Data Layout Oriented Compilation Techniques in Vectorization for Multi-/Many-cores

by

Shixiong Xu

Dissertation

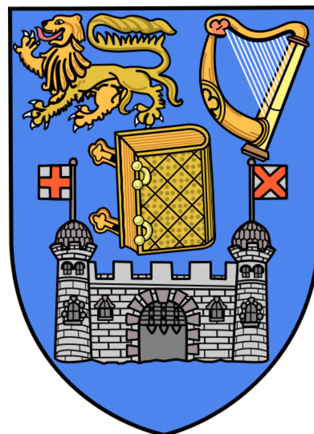
Submitted to the School of Computer Science and Statistics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Science)

School of Computer Science and Statistics

TRINITY COLLEGE DUBLIN

September 2017



Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the Universitys open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgment.

. . . .

Shixiong Xu

Dated: August 26, 2017

Abstract

Single instruction, multiple data (SIMD) architectures are widely adopted in both general-purpose processors and graphic processing units for exploiting data-level parallelism. It is tedious and error-prone for programmers to write high performance code to utilize SIMD execution units on both platforms. Therefore, users often rely on automatic code generation techniques in compilers. However, it is not trivial for compilers to generate high performance code without considering the data layout of the data used in the computation. Data layout determines data access patterns, and in turn have a great impact on the memory performance of the automatically generated code for both CPUs and GPUs.

In this thesis, we demonstrate several data layout oriented compilation techniques for efficient vectorization. We put forward semi-automatic data layout transformation to help users to easily change their program, and exploit the best possible data layout in terms of vectorization. Our proposed vectorization based on hyper loop parallelism provides a way to take advantage the relationship between data layout and computation structure. The experimental results demonstrated that this vectorization technique can yield significant performance gain. In addition, we show that this technique is of great use to boost the memory performance on CUDA GPUs.

We also present pioneering work that uses loop vectorization techniques to handle nested thread-level parallelism (TLP) on CUDA GPUs. As loop vectorization prioritizes vectorizing loops with contiguous data accesses, it is of great help to achieve an efficient mapping strategy for nested TLP on CUDA GPUs.

Our new bitslice vector computing for customizable arithmetic precision on general-purpose processors with SIMD extensions not only breaks the limit of hardware arithmetic precision but also achieves great performance. It also shows the great power of logic optimization widely used in hardware synthesis in optimizing C/C++ code with a large amount of logic operations.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr David Gregg. His enthusiasm and encouragement ensured the completion of this thesis. As English is my second language, he spent substantial amounts of time and effort on correcting grammatical and spelling errors in my papers for conferences and thesis. In addition, his rich experience on research is inevitable to the completion of my PhD. He is one of the most hard-working supervisors I have ever worked with. His dedication to research sets a great example to me. Without him, it is impossible for me to get papers fully revised and submitted in time.

I would also like to thank the members of the Software Tools Group who have been great company and of great assistance through the years. Particular thanks to Andrew Anderson, Aravind Vasudevan, Mircea Horea Ionică, Martin Marinov and Servesh Muralidharan for helping me improve my English significantly and get used to the life in Dublin quickly.

Finally, my greatest appreciation is reserved for my mother for her support during my study for the PhD. This thesis is also dedicated to my late father.

SHIXIONG XU

University of Dublin, Trinity College

September 2017

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **21**

1.1 Our Thesis 21

1.2 Contributions 24

1.3 Thesis Structure 25

1.4 Published Work 25

2 Background and Literature Review **27**

2.1 SIMD Architectures 27

2.1.1 Single Instruction Multiple Data (SIMD) 27

2.1.2 SIMD Instruction Set Extensions for Multimedia 28

2.2 Graphics Processing Units 30

2.3 SIMD Extensions vs SIMT in CUDA GPUs 31

2.3.1 Single instruction, multiple register sets 31

2.3.2 Single instruction, multiple addresses 32

2.3.3 Single instruction, multiple control flow paths 33

2.4 Compilation Techniques for Automatic Vectorization 35

2.4.1 Loop Vectorization 35

2.4.2 Super-word Level Parallelism Vectorization 38

2.4.3 Whole-function Vectorization 39

2.4.4 Other Automatic Vectorization Techniques 40

2.5 CUDA and OpenACC 42

2.5.1 CUDA Programming Model 42

2.5.2 Compiler Directive Based Programming Model — OpenACC 44

2.6 Summary 45

3 Semi-automatic Data Layout Transformations for Loop Vectorization 47

3.1 Introduction . . . 47
3.2 Language Support for Data Layout Transformations . . . 48
3.2.1 Motivating Examples . . . 48
3.2.2 Data Layout Transformation Pragmas . . . 51
3.2.3 Composition of Data Layout Transformations . . . 55
3.3 Data Layout Aware Loop Transformations . . . 57
3.4 Experimental Evaluation . . . 58
3.4.1 Implementation . . . 58
3.4.2 A Case Study: data layout tuning for loop vectorization . . . 59
3.5 Related Work . . . 62
3.6 Summary . . . 64

4 Exploit Computation Structure Exposed by Data Layout in Vectorization 65

4.1 Introduction . . . 65
4.2 Hyper Loop Parallelism in Vectorization . . . 68
4.2.1 Overview . . . 68
4.2.2 Vectorization Analysis . . . 70
4.2.3 Vectorization Transformation . . . 72
4.3 Implementation . . . 77
4.4 Preliminary Experimental Results . . . 79
4.4.1 Experimental Setup . . . 79
4.4.2 Benchmarks . . . 79
4.4.3 Performance . . . 80
4.5 Related Work . . . 81
4.6 Summary . . . 82

5 Boost Memory Performance with HLP in Vectorization for CUDA GPUs 85

5.1 Introduction . . . 85
5.2 Hyper Loop Parallelism in Vectorization . . . 88
5.2.1 Hyper Loop Parallelism . . . 89
5.2.2 Hyper Loop Parallelism in Vectorization . . . 90

5.3 Hyper Loop Parallelism on the CUDA GPU 94
5.3.1 SIMD Vectors 94
5.3.2 SIMD Operations 95
5.3.3 Mapping Execution Model 98
5.4 Implementation 101
5.5 Performance Evaluation 102
5.5.1 Experiment Setup 102
5.5.2 Test-cases 102
5.5.3 Performance Evaluation and Analysis 102
5.6 Related Work 107
5.7 Summary 109

6 Loop Vectorization for Nested Thread-level Parallelism on CUDA GPUs 111

6.1 Introduction 111
6.2 Loop Vectorization for Nested TLP on GPUs 114
6.2.1 Motivation 114
6.2.2 New SIMD vector abstraction of GPU execution model 115
6.2.3 Thread-Reuse Execution Model 116
6.2.4 Advantages of loop vectorization for nested TLP 121
6.3 Loop Vectorization Framework for Nested TLP 122
6.3.1 Language Extension 122
6.3.2 Vectorization Analysis 123
6.3.3 Vectorization Transformation 125
6.4 Evaluation 132
6.4.1 Experimental Methodology 132
6.4.2 Experimental Results 132
6.5 Related Work 137
6.6 Summary 139

7 Fine-grained AoS-to-SoA for Customizable Precision Arithmetic 141

7.1 Introduction 141
7.2 Software Bitslice Representations 143

7.3 Bitslice Vector Computing 144
7.4 Operating on Bitslice Vectors 145
7.4.1 Basic operations 147
7.5 Bitslice Floating Point Vector Operations 148
7.5.1 IEEE-754 Floating Point Format 149
7.5.2 Bitslice Floating-point Operators 150
7.6 Code Generator and Optimization 154
7.6.1 Code Generation Framework 154
7.6.2 Logic Optimization 156
7.7 Experimental Evaluation 159
7.7.1 Experiment Methodology 159
7.7.2 Performance of Building Blocks 159
7.7.3 Performance of BFP Operations 160
7.7.4 Performance of Real-world Applications 164
7.8 Related Work 168
7.9 Summary 170

8 Conclusion and Final Thoughts **173**
8.1 Future Work 174
8.1.1 Integrate Semi-automatic Data Layout Transformation into Performance Auto-tuning Systems 174
8.1.2 Seamless Data Layout Transformations for C++ Code 174
8.1.3 A Source-to-source Vectorizing Compiler for Bitslice Vectors 175
8.1.4 Exploit More Logic Optimization for Bitslice Vector Operators . . . 175
8.2 Final Thoughts 176

List of Figures

1-1 Data layouts: Array-of-Structures(AoS) and Structure-of-Arrays(SoA) . . . 22

1-2 The performance impact of strided memory access on effective memory bandwidth on two Nvidia CUDA GPUs. 23

2-1 Single instruction multiple data (SIMD). 28

2-2 Add two vectors of numbers in SIMD. 31

2-3 Add two vectors of numbers in SIMT. 32

2-4 Sparse matrix vector multiplication in the format of compressed sparse row (CSR) in SIMT. 33

2-5 Computation with control flow in SIMT. 34

2-6 Thread divergence in SIMT. 34

2-7 Vectorization on the loop in transposed matrix vector multiplication with vectorization factor 4. 36

2-8 Example of SIMD-enabled functions in Intel's C/C++ compiler. 39

2-9 CUDA execution model. 43

2-10 CUDA memory model. 43

2-11 The parallel loop annotated with OpenACC pragmas for the transposed matrix vector multiplication (TMV). 45

3-1 The kernel of function tzetar() in the SP of NPB. 49

3-2 Syntax of the data layout transformation pragma. 51

3-3 Loop transformation without considering data layout. 57

3-4 Data layout aware loop transformation. 58

3-5 Performance of `tzetar()` with different data layout transformations 60

3-6 Performance of the SP in different data layouts. 60

3-7 Performance of the SP of the NAS Parallel Benchmarks. 61

3-8 Performance breakdown of the double precision SP of the NAS Parallel Benchmarks. 61

3-9 Performance breakdown of the single precision SP of the NAS Parallel Benchmarks. 62

4-1 C-Saxpy 66

4-2 C-Saxpy by classic loop vectorization. 66

4-3 C-Saxpy by hyper-loop parallelism vectorization 66

4-4 Hyper loop parallelism for vectorization. 69

4-5 Collect program slices. 70

4-6 Group program slices. 71

4-7 Overlapping of fully grouped slices. 73

4-8 Reducible and scatterable computation attributes. 73

4-9 Expand program slices. 74

4-10 Global SIMD lane-wise optimization. 76

4-11 Compilation flow of hyper loop parallelism vectorization. 77

4-12 An example of code generation. 78

4-13 Performance of Group I benchmarks. 80

4-14 Performance of Group II benchmarks. 81

5-1 The performance impact of stride memory access on effective memory bandwidth. 86

5-2 C-Saxpy on the data organized in an array of structures (AoS). 88

5-3 CUDA code for C-Saxpy generated by the Cetus compiler. 88

5-4 Identification of parallel hyper loop iterations with backward program slicing. `y[2*i]` and `y[2*i+1]` are the slicing criteria. This is the same as HLP vectorization for CPUs shown in Fig. 4-5. 90

5-5 Group slices in hyper loop parallelism vectorization. This is the same as HLP vectorization for CPUs shown in Fig. 4-6. 91

5-6 Illustration of vectorization expansion with a loop unrolling factor 32. The CUDA warp size is 32. Note the loop unrolling factor is different from the one used in HLP vectorization for CPUs depicted in Fig. 4-9. 92

5-7 Global SIMD lane optimization. Note the length of SIMD vectors is different from the one used in HLP vectorization for CPUs depicted in Fig. 4-10. 92

5-8 Vector dot operation. 96

5-9 Stride-3 data layout transformation via shared memory. 97

5-10 Runtime generation of masks for intra-vector shuffle operations. 98

5-11 Mapping SIMD lanes representing hyper loop parallelism to GPU threads. 99

5-12 The overall compilation flow of hyper loop parallelism vectorization for the CUDA GPU. The compiler passes in the dotted boxes are what we introduce to the Cetus compiler. 101

5-13 The performance of loops with unit-stride data access on Jetson TK1. The CUDA block size is 128. 103

5-14 The performance of loops with unit-stride data access on GeForce GTX 645. The CUDA block size is 128. 104

5-15 The performance of C-Saxpy on Jetson K1 and GeForce 645. The CUDA block size is 128. 105

5-16 The performance of kernels with stride-3 data access on Jetson TK1. The CUDA block size is 128. 106

5-17 The performance of kernels with stride-3 data access on GeForce GTX 645. The CUDA block size is 128. 107

6-1 Naive mapping from parallel loop nest to the CUDA execution model. . . 112

6-2 The parallel loop annotated with OpenACC pragmas for the transposed matrix vector multiplication (TMV). Note that the `vector` clause in the example is our language extension for nested TLP (discussed in Sec. 6.3.1). 114

6-3 Our proposed hierarchical segmented vectors for CUDA GPUs (warp size is 4 and thread block size is 8). 116

6-4 Comparison of execution models for nested TLP on CUDA GPUs. Assume *i*-loop is the outermost parallel loop, *j*-loop is the nested parallel loop, data access is contiguous across iterations of *j*-loop. For simplicity, each thread block contains 8 GPU threads and the number of iterations of *j*-loop is 5. . 119

6-5 Our proposed two execution modes of thread-reuse execution model, *partial* and *full*. 120

6-6 Virtual simply nested loops in vectorization analysis. Assume *k*-loop is suitable for inner-loop vectorization while *h*-loop is suitable for outer-loop vectorization. 124

6-7 Optimization on the execution guard for nested TLP. The execution guard `i < SIZE` is distributed over the three parts of the original loop body. . . . 127

6-8 The generated CUDA code by inner-loop vectorization in partial mode. The vectorization factor is 32 and loop unrolling factor is 4. 128

6-9 The generated CUDA code by outer-loop vectorization in full mode. The vectorization factor is 32 , block size (TB) is 128, and the max block size (TB_{max}) is 1024. 130

6-10 Performance comparison between our vectorization approach and Yang and Zhou's method [Yang and Zhou, 2014] (TB = 128) 134

6-11 Performance comparison between our approach and PGI compiler. The `loop seq` directive is used to disable the default handling of nested TLP in PGI compiler. 135

6-12 Performance impact of loop unrolling factor in partial mode. VF for Kmeans and MV is 16 and 32. 135

6-13 Performance impact of vectorization factor and TB_{max} in full execution mode in Backprop. The block size TB is set to vectorization factor VF. . . . 136

7-1 Standard and bitslice representation of an array of sixteen 8-bit floating-point numbers. 143

7-2 Bitslice adder for two arrays of unsigned integers. Each integer has ADD_-BITS bits. The size of uint32_t decides the number of array elements are being processed. 144

7-3 Two versions of code that negate the n'th bit of each element of a vector of thirty-two 16-bit integers. The first code fragment operates on the standard representation of arrays of numbers. The second operates on bitslice vectors. 146
7-4 Floating-point word. 149
7-5 Bitslice floating-point vector types for FP32. 150
7-6 AIG representation of boolean network 156
7-7 Output of ABC logic optimization for a 4-bit unsigned integer multiplication. AND2X1, OR2X1, XOR2X1, ANDNOT2X1, INVX1 are C macros for logic instructions supported by the processor. 158
7-8 Performance of unsigned integer addition with bit sizes from 4 to 32. . . . 161
7-9 Performance of unsigned integer subtraction with bit sizes from 4 to 32. . . 161
7-10 Performance of unsigned integer multiplication with bit sizes from 4 to 32. 162
7-11 Performance of bitslice floating-point addition/subtraction with bit sizes from 8 to 28. 163
7-12 Performance of bitslice floating-point multiplication with bit sizes from 8 to 28. 163
7-13 Performance of bitslice floating-point division with bit sizes from 8 to 28. . 164
7-14 Performance of BLAS-1 xSCALE in bitslice floating-point operations with bit sizes from 8 to 16. 166
7-15 Performance of BLAS-1 xAXPY in bitslice floating-point operations with bit sizes from 8 to 16. 166
7-16 Performance of BLAS-2 xGEMV where $y = 0$ in bitslice floating-point operations with bit sizes from 8 to 16. 167
7-17 Performance of 1D Blur in bitslice floating-point operations with bit sizes from 8 to 16. 167

8-1 The customized output from ABC logic optimizer for the 4-bit integer multiplication with a 3-LUT mapping strategy. 176

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

3.1 Data layout schemes and vectorization strategies. 50
3.2 Data layout transformations assuming the array `u` is originally in the `Pure AoS`. 56

5.1 Test-cases with three representative data access strides: 1, 2 and 3. 103

7.1 Types of floating-point numbers. 149

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Our Thesis

Single instruction, multiple data (SIMD) vector computational units are widely available in commodity processors from energy-efficient embedded systems to large supercomputers. For example, Intel Haswell processors provide the AVX2 SIMD instruction set, which supports up to eight 32-bit integer and single precision floating point SIMD operations. In order to find sufficient data parallelism to feed the SIMD execution units, programmers usually rely on compilers to automatically vectorize hot-spot parallel loops and generate efficient SIMD instructions. Classic loop vectorizaton [Kennedy and Allen, 2002][Nuzman and Zaks, 2008] first determines if a loop in a loop nest is vectorizable according to data dependence analysis, then examines every statement enclosed in the loop to check if it is feasible and profitable to be transformed into a SIMD operation. The payoff of forming SIMD operations is decided by several factors, such as data alignment [Eichenberger et al., 2004], and data access patterns [Nuzman et al., 2006].

General Purpose Graphics Processing Units (GPGPUs), in particular, the Nvidia CUDA GPU, are also widely used in a variety of machines from embedded systems (e.g. Nvidia Tegra K1) to super-computers. The deep hierarchy of both execution model — warp, thread block, grid — and memory organization – local, shared and global memories — makes manually writing high performance code for the CUDA GPU error-prone and tedious. To reduce the programming efforts, GPU manufacturers have put forward low-level programming models such as CUDA and OpenCL. Meanwhile, a long

#define SIZE 128
struct vec_type {
  int a;
  int b;
  int c;
};
struct vec_type vectors[SIZE];
 	
for (i = 0 ; i < 128; i++) {
   vectors[i]. a = …
   vectors[i]. b = …
   vectors[i]. c = …
}
a) An array of structures (SoA)
	#define SIZE 128
	struct vec_type {
	   int a[SIZE];
	   int b[SIZE];
	   int c[SIZE];
	} ;
	struct vec_type vectors;
	
	for (i = 0 ; i < 128; i++) {
	   vectors.a[i] = …
	   vectors.b[i] = …
	   vectors.c[i] = …
	}
	b) A structure of arrays (SoA)

Figure 1-1: Data layouts: Array-of-Structures(AoS) and Structure-of-Arrays(SoA)

standing research goal has been to automatically generate GPGPU code from auto-parallelization [Verdoolaege et al., 2013], compiler directive based languages (e.g. OpenMP, OpenACC) and domain-specific languages (e.g. Halide [Ragan-Kelley et al., 2013] for image processing).

Data layout has a great impact on the effectiveness of both automatic loop vectorization for the SIMD execution units of CPUs and automatic code generation for GPUs. Two popular ways are commonly used to organize data in memory, an array of structures (AoS) and a structure of arrays (SoA). As depicted in Figure 1-1, compared to SoA, data is in AoS is a more intuitive way to represent physical entities. However, data in AoS often exposes interleaved data access patterns with non-unit strides. For the given example, computation on the data in AoS reveals stride-3 interleaved data access. On the other hand, computation on the data in SoA presents contiguous memory access.

For contiguous data access, when vectorizing loops for CPUs, compilers can simply transform a scalar load or store operation to a vector load or store operation to load or store a full vector of data. By contrast, it is not trivial for the compiler to generate efficient instructions to reorganize interleaved data into a vector. Vectorizing compilers based on classic loop vectorization usually generate a sequence of data shuffling instructions (e.g., pshuffle, pblend in Intel SSE) for data reorganization [Nuzman et al., 2006][Ren et al.,

2006]. Nonetheless, as long as data is accessed in a non-unit stride pattern, there will always be a cost of shuffling or gathering data for vectorization.

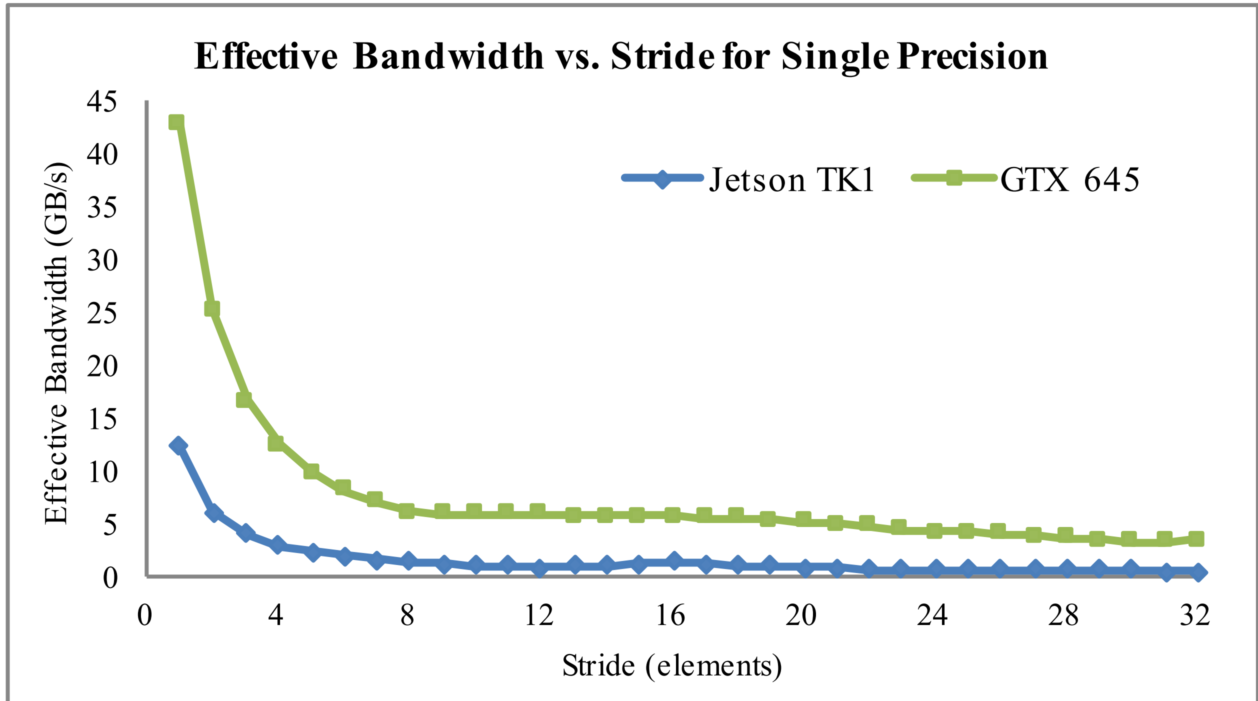


Figure 1-2: The performance impact of strided memory access on effective memory bandwidth on two Nvidia CUDA GPUs.

Similarly, for parallel loops with data in AoS layout, directly mapping each loop iteration to a GPU thread also exposes non-unit stride data access. When the data access pattern has a unit stride, memory access to global memory can be easily coalesced. Conversely, non-unit stride access leads to low memory utilization and can have a great impact on the effective memory bandwidth [Wilt, 2013]. As shown in Figure 1-2, all strides except the unit one greatly decrease the effective memory bandwidth. Therefore, optimizing non-unit stride memory access is of great importance to the memory performance of CUDA GPU programs.

In this thesis, we first present three new solutions to the problems mentioned above — semi-automatic data layout transformation, vectorization based on the computation structure exposed by data access patterns and loop vectorization for nested thread-level parallelism. Then, we demonstrate how fine-grained SoA-to-AoS transformation is useful to customizable precision arithmetic on general purpose processors with SIMD extensions, and introduce our new programming method based on bitslice vectors and related

optimization techniques.

1.2 Contributions

The principal contributions of this thesis are as follows:

- We put forward a new program annotation (using C language pragmas) to enable programmers to specify data layout transformations. The primitive data layout transformations presented are suitable to be composed into more complex data layout transformations. With data layout aware loop transformations, compilers are able to do better vectorization.

- We introduce a new vectorizing technique based on hyper loop parallelism, which is revealed by hyper loops. The hyper loops recover the loop structures of the vectorizable loop and help vectorization to employ global SIMD lane-wise optimization. Experimental results demonstrate that our hype loop parallelism vectorization can achieve significant speedups over the non-vectorized code in our test cases.

- We present a compiler framework to extract hyper loop parallelism in vectorization and map the parallelism efficiently on CUDA GPUs. Our method achieves thread coarsening, which can reduce memory operations in the presence of data locality, and optimizes uncoalesced memory access to global memory. In addition, the introduction of hyper loop parallelism further refines the mapping granularity between coarse-grain loop parallelism and GPU threads.

- We advocate a loop vectorization approach to nested TLP in C-to-CUDA compilation for CUDA GPUs. Our vectorization approach is designed to reuse the GPU threads for outer parallel loop(s) to execute nested TLP.

- We propose a new model of SIMD vector computation based on software bitslicing that allows bit-level customizable precision of numeric types on processors with SIMD extensions We use ABC, a logic optimization and synthesis tool that is intended to optimize hardware circuits, to optimize our arithmetic operators and significantly improve their performance. Our results show that circuit optimization techniques are applicable to both hardware and software.

1.3 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 provides some background on the compilation techniques on automatic vectorization in modern compilers and the programming model for compute unified device architecture (CUDA) GPU by Nvidia.

Chapter 3 introduces a new program annotation (using C language pragmas) to enable programmers to specify data layout transformations for semi-automatic data-layout transformations.

Chapter 4 presents hyper-loop parallelism, which is used to exploit the computation structure exposed by data-layout in vectorization.

Chapter 5 improves the memory performance of CUDA programs with hyper-loop parallelism on CUDA GPUs

Chapter 6 proposes a novel loop vectorization approach to nested thread-level parallelism in C-to-CUDA compilation for CUDA GPUs.

Chapter 7 illustrates a new approach to vector computing based on bitslice vector formats for processors with SIMD extensions and building arithmetic operators from bitwise instructions.

Chapter 8 highlights some of the most notable contributions, and identifies some interesting aspects arising from the work that warrant further research.

1.4 Published Work

Chapter 3, 4, 5 and 6 are based on the published work as follows:

[1] **Shixiong Xu** and David Gregg. An Efficient Vectorization Approach to Nested Thread-level Parallelism for CUDA GPUs. In 2015 International Conference on Parallel Architectures and Compilation Techniques, PACT 2015, San Francisco, CA, USA, October 18-21, 2015, 2015. (DOI: 10.1109/PACT.2015.56) (Extended Abstract)

[2] **Shixiong Xu** and David Gregg. Bitslice Vectors: A Software Approach to Customizable Data Precision on Processors with SIMD Extensions. In 46th International Conference on Parallel Processing, ICPP 2017, Bristol, UK, August 14-17, 2017.

[3] **Shixiong Xu** and David Gregg. Exploiting hyper-loop parallelism in vectorization to improve memory performance on CUDA GPGPU. In 2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 3, pages 5360, 2015. (DOI: 10.1109/Trustcom.2015.612)

[4] **Shixiong Xu** and David Gregg. Efficient exploitation of hyper loop parallelism in vectorization. In Languages and Compilers for Parallel Computing - 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers, pages 382396, 2014. (DOI: 10.1007/978-3-319-17473-0_25)

[5] **Shixiong Xu** and David Gregg. Network and Parallel Computing: 11th IFIP International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings, chapter Semi-automatic Composition of Data Layout Transformations for Loop Vectorization, pages 485496. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. (DOI: 10.1007/978-3-662-44917-2_40)

Chapter 2

Background and Literature Review

In this chapter, we present an account of the relevant background material necessary for this dissertation. We first describe three variants of single instruction, multiple data (SIMD) architectures. Then, we summarize compilation techniques on automatic vectorization in modern compilers. We briefly explain Compute Unified Device Architecture (CUDA) GPU by Nvidia and a compiler directive based programming model — OpenACC. For readers familiar with vectorization techniques and GPGPU programming, this present chapter serves to introduce some terminology used in this dissertation.

2.1 SIMD Architectures

2.1.1 Single Instruction Multiple Data (SIMD)

Single instruction, multiple data (SIMD), is a class of parallel computer architecture in Flynn's taxonomy, proposed by Michael J. Flynn [Flynn, 1972][Duncan, 1990]. It describes computer architectures with multiple processing elements that perform the same operation on multiple data points simultaneously. Therefore, such machines are mainly used to exploit data level parallelism. In this model, only a single instruction is used for multiple parallel computations at a given moment, as depicted in Fig. 2-1.

There are three variants of of SIMD architectures: vector architectures, SIMD instruction set extensions for multimedia and graphics processing units (GPUs) [Hennessy and Patterson, 2011].

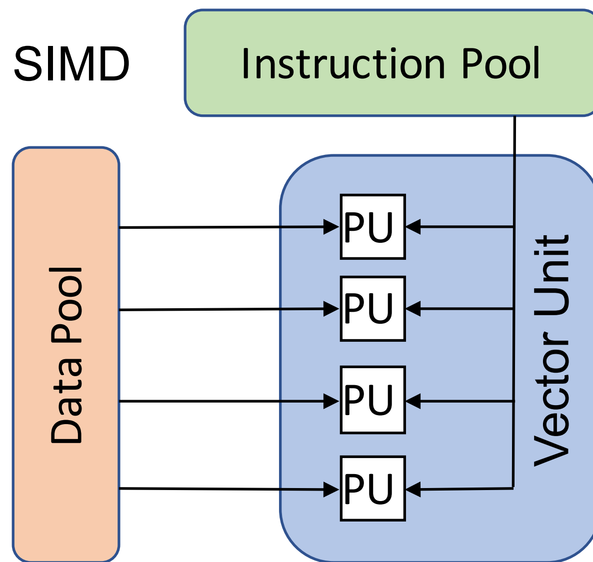


Figure 2-1: Single instruction multiple data (SIMD).

Vector architectures predate the other two SIMD architectures by more than 30 years. Vector processors implements an instruction set that operates on 1-D arrays, called vectors. Vectors contain multiple data elements and the number of data elements per vector is typically referred to as the vector length. Both instructions and data are pipelined to reduce decoding time. Cray platforms were the most notable vector supercomputers, for example, the Cray-1, Cray-2, Cray X-MP and Cray Y-MP.

In the following sections, we present the other two SIMD architectures as well as the difference and similarities between them.

2.1.2 SIMD Instruction Set Extensions for Multimedia

SIMD instruction set extensions for multimedia borrows the SIMD name to express simultaneous parallel data operations. This SIMD architecture was motivated by the narrow data types such as 8 and 16 bits integers used in many media applications. 32-bit processors were not optimized for such narrow types. However, if the carry chains of a 32-bit adder were partitioned, a processor could perform parallel operations on short vectors of four 8-bit operands, or two 16-bit operands. The additional cost of such partitioned adders was small. A SIMD instruction performs the same operation on vectors of data similar to vector instructions used in vector architectures. But compared to the large register of traditional vector architectures, SIMD instructions tend to have fewer

operands and thus operate on narrower registers.

Many widely used instruction sets today provide SIMD instructions on short vectors as an extension of the scalar instruction set. For instance, NEON technology is a 128-bit SIMD architecture extension for the ARM processors. Separate functional units are typically used to implement these short vector extensions with their own pipeline and register files. These functional units provide parallel vectors of scalar arithmetic, data movement and other functions.

SIMD extensions to the instruction set of general-purpose processors (GPPs) were designed to exploit fine-grained data parallelism exposed by applications. They are driven by the demand for accelerating rich multimedia applications. Such applications typically incorporate computationally intensive operations like audio and video decoding [Lee, 1995] and image processing [Cypher and Sanz, 1989].

The width of SIMD vectors in general-purpose processors is getting wider and wider. For example, Intel Xeon Phi co-processors with Intel AVX-512 SIMD instructions have 512-bit SIMD vectors, and Vision P6 digital signal processors from Cadence have 64-way 8-bit SIMD vectors for applications in computer vision.

In addition to the increasingly wider SIMD width, general-purpose processors tend to be equipped with a richer set of SIMD instructions to meet the demand of emerging applications. For example, Intel AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions for x86 architecture with several kinds of new SIMD instructions. Among these new instructions, bitwise ternary logic instructions can logically implement all possible bitwise operations between two inputs. These instructions in turn provide better support for integer bit manipulation operations that are the major operations in cryptography algorithms (e.g., Data Enryption Standard (DES), Advanced Encryption Standard (AES) [Biham, 1997]).

Apart from the SIMD instructions for arithmetic operations, SIMD memory and data permutation instructions play a critical role in accelerating applications with SIMD computational units. Traditional vector processor such as the Cray-1 provided rich data access instructions such as gather and scatter for strided memory operations. In contrast, general-purpose processors with SIMD extensions typically have poor support for non-contiguous memory access. For such memory access, we need data permutation

instructions to pack the non-contiguous data access into a vector register. It is not trivial to derive an optimal sequence of data permutation instructions to transform non-contiguous memory access into contiguous memory access [Nuzman et al., 2006][Ren et al., 2006][Anderson et al., 2015].

2.2 Graphics Processing Units

Graphics Processing Units (GPUs), in particular, Nvidia's GPUs, have a unique execution model — single instruction, multiple threads (SIMT). SIMT was coined by the Nvidia and first implemented in the Nvidia G80 GPU chip in 2006. It describes the execution model of the Nvidia's Compute Unified Device Architecture (CUDA).

CUDA GPU is a many-core architecture and consists of a number of key blocks — memory, streaming multiprocessors (SMs), streaming processors (SPs). Both SMs and SPs form a two-level hierarchy of execution model. A GPU contains several SMs. An SM is also refereed to as a next generation SM (SMX) in Nvidia's Kepler architecture. Each SMX consists of multiple SPs. An SMX can support thousands of threads running concurrently. Groups of threads with consecutive thread indexes are bundled into warps; one full warp is executed on the CUDA cores in SMs. The size of a warp depends on the hardware. For example, on the K20 CUDA GPUs, thread blocks are divided into warps of 32 threads for execution.

Because threads (and not data) are mapped to the processor and executed in the SIMD-like fashion, the style of execution is called single instruction multiple thread (SIMT).

SIMT is very similar to SIMD extensions. In SIMD extensions, multiple data can be processed by a single instruction. In SIMT, multiple threads are processed by a single instruction in lock-step. Each thread executes the same instruction, but possibly on different data. In the next section, we discuss the similarities and differences between SIMT in CUDA GPUs and SIMD extensions in GPPs.

2.3 SIMD Extensions vs SIMT in CUDA GPUs

In this thesis, the similarity between SIMD extensions and SIMT in CUDA GPUs is the foundation of the vectorization techniques for CUDA GPUs. Therefore, in this section, we give a brief comparison between SIMD extensions and SIMT.

Both SIMD extensions and SIMT broadcast the same instruction to multiple execution units. In other words, they both share the same instruction fetch/decode hardware with replicated execution units.

There are three key features that distinguish SIMT from SIMD extensions.

2.3.1 Single instruction, multiple register sets

To utilize SIMD execution units in GPPs with SIMD extensions, we need explicitly to write SIMD operations on data in SIMD data types. For example, with the GCC vector extensions, adding two vectors of numbers can be implemented as in Fig. 2-2. The compiler generates SIMD memory operations to load data into vector registers, and translates the operations on SIMD data types to SIMD instructions taking vector registers as input operands. For other computations, such as address computation (e.g., &A[i]), loop index increment, we can simply keep a single scalar copy of them.

```
1  #define N 1024
2  int a[N];
3  int B[N];
4  int C[N];
5
6  typedef int v4si __attribute__ ((vector_size (16)));
7
8  for (int i = 0; i < N; i+=4) {
9    v4si vec_A = *(v4si *)&A[i];
10   v4si vec_B = *(v4si *)&B[i];
11   v4si vec_res = A[i] + B[i ];
12   *(v4si *)&C[i] = vec_res;
13 }
```

Figure 2-2: Add two vectors of numbers in SIMD.

On the other hand, in SIMT of CUDA GPUs, we need keep a copy for every computation in each thread for the code written in CUDA as shown in Fig. 2-3. This requires each thread has its own registers to hold data across threads. The data in registers sometimes

```
__global__ void vec_add(int *C, int *A, int *B) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  C[i] = B[i] + A[i];
}
```

Figure 2-3: Add two vectors of numbers in SIMT.

can be redundant if the data used in different threads is the same. For example, the intermediate result of `blockIdx.x * blockDim.x` is the same over the threads in a thread block. Consequently, keeping redundant data in registers unnecessarily wastes registers.

2.3.2 Single instruction, multiple addresses

Random data access from applications like sparse matrix computation and loop-up tables can be quite problematic to general-purpose processors with SIMD extensions. As discussed in Sec. 2.1.1, such processors typically have poor support for non-contiguous memory access. For example, in the sparse matrix vector multiplication shown in Fig. 2-4, the indirect data access to array `x` presents a random access pattern. In this case, if the access pattern is known statically, random data access can be implemented with vector load and data permutation instructions. Otherwise, we need hardware support for gather/scatter instructions. Although some processors with SIMD extensions indeed provide gather/scatter instructions, the performance of these instructions is still not good enough and thus the use of such instructions can only be beneficial in certain scenarios [Intel, 2016]. In addition, in order to use gather/scatter instructions, we need to explicitly put the addresses required in a vector register.

In contrast, thanks to the registers in each thread being able to keep a separate address variable for memory loads and stores, writing applications like look-up table in SIMT is much easier than SIMD extensions. However, similar to SIMD extensions, random data access from SIMT can also be slow due to the characteristics of memory systems of CUDA GPUs that we describe in Section 2.5.1. For global memory, if the memory access from a warp cannot be coalesced, several memory transactions might be required depending on the randomness. This may result in great performance loss from the reduced effective DRAM memory bandwidth. Shared memory in CUDA GPUs is organized in banks, and the hardware can only service one access at a time. As a result, random data access may

```
__global__ void csr_matvec_s(ptr, indices, data, x, y) {
  int row = blockDim.x * blockIdx.x + threadIdx.x ;
  if (row < num_rows) {
    float dot = 0;
    int row_start = ptr[row];
    int row_end = ptr[row + 1];
    for (int jj = row_start; jj < row_end; jj++) {
      dot += data[jj ] * x[indices[ jj ]];
    }
    y[row] += dot;
  }
}
```

Figure 2-4: Sparse matrix vector multiplication in the format of compressed sparse row (CSR) in SIMT.

lead to bank conflicts and thus memory transactions can become serialized.

Despite the performance penalties from non-coalesced memory access and bank conflicts, for large problem sizes we can still get decent speedups because of the latency hiding strategy in CUDA GPUs. All the CUDA cores are oversubscribed with computation tasks due to the massive number of threads created. The hardware is able to quickly switch between tasks when it would otherwise have to wait on memory. This latency hiding strategy is similar to multi-threading in CPUs [Hennessy and Patterson, 2011].

2.3.3 Single instruction, multiple control flow paths

For computations with control flows, in order to use SIMD instructions, we have to either convert the control flow into data flow in the form of conditional execution, or annotate the computation with predicates for the condition internally in the process of vectorization. Some SIMD extensions have support for conditional execution through vector select operation `c = vselect a, b, cond`, where the element of `cond` is the condition for selecting element from vector `a` or `b`. However, such vector select operations may introduce redundant stores. For instance, array `data` in Fig. 2-5 should only be updated when the condition is true. When the if statement is converted into a vector select statement, array `data` will be always updated regardless the condition.

It is simpler and cleaner to write code with control flow in SIMT. As shown in Fig. 2-5, explicit conversion from control flow to data flow is not required for SIMT. The control flow is dealt with by the underlying hardware. SIMT executes the same instruction in

```
1  __global__ void control_flow_example(data, x, y) {
2    int row = blockDim.x * blockIdx.x + threadIdx.x ;
3    if (row % 2 == 0) {
4      data[row] = x[row] + y[row];
5    }
6  }
```

Figure 2-5: Computation with control flow in SIMT.

lock-step with a group of threads. When it comes to the control flow, only one flow path can be executed at a time, and threads that are running must wait, as depicted in Fig. 2-6. By doing this, control flow divergence is handled correctly but slowly. Deeply nested control flows can occur significant costs as they effectively serialize the thread execution.

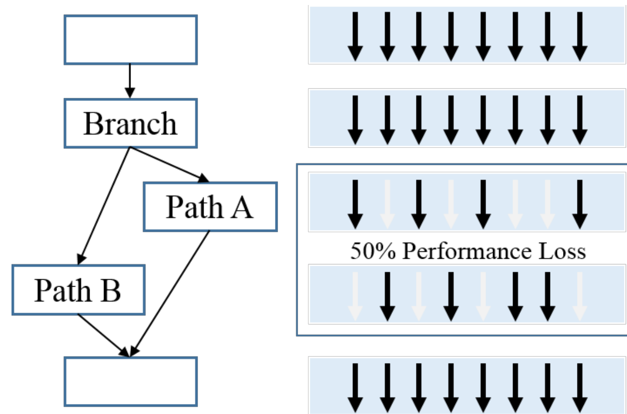


Figure 2-6: Thread divergence in SIMT.

Thread divergence also brings another performance issue in terms of memory access. When the thread execution is serialized, the memory access can present random access patterns. For example, memory accesses to array `data`, `x` and `y` in Fig. 2-5 show a random access pattern and the randomness is determined by the condition `row % 2 == 0`. As discussed above, random access to both global memory and shared memory will lead to significant performance loss.

In summary, both SIMT and SIMD extensions are good at exploiting data parallelism in applications. SIMT in CUDA GPUs is more flexible than SIMD extensions in general-purpose processors in the respect of programmability and performance.

2.4 Compilation Techniques for Automatic Vectorization

In order to take advantage of SIMD extensions, programmers not only have to find sufficient data parallelism to feed the SIMD computational units, but also need express the computation in SIMD instructions taking into account of the memory patterns. Automatic vectorization in compilers alleviates such difficulties to certain extent by automatically extracting data parallelism and translating scalar computations into SIMD instructions.

Automatic vectorization has been part of high performance compilers since the compilers for vector supercomputers [Zima and Chapman, 1991][Kennedy and Allen, 2002]. Ever since the commodity general-purpose processors started to include SIMD extensions, for example, the Hewlett-Packard PA-RISC processor family [Lee, 1995] and Sun Microsystems UltraSPARC [Kohn et al., 1995], SIMD has been the mainstream approach to exploit data parallelism. As the instructions set of SIMD gets richer, automatic vectorization becomes more and more important in optimizing compilers.

There are several ways of automatic vectorization in modern optimizing compilers. These methods attempt to exploit data parallelism in different scopes of programs. Classic loop vectorization performs vectorization on the loop level [Kennedy and Allen, 2002] [Nuzman and Zaks, 2008] [Nuzman et al., 2011] [Kim and Han, 2012]. Superword level parallelism vectorization vectorizes straight line code in basic blocks [Larsen and Amarasinghe, 2000] [Liu et al., 2012]. Whole function vectorization [Karrenberg and Hack, 2011] transforms a function into a SIMD form.

In this section, we summarize the mainstream automatic vectorization techniques adopted in both commercial and open-source compilers, as well as some experimental approaches to automatic vectorization.

2.4.1 Loop Vectorization

Loop vectorization is an effective way of utilizing SIMD execution units on contemporary CPUs with SIMD extensions. The simplest form of loop vectorization deals with the innermost loops and is most common among optimizing compilers (e.g., GCC and LLVM). However, in some cases, vectorization on the innermost loop is possible but not

profitable due to small trip counts or other characteristics (e.g., contiguous memory access). Therefore, it might be profitable to vectorize the enclosing outer-loop [Nuzman and Zaks, 2008]. Outer-loop vectorization refers to vectorizing a level of a loop nest other than the innermost.

```
1 for (i=0; i< SIZE; i++) {
2   float sum = 0.0f;
3   for (j = 0; j < SIZE; j++) {
4     sum += M[j][i] * V[j];
5   }
6   C[i] = sum;
7 }
```

a) non-vectorized version

```
1 for (i=0; i< SIZE; i++) {
2   float vsum = {0.0f, 0.0f, 0.0f, 0.0f};
3   for (vj = 0; vj < SIZE; vj+=4) {
4     vsum += M[vj:vj+3][i] * V[vj:vj+3];
5   }
6   C[i] = reduction_sum(vsum);
7 }
```

b) inner-loop vectorization

```
1 for (vi=0; vi< SIZE; vi+=4) {
2   float vsum = {0.0f, 0.0f, 0.0f, 0.0f};
3   for (j = 0; j < SIZE; j++) {
4     vsum += M[j][vi:vi+3] * V[j];
5   }
6   C[vi:vi+3] = vsum[0:3];
7 }
```

c) outer-loop vectorization

Figure 2-7: Vectorization on the loop in transposed matrix vector multiplication with vectorization factor 4.

Inner-loop Vectorization Vectorizing an innermost loop can be described as a process of

1. unrolling the loop by the vectorization factor VF;

2. scheduling the instructions of the unrolled loop in order to make VF instances

of each instruction become adjacent. The adjacent instructions can be executed concurrently by a SIMD execution unit, and

3. using a corresponding vector instruction to replace these VF instances of each instruction.

Data dependence analysis is used to check the validity of this transformation by proving that VF consecutive iterations of the original loop can be jammed together (i.e., step (2)). When there are short cross-iteration dependencies, as in the case of reduction operations (in Fig. 2-7b), this transformation is not valid anymore and thus special treatment for the reduction operations is required. Vectorizing an innermost loop computing a reduction can be done by computing VF partial reductions in parallel, assuming the operation is associative, and combining them together at the end. For example, Fig. 2-7(a) is the non-vectorized loop for transposed matrix vector multiplication. The vectorized results with VF = 4 from inner-loop vectorization is illustrated in Fig. 2-7(b).

Outer-loop Vectorization Outer-loop vectorization refers to vectorizing a level of a loop nest other than the innermost, and can be described as a process of

1. unrolling the chosen outer-loop by the vectorization factor VF;
2. jamming the contents of the unrolled loop together (including inner loops), so that VF instances of each instruction in the outer and inner loops become adjacent, and
3. using a vector instruction to replace these VF instances of each instruction.

Steps 1 and 2 are similar to the classical compiler optimization loop unroll-and-jam [Kennedy and Allen, 2002]. Data dependence analysis is also critical to check the validity of this transformation by proving that VF consecutive iterations of the original loop can be jammed together. Nuzman and Zaks summarized five ways to achieve outer-loop vectorization such as vectorize innermost and enclosing outer loops, interchange outer loop to innermost position for innermost vectorization, direct outer-loop vectorization in-place [Nuzman and Zaks, 2008]. With the third method mentioned above, the vectorized loop after outer-loop vectorization on the *i*-loop in Fig. 2-7(a) is shown in Fig. 2-7(c).

2.4.2 Super-word Level Parallelism Vectorization

As the finest operating units for loop vectorization are loop iterations, in the cases of unrolled loops or computations on tuples of data (e.g., 3D vectors), loop vectorization techniques in Sec. 2.4.1 may not be able to find sufficient data parallelism for SIMD. To supplement this inefficiency, super-word parallelism vectorization (SLP) is put forward to exploit data parallelism in straight line code in basic blocks [Larsen and Amarasinghe, 2000].

SLP vectorization identifies groups of isomorphic instructions exposing super-word level parallelism, and combines them into equivalent vector instructions [Larsen and Amarasinghe, 2000]. It is a greedy algorithm and packs instructions into SIMD instructions based heuristics, such as contiguous data access.

SLP vectorization has been incorporated into several vectorizing compilers since its introduction. Liu et al. improved SLP vectorization by first trying to find all the possible groups of instructions suitable for packing and then making decisions on packing according to the data reuse between packed groups [Liu et al., 2012]. The improved vectorization not only helps discover more super-word parallelism for vectorization, but also enables better data layout transformation for efficient vectorization. However, both SLP vectorization algorithms work similar to instruction scheduling, which is dealing with instructions separately while ignoring the structure of computation done by the instructions.

As the data parallelism in a single basic block is often quite limited, loop unrolling is sometimes a prerequisite to form a larger basic block for SLP vectorization. The implementation of SLP vectorization in GCC [Ira Rosen and Zaks, 2007] makes loop unrolling and SLP vectorization work seamlessly by introducing loop-aware SLP.

A flexible version of SLP is proposed by Porpodas to address the problem of partial isomorphic operations [Porpodas et al., 2015]. They attempted to introduce redundant operations to make partial isomorphic subgraphs of the data dependence graph isomorphic. Our work in Chapter 4 also tries to solve a similar problem but with different approach. Park et al. introduced a vectorization technique based on sub-graph level parallelism (SGLP), a coarser level of vectorization within basic blocks [Park et al., 2012].

An integrated SIMDization framework is put forward by Wu et al. to address several

orthogonal aspects of SIMDization, including SIMD parallelism extraction from different program scopes (from basic blocks to inner loops), vectorization on loops with mixed data lengths and alignment handling [Wu et al., 2005].

2.4.3 Whole-function Vectorization

Compared to loops in loop vectorization and basic blocks in SLP vectorization, whole-function vectorization works on a larger scope of programs — functions [Karrenberg and Hack, 2011]. This vectorization techniques transforms a scalar function in such a way that it computes W executions of the original code in parallel using SIMD instructions, where W is the size of SIMD vectors. It can vectorize arbitrary control flow structures in a control-flow graph in static single assignment (SSA) form even on architectures without explicit predicated execution.

Intel's C/C++ compiler supports a general language construct — SIMD-enabled functions (formerly called elemental functions) — to express a data parallel algorithm [Geva, 2011]. A SIMD-enabled function is written as a regular C/C++ function, and the algorithm within describes the operation on one element, using scalar syntax. The function can then be called as a regular C/C++ function to operate on a single element or it can be called in a data parallel context to operate on many elements. When programmers write a SIMD-enabled function, Intel's C/C++ compiler generates a short vector form of the function, which can perform the given function's operation on multiple arguments in a single invocation.

```
1 __declspec (vector) double ef_add_doubles(double x, double a)
2 {
3  return x + a;
4 }
5 //invoke the function
6 for (i = 0; i < n; ++i) {
7  y[i] = ef_add_doubles(x[i],42);
8 }
```

Figure 2-8: Example of SIMD-enabled functions in Intel's C/C++ compiler.

The short vector version is implemented with the SIMD vector instruction set architecture in the CPU and can be used by loop vectorization to vectorize funtion calls. As shown in Fig. 2-8, the function `ef_add_doubles` is a SIMD-enabled function. When

vectorizing the function call to `ef_add_doubles` in the `i`-loop, the vector version of `ef_-add_doubles` will be used to replace the scalar version.

2.4.4 Other Automatic Vectorization Techniques

In addition to the three widely adopted techniques discuss above, there are also some interesting experimental approaches to automatic vectorization in compilers.

Loop vectorization in polyhedral model

Loop vectorization starts with a data dependence analysis, which detects whether operators in a loop to be vectorized can run in parallel or not [Padua and Wolfe, 1986]. Many classic scalar optimization and loop transformation techniques are often applied before loop vectorization to make the loops easier to vectorize. [Muchnick, 1997][Kennedy and Allen, 2002]. For example, *if conversion* is a widely used approach to converting control flow in loops into data flow, making loop vectorized less difficult vectorize loops with control flows.

Compilers sometimes use abstract syntax trees (ASTs) as intermediate representation for programs. ASTs are not appropriate for complex program restructuring [Shirako et al., 2014]. Complex transformations such as loop inversion, skewing, tiling and so on. modify the execution order and this is far away from the syntax. The polyhedral (or polytope) model based on a linear-algebraic representation of programs and transformations emerged in the eighties to address this issue [Bastoul, 2004][Shirako et al., 2014]. Some compilers such as LLVM and GCC implement optimizers based on the polyhedral model such as Polly [Grosser et al., 2012] and Graphite [Pop et al., 2006].

The polyhedral framework works in three steps as follows:

1. takes programs in a compiler intermediate representation (IR) such as GIMPLE in GCC and LLVM IR in LLVM, and translates the program parts that fit the model into the linear-algebraic representation;

2. selects a new execution order by using a reordering function (a schedule, or a placement, or a chunking;

3. the code generation creates an optimized implementation of the routine in a compiler intermediate representation or as source code implementing the execution order implied by the reordering function.

Trifunovic et al. examined the interactions between loop transformations of the polyhedral framework and subsequent vectorization of programs in GCC's IR GIMPLE [Trifunovic et al., 2009]. They modeled the performance impact of the different loop transformations and vectorization strategies, and then showed how this cost model can be integrated seamlessly into the polyhedral representation. However, in this work, loop vectorization is done on programs in GCC's IR GIMPLE translated back from the the linear-algebraic representation in the polyhedral model.

Kong et al. stepped further to make the analysis and transformations in polyhedral model and SIMD code generation work more seamlessly by putting forward vectorizable codelet [Kong et al., 2013]. The vectorizable codelet is a tile of code with specific, SIMD-friendly properties. SIMD code generation can take advantage of information expressed by vectorizable codelets for better ISA-specific vector instruction selection, scheduling, and register promotion.

Vectorization based on instruction selection and scheduling

When loops contain a mix of vectorizable and nonvectorizable operations, the classic loop vectorization approach generates separate loops for the vector and scalar operations. Scalar operations result in low utilization of scalars resources in the vectorized loops, and vector resources in scalar loops.

To address this issue, Larsen et al. put forward a novel approach of exploiting SIMD parallelism in software pipelined loops [Larsen et al., 2005]. Software pipelining is an effective instruction scheduling technique on loops to overlap instructions from different loop iterations and maximize resource utilization. Their work enables software pipeling to fully exploit the potential of a multimedia architecture by explicitly selecting instructions for vectorizing operations. However, this work did consider mutating instructions to possibly expose more SIMD parallelism for vector instructions [Novack and Nicolau, 1995].

Barik et al. presented an instruction selection based auto-vectorization framework in

the back-end of a dynamic compiler [Barik et al., 2010]. Their work not only generates optimized vector code but is also well integrated with the instruction scheduler and register allocator. They adopted a compile-time efficient dynamic programming-based vector instruction selection algorithm for straight-line code. This algorithms expands opportunities for vectorization in several ways such as exploring more opportunities of packing multiple scalar variables into short vectors and judicious use of shuffle and horizontal vector operations.

2.5 CUDA and OpenACC

In this section, we give a brief review of the CUDA programming model and a representative compiler directives based approach OpenACC. For more advanced details,

2.5.1 CUDA Programming Model

The CUDA programming model is an explicit parallel programming model in a manner of single program multiple data (SPMD). All the threads are organized in a two-level hierarchy — thread grids and thread blocks (TBs) — according to the physical execution model shown in Fig. 2-9.

The threads in a thread block can be logically organized in up to three dimensions, and indexed by the identifiers `threadIdx.x, threadIdx.y, and threadIdx.z`. The threads aligned along the x dimension are adjacent to each other. Similarly, the thread blocks can also be logically arranged in up to three dimensions and indexed by the identifiers `blockIdx.x, blockIdx.y, and blockIdx.z`. When launching a GPU program, also called a kernel, the shape of both the threads in a thread block and the grid of thread blocks is configured and fixed during the whole execution.

CUDA GPUs provides five types of memories – shared, global, local, constant and texture memory, as shown in Fig. 2-10. For each different memory type there are tradeoffs that must be considered when designing the algorithm for your CUDA kernel. For example, global memory has a very large address space, but the latency to access is very high. Shared memory has a very low access latency but the memory address is small compared to global memory.

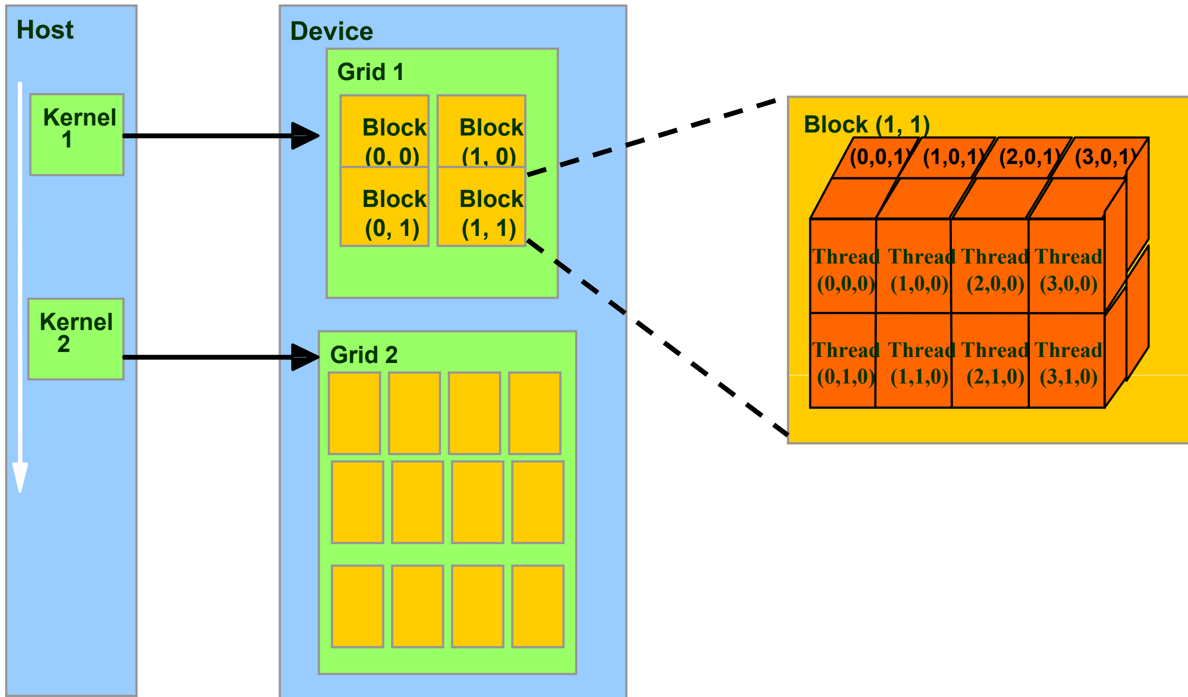


Figure 2-9: CUDA execution model.

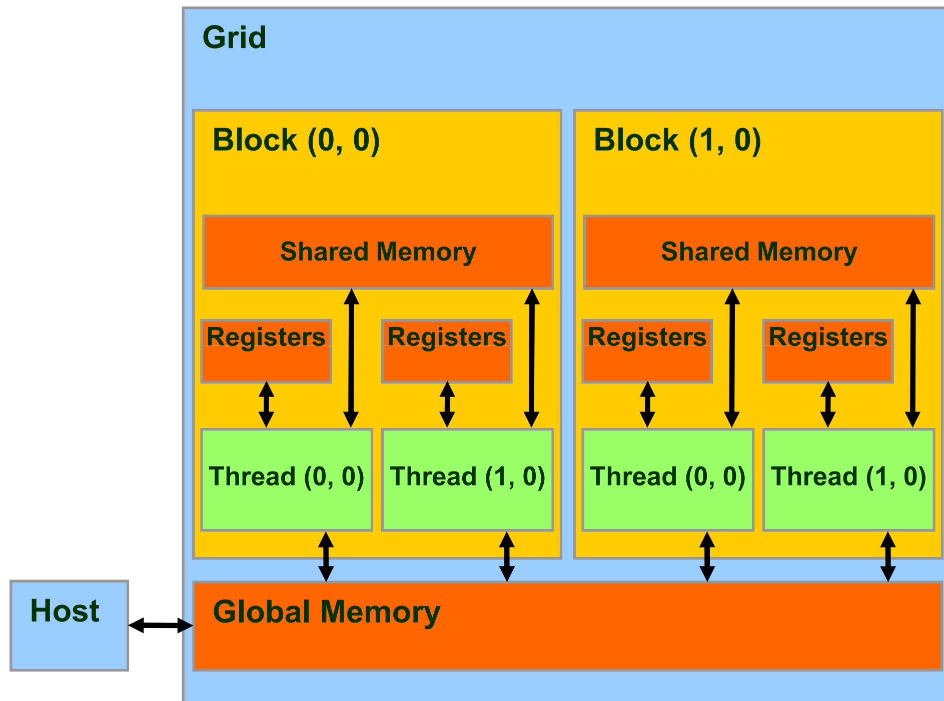


Figure 2-10: CUDA memory model.

Dynamic parallelism is introduced to CUDA GPUs with compute capability 3.5 or higher. With this support, a GPU thread is able to launch another kernel during execution. This kernel exhibits an extra level of parallelism — thread-level parallelism (TLP). Dynamic parallelism is an easy way to deal with nested TLP. But it requires the kernel launched by a GPU thread to have a very high number of threads so that the overhead of launching a kernel can be amortized. In addition, the parent thread communicates with its child threads only through global memory.

2.5.2 Compiler Directive Based Programming Model — OpenACC

While the CUDA programming model makes GPU computing feasible, it still asks programmers to explicitly manage parallelism and usage of the GPU's memory. This kind of explicit programing model for GPU is often considered to be a low-level approach. In order to take advantage of GPUs, specially trained programmers are often required, and significant modifications of source code are inevitable. To alleviate the programming burden, tools based on automatic parallelization and languages based on compiler directives are put forward to reduce programming efforts on accelerators while delivering high performance.

OpenACC is a directive based extension for C, C++, and FORTRAN [OpenACC, 2011] . It adopts a work-sharing model where loops are distributed over multiple execution units. With OpenACC, programmers can use either `parallel loop` or `kernel` directives to mark computationally intensive sections of the code, a compute region, to be executed on an accelerator. The `parallel loop` directive is used for explicit parallelism so that the compiler can simply follow the developer's instructions. On the other hand, the `kernel` directive is used for implicit parallelism; the compiler uses automatic parallelization techniques. Fig. 6-2 gives an example of OpenACC directives. The `parallel loop` directive is used to mark the outermost *i*-loop as a parallel loop for executing on GPUs.

Inside a compute region, loops can be distributed over three different levels of parallelism. These levels are gang, worker, and vector. Gang is used for coarse-grain parallelism. Iterations of a loop executing on different gangs do not share memory or synchronization primitives. Each gang is composed of workers. Workers are used for fine-grain parallelism. They can share local memory. Workers in the same gang can synchronize

```
#pragma acc parallel loop private(i, j)
for (i=0; i<SIZE; i++){
  float sum = 0.0f;
  for (j = 0; j < SIZE; j++){
    sum += M[j][i] * V[j];
  }
  C[i] = sum;
}
```

Figure 2-11: The parallel loop annotated with OpenACC pragmas for the transposed matrix vector multiplication (TMV).

with each other. Vector parallelism is for vector operation within one worker. The mapping of gang, worker and vector to the CUDA GPU execution hierarchy is discussed in [Tian et al., 2013] in different scenarios.

2.6 Summary

In this chapter, we give an account of the relevant background material necessary for this thesis. In addition to the background and related work presented in this chapter, for each of our proposed data layout oriented compilation techniques for efficient vectorization in the following chapters, we include more related work and detailed comparison between our new approaches and existing ones.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Semi-automatic Data Layout Transformations for Loop Vectorization

3.1 Introduction

Single instruction multiple data (SIMD) vector computational units are widely available in processors from large supercomputers to energy-efficient embedded systems. Programmers often depend on compilers to auto-vectorize key loops. However, some program features can hinder the compilers from fully unleashing the power of SIMD. One important feature is interleaved data access coming from the data organized in the manner of an array of structures (AoS). In order to efficiently deal with interleaved data access, vectorizing compilers generate a sequence of data shuffling instructions (e.g. *pshuffle*, *pblend* in Intel SSE) for data reorganization. As long as data is accessed in a non-linear pattern, there will always be a cost of shuffling or gathering data for vectorization.

We observe that for many scientific computing applications with data in AoS, different loops in the program often repeat the same patterns of data permutation during loop vectorization. These patterns usually first do data permutations on a small portion of the whole data needed before the computation in each loop iteration, and apply data permutations on the results after the computation is done. One way of getting rid of these repeated data permutation operations is to transform the layout of the data throughout the program. There are two main approaches to transforming array layouts in programs: automatic transformation by the compiler, or manual changes by the programmer.

Compilers face two major challenges when performing automatic data layout transformations for vectorization. First, the compiler needs a very sophisticated **whole-program** data dependency and pointer aliasing analysis to make sure that the transformation is safe. Secondly, it is difficult for the compiler to choose the best layout. It is perhaps easier for the programmer to determine whether modifying the data layout is safe. But it is tedious and error-prone for programmers to change their code by hand. They may have to change the type declarations and any code that operates on the array. This may involve modifications to many parts of the program, and may result in changes to array indexing, and even the introduction of new statements and loops.

To allow compositions of data layout transformations and evaluate the performance impact of data layout transformations on vectorization, in this chapter we put forward a new program annotation (using C language pragmas) to enable programmers to specify a sequence of data layout transformations. This data layout transformation pragma is implemented in the Cetus source-to-source compiler framework [Bae et al., 2013]. Our prototype implementation currently supports static arrays but can be easily extended to support dynamically allocated arrays using Sung et al.'s approach [Sung et al., 2010]. Our compiler changes data type declarations for all modified arrays, rewrites all functions that operate on modified arrays to change array indexing, and introduces additional loops and other code. Similar to other pragma annotation systems, such as OpenMP, we assume that where the programmer requests a transformation, that transformation is safe.

3.2 Language Support for Data Layout Transformations

3.2.1 Motivating Examples

In this section, we take the kernel of `tezar()` in the SP (**S**calar **P**enta-diagonal), one of the benchmarks in the NAS Parallel Benchmarks (NPB) to demonstrate the advantage of data layout transformations for efficient loop vectorization. This kernel conducts block-diagonal matrix-vector multiplication on the data. For simplicity, we don't consider other cache optimizations, such as array padding.

There is a loop nest of depth three enclosing the main computations and all these

```
double us       [KMAX][JMAXP][IMAXP];
double vs       [KMAX][JMAXP][IMAXP];
double ws       [KMAX][JMAXP][IMAXP];
double speed    [KMAX][JMAXP][IMAXP];
double qs       [KMAX][JMAXP][IMAXP];
double rhs      [KMAX][JMAXP][IMAXP][5];
double u        [KMAX][JMAXP][IMAXP][5];

for (k = 1; k <= nz2; k++) {
  for (j = 1; j <= ny2; j++) {
    for (i = 1; i <= nx2; i++) {
    xvel = us[k][j][i];
    yvel = vs[k][j][i];
    zvel = ws[k][j][i];
    ac   = speed[k][j][i];
    ac2u = ac*ac;
    r1 = rhs[k][j][i][0];
    r2 = rhs[k][j][i][1];
    r3 = rhs[k][j][i][2];
    r4 = rhs[k][j][i][3];
    r5 = rhs[k][j][i][4];
    uzik1 = u[k][j][i][0];
    btuz  = bt * uzik1;
    t1 = btuz/ac * (r4 + r5);
    t2 = r3 + t1;
    t3 = btuz * (r4 - r5);
    rhs[k][j][i][0] = t2;
    rhs[k][j][i][1] = -uzik1*r2 + xvel*t2;
    rhs[k][j][i][2] =  uzik1*r1 + yvel*t2;
    rhs[k][j][i][3] =  zvel*t2  + t3;
    rhs[k][j][i][4] =  uzik1*(-xvel*r2 + yvel*r1) + qs[k][j][i]*t2 + c2iv*ac2u*t1 + zvel*t3;
    }
  }
}
```

Figure 3-1: The kernel of function tzetar() in the SP of NPB.

loops are parallel, shown in Fig. 3-1. When vectorizing the innermost parallel loop i, compilers directly generate vector loads and stores for the data references to array `us`, `vs`, `ws`. On the contrary, the inter-leaved data access exposed by the references to array u and *rhs* may require compilers to apply suitable data reorganization. Compilers can treat these inter-leaved loads as gather operations. But the support for these gather operations in modern commodity processors is still not good enough [Ramachandran et al., 2013]. Instead, the compiler may utilize available data permutation instructions to transform the inter-leaved data access into consecutive data access. On the other hand, the cost of data permutation instructions introduced by the data reorganization may not be well offset by the performance benefits gained by vectorization on the computations.

Table 3.1: Data layout schemes and vectorization strategies.

Description	Declaration	Vectorization Strategy
Pure AoS	double u [KMAX][JMAXP][IMAXP][5];	Data permutation with stride 5
Split AoS (1:4)	double u1 [KMAX][JMAXP][IMAXP]; double u2 [KMAX][JMAXP][IMAXP][4];	Consecutive data accesses Data permutation with stride 4
Split AoS (4:1)	double u1 [KMAX][JMAXP][IMAXP][4]; double u2 [KMAX][JMAXP][IMAXP];	Data permutation with stride 4 Consecutive data accesses
Split AoS (1:2:2)	double u1 [KMAX][JMAXP][IMAXP]; double u2 [KMAX][JMAXP][IMAXP][2]; double u3 [KMAX][JMAXP][IMAXP][2];	Consecutive data accesses Data permutation with stride 2 Data permutation with stride 2
Split AoS (2:2:1)	double u1 [KMAX][JMAXP][IMAXP][2]; double u2 [KMAX][JMAXP][IMAXP][2]; double u3 [KMAX][JMAXP][IMAXP];	Data permutation with stride 2 Consecutive data accesses Consecutive data accesses
Pure SoA	double u [5][KMAX][JMAXP][IMAXP];	Consecutive data accesses
Hybrid SoA	double u [KMAX][JMAXP][IMAXP/4][5][4];	Consecutive data accesses

Instead of compilers generating data permutation instructions to reorganize data, programmers can change the data layout into a form amenable to vectorization. Table 3.1 gives several possible data layout schemes of array u and their related vectorizing strategies compilers may take. The vectorizing strategies shown in Table 3.1 illustrate that some data layout transformations may simplify the vectorization of interleaved data

access. For instance, compilers deal with the inter-leaved data access with stride 2 in `Split AoS` instead of stride 5 in `Pure AoS`, demonstrated in Section 4. Similarly, since the data references to the array *rhs* are inter-leaved with stride 5, the array *rhs* could also have same data layout transformation schemes as the array *u*.

3.2.2 Data Layout Transformation Pragmas

In this chapter, we put forward a program annotation, *array transform*, a C language pragma to express data layout transformations on static arrays. The syntax of this new pragma is shown in Fig. 3-2.

⟨*pragma*⟩ ::= #pragma array_transform ⟨*array_name*⟩ ⟨*descriptor*⟩ ⟨*actions*⟩

⟨*descriptor*⟩ ::= [⟨*identifier*⟩] ⟨*descriptor_list*⟩

⟨*descriptor_list*⟩ ::= [⟨*identifier*⟩] ⟨*descriptor_list*⟩ | ⟨*empty*⟩

⟨*actions*⟩ ::= -> ⟨*pre_actions*⟩ ⟨*post_actions*⟩

⟨*pre_actions*⟩ ::= ⟨*strip_mine*⟩ | ⟨*interchange*⟩ | ⟨*pad*⟩ | ⟨*pre_actions*⟩ | ⟨*empty*⟩

⟨*post_actions*⟩ ::= ⟨*peel*⟩ | ⟨*empty*⟩ | ⟨*post_actions*⟩

⟨*strip_mine*⟩ ::= STRIP_MINE (⟨*identifier*⟩ , ⟨*stride_size*⟩ , ⟨*identifier*⟩)

⟨*interchange*⟩ ::= INTERCHANGE (⟨*identifier*⟩, ⟨*identifier*⟩)

⟨*pad*⟩ ::= PAD (⟨*identifier*⟩, ⟨*pad_size*⟩)

⟨*peel*⟩ ::= PEEL (⟨*identifier*⟩, ⟨*peel_size*⟩)

Figure 3-2: Syntax of the data layout transformation pragma.

The *array transform* pragma consists of *array descriptor* and *transform actions*. The *array descriptor* gives a name to each array dimension, and these names are used in the *transform actions* to record the related data layout transformations. The *transform actions* present the basic data layout transformations. In this chapter, we define four basic data layout transformations, *strip-mining*, *interchange*, *pad*, and *peel*. These terms for data layout transformations are borrowed from the classic loop transformations [Bacon et al., 1994].

The data storage of an array A can be viewed as a rectangular polyhedron. In [O'Boyle

and Knijnenburg, 1997], formal indices $\vec{\mathcal{I}}$ are introduced to describe the array index space

$$\vec{\mathcal{I}} = [i_1, i_2, \ldots, i_n]^T \tag{3.1}$$

where n is the dimension of the array A. The range of the formal indices $\vec{\mathcal{I}}$ describes the size of the array, or index space, as follows:

$$\vec{\lambda} \leq \vec{\mathcal{I}} < \vec{\mu} \tag{3.2}$$

where the lower bound vector $\vec{\lambda} = [\lambda_1, \ldots, \lambda_n]^T$ and the upper bound vector $\vec{\mu} = [\mu_1, \ldots, \mu_n]^T$ are $n \times 1$ vectors. The array index in C language can only start from 0, therefore, the lower bound vector $\vec{\lambda}$ in this chapter is $\vec{0}$. As each array dimension is given a name by the *array descriptor*, these names can be treated as the formal indices to the arrays.

In contrast to the loop transformations which transform the loop iteration space formed by the loop indices, data layout transformations change the array index space. Since the array index space is changed, the subscripts in references to the array also have to be transformed accordingly.

The subscripts in a reference to an array in loops represent a function that maps the values of the loop iteration space to the array index space and this function is often expressed in the form of a memory access matrix [Jang et al., 2010]. Consider a data reference to an M dimensional array in the loop nest of depth D, where D and M do not need to match. The memory access pattern of the array in the loop is represented as a memory access vector, $\vec{m}$, which is a column vector of size M starting from the index of the first dimension. The memory access vector is then decomposed as an affine form:

$$\vec{m} = \mathbf{M}\vec{i} + \vec{o} \tag{3.3}$$

where $\mathbf{M}$ is a memory access matrix whose size is $M \times D$, $\vec{i}$ is an iteration vector of size D traversing from the outermost to the innermost loop, and $\vec{o}$ is an offset vector that is a column vector of size M and determines the starting access point in an array.

The semantics of the four data layout transformations are defined as follows:

Strip-mining : STRIP_MINE (*id_1*, *stride_size*, *id_2*)

This transformation splits the array dimension i indicated by the id_1 into tiles of size *stride_size* and creates a new formal indices vector $\vec{\mathcal{I}'}$ and two new dimension range vectors $\vec{\lambda'}$ which is $\vec{0}$ and $\vec{\mu'}$. Intuitively, the strip-mining splits the array dimension into two adjacent dimensions with dimension name id_1 and id_2, respectively. The new dimension id_1 takes the position of i and the new dimension id_2 takes the position of $i+1$ in the $\vec{\mathcal{I}'}$. $\vec{\mu'}$ is created by dividing $\vec{\mu}_i$ into $\vec{\mu}_h$ and $\vec{\mu}_l$, where $\vec{\mu}_h = \lceil \vec{\mu}_i / stride_size \rceil$ and $\vec{\mu}_l = stride_size$. For each reference with subscripts $\vec{s}$ to the target array in the corresponding scope, new subscripts $\vec{s'}$ for each reference are created by dividing $\vec{s}_i$ into $\vec{s}_h$ and $\vec{s}_l$, where $\vec{s}_h = \lfloor \vec{s}_i / stride_size \rfloor$ and $\vec{s}_l = \vec{s}_i \bmod stride_size$. Note that, when the original dimension size is not a multiple of block size *stride_size*, padding is introduced automatically at dimension i.

Interchange : INTERCHANGE (*id_1*, *id_2*)

This transformation interchanges the array dimensions i, j indicated by id_1 and id_2 and creates a new formal indices vector $\vec{\mathcal{I}'}$ and two new dimension range vectors $\vec{\lambda'}$ which is $\vec{0}$ and $\vec{\mu'}$. The upper bound vector $\vec{\mu'}$ is created by interchanging $\vec{\mu}_i$ and $\vec{\mu}_j$. For each reference with subscripts $\vec{s}$ to the target array in the corresponding scope, new subscripts $\vec{s'}$ for each reference are created by interchange $\vec{s}_i$ and $\vec{s}_j$.

Pad : PAD (*id*, *pad_size*)

This transformation pads the array dimension i indicated by *id* by the size of $|pad_size|$ either from the beginning if the integer *pad_size* is negative or from the end if the integer *pad_size* is positive. Two new dimension range vectors $\vec{\lambda'}$ and $\vec{\mu'}$ are created, where $\vec{\lambda'}$ is $\vec{0}$ and $\vec{\mu'}$ is formed by increasing $\vec{\mu}_i$ by $|pad_size|$. If the pad_size is negative, for each reference with subscripts $\vec{s}$ to the target array in the corresponding scope, new subscripts $\vec{s'}$ for each reference are created, where $\vec{s'}_i = \vec{s}_i + |pad_size|$.

Peel : PEEL (*id*, *peel_size*)

This transformation peels the dimension i of an array $\mathcal{A}$ indicated by *id* by reducing the dimension size by $|peel_size|$ and creates two arrays $\mathcal{A}_1, \mathcal{A}_2$. Two pairs of range vectors $(\vec{\lambda'}_h, \vec{\mu'}_h)$,$(\vec{\lambda'}_l, \vec{\mu'}_l)$ are created for resulting arrays $\mathcal{A}_1, \mathcal{A}_2$, respectively,

where $\vec{\lambda'}_h, \vec{\lambda'}_l$ are $\vec{0}$, and $\vec{\mu'}_h, \vec{\mu'}_l$ are as follows:

$$\vec{\mu'}_h = \begin{cases} |peel_size| & \text{if } peel_size > 0 \\ \vec{\mu}_i - |peel_size| & \text{otherwise} \end{cases}$$

$$\vec{\mu'}_l = \begin{cases} |peel_size| & \text{if } peel_size < 0 \\ \vec{\mu}_i - |peel_size| & \text{otherwise} \end{cases}$$

For each reference with subscripts $\vec{s}$ to the target array $\mathcal{A}$ in the corresponding scope, new subscripts $\vec{s'}$ are created by first choosing the right array, $\mathcal{A}_1$ if $\vec{s}_i$ is less than μ_i of array $\mathcal{A}_1$ or $\mathcal{A}_2$ otherwise; then new subscripts are calculated as follows:

$$\vec{s'}_i = \begin{cases} \vec{s}_i & \text{if refers to } \mathcal{A}_1 \\ \vec{s}_i - \vec{\mu'}_{h_i} & \text{otherwise} \end{cases}$$

Note that, according to the semantics of array peeling, the subscripts in the dimension i of all the references to the array $\mathcal{A}$ should be compile-time constants. As the array peeling transformations can be chained together, in this case, all these chained array peeling actions should apply on the same array dimension. The input to the next array peeling transformation is decided by the current peeling size. If the current peeling size is positive, which means the target array dimension is peeled off from the beginning, the remaining array $\mathcal{A}_2$ will be the input for the next array peeling action. Otherwise, the target array dimension is peeled off from the end and thus the remaining array $\mathcal{A}_1$ will be the input for the next array peeling action, demonstrated by the `Split AoS` in Table 3.2.

The four data layout transformations are classified into two classes, *pre-action* and *post-action*. The *post-action* means all actions of this class can only be added after all the actions in the class of *pre-action*. We define *array peeling* as a member of the class *post-action* because we observe that for vectorization, array peeling is mainly used to split one array dimension for the data alignment or making the size of the array dimension power-of-two.

3.2.3 Composition of Data Layout Transformations

Our proposed *array transform* supports four primitive data layout transformations on static arrays. More complex data layout transformations can be achieved by composing these primitive transformations.

Array permutation permutes several array dimensions according to a given permutation command. It is more general than array interchange, which only swaps two array dimensions indicated by the dimension names. It is intuitive that array permutation can be decomposed as a sequence of array interchange actions. For example, given an array: `float A[SIZE_I][SIZE_J][SIZE_K]`, where i, j, k are the dimension names for each array dimension from the first to the last dimension, the permutation command (k, i, j), which rearranges the array dimensions indicated by i, j, k into a new order k, i, j, can be decomposed into a sequence of array interchange transformations, $(k, j) -> (i, k)$. Therefore, programmers can put the array transform pragma as `#pragma array_transform A[i][j][k] -> INTERCHANGE(k, j) -> INTERCHANGE(i, k)`

Rectangular array tiling blocks array dimensions into tiles, and thus decomposes the whole array into blocks which may help improve data locality. Array tiling is a process of choosing suitable hyperplanes according to certain conditions (e.g. data reuse distance) and partitioning the array data space with these hyperplanes. Here, *rectangular array tiling* means the determined tiling hyperplane for each array dimension is perpendicular to the axis of the array dimension to be tiled. Similar to the loop tiling which is a combination of loop strip-mining and loop interchange, *rectangular array tiling* can be decomposed into a sequence of array strip-mining, and array interchange, which are the primitive transformations defined in the *array transform* pragma.

As listed in Table 3.1 in Section 2.1, there are seven possible data layout transformation schemes for the motivating example. With our proposed *array transform* pragma, programmers can easily specify these data layout schemes by giving varying sequences of valid transformation actions, as shown in Table 3.2.

Table 3.2: Data layout transformations assuming the array u is originally in the Pure AoS.

Description	Declaration	Data Layout Transformation
Pure AoS	double u [KMAX][JMAXP][IMAXP][5];	NA
Split AoS (1:4)	double u1 [KMAX][JMAXP][IMAXP]; double u2 [KMAX][JMAXP][IMAXP][4];	#pragma array_transform u[i][j][k][m]-> PEEL(m, 1)

Split AoS (4:1)	double u1 [KMAX][JMAXP][IMAXP][4];
	double u2 [KMAX][JMAXP][IMAXP];
		#pragma array_transform u[i][j][k][m]->
		PEEL(m, -1)

Split AoS (1:2:2)	double u1 [KMAX][JMAXP][IMAXP];
	double u2 [KMAX][JMAXP][IMAXP][2];
	double u3 [KMAX][JMAXP][IMAXP][2];
		#pragma array_transform u[i][j][k][m]->
		PEEL(m, 1) -> PEEL(m, 2)

Split AoS (2:2:1)	double u1 [KMAX][JMAXP][IMAXP][2];
	double u2 [KMAX][JMAXP][IMAXP][2];
	double u3 [KMAX][JMAXP][IMAXP];
		#pragma array_transform u[i][j][k][m]->
		PEEL(m, 2) -> PEEL(m, 2)

Pure SoA	double u [5][KMAX][JMAXP][IMAXP];	#pragma array_transform u[i][j][k][m]-> INTERCHANGE(m, k) -> INTERCHANGE(m, j) -> INTERCHANGE(m, i)

Hybrid AoS	double u [KMAX][JMAXP][IMAXP/4][5][4];	#pragma u[i][j][k][m]-> STRIP_MINE(k, 4, kk) ->
		INTERCHANGE(m, kk)

3.3 Data Layout Aware Loop Transformations

Array strip-mining introduces modulus operations to get offsets in the resulting tiles, illustrated in line 8 of Fig. 3-3. This kind of operation is not friendly to vectorization, because it might hinder the native compiler from detecting possible consecutive data access. Both the Intel C compiler and GCC are not able to identify that the data references to the transformed array are consecutive. Although the Intel C compiler is able to vectorize the code with the SIMD pragma `#pragma simd` annotated around the loop, the vectorized code strictly conforms to the semantics of the modulus operations. Therefore, the performance is not so good as the one with consecutive data accesses.

```
1    #pragma ary[i] -> STRIP_MINING(i, 4)
2    float ary[32];
3    /* before transformation: */
4    for (i = 1; i < 31; i++)
5      ... = ary[i];
6    /* after transformation: */
7    for (i = 1; i < 31; i++)
8      ... = ary[i/4][i%4];
```

Figure 3-3: Loop transformation without considering data layout.

The modulus operations in the data references to the transformed arrays are from the array strip-mining. Therefore, if the data references to the target array to be transformed are enclosed in loops, one easy way to get rid of the modulus operations is to strip-mine the corresponding loops. In this chapter we only consider the case where all the references to the arrays to be transformed have uniform effects to the surrounding loops. By which it means, if a loop is strip-mined with stride δ according to one data reference, there should be no other data references which require the same loop to be strip-mined with stride other than δ.

Data layout aware loop strip-mining according to the array strip-mining may include pre-loop peeling and post-loop peeling depending on whether the loop iteration space and the data index space are aligned, as shown in line 6-8, 14-16 of Fig. 3-4. If a loop starts from 0 and ends at `SIZE-1` and the corresponding array dimension has a range from 0 to `SIZE-1`, in this case, the loop iteration space and the data index space are aligned, otherwise they are unaligned. Regarding the legality of these data layout aware loop peeling and loop strip-mining, they are always legal because these loop transformations

```
#pragma ary[i] -> STRIP_MINING(i, 4)
float ary[32];

/* data layout aware transformation:*/
/* from pre-loop peeling */
for (i = 0; i < 1; i++)
  for (ii = 1; ii < 4; ii++)
    ... = ary[i][ii];
/* from loop strip-mining */
for (i = 1; i < 7; i++)
  for (ii = 0; ii < 4; ii++)
    ... = ary[i][ii];
/* from post-loop peeling*/
for (i = 7; i < 8; i++)
  for (ii = 0; ii < 3; ii++)
    ... = ary[i][ii];
```

Figure 3-4: Data layout aware loop transformation.

inherently will not change the data dependencies across loop iterations.

In addition to the elimination of the modulus operations, the data layout aware loop strip-mining helps solve the alignment issue in vectorization. If the loop iteration space and the data index space are not aligned, pre-loop peeling and post-loop peeling are applied according to the boundaries of tiles from the array strip-mining. If the array starting address is aligned to 32 bytes and the tile size is 32 bytes, for instance, all the boundaries of tiles will be aligned to 32 bytes as well. As a result, all the loads from these boundaries are aligned to 32 bytes.

3.4 Experimental Evaluation

3.4.1 Implementation

Our proposed array transform pragma is implemented in the Cetus source-to-source C compiler. All the *transform actions* are processed and collected in the pragma parsing phase. The actual data layout transformations and the data layout aware loop optimizations are done as transform passes in the Cetus compiler. We also introduce loop unrolling and constant propagation as pre-processing passes. The high-level internal presentation in the Cetus compiler keeps the array access close to the source code and thus simplifies the array transformation and the substitution of subscripts in array references.

3.4.2 A Case Study: data layout tuning for loop vectorization

We use the SP in the NAS Parallel Benchmarks [Bailey et al., 1991b] as a case study to show the performance impact of data layout transformations upon loop vectorization. **SP** is one of the simulated CFD applications that solve the discretized compressible Navier-Stokes equations. We choose the data set of Class A in NPB, which has the size of $64 \times 64 \times 64$ with 400 iterations. Note that, we don't consider other cache optimizations (e.g. array padding) in our evaluation. All the experiments are conducted on an Intel Haswell platform (Intel Core i7-4770) running the Ubuntu Linux 13.04. We choose the Intel C compiler 13.1.3 to compile both the original and transformed code with the compiler option `-march=core-avx2 -O3 -fno-alias` for vectorization.

Performance of the Motivating Example

Fig. 3-5 gives the performance of the motivating example in different data layouts shown in Table 3.2. The results show that the best vectorization performance is given by the data layout transformation `Split 1:2:2`. Splitting the last dimension of the array `u` (line 22 in Fig. 3-1) into three parts with sizes of 1, 2 and 2 helps the native compiler vectorize the load of array `u` with a contiguous vector load. In the mean time, data permutation instructions (e.g. `vperm2f128, vunpacklpd`) are used for the data reorganization of the array `rhs` (line 17 - 21 in Fig. 3-1) instead of gather instructions.

Overall Performance

We manually tune the data layout transformations for the SP and constrain the search space of data layout transformations to the ones mentioned in Table 3.2. Fig. 3-6 presents the overall performance of the SP in different data layouts. Among the seven data layouts, the `Hybrid SoA` gives the best overall performance.

We also evaluated the performance of the single precision SP with the data layout `Hybrid SoA`, where the strip-mining size is 8. Compared to the double precision SP, the performance boost from vectorization for the single precision SP is more significant, as depicted in Fig. 3-7.

Fig. 3-8 and Fig. 3-9 give the performance breakdown of the single precision and double precision SP, respectively. With naive manual tuning of data layouts, for the SP,

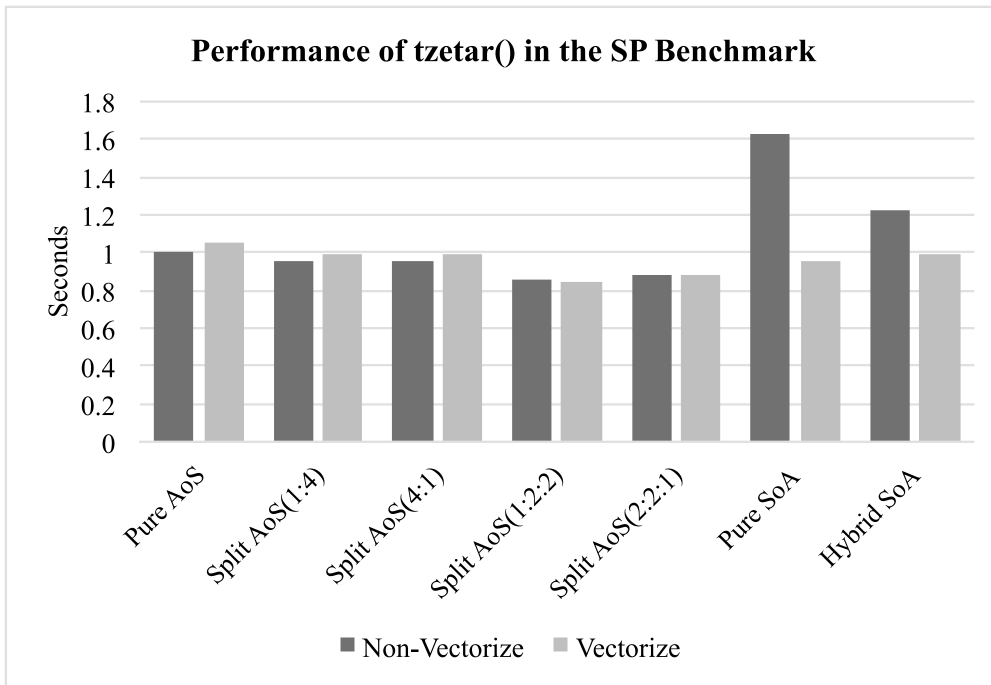


Figure 3-5: Performance of `tzetar()` with different data layout transformations .

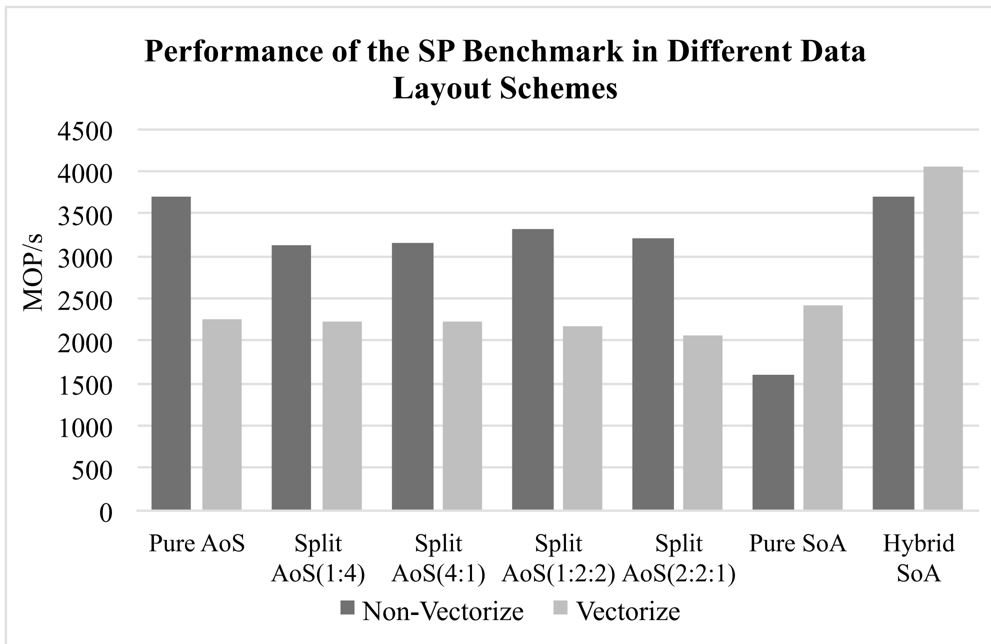


Figure 3-6: Performance of the SP in different data layouts.

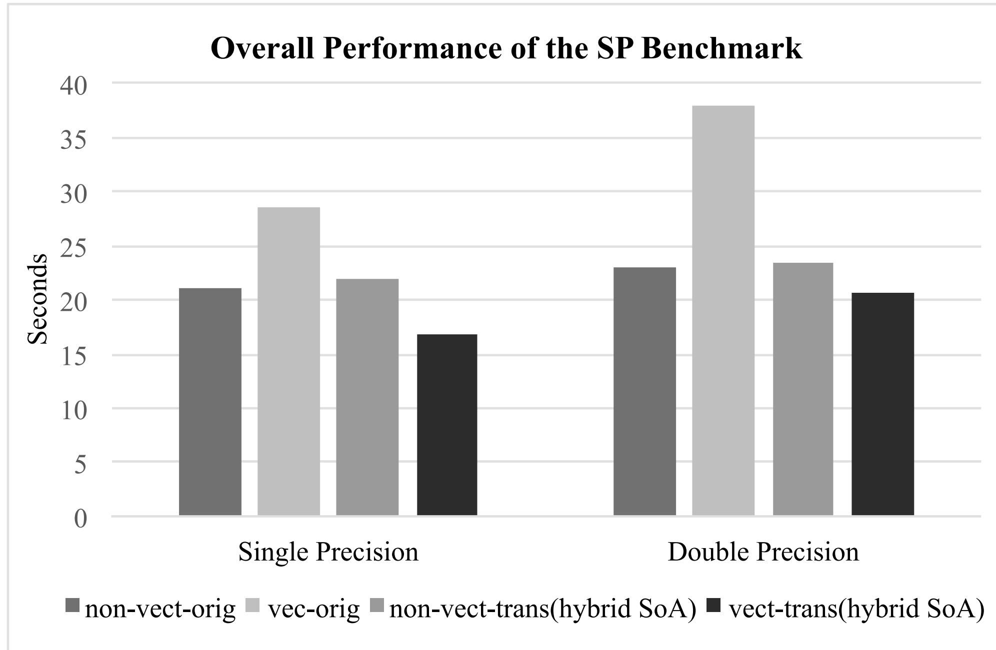


Figure 3-7: Performance of the SP of the NAS Parallel Benchmarks.

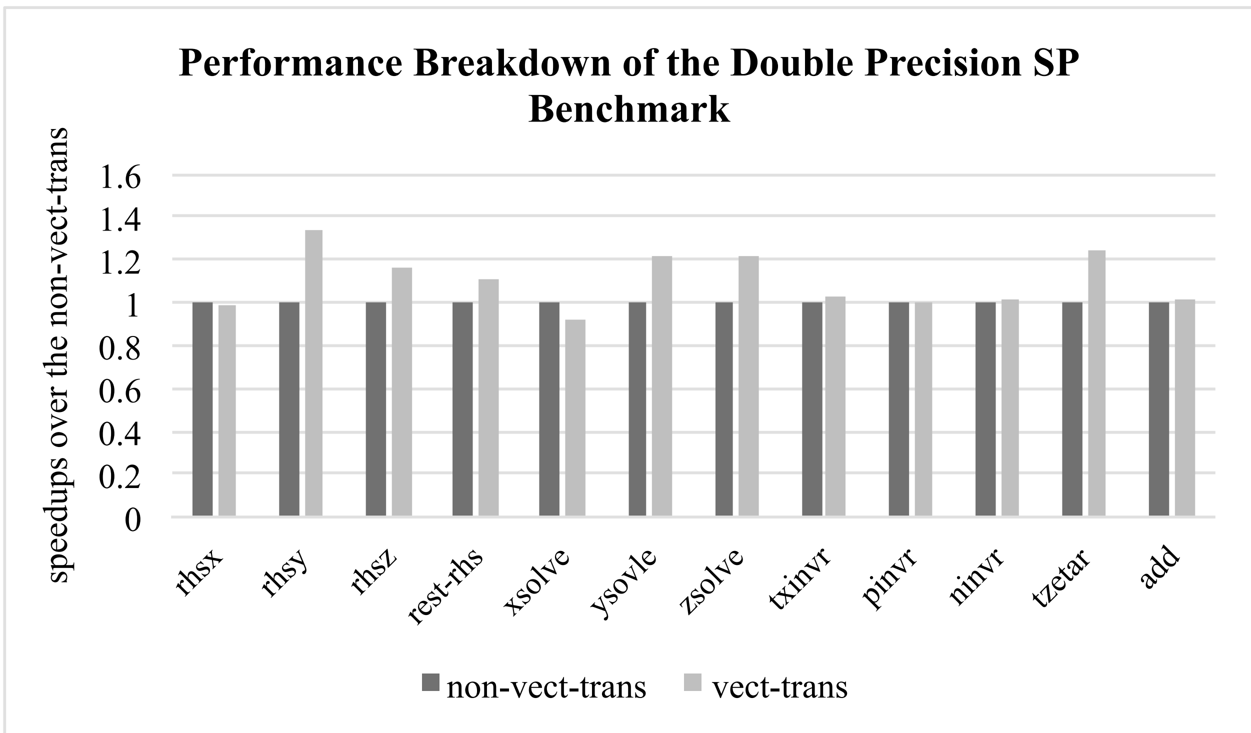


Figure 3-8: Performance breakdown of the double precision SP of the NAS Parallel Benchmarks.

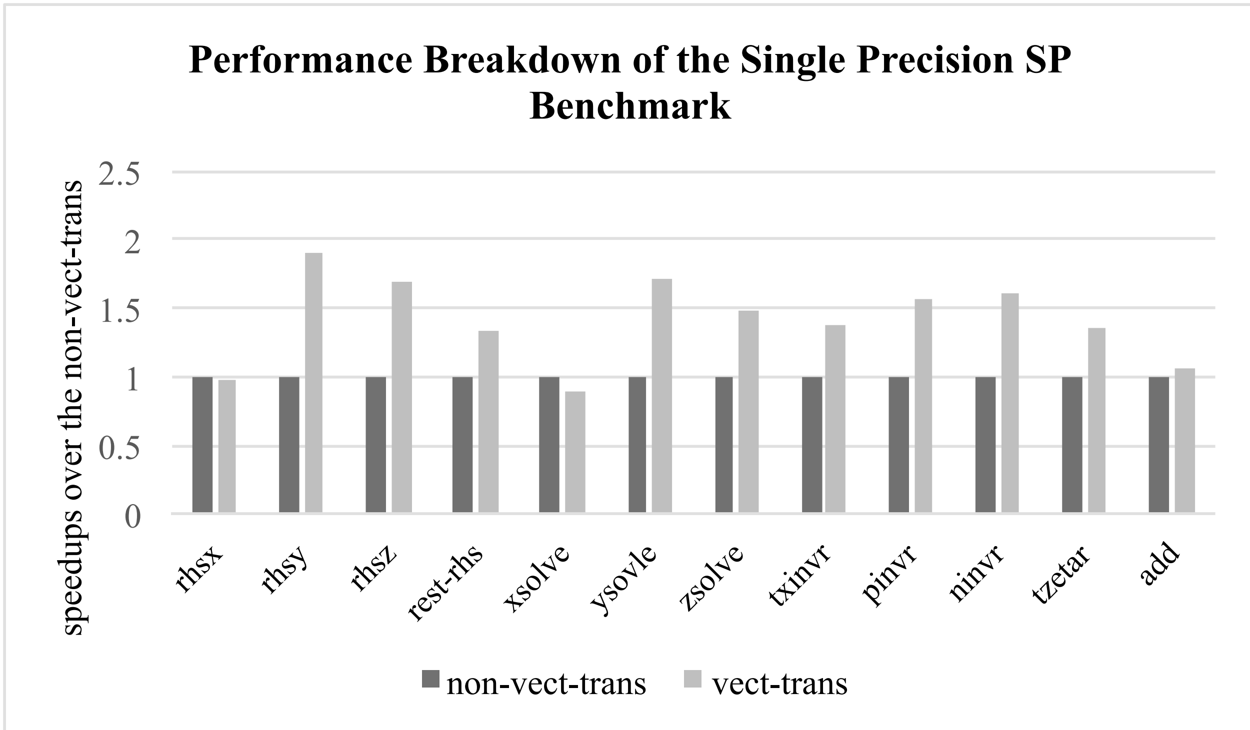


Figure 3-9: Performance breakdown of the single precision SP of the NAS Parallel Benchmarks.

vectorization on the transformed data can outperform the vectorization on the untransformed data by a factor of 1.8. The experimental results demonstrate that it is worthwhile to introduce data layout tuning into existing performance auto-tuning systems, in particular, for the better performance of vectorization.

3.5 Related Work

Data layout transformations have primarily been applied to improving cache locality and localizing memory accesses in nonuniform memory architectures and clusters [Kennedy and Kremer, 1998]. Maleki et al. evaluated the vectorizing compilers and found that manually changing the data layout is a valuable way to help compilers to efficiently vectorize loops with non-unit stride accesses [Maleki et al., 2011]. However, due to the limitation of inter-procedural analysis, most current compilers (e.g. Intel C Compiler, GCC) do not apply this transformation or apply it conservatively (e.g. IBM XLC). As a result, compilers rarely automatically perform the memory layout transformations. Our approach is based on the observation that programmers often know that some data layout

transformations are safe but implementing the changes manually is impractical because it requires many changes throughout the code. With our proposed array transform pragma, programmers can specify the data layout transformations easily and our source-to-source compiler propagates the changes throughout the program.

Our work is mainly inspired by the the work on semi-automatic composition of loop transformations for deep parallelism and memory hierarchies [Girbal et al., 2006]. The main approach of previous work is introducing a script language to control the loop transformations upon the target loops. As far as we know, there are no such script languages available to control the data layout transformations. Similar language support for data layout transformations is designed mainly for optimizing data locality, such as the align and distribute directives in HPF [Rice University, 1993].

Henretty et al. propose a novel data layout transformation, dimension-lifted transposition, for stencil computations [Henretty et al., 2011]. This domain-specific technique solves the memory stream alignment issue. On the contrary, our work is a general solution to manual data layout transformations. Although our work currently applies data layout transformations according to the scopes of the target arrays rather than the computation structures, our array transform pragma can be easily extended to work for a specific region of code.

Our work is very close to the work by Sung et al., which presents a framework that enables automatic data layout transformations for structured grid code in CUDA [Sung et al., 2010]. Our work not only supports more data layout transformations, but also presents data layout aware loop transformations for loop vectorization.

Jang et al. optimize memory access into DRAM bursts (i.e. coalescing) by creating unit-stride accesses with data layout transformations in the case of GPGPUs [Jang et al., 2010]. Majeti et al. put forward a meta-data framework that allows both programmers and tuning experts to specify architecture specific and domain specific information for parallel-for loops of programs [Majeti et al., 2014]. The data layout transformations considered in this work are only AoS-to-SoA and SoA-to-AoS. Sinkarovs et al. also present a compiler driven approach towards automatically transforming data layouts into a form that is suitable for vectorization [Sinkarovs and Scholz, 2013]. Their work is studied in the case of a first-order functional array programming language while our

work focuses on the imperative C language.

3.6 Summary

In this chapter, we put forward a new program annotation (using C language pragmas) to enable programmers to specify data layout transformations and implemented it in the Cetus source-to-source compiler. In terms of loop vectorization, we introduce data layout aware loop transformations to help native compilers to do better vectorization as well. The four primitive data layout transformations presented are suitable to be composed into more complex data layout transformations. The experimental results indicate that it is necessary to introduce semi- or fully automatic tuning of data layout transformations in order to help compilers to achieve better performance on vectorization.

Chapter 4

Exploit Computation Structure Exposed by Data Layout in Vectorization

4.1 Introduction

The introduction of Single Instruction Multiple Data (SIMD) units in processors increases the levels of parallelism in hardware, and results in a three-level hierarchy of parallelism, instruction level parallelism, SIMD parallelism, and thread-level parallelism. In order to take advantage of the SIMD parallelism, users usually resort to the automatic vectorization in compilers. So far, there are mainly two vectorizing approaches available in compilers, classic loop vectorization [Kennedy and Allen, 2002] and super-word level parallelism (SLP) vectorization [Larsen and Amarasinghe, 2000]. These two methods usually supplement each other. Classic loop vectorization works on each statement in the vectorizable loop while SLP vectorization attempts to pack the isomorphic operations in the basic blocks based on some heuristics (contiguous memory access [Larsen and Amarasinghe, 2000] or data reuse [Liu et al., 2012]). What these two methods have in common is that they both ignore the overall computation structure exposed by the vectorizable loop.

With the advance of SIMD support in modern commodity processors with short vectors, more and more advanced features are introduced to programmers and compiler designers to exploit the performance of SIMD, such as the flexible lane-wise operations (e.g. masking load/store, blend instructions). When using these SIMD lane-wise opera-

tions, we have to consider how the SIMD lanes change between SIMD instructions. With the computation structure of the vectorizable loop, we can have a global view of how the SIMD lanes can be allocated in each SIMD instruction. This view of SIMD lanes helps us to achieve global SIMD lane-wise optimization, which may reduce unnecessary shuffling operations on SIMD lanes.

```
1    float y[128], x[128], C[128];
2    for (int i = 0; i < 64; i++) {
3      y[2*i]  += x[2*i] * C[2*i] -  x[2*i+1] * C[2*i+1];
4      y[2*i+1] += x[2*i] * C[2*i+1] + x[2*i+1] * C[2*i];
5    }
```

Figure 4-1: C-Saxpy

```
1    y[0:126:2] += x[0:126:2] * C[0:126:2] - x[1:127:2] * C[1:127:2];
2
3    y[1:127:2] += x[0:126:2] * C[1:127:2] + x[1:127:2] * C[0:126:2];
```

Figure 4-2: C-Saxpy by classic loop vectorization.

```
1    // take full lanes
2    tmp0[0:127] = x[0:127:1] * C[0:127:1];
3    tmp1[0:127] = SwapEvenOddLanes (tmp0);
4    // actual computation on the even lanes
5    tmp1[0:127:1] = tmp0 - tmp1;
6    tmp2[0:127:1] = SwapEvenOddLanes (C[0:127:1]);
7    // take full lanes
8    tmp3[0:127:1] = x[0:127:1] * tmp2[0:127:1];
9    tmp4[0:127:1] = SwapEvenOddLanes (tmp3);
10   // actual computation on the odd lanes
11   tmp5[0:127:1] = tmp3 + tmp4;
12   // merge the results from both even and odd lanes
13   y[0:127:1] += MergeEvenOddLanes (tmp1, tmp5);
```

Figure 4-3: C-Saxpy by hyper-loop parallelism vectorization

Take the C-Saxpy, which multiplies a complex vector by a constant complex vector and adds it to another complex vector, as an example, as shown in Fig. 5-2. When classic loop vectorization attempts to vectorize the loop, it tries to aggressively squeeze all the data needed by each memory operation into a SIMD vector regardless of how the data will be used throughout the loop body. As shown in Fig. 4-2, all memory operations are either interleaved loads (gather) or interleaved stores (scatter). The hardware support

for native gather and scatter instructions is still not good [Ramachandran et al., 2013], therefore, most compilers use data permutation instructions to achieve gather and scatter operations.

If we carefully examine the computation structure of the loop body in Fig. 5-2, we can derive a vectorizing scheme with fewer data permutation instructions than the one by classic loop vectorization. As we can see from Fig. 4-3, all the memory operations are contiguous memory loads and stores, and only two `SwapEvenOddLanes` and a `MergeEvenOddLanes` operations defined in Section 4.2.3 are required to reorganize data. This vectorizing scheme is obtained by putting in data reorganization operations to adjust the data needed by the SIMD computation according to the overall computation structure.

Two key components are required by the vectorizing scheme shown in Fig. 4-3. One is the computation structure recognition and the other is SIMD lane-wise mapping. Computation structures can be obtained by program slicing with suitable slicing criteria. On the other hand, SIMD lane-wise mapping requires detailed information on how to position data in SIMD lanes along the computation structure. For classic loop vectorization, as it strip-mines the vectorizable loop for vectorization, the numbering of the loop iterations of the resulting loop determines which SIMD lane a loop iteration will take. Inspired by this mapping between loop iterations and SIMD lanes, we put forward hyper loops based on program slices to recover the loop structure of the vectorizable loop. With hyper loops, we can apply global SIMD lane-wise optimization by taking advantage of the mapping between loop iterations and SIMD lanes.

We define the program slices that can be partitioned into groups with respect to certain relationships (i.e. contiguous memory stores) as hyper loop iterations. The computations in each hyper loop iteration of a group do not have to be isomorphic. As all the program slices are independent of each other, hyper loop iterations are all parallel. The parallelism exposed by the hyper loop iterations is hyper loop parallelism. In this chapter, we put forward a vectorizing technique based on the hyper loop parallelism. Our vectorizing method addresses the problems of extracting hyper loop parallelism and efficiently mapping it onto the target processor. We implemented our vectorizing approach as a source-to-source compiler in the Cetus source-to-source compiler. The

preliminary experimental results show that our vectorizing technique can achieve significant speedups over the non-vectorized code.

4.2 Hyper Loop Parallelism in Vectorization

4.2.1 Overview

Classic loop vectorization strip-mines vectorizable loops. The loop iterations of the resulting loops correspond to the SIMD lanes in the SIMD vectors. In order to take advantage of the instructions that have flexible control of the SIMD lanes in modern commodity processors, we put forward hyper loops to recover the implicit loop structures of the loop body.

The loop body of a vectorizable loop generally can be partitioned into parts in terms of the downwards-exposed definitions. Program slicing is a widely used technique to compute a set of program statements, *a program slice*, which may affect the values at some point of interest (aka. *a slicing criterion*). Choosing the downwards-exposed definitions of the vectorizable loop as the set of slicing criteria, with the backward program slicing, we can derive a set of program slices, each of which represents a partition of statements of the loop. Without considering control dependence, a program slice within a loop body is essentially a sub-graph of the data dependence graph of the loop body. As each slice is collected within the loop body, a slice is a direct acyclic graph (DAG) $G(V, E)$, where V is the set of computations within the slice, and E are the define-use relationships between nodes in V.

There are three slices after program slicing in Fig. 4-4. Without considering the relationships between the slices, we can treat each slice as a loop with only one iteration. However, in real world applications, there usually exist relationships between the slices. The relationships between the slices often come from two aspects: 1) unrolled loops from the loops with no loop carried dependence; and 2) computations on the tuples of data organized in an array of structures. For the former case, each unrolled loop iteration is a slice and all the slices are isomorphic. In other words, the DAGs representing the unrolled loop iterations have the same structure and computations on each DAG are isomorphic correspondingly. On the other hand, for the computations on the fields of

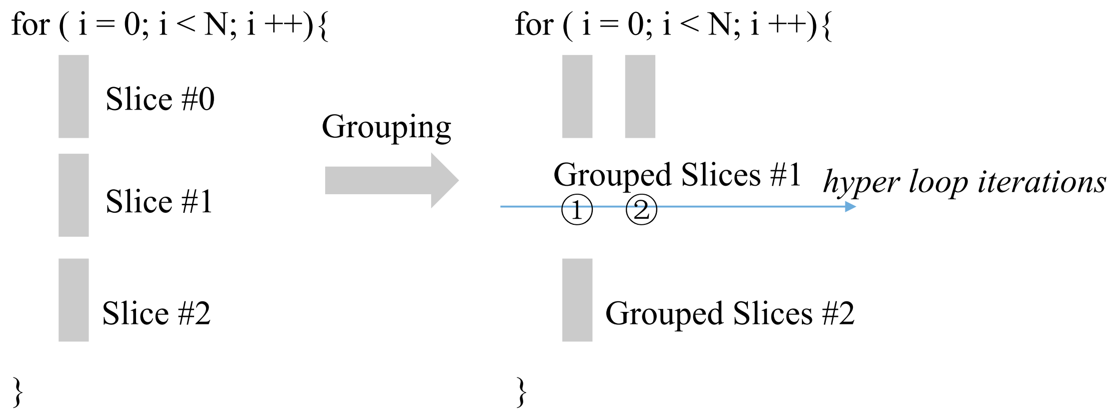


Figure 4-4: Hyper loop parallelism for vectorization.

data organized in an array of structures, the DAGs for the elements of the tuple may have different structures depending on the computation (e.g. C-Saxpy in Fig. 5-2). However, as each slice is for the computations regarding an element of the tuple, the relationships between elements (aka. contiguous memory access) build the relationships between the slices.

The relationships between slices (aka. contiguous downwards-exposed definitions) can be used to group slices into grouped slices, or grouped DAGs. We can deem a slice group as a hyper loop where the number of hyper loop iterations is the same as the number of slices in the group. As each slice is an independent partition of the loop body, hyper loops are all parallel and eligible to vectorization. Grouped slices help vectorization to achieve flexible control on SIMD lanes. For instance, as shown in Fig. 4-4, according to the iteration number of the hyper loop, when mapping the grouped slices to the SIMD vector, the two slices in the grouped slices #1 prefer to take the even and odd lanes, respectively. With this precise information on SIMD lanes, vectorization can apply global SIMD lane-wise optimization when mapping the slices to the SIMD vector in order to reduce the number of shuffling operations on SIMD lanes.

In this chapter, we propose a vectorizing technique by exploiting the hyper loop parallelism exposed by the hyper loop. Similar to other vectorization frameworks, our vectorizing technique consists of two stages, vectorization analysis and vectorization transformation.

4.2.2 Vectorization Analysis

Before collecting program slices for hyper loop parallelism, we use existing data dependence analysis to analyze whether a loop is vectorizable or not. Moreover, we apply data-flow analysis to find the downwards exposed definitions in the vectorizable loop and identify the types of the definitions, *reduction definition* or *ordinary definition*.

Collect Slices

All the downwards-exposed definitions in the loop are used as the slicing criteria for program slicing. As the data dependence graph is already built in the vectorization analysis, backward program slicing can be easily applied. As shown in Fig. 4-5, there are two ordinary definitions, `y[2*i]` and `y[2*i+1]`. We can get two slices from program slicing. Note that, the dash lines depict the define-use relationships among statements and connect a node to its parent in the DAGs representing the slices.

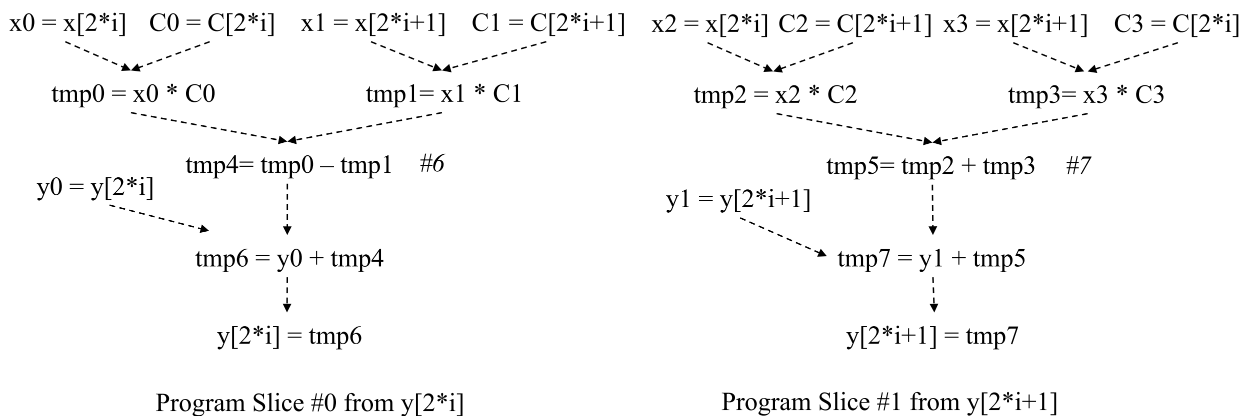


Figure 4-5: Collect program slices.

Group Slices

Grouping slices is a key stage for discovering hyper loop parallelism. In this stage, slices collected are first partitioned into two sets according to the types of downward exposed definitions.

Grouping slices works similar to the super-word level parallelism (SLP) vectorization that tries to pack isomorphic instructions into groups for vectorization [Larsen and Amarasinghe, 2000]. In contrast to the SLP vectorization, the grouping of slices starts from

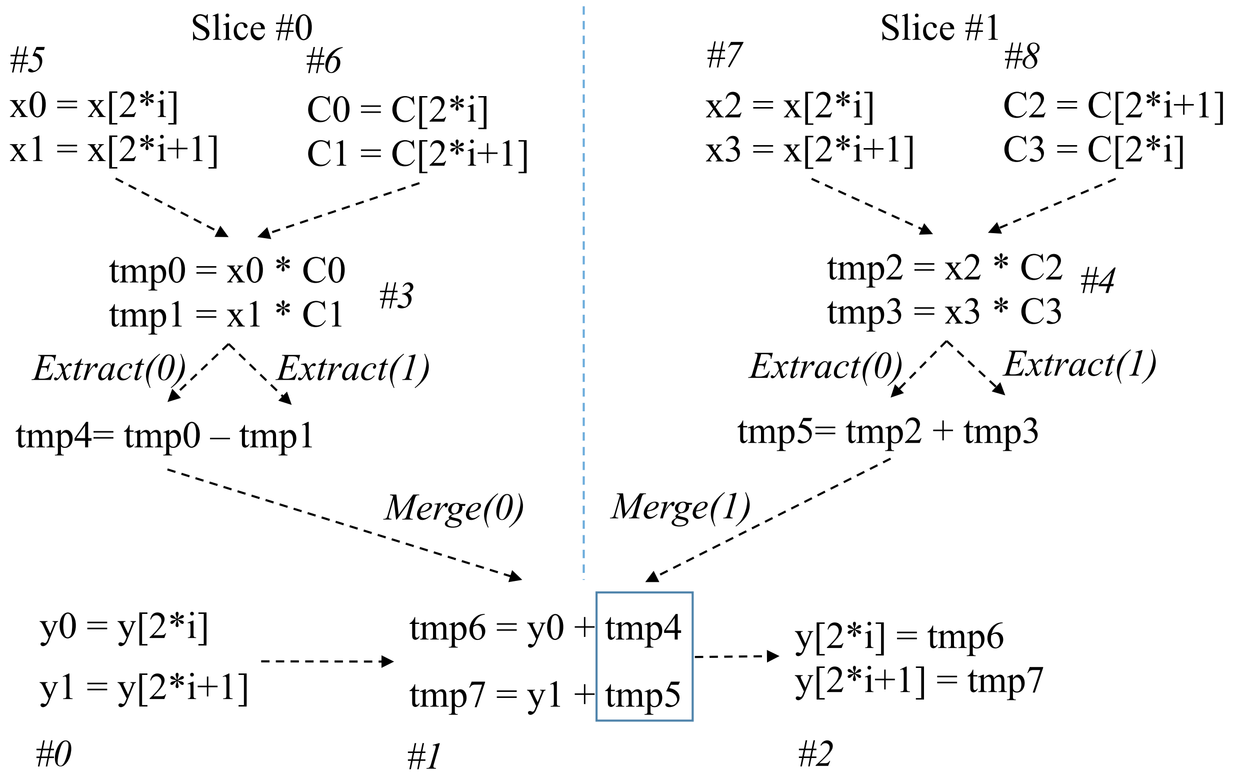


Figure 4-6: Group program slices.

contiguous memory stores which are the downwards exposed definitions for program slicing, and packs isomorphic operations from different slices. As stated in Section 4.2.1, two slices in the same group do not necessarily have the same computation structure. Thus, it is possible that some computations are not isomorphic. We define two types of grouping, *fully grouped* and *partially grouped*. If all the computations from two slices are isomorphic correspondingly, we call it *fully grouped*, otherwise *partially grouped*.

For partially grouped slices, in order to find more opportunities for vectorization, we apply grouping to the parts which are not grouped when grouping different slices. For example, when grouping the node #6 and node #7 in Fig. 4-5, as the computations from both nodes are not isomorphic. Hence, the grouping on both slice #0 and slice #1 terminates. In order to find more grouping opportunities, the grouping continues on each slice separately, and groups nodes with isomorphic operations within each slice.

Moreover, when dealing with partially grouping, we attach actions on the edges between two nodes in the grouped DAGs. We put forward two actions, *extract* and *merge*, to depict how the data flows. The `extract(`*number*`)` deals with data-flow from a grouped node to a non-grouped node while the `merge(`*number*`)` handles the data-flow from a

non-grouped node to a grouped node. The parameter *number* in both actions specifies the position of definition in the source node or the position of use in the destination node.

For the slices collected from the C-Saxpy, as shown in Fig. 4-5, Fig. 4-6 illustrates the results of grouping slices. Because the computations for the definitions of the two slices are different in some parts, the two slices are not fully grouped. Three grouped nodes (node #0 - node #2) are created by the grouping on the two slices while six grouped nodes (node #3 - node #8) are created by the grouping on the parts of slices which cannot be grouped.

Calculate Computation Attributes

Slices for grouping may overlap with each other depending on the computations. For fully grouped slices, the overlapping may lead to a grouped DAG that is not efficient for directly vectorization transformation. For example, the grouped DAG of vector normalization is shown in Fig. 4-7. All the nodes in the dashed boxes are from the overlapped parts of the three slices. If this grouped DAG is directly used for vectorization transformation, there would be a lot of redundant computation within SIMD lanes that may not be optimized out by compilers.

In order to achieve better vectorization transformation on the fully grouped slices, we calculate the computation attributes from the data access of each node in the grouped DAGs. As memory loads are in the *leaf* nodes of the DAGs, calculation starts with *leaf* nodes, and propagates the computation attributes to the root nodes. Each node by default has an *implicit* computation attribute decided by the data accesses pattens (e.g. consecutive, gathering). Two more *explicit* computation attributes are calculated for vectorization transformation, *reducible* and *scatterable*, as shown in Fig.4-8.

4.2.3 Vectorization Transformation

Expand Grouped Slices

After all the grouped DAGs have been collected and computation attributes for each node in the fully grouped DAGs are calculated, the vectorization transformation transforms

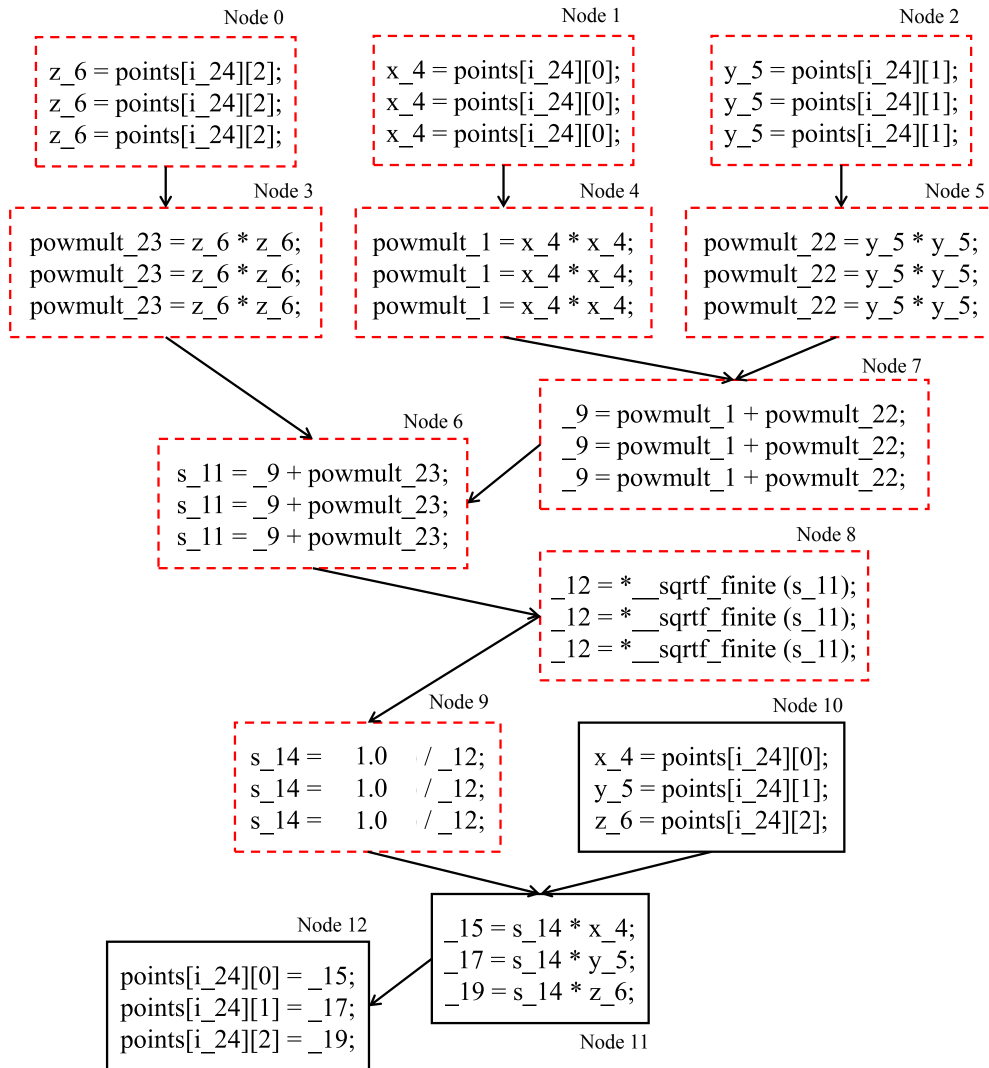


Figure 4-7: Overlapping of fully grouped slices.

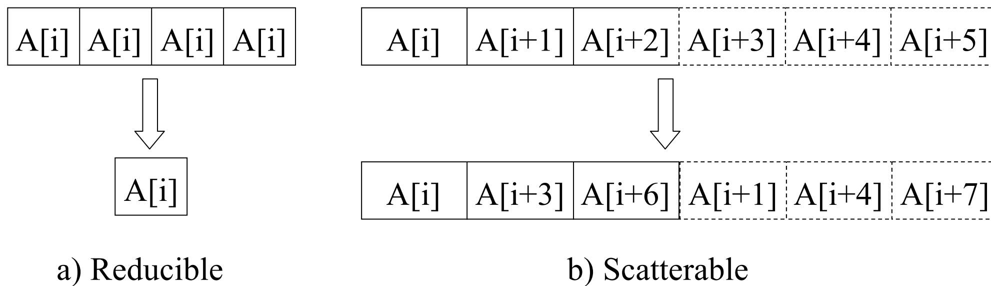


Figure 4-8: Reducible and scatterable computation attributes.

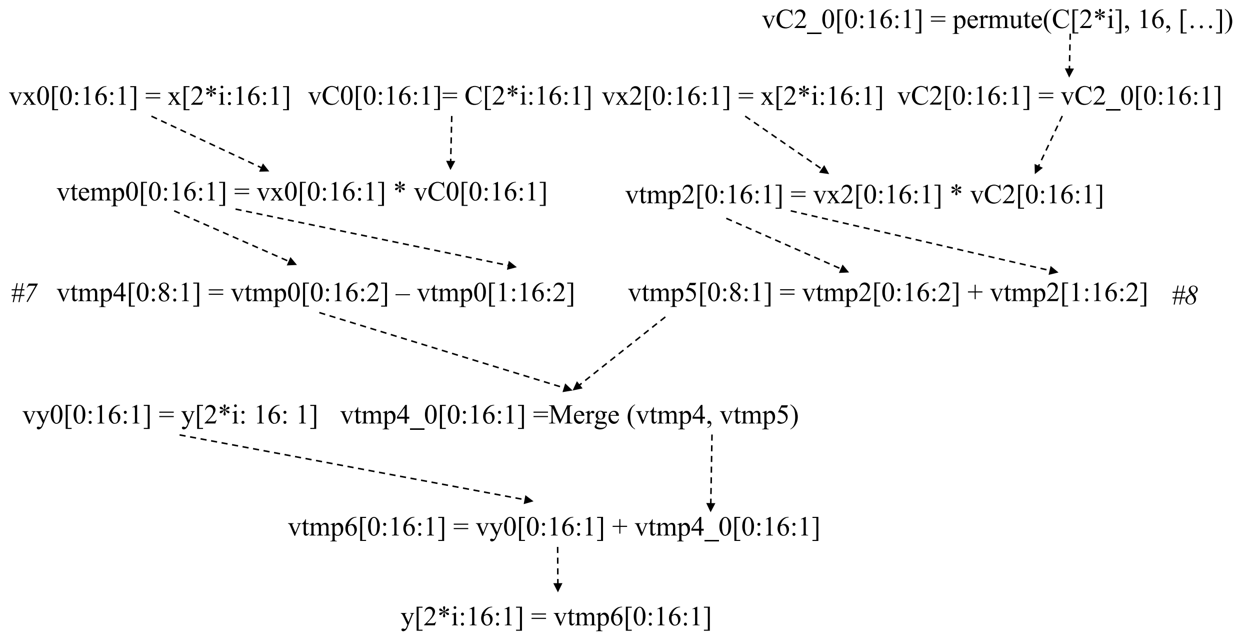


Figure 4-9: Expand program slices.

each grouped DAG into a vectorized DAG with virtual vector operations on virtual registers. We use the idea of virtual vector registers and vector operations similar to [Bocchino and Adve, 2006]. The loop unrolling factor for vectorization transformation is calculated by first finding the least common multiple (L.C.M.) value of the width of the physical vector register and the size of the grouped node with the minimum number of isomorphic operations, then dividing the value by the size of the smallest grouped node. The width of the virtual register of each node is decided by the multiplication of the loop unrolling factor and the size of the node.

For the fully grouped DAGs, since each node is already annotated with computation attributes, the vectorization transformation makes decisions on how to schedule data operations and computation along with generating virtual vector operations. In other words, the vectorization transformation decides when, where, and which kind of data operation is needed, such as consecutive load/store, gathered load.

The data and computation scheduling is made by the simple heuristics as follows: **1)** All the reducible *leaf* nodes of the DAGs are always reduced into nodes with a single operation; the data accesses in the reduced leaf nodes can be gatherable, consecutive (or replicable for constants) depending on the data access pattern; **2)** According to the cost of data permutation, consecutive loads have higher priority than gathered loads;

consecutive stores have higher priority than scattered stores. **3)** If the child nodes of a node are all reduced, the node is also reduced; **4)** If one of the child nodes of a node is reduced and expanded as gathered and the other child nodes are not reduced and but scatterable, all these non-reduced child nodes will be scattered and the corresponding computation sequence in the parent node will be scattered as well.

For the fully grouped DAG in Fig. 4-7, according to the heuristics mentioned above, the reducible leaf nodes 0-2 are first reduced. As the data accesses in the nodes 0-2 are interleaved with stride 3, data gathering operations are introduced when these reduced nodes are expanded. According to the rule 3, the reducible nodes 3-9 are reduced and expanded with gathered data thanks to the reduced child nodes. For the join node 11, according to the rule 4, although node 10 has consecutive data accesses, it is transformed into a node with scattered loads. As a result, the computation sequence in node 11 is skewed correspondingly. Because node 12 requires a consecutive store, data permutation is needed to transform the data from the skewed computation in node 11 back to consecutive data for the store operation. As we can see, rule 4 helps defer the data permutation operations needed to the final store operation, which may cut the number of vector registers required by data reorganization optimization and reduce the register pressure in the generated code.

When expanding the grouped DAGs into the vectorized DAGs, we use *SIMD lane descriptor*s to describe the patterns of SIMD lanes for each node. SIMD lane descriptors have the format of `id[start_position: size: stride]`, where `id` is the name of an array, a pointer or a virtual vector, `size` is the number of lanes, `stride` is the lane pattern. In this chapter, we consider strided SIMD lane pattens. The support for arbitrary SIMD lane patterns is beyond the scope of this thesis. For the grouped DAGs in Fig. 4-6, the vectorized DAG after expanding is shown in Fig. 4-9.

Global SIMD Lane-wise Optimization

If all the nodes in the expanded grouped DAGs have valid SIMD lane descriptors, the vectorization transformation applies global SIMD lane-wise optimization on the expanded grouped DAGs. The global SIMD lane-wise optimization tries to optimize the allocation of SIMD lanes according to the changes of SIMD lanes between nodes in the DAGs by

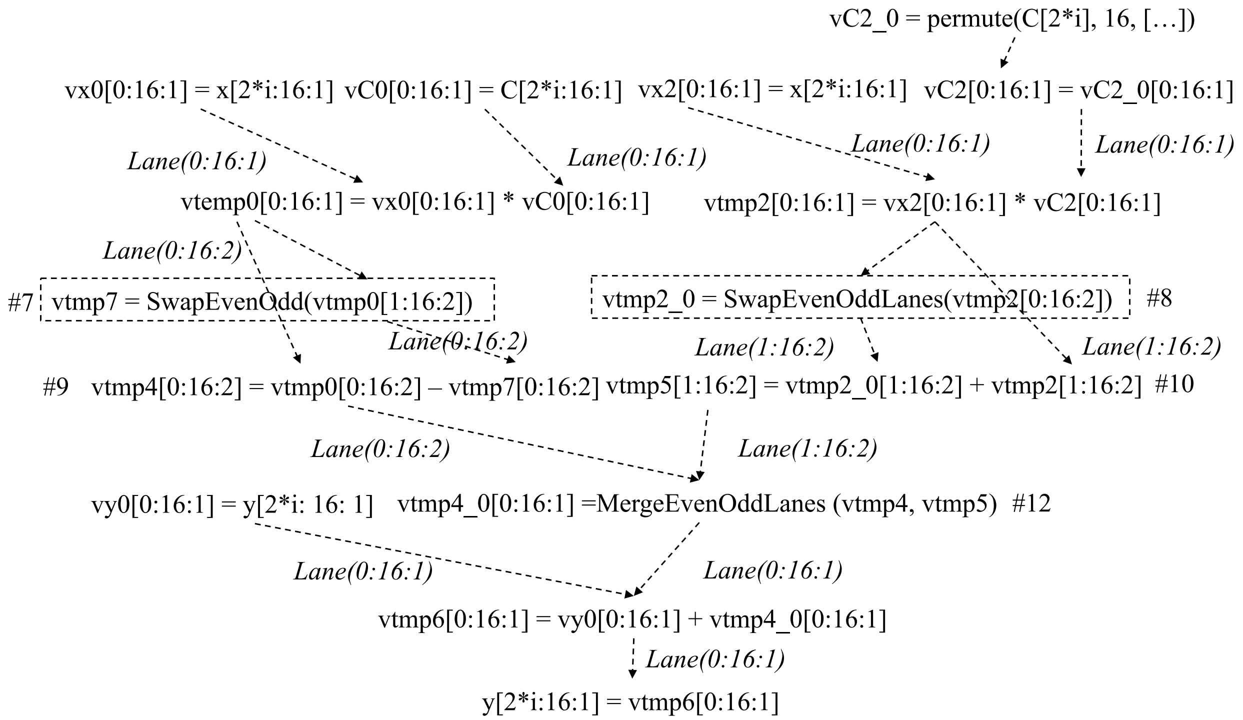


Figure 4-10: Global SIMD lane-wise optimization.

inserting new nodes for four SIMD lanes operations - `pack`, `unpack`, `merge` and `permute`. `pack` and `unpack` deal with the changes of the vector size. `merge` performs blending of two vectors with the given SIMD lane information. `permute` handles the changes of ordering of SIMD lanes between two vectors in the same size. The operations `SwapEvenOddLanes` and `MergeEvenOddLanes` in Fig. 4-10 are concrete instances of the operations `permute` and `merge`, respectively.

The global SIMD lane optimization consists of two passes, a top-down pass and a bottom-up pass on the expanded DAGs. The top-down pass tries to adjust the widths of virtual vectors and SIMD lane patterns according the memory loads in the leaf nodes in the grouped DAGs. For example, the node #8 in the expanded grouped DAG shown in Fig. 4-9 has a destination vector `vtmp5` with the SIMD lane pattern of `[0:8:1]`. The top-down pass changes the SIMD lane pattern into `[0:16:2]` according to the operand `vtmp2[0:16:2]` because both operands have strided SIMD lane patterns. Note that, since there is no other information to guide the choosing of SIMD lane patterns, the top-down pass always picks the SIMD lane pattern of the first operand as the pattern of the destination vector.

On the other hand, the bottom-up pass propagates the SIMD lane information of

the root nodes to the leaf nodes and inserts the four SIMD lane operations accordingly. The bottom-up pass, in particular, takes care of the join nodes represented by `Merge`. For instance, after the top-down pass, the destination vectors `vtmp4` and `vtmp5` have the same SIMD lane pattern of `[0:16:2]`. When comes to the merge node #12 in Fig. 4-10, according to the relationships between hyper loop iterations and SIMD lanes, the optimization will assign the even lanes to the `vtmp4` while giving odd lanes to the `vtmp5`. Thus, the desirable SIMD lane pattern `[0:16:2]` and `[1:16:2]` are propagated to the node #9 and node #10, respectively. Guided by the desirable SIMD lane patterns, a `SwapEvenOddLanes` operation is introduced to transform the SIMD lane pattern of `vtmp2` from `[0:16:2]` to `[1:16:2]` as the node #8.

4.3 Implementation

We implemented our proposed vectorization approach as a source-to-source compiler based on the Cetus compiler infrastructure [Bae et al., 2013]. The compilation flow for our vectorization approach is shown in Fig. 4-11.

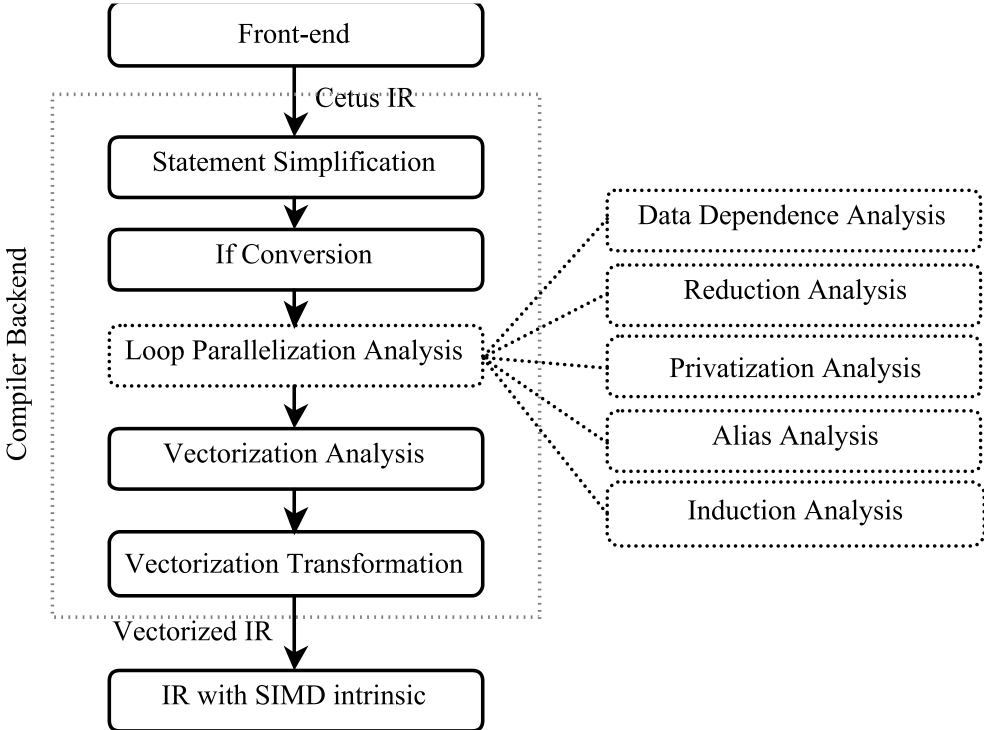


Figure 4-11: Compilation flow of hyper loop parallelism vectorization.

The Cetus compiler uses a single level internal representation (IR) which contains all

the information needed for high-level loop optimization. Although the IR closely conforms to the source code, expressions in this IR may have multiple levels which hinders compilers from detecting whether the expressions in two statements are isomorphic or not. To tackle this problem, we introduce a *Statement Simplification* pass to lower each statement into short statements with only one unary, binary or ternary expression and add temporary variables to hold the immediate values of these resulting expressions. In addition, we introduce a simple *If-conversion* pass to eliminate part of control dependence by replacing *if* statements with conditional statements.

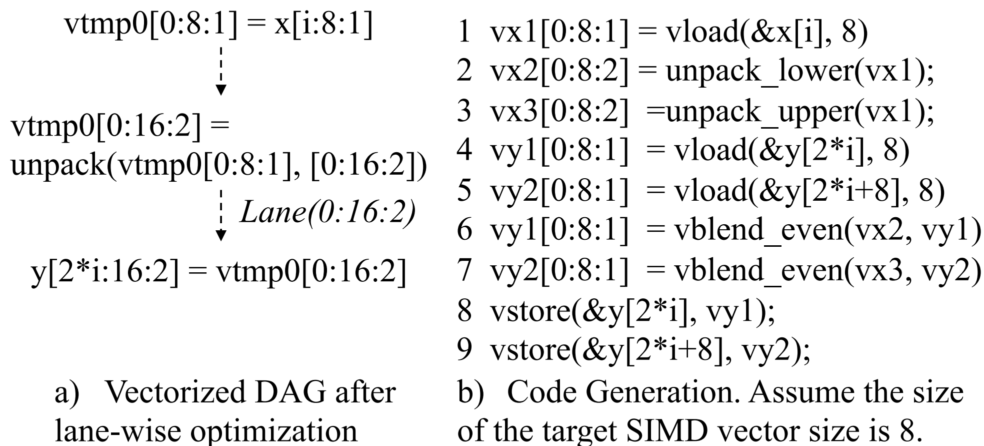


Figure 4-12: An example of code generation.

The vectorization analysis and transformation are applied as described in Section 4.2. After vectorization transformation, we lower the virtual vector operations to Intel AVX2 SIMD intrinsics. As the code generator is independent of the target architecture, our vectorizer can be easily extended to support other architectures (e.g., Intel AVX-512). When lowering the SIMD lane-wise operations to the SIMD intrinsics, our compiler uses data permutation and blend instructions to implement these operations. As shown in Fig. 4-12, when dealing with strided stores, the code generator emits contiguous vector loads (line 4-5), blends the results to be stored with the load vectors according to the stride (line 6-7), and stores the blended results with contiguous vector stores (line 8-9).

In the code generation, data permutation optimization is applied to the interleaved data access as well. Instead of general optimization on data permutation [Nuzman et al., 2006][Ren et al., 2006], such as the one specific to strides of power-of-two [Nuzman et al., 2006], we treat each specific case of interleaved data access separately. For example,

when dealing with interleaved data accesses with stride 3, we adopt the data permutation scheme considered optimal for this case [Melax, 2012].

4.4 Preliminary Experimental Results

4.4.1 Experimental Setup

As our compiler generates C code with SIMD intrinsics for Intel AVX2, all the experiments are conducted on an Intel Haswell platform, Intel(R) Core(TM) i7-4770, with Intel AVX2 running Ubuntu Linux 13.10. We use the Intel C compiler (ICC) 14.02 for automatic vectorization with compiler options `-march=core-avx2 -O3 -fno-alias` for performance comparison. The non-vectorized execution time is collected by ICC with compiler options `-march=core-avx2 -O3 -no-vec`
`-fno-alias`.

4.4.2 Benchmarks

We choose two groups of benchmarks to evaluate the effectiveness of our proposed vectorizing technique based on the hyper-loop parallelism. The Group I benchmarks are all suitable for fully grouping and some of them require the data and computation scheduling guided by the computation attributes (in Section 4.2.3). The Group II benchmarks contain some vectorizable loops that can only be partially grouped, and most of the vectorizable loops can benefit from the global SIMD lane-wise optimization.

- **Group I:** Five basic operations on 3D-vectors, **multiplication, dot product, normalization, rotation and cross product**, are often encapsulated as library functions in widely used libraries, such as Open Source Computer Vision Library (OpenCV). **YUVtoRGB** and **RGBtoYUV** are important applications in image processing. The 3D-vectors used in these benchmarks is organized in an array of structures.

- **Group II: C-Saxpy**, which multiplies a complex vector by a constant complex vector and adds it to another complex vector. Two benchmarks from the NAS Parallel Benchmarks, FT and MG. **FT** contains the computational kernel of a 3-D Fast

Fourier Transform (FFT). **MG** uses a V-cycle Multi Grid method to compute the solution of the 3-D scalar Poisson equation.

4.4.3 Performance

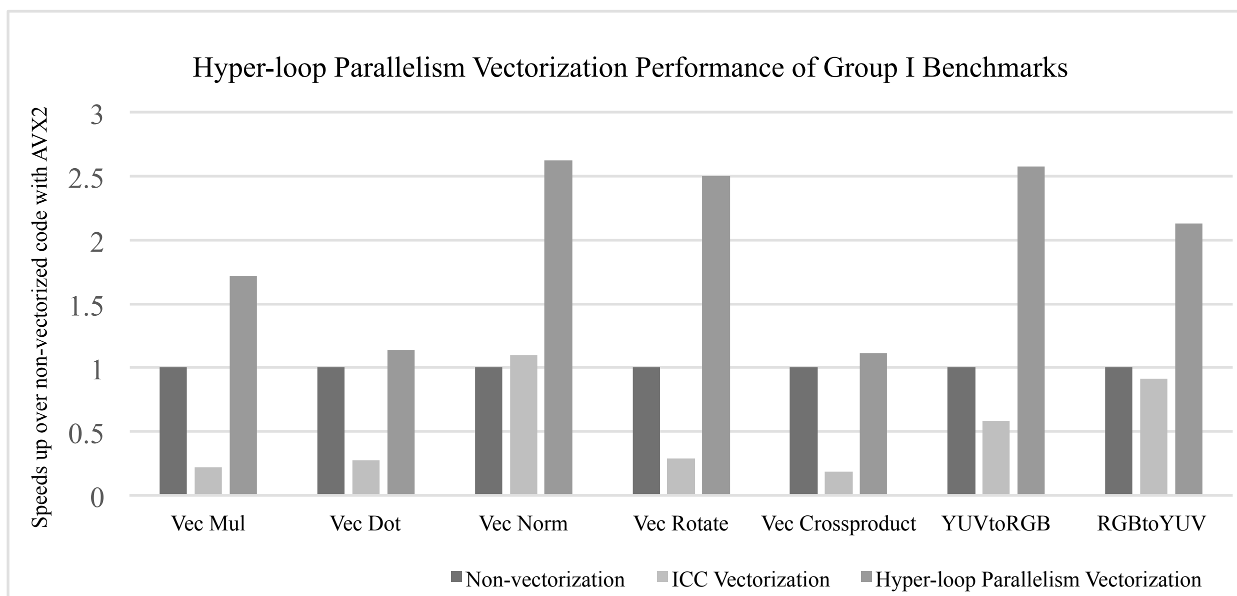


Figure 4-13: Performance of Group I benchmarks.

The overall performance of the Group-I benchmarks is given in Fig. 4-13. As we can see, the performance of vectorized vector multiplication, dot product, rotation and cross product, YUVtoRGB, RGBtoYUV by ICC is all worse than the non-vectorized code. The reasons for the performance degradation are 1) ICC by default chooses gather instructions (aka. *vgather*) to deal with interleaved data accesses with stride 3, and these instructions are not efficiently supported by the hardware [Pennycook et al., 2013]; 2) ICC has no support of optimization on data scattering with stride 3, thereby it generates a sequence of scalar instructions to extract data out of vector registers. The vectorized vector normalization by our method outperforms ICC because of the data permutation optimization specific to interleaved access with stride 3.

Fig. 4-14 presents the overall performance of the Group-II benchmarks. This group of benchmarks mainly test the effectiveness of the global SIMD-lane wise optimization. For the C-Saxpy, as we can see from Fig. 4-3, fewer data permutation instructions are required by the SIMD lane-wise optimization than the loop vectorization in Fig. 4-2.

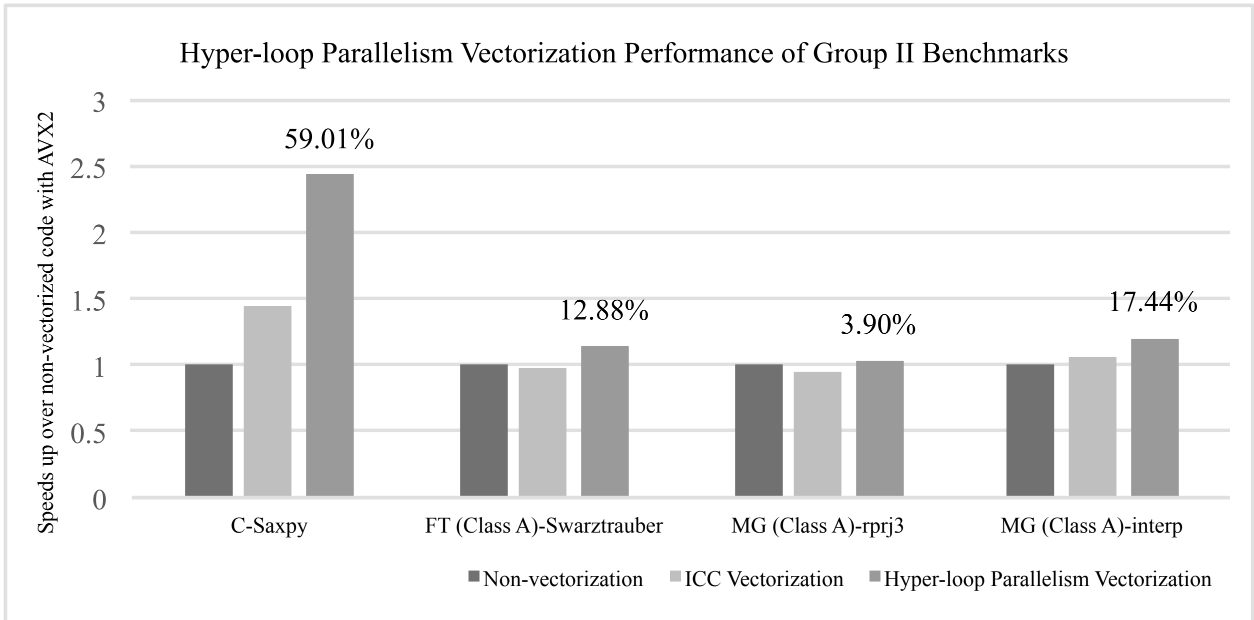


Figure 4-14: Performance of Group II benchmarks.

The reduction of data permutation instructions leads to a great speedup. Similar to the C-Saxpy, our vectorizing technique achieves great performance improvement over the non-vectorized execution for the functions from FT and MG while the vectorization by ICC degrades the performance of FT-Swarztrauber and MG-rprj3. The performance gains of the Group-II benchmarks by our vectorizing technique demonstrate the effectiveness of the global SIMD-lane wise optimization.

4.5 Related Work

Most prior work on automatic vectorization is performed on the loop level [Kennedy and Allen, 2002] [Nuzman and Zaks, 2008] [Nuzman et al., 2011] [Kim and Han, 2012], the basic block level [Larsen and Amarasinghe, 2000] [Liu et al., 2012], and the whole function level [Karrenberg and Hack, 2011]. Some of these vectorizing techniques are adopted in both commercial and open-source compilers such as Intel Compiler, Open64 [Dibyendu Das, 2012], GCC, LLVM. There is also extensive work on automatic vectorization with polyhedral model [Trifunovic et al., 2009]. Our hyper loop parallelism (HLP) vectorization resembles the classic loop vectorization by taking advantage of the mapping between loop iterations and SIMD lanes.

Super-word level parallelism (SLP) [Larsen and Amarasinghe, 2000] vectorization is the closest related work but it cannot handle complex computation patterns, such as intra-loop reduction. Although the variant of SLP in GCC handles intra-loop reduction, it may incur redundant computations similar to the one in Fig. 4-7. Besides, the implementation of SLP in GCC [Ira Rosen and Zaks, 2007] is limited to only the cases where the number of operations for packing is power-of-two. Park et al. introduces vectorization based on sub-graph level parallelism (SGLP), a coarser level of vectorization within basic blocks [Park et al., 2012]. Our proposed HLP is similar to the SGLP, but we consider HLP as a complement to classic loop parallelism. Besides, SGLP tries to identify opportunities for vectorization within the already vectorized basic blocks, while our work focuses on vectorization of non-vectorized code. The most significant difference between HLP and SGLP is that when mapping the SIMD parallelism to the target architecture, our method takes into account the instructions that flexibly control the SIMD lanes.

An integrated SIMDization framework is put forward by Wu et al. to address several orthogonal aspects of SIMDization, including SIMD parallelism extraction from different program scopes (from basic blocks to inner loops) [Wu et al., 2005]. Our HLP vectorization achieves the same goal of the basic block aggregation in this work. Furthermore, our vectorization transformation and code generation is similar to the length de-virtualization in [Wu et al., 2005] which also works on virtual vector registers.

General code generation for interleaved data accesses with strides of power-of-two is presented in [Nuzman et al., 2006] and implemented in GCC. This approach achieves portability but does not always gives the optimal code for a specific target architecture. Ren et al. work on optimizing data permutations on vectorized code [Ren et al., 2006]. Instead of general data permutation optimization, our approach directly generates well-known optimal code for a specific case of interleaved data access in order to achieve high performance.

4.6 Summary

In this chapter, we put forward a vectorizing technique based on the hyper loop parallelism, which is revealed by the hyper loops. The hyper loops recover the loop structures

of the vectorizable loop and help vectorization to employ global SIMD lane-wise optimization. We implemented our vectorizing technique in the Cetus source-to-source compiler to generate C code with SIMD intrinsics. The preliminary experimental results show that our vectorizing technique can achieve significant speedups over the non-vectorized code in our test cases.

In the next chapter, we are going to demonstrate how our proposed hyper loop parallelism based vectorizing compilation technique can be used to improve the memory performance of applications on CUDA GPUs.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Boost Memory Performance with HLP in Vectorization for CUDA GPUs

5.1 Introduction

General Purpose Graphics Processing Units (GPGPUs), in particular, Nvidia CUDA GPUs, are widely used in a variety of machines from super-computers to embedded systems (e.g., Nvidia Tegra K1). The deep hierarchy of both execution model and memory organization makes manually writing high performance code for the CUDA GPU error-prone and tedious. To reduce programming efforts, GPU manufacturers have put forward low level programming models such as CUDA, OpenCL. Meanwhile, a long standing research goal has been to automatically generate GPGPU code from either auto-parallelization, compiler directive based languages (e.g. OpenMP [OpenMP, 2013], OpenACC [OpenACC, 2011]) or domain-specific languages (e.g. Halide for image processing [Ragan-Kelley et al., 2013]).

For parallel loops with data in an array of structures (AoS), directly mapping each loop iteration to a GPU thread may expose non-unit stride data access. When the data access pattern has unit stride, global memory accesses can be easily coalesced. However, non-unit stride accesses have a great impact on the effective memory bandwidth. As shown in Fig. 5-1, all strides except the unit greatly decrease the effective memory bandwidth. Therefore, optimizing non-unit stride memory access is of great importance to the performance of CUDA programs. One popular solution is to apply global data layout

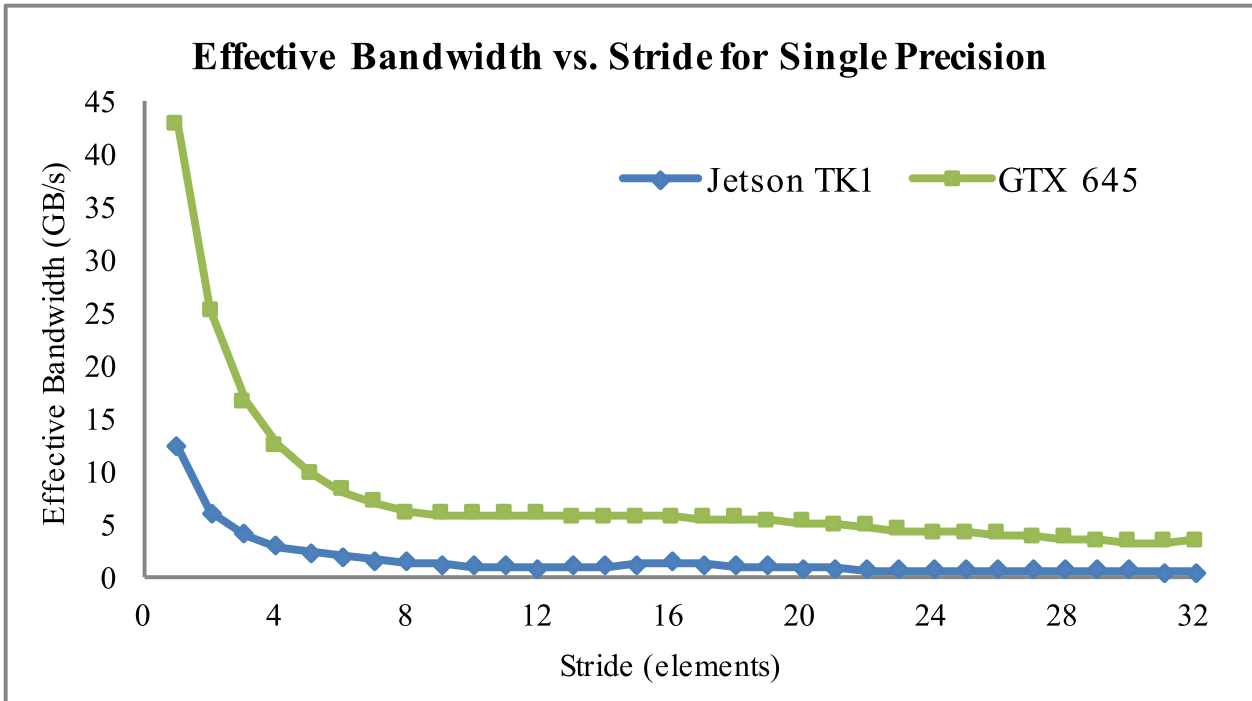


Figure 5-1: The performance impact of stride memory access on effective memory bandwidth.

transformation to transform the data layout from an array of structures to a structure of arrays, resulting in unit-stride data access. However, automatic global data layout is not trivial due the complexity of whole program analysis as discussed in Chapter 3 [Xu and Gregg, 2014b].

Non-unit stride data access is also an obstacle for efficient vectorization on CPUs. Apart from optimization with data reorganization instructions, super-word level parallelism (SLP) vectorization [Larsen and Amarasinghe, 2000] [Liu et al., 2012] is often used as an effective way to vectorize loops with data in AoS. However, to the best of our knowledge, dealing with non-unit stride memory access in C-to-CUDA with SLP vectorization has not been explored in either the research literature or in existing compilers. In this chapter, we propose an improved SLP vectorization technique based on *hyper loop parallelism* to deal with the non-unit stride data access in C-to-CUDA. Hyper loop parallelism (HLP) vectorization is used to deal with semi-isomorphic sub-graphs of data flow graphs of a vectorizable loop and has proven effective on the CPU in Chapter 4 [Xu and Gregg, 2014a]. In addition to optimizing non-unit stride access, HLP vectorization can achieve thread coarsening, which is an important technique to reduce memory

operations to global memory by exploiting data locality. In this chapter, we argue that efficient exploitation of hyper loop parallelism in vectorization can significantly improve the memory performance on the CUDA GPU. Moreover, the abstraction of hyper loops gives a way to further refine the mapping granularity between loop iterations and GPU threads.

In this chapter, we put forward a compiler framework to efficiently exploit hyper loop parallelism in vectorization to improve the memory performance on Nvidia CUDA GPUs. Our approach consists of identification of hyper loop parallelism and efficient mapping to the GPU. We present a scheme to map conventional SIMD operations to the GPU. Based on this mapping scheme, we introduce a code generation technique to generate efficient CUDA code. Moreover, we examined three mapping strategies for offloading vectorized code to the GPU. We implemented our technique on top of the Cetus source-to-source compiler [Bae et al., 2013] [Lee and Eigenmann, 2010], which already contains a basic code generation framework from C to CUDA. Though our work is based on C to CUDA compilation, the core techniques are not specific to any programming models. Thus, our compilation technique can be used in other tools or programming models to complement the existing mapping method. The experimental results demonstrate that our vectorization technique based on hyper loop parallelism can yield performance speedups up to 2.5 $\times$ compared to direct coarse-grain loop parallelism mapping.

5.2 Hyper Loop Parallelism in Vectorization

In this section, we first give a brief description of our proposed vectorization technique based on hyper loop parallelism (HLP) for CPUs in Chapter 4, and then demonstrate how HLP vectorization can be used to expose more coalesced memory accesses for CUDA GPUs.

Same as Chapter 4, we take an important kernel, C-saxpy, in scientific computing as the motivating example to present our proposed concept of hyper loop parallelism [Xu and Gregg, 2014a] and how it helps tackle the memory performance issue due to the AoS data layout for CUDA GPU. C-saxpy multiplies a complex vector by a constant complex vector and adds it to another complex vector, as presented in Fig. 5-2. As the data is in AoS layout, if the parallel loop in the kernel is directly mapped as a CUDA kernel, the data layout leads to non-unit stride of 2 memory accesses to global memory as shown in Fig. 5-3. The non-unit stride data access will have a great negative impact on the effective memory bandwidth as depicted in Fig. 5-1.

```
1  float y[SIZE * 2], x[SIZE * 2], C[SIZE * 2];
2  for (int i = 0; i < SIZE; i++) {
3    y[2*i]  += x[2*i] * C[2*i] - x[2*i+1] * C[2*i+1];
4    y[2*i+1] += x[2*i] * C[2*i+1] + x[2*i+1] * C[2*i];
5  }
```

Figure 5-2: C-Saxpy on the data organized in an array of structures (AoS).

```
1  float x[SIZE * 2], y[SIZE * 2], c[SIZE * 2];
2  __global__ void csaxpy_kernel0(float * c, float * x, float * y)
3  {
4    int i;
5    int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
6    int _gtid = (threadIdx.x+(_bid*blockDim.x));
7
8    if (i< SIZE) {
9      y[((2*i)+0)]=(y[((2*i)+0)]+((x[((2*i)+0)]*c[((2*i)+0)])
10                  -(x[((2*i)+1)]*c[((2*i)+1)])));
11     y[((2*i)+1)]=(y[((2*i)+1)]+((x[((2*i)+0)]*c[((2*i)+1)])
12                  +(x[((2*i)+1)]*c[((2*i)+0)])));
13    }
14 }
```

Figure 5-3: CUDA code for C-Saxpy generated by the Cetus compiler.

5.2.1 Hyper Loop Parallelism

The loop body of a vectorizable loop generally can be partitioned according to the downwards-exposed definitions. With the downwards-exposed definitions as the slicing criteria, we can use backward program slicing to derive a set of program slices. Each slice represents a subset of statements of the loop. Without considering the control dependence, a program slice within a loop body is essentially a sub-graph of the data dependence graph of the loop body. As each slice is collected within the loop body, a slice is a direct acyclic graph (DAG) $G(V, E)$, where V is a set of nodes representing the computation expressions within the slice, and E are the define-use relationships between the nodes in V.

For the motivating example, as shown in Fig. 5-4, there are two slices after program slicing. Without considering the relationships between the slices, we can treat each slice as a loop with only one iteration. However, there exist specific relationships between the slices. For C-saxpy, as the data is in AoS layout, each slice is for the computations with respect to a structure component. Therefore, the relationships between the structure components (aka. contiguous memory access) build the relationships among slices.

The relationships between slices (aka. contiguous downwards-exposed definitions) can be is used to group slices into grouped slices, or grouped DAGs. We define a slice group to be a *hyper loop*, where the number of hyper loop iterations is the same as the number of slices in the group. As each slice is an independent partition of the loop body, hyper loops are all parallel and potentially eligible for vectorization. We define the parallelism exposed by the hyper loops is *hyper loop parallelism*.

5.2.2 Hyper Loop Parallelism in Vectorization

Similar to other vectorization frameworks, hyper loop parallelism (HLP) vectorization consists of two phases, vectorization analysis and transformation.

Vectorization Analysis

Before HLP vectorization analysis, data dependence analysis is first applied to analyze whether a given loop is vectorizable or not. Data-flow analysis is also used to collect the downwards exposed definitions which are classified into *ordinary definition*s and *reduction definition*s. The HLP vectorization analysis contains two steps, collect slices and group slices. The results of these two steps are shown in Fig. 5-4 and Fig. 5-5, respectively.

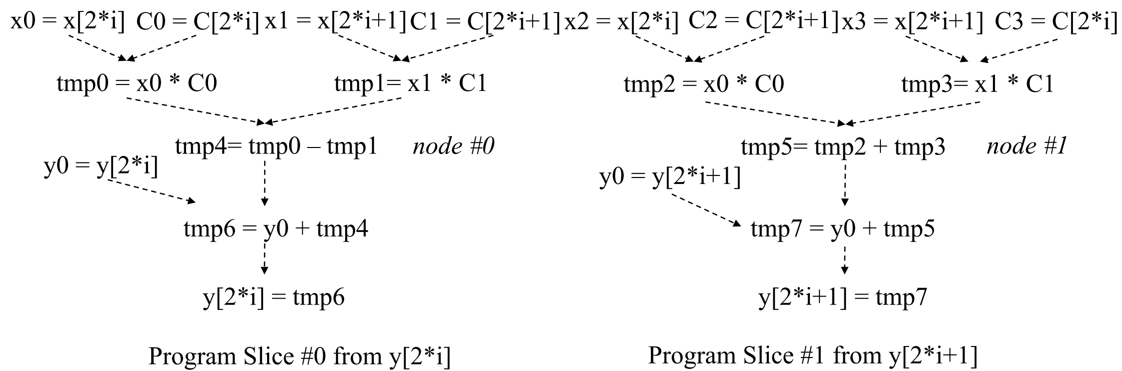


Figure 5-4: Identification of parallel hyper loop iterations with backward program slicing. `y[2*i]` and `y[2*i+1]` are the slicing criteria. This is the same as HLP vectorization for CPUs shown in Fig. 4-5.

Slice grouping works similar to the super-word level parallelism (SLP) vectorization that tries to pack isomorphic instructions into groups for vectorization [Larsen and Amarasinghe, 2000]. In contrast to the SLP vectorization, the grouping of slices starts from contiguous memory stores that are the downwards exposed definitions, and packs isomorphic operations from different slices. Either fully grouping or partially grouping is applied according to whether the computation structures of all the slices are the same or not. For the motivating example, because the computations at the node #0 and node #1 in Fig. 5-4 are not isomorphic, only partially grouping is employed. As illustrated in Fig. 5-5, two kinds of actions *extract* and *merge* are annotated on some edges to depict how the data flows between the connected nodes.

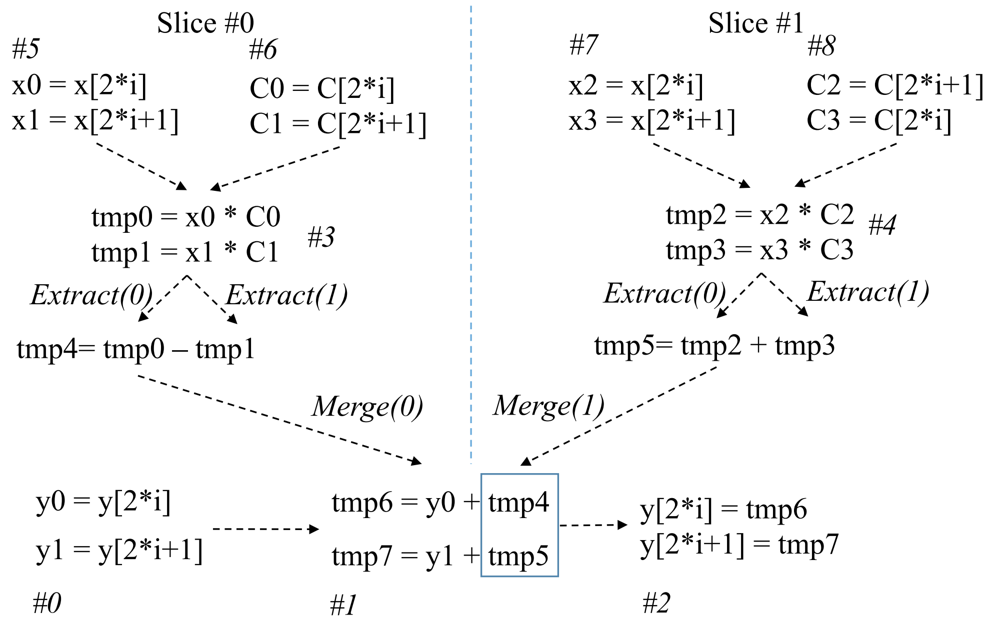


Figure 5-5: Group slices in hyper loop parallelism vectorization. This is the same as HLP vectorization for CPUs shown in Fig. 4-6.

Vectorization Transformation

HLP vectorization transformation includes two phases before generating CUDA code, expand grouped slices and global SIMD lane optimization.

Expand Grouped Slices Each grouped DAG is transformed into a vectorized DAG with vector operations on virtual vectors with respect to a loop unrolling factor. The width of the virtual vector of each node is decided by multiplying the loop unrolling factor and the size of the node. For the CUDA GPU, we choose the warp size as the width of the physical vector. In the expansion, *SIMD lane descriptor*s are annotated to describe the patterns of SIMD lanes for each node. SIMD lane descriptors are in the format of `id[start_position: size: stride]`, where `id` is the name of an array, a pointer or a virtual vector, `size` is the number of lanes, `stride` is the lane pattern. For the grouped DAG in Fig. 5-5, the vectorized DAG after expansion is shown in Fig. 5-6.

Global SIMD Lane Optimization The global SIMD lane-wise optimization tries to optimize the allocation of SIMD lanes according to the changes of SIMD lanes between nodes by inserting new nodes corresponding to any of the four SIMD lanes operations — `pack`, `unpack`, `merge` and `permute`. The `pack` and `unpack` operations deal with the changes of the vector size. The `merge` operation performs blending of two vectors with

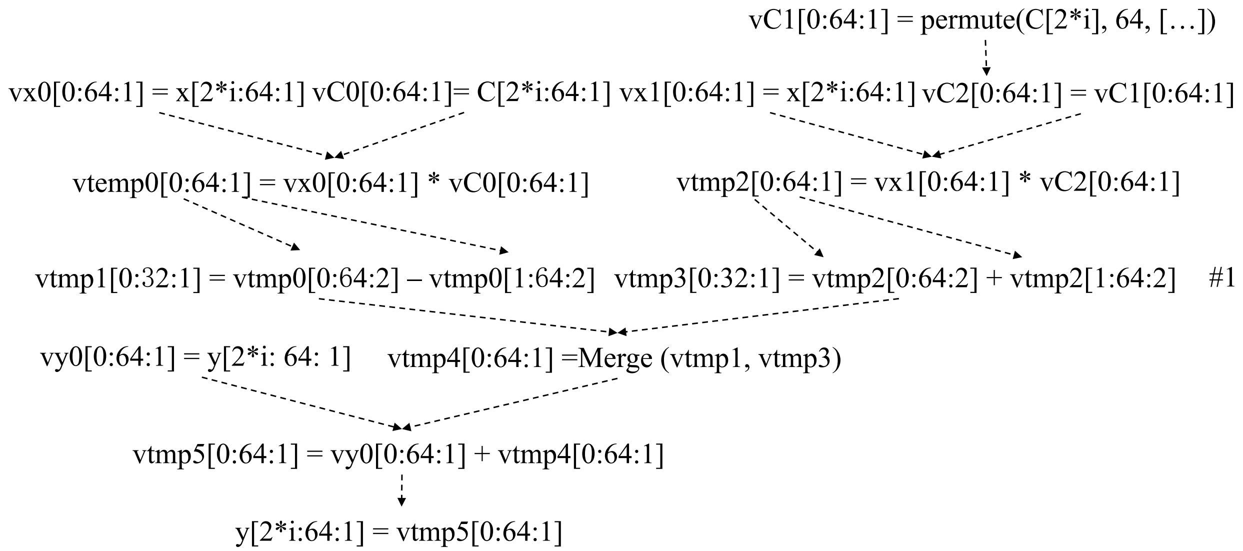


Figure 5-6: Illustration of vectorization expansion with a loop unrolling factor 32. The CUDA warp size is 32. Note the loop unrolling factor is different from the one used in HLP vectorization for CPUs depicted in Fig. 4-9.

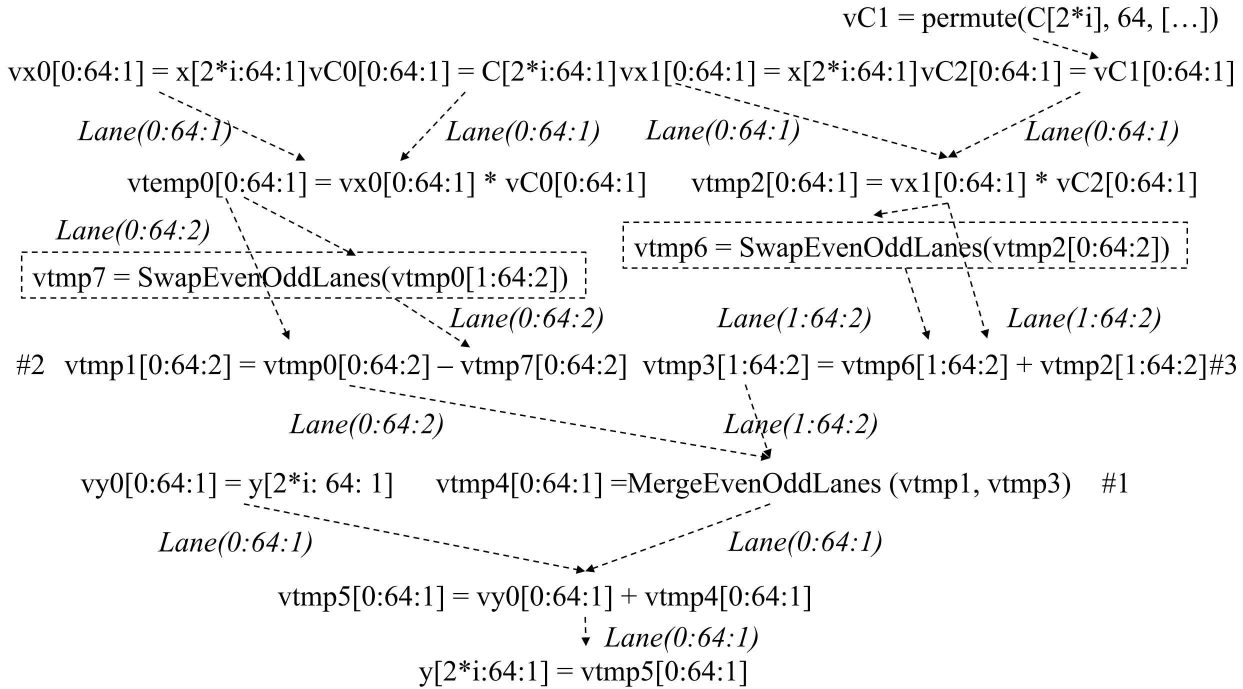


Figure 5-7: Global SIMD lane optimization. Note the length of SIMD vectors is different from the one used in HLP vectorization for CPUs depicted in Fig. 4-10.

the given SIMD lane information. The `permute` operation handles the changes of ordering of SIMD lanes between two vectors in the same size. The operations `SwapEvenOddLanes` and `MergeEvenOddLanes` in Fig. 5-7 are concrete instances of the `permute` and `merge`, respectively.

The global SIMD lane optimization consists of two passes, a top-down pass and a bottom-up pass on the expanded DAGs. The top-down pass tries to adjust the widths of virtual vectors and SIMD lane patterns according the memory loads in the leaf nodes. For example, the node #1 in Fig. 5-6 has a destination vector `vtmp3` with the SIMD lane pattern of `[0:32:1]`. The top-down pass changes the SIMD lane pattern into `[0:64:2]` according to the operand `vtmp2[0:64:2]` because both operands have strided SIMD lane patterns.

On the other hand, the bottom-up pass propagates the SIMD lane information of the root nodes to the leaf nodes and inserts the four SIMD lane operations accordingly. The bottom-up pass, in particular, takes care of the join nodes represented by `Merge`. For instance, after the top-down pass, the destination vectors `vtmp1` and `vtmp3` have the same SIMD lane pattern of `[0:64:2]`. When comes to the merge node #1 in Fig. 5-7, according to the relationships between hyper loop iterations and SIMD lanes, the optimization will assign the even lanes to the `vtmp1` while giving odd lanes to the `vtmp3`. Thus, the desirable SIMD lane pattern `[0:64:2]` and `[1:64:2]` are propagated to the node #2 and node #3, respectively. Guided by the desirable SIMD lane patterns, a `SwapEvenOddLanes` operation is introduced to transform the SIMD lane pattern of `vtmp2` from `[0:64:2]` to `[1:64:2]` as the node #3.

5.3 Hyper Loop Parallelism on the CUDA GPU

In this section, we discuss how to map hyper loop parallelism in the form of SIMD vectors and operations onto the CUDA GPU.

5.3.1 SIMD Vectors

Virtual SIMD vectors via thread local variables

It is intuitive that a thread local variable in a group can present a virtual SIMD vector. According to the CUDA execution model, all GPU threads are divided into blocks, and the threads in a block are executed in batch, that is, a warp with 16 or 32 GPU threads. In this case, we have two options to decide the size of SIMD vectors, either the size of a thread block or the size of a warp. Choosing the size of a warp as the vector size is superior to the size of a thread block for two reasons:

1. threads in a warp can executed in lock-step without any explicit synchronization;

2. CUDA devices with computation capability of at least version 3.X support intrinsics to perform data shuffling operations across the threads within in a warp.

In addition to the basic scalar types, CUDA has explicit SIMD vector types, for example, `float2`, `float4`. These data types can help improve the memory efficiency by using instructions such as `LD.64` and `LD.128` loading 8 and 16 bytes of data, respectively. When using thread local variables in vector types, the size of the virtual SIMD vector increases by a factor of the size of the vector type.

Explicit SIMD vectors via shared memory

In order to support both intra-vector and inter-vector SIMD operations across SIMD lanes on the GPU, we also need explicit SIMD vectors in memory. Without shuffle intrinsics, the only way to communicate across GPU threads is through the memory. According to the CUDA memory hierarchy, it is beneficial to represent explicit SIMD vectors with shared memory.

Explicit SIMD vectors via shared memory can be used to form super SIMD vectors with the sizes that are multiples of a SIMD vector size. For example, if three explicit

SIMD vectors are allocated in shared memory contiguously, we can treat these three SIMD vectors as a super SIMD vector with the same starting address as the first SIMD vector. Super SIMD vector plays an important role in optimizing non-unit stride memory access and exploiting data locality.

5.3.2 SIMD Operations

Arithmetic Operations

CUDA devices bundle several threads for execution. Each thread block is partitioned into warps. The execution of warps is implemented by SIMD hardware. The execution core of the processing units not only can perform 32-bit integer and single- and double-precision floating-point arithmetic operations, but also has special function units (SFUs) to compute single-precision approximations of log/exp, sin/cos, and rcp/rsqrt. It makes sense to implement SIMD arithmetic operations performing on the SIMD vectors organized in warps. Due to the lack of SIMD execution units in each streaming core, each SIMD operation on the data in SIMD types is decomposed into a sequence of scalar operations. Therefore, the granularity of a CUDA thread is coarsened by the vectorization when using SIMD data types.

Memory Operations

In this chapter, we only consider unit stride and non-unit stride memory operations.

Unit stride loads/store Each SIMD lane of a SIMD vector is mapped to a thread in a warp. Thus, for a unit stride vector load/store, if it is aligned to 128-bytes, the memory access to the global memory would be coalesced without extra overhead.

Non-unit stride load/store As shown in Fig. 5-1, non-unit stride loads and stores lead to ineffective use of global memory bandwidth. Unlike CPUs, which support many different data permutation instructions, GPUs have quite a limited data permutation capability across GPU threads. Data permutation operations on the GPU usually have to resort to other types of memory, such as shared memory and global memory, to exchange data across threads with necessary synchronization.

In order to optimize non-unit stride loads and stores on GPU, we employ on-the-fly

data layout transformation with the virtual SIMD vectors in shared memory. Our proposed on-the-fly data layout transformation works by first collecting a group of strided loads or stores with spatial locality. For the non-unit stride loads, a sequence of unit-stride vector loads with contiguous virtual SIMD vectors are generated to ensure all the data to be accessed by each stride load in the group is cached in shared memory in the form of virtual SIMD vectors. Then, the contiguous virtual SIMD vectors are converted into a super SIMD vector. Finally, each original stride memory access to the global memory is transformed into a stride access to the super SIMD vector. The transformation works similarly for the non-unit stride stores.

```
1  for (i=0; i<NUMPOINTS; i++){
2    results[i] = points1[i][0] * points2[i][0] +
3                 points1[i][1] * points2[i][1] +
4                 points1[i][2] * points2[i][2];
5  }
```

Figure 5-8: Vector dot operation.

For instance, the vector dot in Fig. 5-8 exhibits stride-3 data access. The result of on-the-fly data reorganization through shared memory is presented in Fig. 5-9. Three stride accesses to `points1` in the global memory are transformed into three coalesced accesses to global memory (line 3, 4, 5 in Fig. 5-9) and six stride accesses to the shared memory (line 3, 5, 7, 12, 14, 16 in Fig. 5-9). Because the access stride 3 has no common factor with the number of banks of shared memory (in this chapter, 32), there is no performance degradation due to the bank conflicts. Another potential optimization technique to further utilize the global memory bandwidth is to combine the first two coalesced accesses into one access with the built-in vector type `float2`. However, the potential performance gain by using vector types is rarely seen in experiments.

Data Reorganization Operations

In this section, we describe how to implement three widely used data reorganization operations, intra-vector shuffle, inter-vector permutation and blend.

Intra-vector Shuffle The intra-vector shuffle operation takes a SIMD vector and a mask vector as the input, and shuffles the data in the SIMD vector according to the mask vector. There are two ways to implement the intra-vector shuffle operation on the GPU:

```
1  volatile __shared__ float vregs_0[REG_NUM * WARP_NUM][32];
2  __simd_vptr_points1_0=((((float *)( & points1[0][0]))+(_bid*blockDim.x))+(_warp_id*(3*32)));
3  vregs_0[((_warp_id*3)+0)][_lane_id]= __simd_vptr_points1_0[((0*32)+_lane_id)];
4  vregs_0[((_warp_id*3)+1)][_lane_id]= __simd_vptr_points1_0[((1*32)+_lane_id)];
5  vregs_0[((_warp_id*3)+2)][_lane_id]= __simd_vptr_points1_0[((2*32)+_lane_id)];
6  vregs_0_ptr_0=( & vregs_0[((_warp_id*3)+0)][0]); /* Treat 3 vregs as a super-vreg. */
7  vregs_0_elem_0=vregs_0_ptr_0[((_lane_id*3)+0)]; /* Extract the vector element */
8  vregs_0_elem_1=vregs_0_ptr_0[((_lane_id*3)+1)]; /* Extract the vector element */
9  vregs_0_elem_2=vregs_0_ptr_0[((_lane_id*3)+2)]; /* Extract the vector element */
10  ...
```

Figure 5-9: Stride-3 data layout transformation via shared memory.

a software approach and a hardware approach.

A mask vector is required by the shuffle operation; thus, the first problem is how to generate and represent the mask vector. There are two options to generate mask vectors. For simple cases, such as the `SwapEvenOddLanes` operation, we can adopt runtime generation at cost of control divergence due to the introduced if statements, as shown in line 4 - 9 of Fig. 5-10. However, sometimes these if statements can be optimized by predicated instructions. On the other hand, we can generate shuffle masks at compile-time and store the masks in global memory. In this case, more complex data shuffling patterns are possible because arbitrary shuffling masks can be generated and accessed without any extra costs.

In order to ensure high performance, immediate results of computations are often kept in thread-local variables rather than shared memory. When performing an intra-vector shuffle operation, all the values participating in the shuffle operation need to be copied into an explicit virtual SIMD register in shared memory. With the data in the virtual SIMD register, the data shuffling operation across threads within a warp is equivalent to random data access to shared memory.

The disadvantage of the earlier software approach through an explicit virtual SIMD register in shared memory is the cost of extra memory accesses to the shared memory. For devices with compute capability of at least 3.0, CUDA introduces a set of `__shfl()` intrinsics to permit exchanging of variables between threads within a warp without use of shared memory [NVIDIA, 2014]. The exchange occurs simultaneously for all active threads within the warp, moving 4 bytes of data per thread.

There are four shuffle instructions available for different purposes - `__shfl()`, `__shfl_up()`, `__shfl_down()` and `__shfl_xor()`. For the `SwapEvenOddLanes` operation,

```
const int idx = threadIdx.x; // local thread id
const unsigned int lane = idx % 32; // lane id
const unsigned int warp_id = idx / 32; // warp id
if (((_lane_id%2)==0)){ // compute the shuffle masks
  perm_mask_0=(_lane_id+1);
}
else{
  perm_mask_0=(_lane_id-1);
}
// intra-vector shuffle
__simd_swap_even_odd_lanes_0=vregs_0[_warp_id][perm_mask_0];
```

Figure 5-10: Runtime generation of masks for intra-vector shuffle operations.

we can use the `__shfl_xor(input, 0x1, 2)`, which partitions the SIMD lanes into subgroups with the size of 2 and gives the source lane ID by a bitwise XOR.

Inter-vector Permutation The Inter-vector permutation operation usually requires two vectors as the source vectors and a mask vector to specify the permutation pattern. Our solution to the inter-vector permutation operation is similar to the software approach of the intra-vector shuffle operation, which is achieved via shared memory. For inter-vector permutation, we combine all the participating explicit virtual SIMD registers into a super-vector, and then replace the inter-vector permutation operation with an arbitrary memory access to the super-vector according the permutation mask computed either at runtime or at compile-time and stored in global memory.

Blend The blend operation takes two vectors and a mask vector as the input, and selects a value for each element of the destination vector from either input vector according to the condition indicated by the mask vector. A simple solution to convert a blend operation into a conditional assignment statement, which would be optimized by predicated instructions. Another solution is to convert the two input vectors into a super-vector in shared memory similar to the inter-vector permutation operation, and replace the blend operation with a memory gather operation from the super-vector.

5.3.3 Mapping Execution Model

In this section, we present three strategies of mapping SIMD lanes to GPU threads, *direct mapping*, *flatten mapping* and *nested SIMD mapping*, as shown in Fig. 5-11.

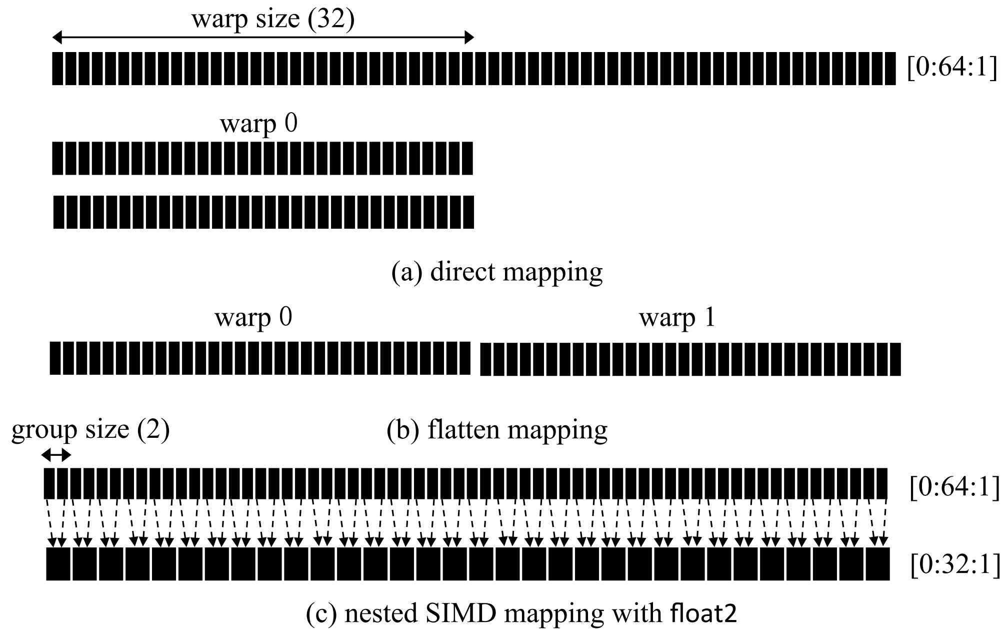


Figure 5-11: Mapping SIMD lanes representing hyper loop parallelism to GPU threads.

Direct Mapping

When a hyper loop is identified and the hyper loop parallelism is expressed as SIMD operations on SIMD vectors, it is intuitive to directly map each SIMD operation to a GPU thread as discussed in Sec. 5.3.2. However, the width of SIMD lanes after our global SIMD lane optimization may differ from the width of SIMD units (the warp size). Given that we treat a warp of GPU threads as a SIMD computation unit, when mapping SIMD operations with larger width than the warp size, we can partition the SIMD operations into several SIMD operations with width equal to the warp size. We call this mapping strategy as *direct mapping*.

For instance, as shown in Fig. 5-11 (a), the SIMD operation to be mapped has a SIMD lane descriptor [0:64:1], indicating the width of this SIMD operation is twice larger than the warp size. Therefore, execution of this SIMD operation requires two warps of threads. In direct mapping, the SIMD operation is partitioned into two SIMD operations, and both of them are scheduled in the same warp and executed one after another.

Flatten Mapping

In contrast to direct mapping, another way of dealing with the SIMD operation with size larger than the warp size is to partition the SIMD operation into small SIMD operations

and spread the resulting SIMD operations into different warps of GPU threads. As shown in Fig. 5-11 (b), the SIMD operation with a SIMD lane descriptor [0:64:1] is split into two SIMD operations with the SIMD lane descriptor [0:32:1]. We call this mapping strategy *flatten mapping*.

It is not always possible to apply flatten mapping due to the across SIMD lane operation. We consider the flatten mapping to be legal only when the following two conditions are satisfied:

1. the number of SIMD lanes of every node after global SIMD optimization is the same, and this number is a multiple of the warp size;

2. all the data shuffling operations in the nodes are qualified to be split into several data shuffling operations, and these operations only perform data permutation on SIMD vectors of which the size is the same as the warp size.

The second condition is to ensure that no matter which kind of across SIMD lane operation it is, the GPU threads engaged are never distributed into two different warps. Because there is no efficient way to reorganize data across warps except through shared memory with explicit synchronization, as discussed in Sec. 5.3.2,

Nested SIMD Mapping

Both direct and flatten mapping may introduce data reorganization across GPU threads within a warp. Even with the CUDA 3.0 hardware support for data shuffling within a warp without explicit shared memory access, the cost may be prohibitive. Using hyper loops to represent the inner-loop computation structure, the data reorganization in fact occurs only among the hyper loop iterations or within a hyper loop iteration when nested SIMD parallelism is discovered. Therefore, one way to get rid of the data shuffling operations across threads within a warp is to enlarge the the mapping granularity from a hyper loop iteration to a hyper loop. As shown in Fig.5-11(c), each hyper loop instance with two hyper loop iterations is scheduled to a GPU thread. Because the main purpose of hyper loops is to expose an extra degree of parallelism for SIMD, we call this mapping strategy *nested SIMD mapping*.

Representing inner-loop computation structure as hyper loops could logically achieve AoS to SoA data layout transformation for the loops with data in AoS. When choosing the hyper loop as a schedule unit, it is of great importance to preserve the contiguous memory access exposed by hyper loops, because contiguous accesses to global memory could be coalesced. In order to achieve contiguous memory access within a schedule unit, we need to use vectorized loads and stores through the CUDA vector types.

5.4 Implementation

Our hyper loop parallelism vectorization for GPU is built on top of the Cetus source-to-source C compiler. The Cetus compiler is capable of auto-parallelization of C programs and generates C programs with OpenMP and Cetus directives [Lee and Eigenmann, 2010]. The overall compilation flow of hyper loop parallelism vectorization for the GPU is illustrated in Fig. 5-12.

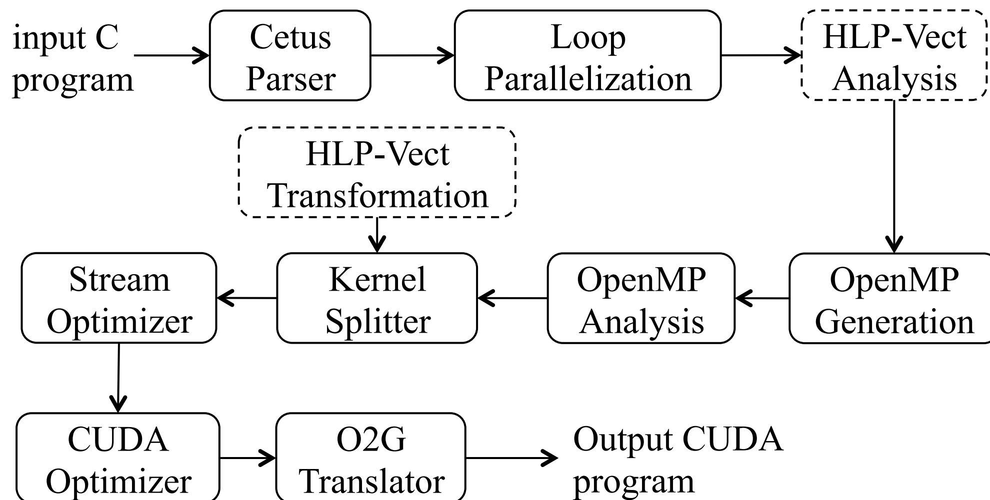


Figure 5-12: The overall compilation flow of hyper loop parallelism vectorization for the CUDA GPU. The compiler passes in the dotted boxes are what we introduce to the Cetus compiler.

Our hyper loop parallelism vectorization analysis follows directly after loop parallelization, which identifies the parallel loop nests and annotates the parallel loops with Cetus internal directives. These parallel loop nests are the candidates to become GPU kernels if the performance prediction model determines that annotation with OpenMP directives is likely to be profitable. The HLP vectorization transformation is applied dur-

ing the kernel splitter phase, where code generation for CUDA takes place. A kernel function call is generated and its execution configuration is decided as well. The execution configuration decides how many explicit virtual SIMD vectors via shared memory are required.

5.5 Performance Evaluation

5.5.1 Experiment Setup

Platforms

We evaluated our hyper loop parallelism vectorization for CUDA GPU on the following two platforms.

Jetson TK1 is a fully-featured embedded system designed for development of embedded and mobile applications. Jetson K1 incorporates a Tegra K1 that has a Kepler GPU with 192 CUDA cores, an Nvidia 4-plus-1 quad-core ARM Cortex-A15 CPU. The significant difference between Tegra K1 and other desktop GPUs is that the memory on Tegra K1 is physically unified but with separate CPU and GPU caches.

GeForce GTX 645 is a desktop GPU with the architecture of Kepler GK106. It supports computation capability 3.0 and has 576 CUDA cores and 1024MB GDDR5 memory.

5.5.2 Test-cases

We chose a range of parallel loops featuring both unit and non-unit strides as listed in Table 5.1 to demonstrate the performance of our proposed technique. For non-unit stride tests, we consider two representative strides: 2 and 3. For other strides, our technique can work similarly. .

5.5.3 Performance Evaluation and Analysis

We compare the performance of HLP vectorization for CUDA GPU against the direct coarse-grain loop parallelism mapping. When generating the CUDA code for a parallel loop nest, the most common mapping strategy adopted is to directly map a loop iteration

Table 5.1: Test-cases with three representative data access strides: 1, 2 and 3.

Stride	Test-case	Description
1	Array Copy 1D Dot Blur	Copy 1D array. Dot operation on 1D vectors. Blur in image processing on an 1D linearized array.
2	C-Saxpy	Complex vector operation.
3	Vec-Copy Vec-Dot Vec-Crossproduct Vec-Rotation Vec-Normalization	Basic operations on 3D vectors in AoS in image processing

to a CUDA thread. The baseline CUDA code for performance comparison is generated by OpenMPC [Lee and Eigenmann, 2010] with the direct coarse-grain loop parallelism mapping.

Unit Stride Tests

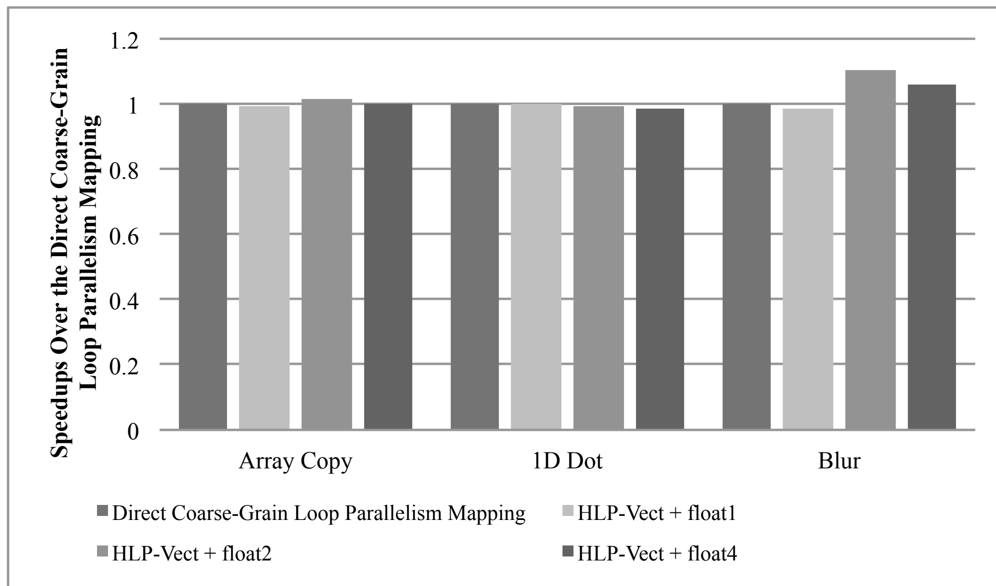


Figure 5-13: The performance of loops with unit-stride data access on Jetson TK1. The CUDA block size is 128.

Array copy is a trivial test-case to demonstrate whether our nested SIMD mapping could yield any performance improvement. As shown in Fig. 5-14, vectorized loads/s-

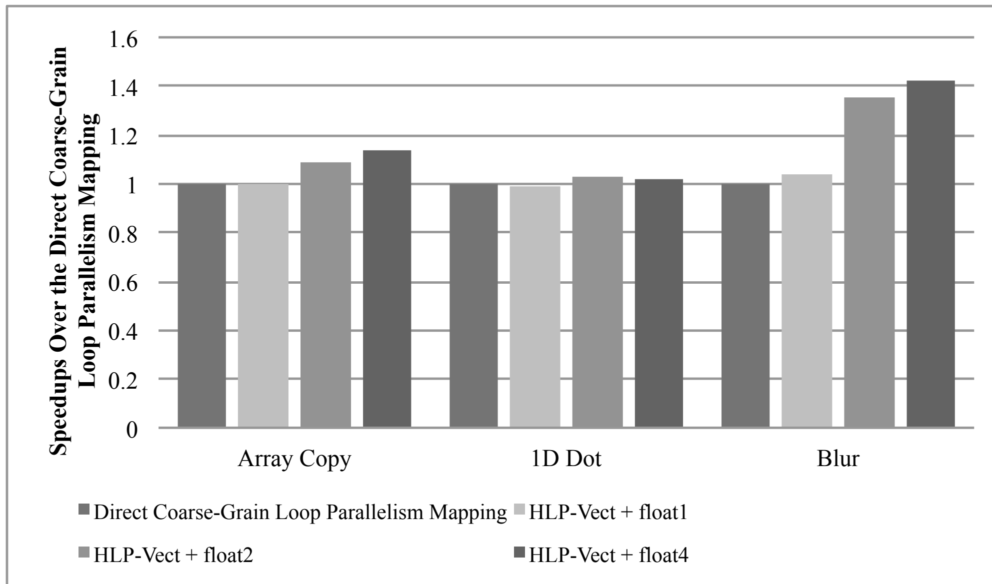


Figure 5-14: The performance of loops with unit-stride data access on GeForce GTX 645. The CUDA block size is 128.

tores using CUDA built-in vector type `float2/float4` would not give any performance gain on the Jetson TK1 platform while a speedup of up to 1.1 $\times$ on GeForce GTX 645. We suspect that the vectorized loads and stores are not well supported on the Jetson TK1 platform.

Although performing only vectorized loads and stores improves performance on GeForce GTX 645, when computation is involved, the performance gain is reduced by the long latency of the vectorized load instructions. For example, `LD.E.64` can load 2 floats into two registers, and all the instructions using these two registers explicitly have to wait until the load is finished. The 1D dot has no other computations to overlap the memory latency; thus, regardless of the kind of mapping, the performance stays the same.

Compared to 1D dot, which has no temporal locality, blur has lots because of the stencil computation. Temporal locality is often exploited to handle unaligned vectorized memory operations in vectorization for CPUs [Eichenberger et al., 2004]. Similarly, we could exploit temporal locality to optimize memory access to global memory. When the nested SIMD mapping is applied, a group of threads that share some data are squeezed into a single thread. If there is shared data between the threads being coarsened, coarsening them will reduce the number of memory operations to global memory. As shown in Fig. 5-14, nested SIMD mapping improves the performance of blur by 1.4$\times$.

Non-unit Stride Tests

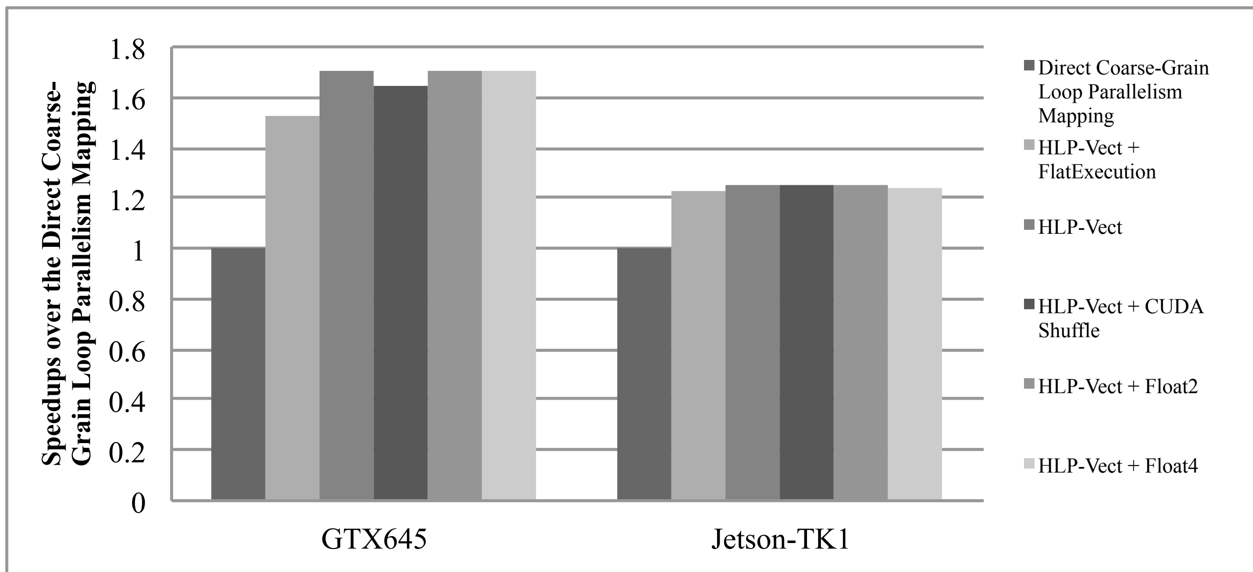


Figure 5-15: The performance of C-Saxpy on Jetson K1 and GeForce 645. The CUDA block size is 128.

Stride 2 - C-saxpy C-saxpy is a good candidate to evaluate different execution mapping strategies on the CUDA GPU. As shown in Fig. 5-15, all the mapping strategies proposed in this chapter yield better performance, when compared to the direct coarse-grain loop parallelism mapping, which is common in existing compilers with semi-/fully automatic code generation for CUDA. Speedups up to 1.7× are obtained from HLP vectorization. The major performance contribution comes from the global memory load efficiency that is boosted from 50% to 100%. The performance difference between the flatten mapping and direct mapping demonstrates that instruction level parallelism is important for performance as well. The flatten mapping splits the SIMD operations with width 64 into two short operations with width 32, and maps the resulting operations onto two warps. Consequently, the ILP in each warp is reduced.

We are surprised to see that the hardware data shuffle degrades the performance of HLP vectorization on GeForce GTX 645. Exchanging data via explicit shared memory causes two extra memory operations to shared memory, but the compiler could schedule these instructions and make them pipelined. On the other hand, the single shuffle instruction may be a barrier to instruction scheduling.

Similar to the 1D dot, nested SIMD mapping does not give extra performance gain for

the csaxpy over the direct mapping. The nested SIMD mapping with `float2` and `float4` coarsens the thread granularity of the direct mapping by a factor of 2 and 4, respectively. The thread coarsening reduces the overall number of threads. But the coalesced global memory access via vectorized loads and stores and the extra ILP by threading coarsening make the performance on a par with the direct mapping. However, casxpy is a memory bounded kernel so that the performance difference between `float2` and `float4` is slight.

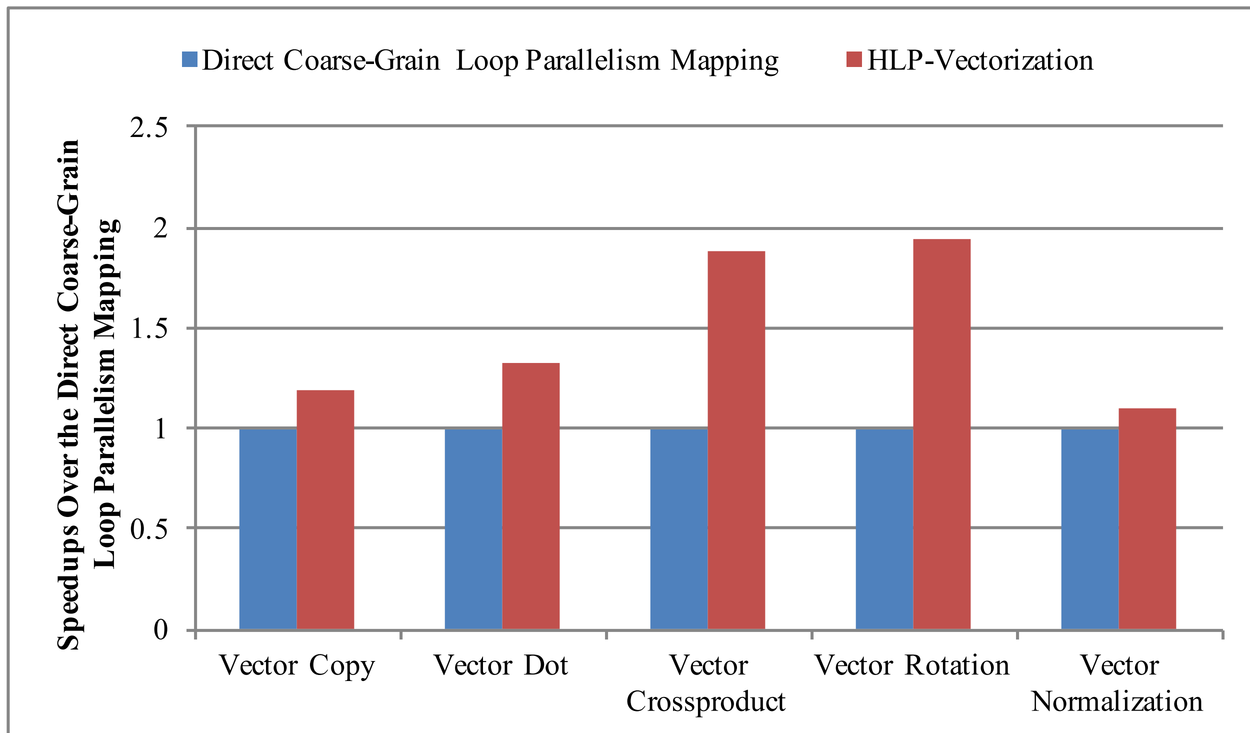


Figure 5-16: The performance of kernels with stride-3 data access on Jetson TK1. The CUDA block size is 128.

Stride 3 As shown in Fig. 5-16 and Fig. 5-17, HLP vectorization can achieve significant speedups for the stride-3 test-cases on both platforms. The profiling data obtained by the `nvprof` in the CUDA toolkit shows that the global memory load efficiency is improved to 100% from 33.3%. For `Vector Copy`, all the stride memory accesses to global memory from the direct mapping are coalesced thanks to the hyper loop structure. Other stride-3 test-cases require on-the-fly data layout transformation via shared memory for load and store operations. The great speedups achieved — up to a factor of 2.5 — demonstrate that HLP vectorization can be an effective way to improve the memory performance on the CUDA GPU.

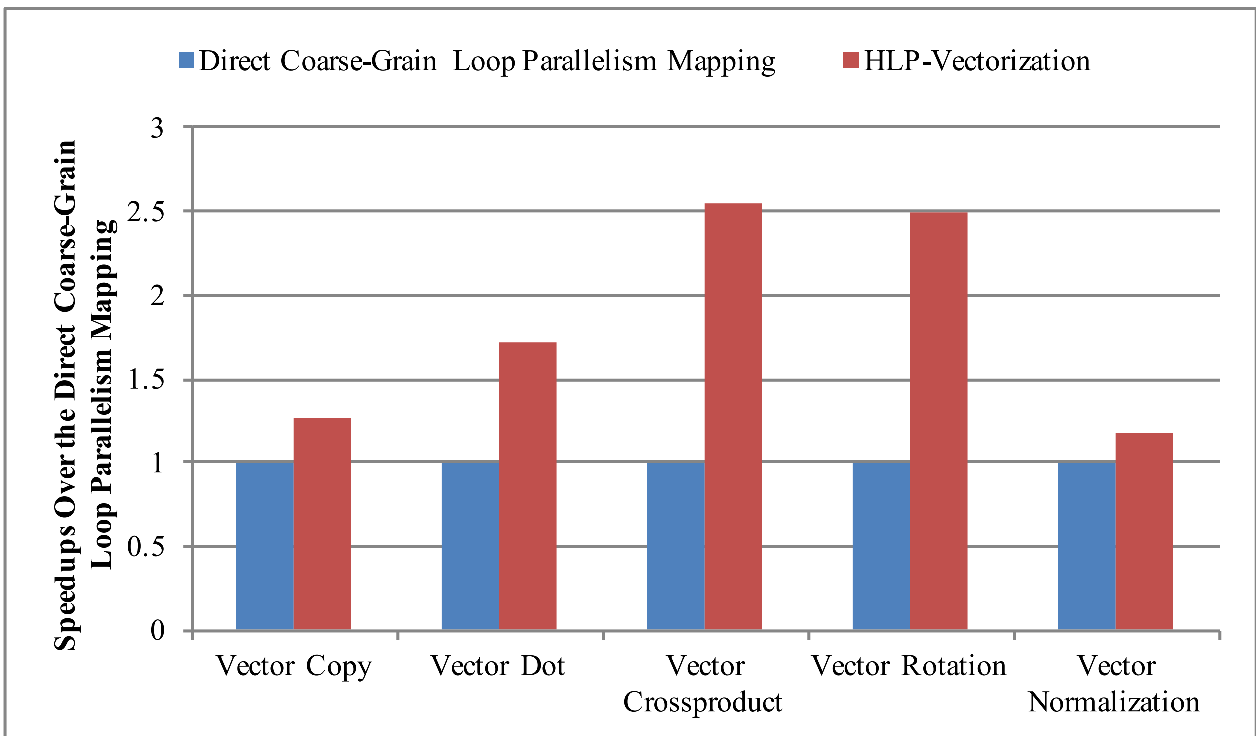


Figure 5-17: The performance of kernels with stride-3 data access on GeForce GTX 645. The CUDA block size is 128.

5.6 Related Work

There is extensive work on optimizing memory performance for the GPGPUs. For example, Jang et al. introduced vectorization via data transformation to benefit vector-based GPU architectures (e.g. AMD GPUs) and algorithmic memory selection for scalar-based GPU architectures (e.g. Nvidia GPUs) [Jang et al., 2011a]. Che et al. proposes a simple API to allow programmers to optimize memory mappings to improve the efficiency of memory accesses on heterogeneous platforms [Che et al., 2011]. In Chapter 3, we introduced a set of compiler directives to help the programmer to apply global data layout transformations for better vectorization. Instead of global data layout transformations, our proposed technique optimizes the memory performance by exploiting the computation structures of the vectorizable loops and applying on-the-fly data layout transformation.

Employing vectorization for the GPU is not a new idea. Kerr et al. put forward a dynamic compiler to compile explicitly data-parallel kernels for SIMD functional units [Kerr et al., 2012] . In contrast, our work focuses on how to exploit SIMD parallelism when

automatically extracting data-parallel kernels and addressing the problem of non-unit stride memory access with our vectorization technique.

Automatic code generation from high-level languages to CUDA has been studied since CUDA was put forward. One pioneering approach is hiCUDA [Han and Abdelrahman, 2009], a high-level directive-based language for CUDA. Other directive based programming method for heterogeneous devices have evolved into programming standards, such as OpenMP 4.0 [OpenMP, 2013] and OpenACC [OpenACC, 2011]. Since these standards have been established, many attempts have been made to explore and evaluate the ways of mapping coarse-grain loop parallelism to the heterogeneous devices, including both the commercial compilers, such as PGI OpenACC compiler, and open-source compilers such as OpenARC [Lee and Eigenmann, 2010], OpenUH-ACC [Tian et al., 2014].

However, existing compiler work for directive based programming methods concentrates only on how to map the execution model of the programming model to the devices. For example, Tian et al. demonstrated their way of mapping the gang, worker, vector execution model of OpenACC to the CUDA execution model [Tian et al., 2014]. Even with explicit annotations for SIMD parallelism, the compiler is not able to employ suitable vectorization and efficiently map the vectorized code to the heterogeneous devices. To the best of our knowledge, our work is the first work on employing a variant of superword level parallelism vectorization [Larsen and Amarasinghe, 2000] [Liu et al., 2012] [Tenllado et al., 2005], HLP vectorization, in automatic C-to-CUDA.

5.7 Summary

In this chapter, we put forward a compiler framework to extract hyper loop parallelism in vectorization and map the parallelism efficiently on the CUDA GPU. Our method achieves thread coarsening, which can reduce memory operations in the presence of data locality, and optimizes uncoalesced memory access to global memory. In addition, the introduction of hyper loop parallelism further refines the mapping granularity between coarse-grain loop parallelism and GPU threads. Our vectorization techniques are general, and could be be adopted in existing directive based programming models for GPUs to improve the memory performance. Our experimental evaluation demonstrates that our proposed approach can deliver speedups of up to 2.5× compared to the direct coarse-grain loop parallelism mapping.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Loop Vectorization for Nested Thread-level Parallelism on CUDA GPUs

6.1 Introduction

General-purpose graphics processing units (GPGPUs) have been widely adopted to accelerate data parallel applications in across a range of problems from embedded computing to supercomputing. GPUs are particularly effective for nested loops that operate on arrays of data. Indeed Nvidia CUDA model of parallelism is specifically aimed at nested loops which may contain massive data parallelism in the outer, middle, and/or inner loops. Such nested loops that expose thread-level parallelism at different levels of the loop nest are common in array-oriented applications. We refer to this kind of parallelism as *nested thread-level parallelism* (TLP).

Nested TLP revealed by nested parallel loops is pervasive in real applications. For example, 75% (14 out of 19) of the applications in the Rodinia benchmark for heterogeneous accelerators contain kernels with nested TLP. It is easy to spawn worker threads for nested TLP on CPU. In contrast, nested TLP contained in parallel programs puts an extra burden on the orchestration of GPU threads. As execution configuration is fixed when a GPU kernel starts, it is prohibitive to spawn new GPU threads for nested TLP despite the support for dynamic parallelism on the latest CUDA devices [Yang and Zhou,

2014].

Mapping nested TLP to the model of parallelism offered by the GPU is not always simple. CUDA GPUs offer a multi-level hierarchy of thread-level parallelism, which suits nested loops with simple patterns parallelism and data access very well. When converting a parallel loop nest to a GPU kernel, it is common practice to directly map the closely nested parallel loops to the GPU execution model. For Nvidia CUDA GPUs, GPU threads are partitioned into thread blocks, and threads within a thread block can be organized in up to three dimensions – x, y and z. Thread blocks make up a grid, and can also be managed in up to three dimensions. In many cases it is possible to simply map the parallel loop nest directly to this hierarchy of parallel thread blocks and threads, as shown in Fig. 6-1.

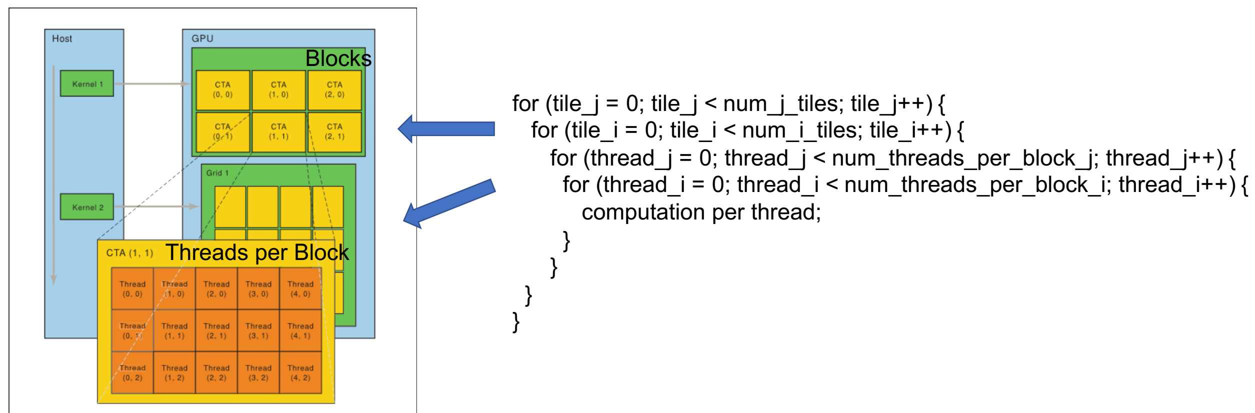


Figure 6-1: Naive mapping from parallel loop nest to the CUDA execution model.

However, there also exists a large class of nested parallel loops where it is less clear how the levels of the loop nest should be mapped to levels of the CUDA GPU parallelism hierarchy. For array-oriented nested loops, we consider three main categories of problems:

- Some loops in the nest may contain data dependences, which limit or prohibit parallelism. Thus, the loop nest may contain both parallel loops and loops than cannot be parallelized.

- Perfectly-nested loops consist of a simple hierarchy with exactly one loop nested inside the next. In contrast, imperfectly-nested loops may contain multiple loops within a single level of the loop hierarchy, or may contain a mixture of loops and

sequential statements at the same level of loop nest. Imperfectly-nested loops at the same level of the loop hierarchy may have quite different parallel processing requirements.

- Some loops within the nest may contain non-sequential or strided data access patterns. As we demonstrate in Fig. 5-1 in Chapter 5, GPUs benefit significantly from locality of data access. Even if the original loop nest maps perfectly to the GPU hierarchy of parallelism, the resulting data access patterns may exhibit poor locality.

Loop nests with these properties can be difficult to map to CUDA GPU's simple hierarchy of parallelism. As a result of these problems, researchers have developed techniques to break the link between the loop hierarchy and CUDA execution hierarchy. For example, Yang and Zhou proposed automatic compiler transformations that attempt to introduce a master-slave execution model for GPUs by restructuring the hierarchy of GPU threads [Yang and Zhou, 2014]. These techniques can achieve good speedups, particularly on loop nests with data dependences. However, in our own experiments we have found some shortcomings with these approaches, particularly for imperfectly nested loops, which we describe in more detail in Section 6.4.2.

In this chapter, we argue that the problem of mapping these difficult kinds of nested TLP to GPUs can be solved by adapting traditional loop vectorization techniques designed for CPUs with short vectors. We propose a novel model of nested vector parallelism, and show how it can be used to break the link between loop nest and the CUDA execution hierarchy. In addition, vectorization techniques are strongly-focused on contiguous data access patterns, which in turn improves memory performance. To this end, we first represent the nested TLP mapping problem as a loop vectorization problem. We then introduce a thread-reuse execution model to support nested TLP on CUDA GPUs. With this execution model, we present an automatic loop vectorization technique for nested TLP that generates CUDA code from C code with pragmas. We implemented our proposed vectorization approach for nested TLP in the Cetus source-to-source compiler and compared the performance against both Yang and Zhou's method and an industrial compiler.

6.2 Loop Vectorization for Nested TLP on GPUs

6.2.1 Motivation

When the execution configuration of GPU threads is given for a GPU kernel, regardless of the sources of loop parallelism exposed by the kernels, all the loop parallelism is eventually handled by the same set of threads spawned at the start of execution. *Mapping the nested TLP contained in parallel loops to GPU threads is a problem of finding a set of threads and deriving an efficient mapping strategy for nested TLP*. In this chapter, we mainly focus on nested TLP exposed by nested parallel loops. Other forms of nested TLP such as coarse-grain task parallelism are beyond the scope of our work.

```
1  #pragma acc parallel loop private(i, j)
2  for (i=0; i<SIZE; i++){
3    float sum = 0.0f;
4    #pragma acc loop vector(32, full, outer-vect) reduction(+:sum)
5    for (j = 0; j < SIZE; j++){
6      sum += M[j][i] * V[j];
7    }
8    C[i] = sum;
9  }
```

Figure 6-2: The parallel loop annotated with OpenACC pragmas for the transposed matrix vector multiplication (TMV). Note that the `vector` clause in the example is our language extension for nested TLP (discussed in Sec. 6.3.1).

An efficient mapping strategy for nested TLP needs take the GPU memory hierarchy into account. On CUDA GPUs, contiguous memory access to global memory can be coalesced which in turn greatly improves performance. Therefore, we need make a decision on which level of loop should be mapped to the consecutive GPU threads for the sake of contiguous memory access. For example, transposed matrix vector multiplication in Fig. 6-2 has an imperfectly nested loop. The outer `i`-loop is embarrassingly parallel while the inner `j`-loop is parallel with a reduction of addition on the scalar `sum`. The mapping strategy depends on whether we would like to make the access to array `M` and `C` contiguous, or array `V` across consecutive GPU threads.

As discussed in Section 2.3 in Chapter 2, the SIMT execution model of GPUs is a more flexible form of SIMD on CPUs. Therefore, mapping each iteration of a single-level parallel loop representing a kernel to a GPU thread is equivalent to vectorizing the parallel

loop for an architecture with vector size equal to the number of GPU threads required by the kernel. This simple SIMD vector abstraction of the GPU execution model suits vectorized loops from outer-loop vectorization for a loop nest and inner-loop vectorization on a single-level parallel loop. As each GPU thread has its own data context for execution, explicit SIMD vector registers are not required.

Note that CUDA supports vector types, such as `float2, float4`. Arithmetic operations on these vector types are decomposed into a sequence of scalar operations due to the lack of SIMD execution units in each SP. In addition, the execution model support for control flow on GPUs simplifies loop vectorization by eliminating the need to vectorize control flow with predication [Shin, 2007].

6.2.2 New SIMD vector abstraction of GPU execution model

The simple SIMD vector abstraction mentioned above becomes problematic when dealing with multi-level parallel loops. For example, for a loop nest has two levels of parallel loops, if we map the outer parallel loop to GPU threads with SIMD vector abstraction, it would be a problem when we want to use SIMD vector abstraction to map the inner parallel loop to GPU threads. Because there will not be any vectors formed by GPU threads available for the vectorized inner loop. In other words, all the GPU threads are occupied by the outer parallel loop. To solve this problem, we abstract GPU threads as *hierarchical segmented vectors* by taking the execution hierarchy of GPU thread into account. Figure 6-3 shows the hierarchy of segmented vectors. Short SIMD vectors with varying sizes are formed by the GPU threads in a thread block, and a SIMD vector pool represents a thread block.

The SIMD vectors in the vector pool of our hierarchical segmented vectors represent a set of GPU threads in a thread block. The number of GPU threads in this set can change dynamically according to the amount of nested TLP. In other words, when nested TLP requires a set of GPU threads for execution, a SIMD vector representing the same number of GPU threads can be dynamically formed. As these GPU threads are occupied by the outer-most parallel loop(s), in order to form a SIMD vector, all the required GPU threads have to be reclaimed from the outer-most parallel loop.

With our new SIMD vector abstraction of the GPU execution model, mapping nested

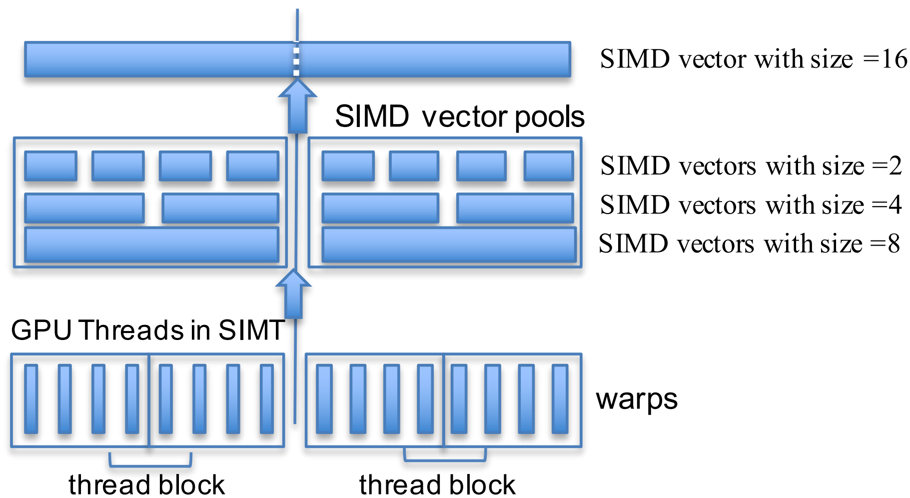


Figure 6-3: Our proposed hierarchical segmented vectors for CUDA GPUs (warp size is 4 and thread block size is 8).

TLP is equivalent to vectorizing the nested loop with SIMD vectors of size VF. In the following sections, we use ***dominant parallel loop*** to refer to the outer parallel loop, of which the iterations are mapped to the GPU threads aligned along the x dimension of the thread block. For instance, the i-loop in Fig. 6-2 is a dominant parallel loop. We call the parallel loop expressing nested TLP a ***target loop***. The j-loop in Fig. 6-2 is a target loop exposing nested TLP.

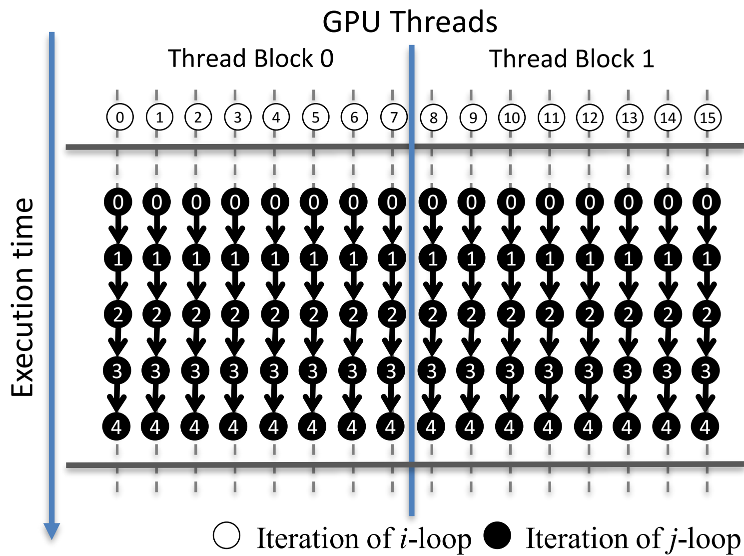
6.2.3 Thread-Reuse Execution Model

Consider a loop nest composed of a dominant parallel loop and a target loop, if we ignore any nested TLP, the target loop is a serial loop. The thread on which an iteration of the dominant parallel loop is running is in charge of executing the serial target loop, as shown in Fig. 6-4(a). When considering nested TLP, if the target loop can be vectorized by either outer-loop or inner-loop vectorization, SIMD vectors are required by the execution of vectorized target loop. Therefore, we need find a set of GPU threads in order to form SIMD vectors in our proposed hierarchical segmented vectors. As all GPU threads are spawned and fixed at the start of kernel execution, we need an execution model in which SIMD vectors can be easily created.

Review of existing execution models for nested TLP

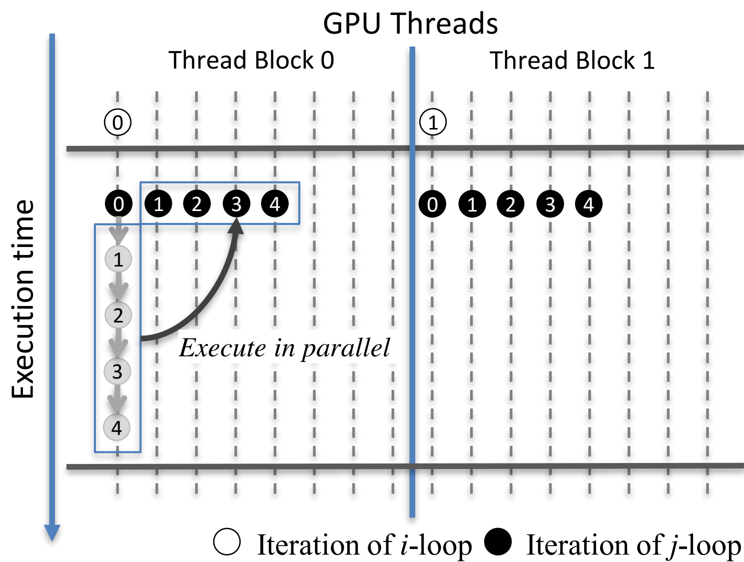
If vectorization factor VF is the thread block size for the dominant parallel loop, one possible execution model for nested TLP is to use only one thread in each thread block to execute the dominant parallel loop, and keep other threads inactive. All the threads in a thread block will become active when they meet the target loop. This *inter-TB master-slave* execution model is depicted in Fig. 6-4(b).

If VF is smaller than the thread block size (TB), another choice is to use (TB/VF) threads in each thread block to execute the dominant parallel loop. As a result, the target loop can get VF threads in each thread block as shown in Fig. 6-4(c). We call this model *intra-TB master-slave*. This model is quite similar to the model adopted in the Yang and Zhou's method, where they treat the GPU threads aligned along one dimension as master threads, and the ones aligned along another dimension as slave threads in multi-dimensional thread blocks [Yang and Zhou, 2014].



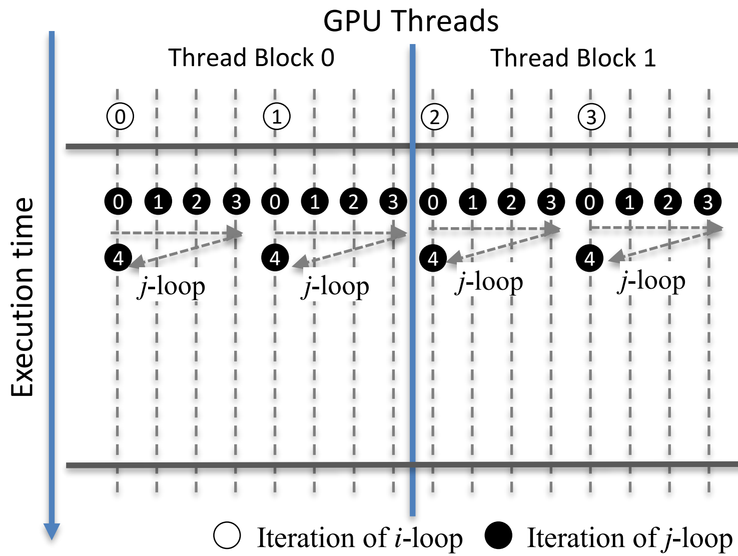
Each iteration of *i*-loop is mapped to a GPU thread; all the iterations of *j*-loop in the same iteration of *i*-loop are executed by a single GPU thread sequentially.

(a) Execution model without nested TLP.



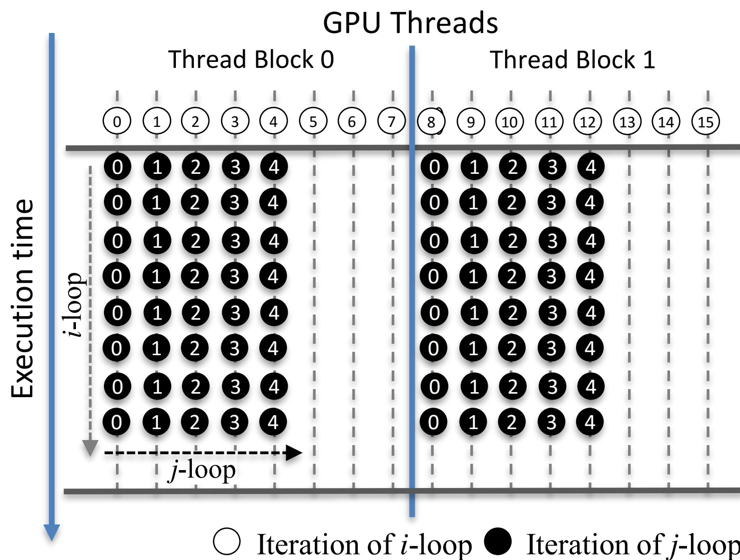
Each iteration of *i*-loop is mapped to a GPU thread block and executed by a single thread in the thread block; all the iterations of *j*-loop in the same iteration of *i*-loop are executed by the GPU threads in a thread block in parallel.

(b) Inter-TB master-slave execution model. This is the default execution model used by PGI compiler for nested TLP.



Each iteration of *i*-loop is mapped to a GPU thread aligned along one dimension (e.g., `threadIdx.x`); all the iterations of *j*-loop in the same iteration of *i*-loop are executed by the GPU threads aligned along another dimension (e.g., `threadIdx.y`).

(c) Intra-TB master-slave execution model (multi-dimensional). This is the execution model used by Yang and Zhou's method for nested TLP.



Each iteration of *i*-loop is mapped to a GPU thread; all the iterations of *j*-loop in the same iteration of *i*-loop are executed by the GPU threads reclaimed from *i*-loop in parallel. The GPU threads used by every 8 iterations of *i*-loop form a SIMD vector with size 8.

(d) Our proposed thread-reuse execution model for nested TLP.

Figure 6-4: Comparison of execution models for nested TLP on CUDA GPUs. Assume *i*-loop is the outermost parallel loop, *j*-loop is the nested parallel loop, data access is contiguous across iterations of *j*-loop. For simplicity, each thread block contains 8 GPU threads and the number of iterations of *j*-loop is 5.

The common weakness of both the inter-TB master-slave model and the intra-TB master-slave model is that they fail to consider the parallelism exposed by the the dominant parallel loop. In both models, the sequential sections before and after the target loop are executed by either one thread or (TB/VF) threads in a thread block. Depending on the amount of computation in the sequential sections, these two execution models may result in low utilization of GPU threads.

Our thread-reuse execution model for nested TLP

To overcome the issue mentioned above, we put forward a *thread-reuse execution model* as shown in Fig. 6-4(d) (VF is equal to TB in the example). In our thread-reuse model, before VF threads of the dominant parallel loop enter the target loop, all these threads are synchronized first. Then, VF iterations I_d of the dominant parallel loop running on these threads are serialized in order to give VF threads to the target loop in each iteration of I_d. However, whether the target loop will be executed by these threads or not depends on the loop vectorization results. If the vectorization is on the dominant parallel loop, these threads are returned back to the dominant parallel loop; otherwise, they are used for the vectorized target loop. Our thread-reuse execution model requires the vectorization factor to be a factor of the thread block size. The vectorization factor is suggested to be the half-warp size, warp size or a multiple of warp size.

The key idea of our thread-reuse execution model is that we want to maximally reuse the threads used for execution of sequential sections of the dominant parallel loop. The granularity of thread reuse is decided by the vectorization factor VF.

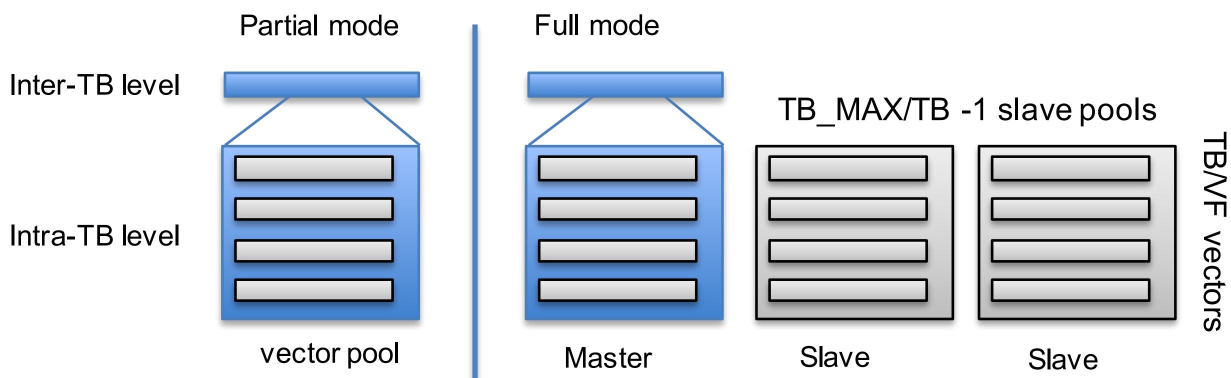


Figure 6-5: Our proposed two execution modes of thread-reuse execution model, *partial* and *full*.

Two execution modes of thread-reuse execution model

Given the vectorization factor VF, for a one dimensional thread block with size TB, the number num_{vect} of existing vectors of size VF is TB/VF, as shown in Fig. 6-5. Without resizing TB, there is only one vector available for each vectorized nested loop. The execution model is in the *partial* mode. This would be a constraint for the vectorized loop due to outer-loop vectorization because the nested loop is kept intact. It would be profitable to enlarge the pools of vectors. The chosen TB is often smaller than the maximum size TB_{max} supported by the CUDA devices. Therefore, if we enlarge the number of GPU threads by TB_{max}/TB times, this would give TB_{max}/TB vectors for each vectorized loop. The execution model is in the *full* mode. These two execution modes can be specified by our language extension for nested TLP (see in Sec. 6.3.1).

6.2.4 Advantages of loop vectorization for nested TLP

The greatest benefit of using loop vectorization to handle nested TLP is that vectorization favors memory performance for GPU. Loop vectorization usually employs either outer-loop vectorization or inner-loop vectorization depending on which loop can help expose contiguous memory access. Contiguous memory access to global memory can be coalesced on GPU and is thus important to the memory performance of CUDA programs. When using loop vectorization to handle nested TLP exposed by a target loop, our compiler decides which type of loop vectorization is profitable for the target loop in terms of contiguous memory access.

Apart from memory performance benefit, loop vectorization helps deal with target loops with a small number of iterations. Vectorization on the target loop, which is the case of inner-loop vectorization, is performed according to a given vectorization factor VF. Therefore, the vectorized target loop can be executed by VF threads rather than the whole thread block.

6.3 Loop Vectorization Framework for Nested TLP

With our proposed thread-reuse execution model, we introduce a compiler framework for our vectorization approach to nested TLP on CUDA GPUs. Similar to other existing vectorization frameworks such as GCC and LLVM, our vectorization approach to nested TLP consists of two major components — vectorization analysis and vectorization transformation. We use the transposed matrix vector multiplication in Fig. 6-2 as the running example.

6.3.1 Language Extension

Sometimes it is very hard for automatic loop vectorization to find the right loop in a loop nest for vectorization due to the lack of a precise enough cost model. In addition, our proposed thread-reuse execution model for nested TLP gives two options for the execution mode (see in Sec. 6.2.3). In order to let users specify the desired vectorization type and execution mode for the execution model, we extend the `vector` clause in the OpenACC loop pragma as follows:

```
vector(vector_length, (partial | full), (inner-vect | outer-vect))
```

, where `vector_length` specifies the vectorization factor VF, `inner-vect` and `outer-vect` indicate inner-loop and outer-loop vectorization, respectively. Vectorization type should be specified according to which loop exposes contiguous memory access that can be coalesced by the hardware. `partial` and `full` are the two execution modes of our thread-reuse execution model.

Note that our new language extension can only be used on the *innermost nested loop* within the outermost parallel loop(s). Therefore, it is orthogonal to the original `vector` clause and will not affect the semantics of the original one. It is illegal to put our language extension around the target loop in the following cases:

1. loop bounds of the enclosing loops of the target loop are not uniform across iterations of the dominant parallel loop;

2. the target loop is nested under decision-making statements (e.g., if-then, if-then-else, switch).

6.3.2 Vectorization Analysis

Vectorization analysis is applied after automatic loop parallelization. Loop parallelization is useful for the OpenACC `kernel` directive, which asks the compiler to automatically identify a parallel loop for a GPU kernel. If automatic loop parallelization is not turned on, programmers have to use the `parallel loop` directive to explicitly annotate the parallel for-loops as GPU kernels. It is common that for a given loop nest, compilers take the outermost closely nested parallel loops as the candidate for a GPU kernel while leaving the identified parallel loops inside the loop nest as sequential loops. Our motivating example in Fig. 6-2 uses the `parallel loop` directive to annotate the outermost *i*-loop to be off-loaded as a computation kernel to GPU. Without considering the parallelism exposed by the *j*-loop, all the computation within *i*-loop is carried out sequentially.

Vectorization for short-vector machines usually works on *simply nested loops* (SNLs), where the loop at each level of the loop nest contains only one loop inside. For example, the loop vectorization in GCC is capable of applying either inner or outer loop vectorization upon a simply nested loop [Nuzman and Zaks, 2008]. If there is no other computation around each loop in the loop nest, the simply nested loop becomes a perfectly nested loop. If a given nested loop is not qualified as an SNL, loop fission can be used to recursively split the loops from the innermost loop to the outermost loop in order to make some of the resultant loops simply nested. For example, the *j*-loop in Fig. 6-6 is not an SNL because it contains two loops, *k*-loop and *h*-loop. However, loop fission often replaces scalar variables with array accesses to break data dependence between two nested loops.

The only loop transformation in our method is loop vectorization. Therefore, instead of applying loop fission to form a simply nested loop, we directly put the dominant parallel loop around the target loop to form *a virtual simply nested loop*. This process avoids the consequences from loop fission, such as scalar expansion. In addition, it is always legal to form virtual simply nested loops because the dominant parallel loop is parallel and moving it inwards in the loop nest will not change the loop dependence. In Figure 6-6, there are two innermost loops nested in the dominant parallel loop. Therefore, with the dominant parallel loop, there are two virtual SNLs annotated as computation region 0 and 1, respectively. Note that for the purpose of vectorization analysis, it is not neces-

```
#pragma acc parallel for
for (i = 0 ; i < i_bound; i++){
  /* computation 1 */
// illegal to put our proposed vector
// directive around j-loop
  for (j = 0; j < j_bound; j++){
     /* computation 2 */
     for (k = 0; k < k_bound; k++){
       /* computation 3 */
     }
     for (h = 0; h < h_bound; h++){
       /* computation 4 */
     } // end-j
  }// end-k
}// end-h
```

Vectorization analysis

```
#pragma acc parallel for
for (i = 0 ; i < i_bound; i++){
  /* computation 1 */
// illegal to put our proposed vector directive around j-loop
  for (j = 0; j < j_bound; j++){
     /* computation 2*/
#pragma acc loop vector (32, partial, inner-vect)
   // only for vectorization analysis
   for (i = 0 ; i < i_bound; i++){
        for (k = 0; k < k_bound; k++){
           /* computation 3 */
  }}
#pragma acc loop vector (32, partial, outer-vect)
  // only for vectorization analysis
  for (i = 0 ; i < i_bound; i++){
     for (h = 0; h < h_bound; h++){
        /* computation 4 */
  }}}}
```


Figure 6-6: Virtual simply nested loops in vectorization analysis. Assume *k*-loop is suitable for inner-loop vectorization while *h*-loop is suitable for outer-loop vectorization.

sary to change the loop bounds of the dominant parallel loop in terms of vectorization factor VF. Because the virtual SNLs are only used for the compiler to decide which loop to vectorize.

The classic loop vectorization analysis – inner-loop or outer-loop vectorization – performs on the resultant virtual simply nested loops mentioned above. Data dependence analysis checks whether the loop at a certain level of the loop nest is legal for vectorization. In the case that several loops are all permitted for vectorization, compilers take contiguous memory access into account to decide the final candidate. With the notations defined in 6.2.2, in this chapter we refer to inner-loop vectorization as the vectorization on the target loop and outer-loop vectorization as the vectorization on the dominant parallel loop in a virtual simply nested loop. Without any explicit vector directives, after the vectorization analysis, vector directives with default configuration – warp size as the vectorization factor, partial mode for the execution model – are annotated around the target loop for vectorization transformation. For example, the vector directives in Fig. 6-6 are the results from automatic vectorization analysis.

In vectorization analysis, our compiler also checks the legality of the usage of our proposed vector clause for nested TLP. First, the vector pragma should be annotated only on an innermost loop nested in the dominant parallel loop. For example, it is illegal to put a vector pragma on the j-loop in Fig. 6-6. Second, the loop bounds of the

loops enclosing the target loop have to be the same at runtime across iterations of the dominant parallel loop. This is the same as the uniform scalar analysis in the whole function vectorization [Karrenberg and Hack, 2011]. The compiler then can guarantee it is always feasible to reclaim SIMD vectors for the vectorized loops. Third, it is illegal to put the target loop under an *if* statement either directly or indirectly or other decision-making statements (e.g., switch in C). This can be eliminated in the future work when our proposed loop vectorization approach is extended to support parallel loops with control flows.

6.3.3 Vectorization Transformation

Given a parallel loop nest with nested loops annotated with our extended vector clause defined in 6.3.1, vectorization transformation on the nested loops is performed in the phase of extracting the outer parallel loops into a GPU kernel. It first splits the computation of the dominant parallel loop into several regions according to the given vector pragmas, and then transforms the target loops contained in the resultant regions into vectorized forms. These two steps can proceed side by side.

Compute Computation Regions

Splitting computation in the dominant parallel loop into several regions is a critical step for vectorization transformation. The thread-reuse execution model for nested TLP requires that all GPU threads spawned at the start of execution must be active when reaching the target loop. The start of each region indicates a synchronization across GPU threads in a thread block. With threads synchronized, the vectorized loop can reuse the existing GPU threads for execution. The algorithm for computing computation regions is presented in Alg. 1.

When generating CUDA code from C programs, the execution guard derived from the targeted parallel loop is often directly put around the computation of the loop body. The execution guard may make some of the GPU threads inactive during the whole execution. Vectorization thus may not be able to get enough active threads for the vectorized loop. When the computation is split into different regions, the execution guard can be distributed to each region. As a result, each region has freedom to decide how to use

Algorithm 1 Compute computation regions for nested TLP

INPUT: *dom_loop*, a parallel loop nest to be extracted as a GPU kernel.
OUTPUT: *ntp_region_list*, regions with nested TLP; *region_list*, regions without nested TLP.

```
pragma_set S ← vector directives in the loop body
exe_guard ← loop condition of dom_loop
if exsits a directive in S has the attribute full then
    exe_guard ← min((bid + 1) * TB, exe_guard)
end if
for each directive annot in S do
    if loop L with annot is directly nested in dom_loop then
        add loop L to partition_stmts and mark L as simply nested
    else
        find the directly nested loop Parent_L in dom_loop enclosing loop L and add Parent_L to
        partition_stmts
    end if
end for
last_stmt_idx ← 0, new_region ← new CompoundStatement
for each loop L in partition_stmts do
    cur_idx ← statement index of L in dom_loop
    if L is simply nested then
        if inner-vect || outer-vect in full mode then
            need_split ← true
        end if
    else
        deeply_nested ← true , need_split ← true
    end if
    add statments from last_stmt_idx to (cur_idx − 1) to new_region
    if need_split == true then
        if_stmt ← an if statement with exe_guard as the condition and new_region as it body
        add if_stmt to region_list; add L to ntp_region_list
        new_region ← new CompoundStatement
    else
        add L into new_region
    end if
    last_stmt_idx ← cur_idx + 1
end for
```

the GPU threads. Furthermore, some instances of the execution guard can be optimized out with redundant computation to reduce control divergence. Fig. 6-7 shows the results of computing computation regions without our vectorization transformation. The execution guard around the region 1 is eliminated with redundant computation.

Transform Computation Regions with Nested TLP

Vectorization transformation is performed on the target loops in each computation region according to the vector directive annotated on the target loop.

```
1   __global__ void tmv_kernel(...)
2   {
3   const int _bid = ...;
4   const int _gtid = ...;
5   const i = _gtid;
6   float sum;
7   sum= 0.0f;                              // region 1
8
9   for (j = 0; j < SIZE && i < SIZE; j++)  // region 2
10    sum += M[j][i] * V[j];
11
12  if (i < SIZE) C[i] = sum;               // region 3
13  }
```

Figure 6-7: Optimization on the execution guard for nested TLP. The execution guard `i < SIZE` is distributed over the three parts of the original loop body.

Step 1: Collect the vectorization information. The compiler first collects vectorization transformation information from the vector directive: vectorization factor, vectorization type, and execution mode of the thread-reuse execution model. The reduction scalar and its reduction operation is also gathered for reduction transformations.

Step 2: Build the thread-reuse execution model. The compiler sets up the GPU threads in a thread block in the form of SIMD vectors. Each GPU thread is given a vector id and a lane id in a vector (line 4-5 in Fig. 6-8). If the thread-reuse execution model is specified in the full mode, master and slave vectors are also built according to the vectorization factor VF, the block size TB, and the max block size TB_{max} (line 2-7 in Fig. 6-9). Note that TB_{max} is the actual thread block size when launching the generated GPU kernel. If one of the computation regions with nested TLP asks the execution model to work in the full mode, the TB_{max} is by default set to the max number of threads that a thread block can accommodate (1024 in this chapter). Otherwise, TB_{max} is the same as TB, which is by default 128 in our compiler.

Step 3: Serialize the dominant parallel loop. The key part of our proposed thread-reuse execution model for nested TLP is reusing existing running GPU threads modeled by SIMD vectors. The given vectorization factor VF decides the granularity of thread-reuse. Before entering the target loop, the existing running GPU threads are distributed over the iterations of the dominant parallel loop. Therefore, in order to claim VF threads

```
1  const int _bid = ...;
2  const int _gtid = ...;
3  float sum;
4  int l_i1_vect_id = (threadIdx.x>>5);
5  int l_i1_lane_id = (threadIdx.x&31);
6  int l_i1_vect_offset = (l_i1_vect_id*32);
7  int l_i1_start = ((_bid*128)+l_i1_vect_offset);
8  int l_i1_end = (l_i1_start+32);
9  __shared__ float _sh_tmp0[128], _sh_tmp1[128], _sh_tmp2[128], _sh_tmp3[128],
       _sh_tmp4[128];
10 i=_gtid;
11 for (seq_l_i1=l_i1_start; seq_l_i1<l_i1_end; seq_l_i1=(seq_l_i1+(1*4))){
12   float red_var_sum=0.0F;
13   float red_var_sum_unroll_v1=0.0F;
14   float red_var_sum_unroll_v2=0.0F;
15   float red_var_sum_unroll_v3=0.0F;
16   for (l_j1_strip=0; l_j1_strip<(2*1024); l_j1_strip=(l_j1_strip+32)){
17   j=(l_j1_strip+l_i1_lane_id);
18   if (j<(l_j1_strip+32)) {
19     red_var_sum+=(M[j][seq_l_i1]*V[j]);
20     red_var_sum_unroll_v1+=(M[j][(seq_l_i1+1)]*V[j]);
21     red_var_sum_unroll_v2+=(M[j][(seq_l_i1+2)]*V[j]);
22     red_var_sum_unroll_v3+=(M[j][(seq_l_i1+3)]*V[j]);
23   }}
24 _sh_tmp0[l_i1_vect_offset+l_i1_lane_id]=red_var_sum;
25 _sh_tmp1[l_i1_vect_offset+l_i1_lane_id]=red_var_sum_unroll_v1;
26 _sh_tmp2[l_i1_vect_offset+l_i1_lane_id]=red_var_sum_unroll_v2;
27 _sh_tmp3[l_i1_vect_offset+l_i1_lane_id]=red_var_sum_unroll_v3;
28 __syncthreads();
29
30 (reduction on _sh_tmp0, _sh_tmp1, _sh_tmp2, _sh_tmp3)
31
32 if ((l_i1_lane_id==0))
33   { put results back to _sh_tmp4}
34 __syncthreads();
35 if (i<SIZE) {
36   sum=_sh_tmp4[((threadIdx.y*32)+threadIdx.x)];
37   C[i]=sum;
38 }
```

Figure 6-8: The generated CUDA code by inner-loop vectorization in partial mode. The vectorization factor is 32 and loop unrolling factor is 4.

for the target loop, the compiler needs to serialize the dominant parallel loop so that threads within the same vector can execute the same iteration of the dominant parallel loop. The lower bound and upper bound of the serialized loop are calculated in terms of VF and the mode of execution model. The loop at line 11 in Fig. 6-8 presents the results of the serialization of the dominant parallel *i*-loop in Fig. 6-2. The lower bound and upper bound are calculated in terms of the vector id at line 7-8 in Fig. 6-8.

For multiple dimensional thread blocks, we treat the loop mapped to the GPU threads aligned along the dimension x (aka. `threadIdx.x`) as the dominant parallel loop. As a result, the compiler is only able to reclaim these threads rather than all the GPU threads in a thread block.

Step 4: Handle live-in/live-out scalars and arrays. The iterations of the dominant parallel loop are executed by different GPU threads, and thus all the scalar and array definitions in the loop body are thread-local. When the dominant parallel loop is serialized to hand over the occupied threads to the target loop, all the live-in scalars and thread-local arrays need to be promoted to thread-block visible arrays in either shared memory or global memory of GPUs. These promoted arrays are initialized before executing the target loop. This data promotion can be optimized when the vectorization type is considered. If outer-loop vectorization in partial mode is applied, it indicates that the dominant parallel loop takes back the threads it gives away; thus, the data context in each thread need no changes.

Step 5: Strip-mine the target loop. As discussed in Sec. 2.4.1, loop vectorization for short SIMD vectors usually first strip-mines a loop with VF and then directly transforms the computation in the resulting element loop to into vector operations. In our vectorization approach to nested TLP, the compiler also applies loop strip-mining upon the target loop according to the given VF, which results in two loops, *strip loop* and *element loop*. Computation in the element loop can be directly transformed into vector operations in the case of inner-loop vectorization. The iterations of the strip loop are useful for scheduling the vector operations over slave vectors in the full mode of execution model.

```
1  (same as inner-vect in partial)
2  const int master_vect_id_l1 = (l_i1_vect_id>>3);
3  const int slave_vect_id_l1 = (l_i1_vect_id&7);
4  const int slave_offset_id_l1 = (master_vect_id_l1*8);
5  int l_i1_master_vect_offset = (master_vect_id_l1*32);
6  int l_i1_start = ((_bid*128)+l_i1_master_vect_offset);
7  int l_i1_end = (l_i1_start+32);
8  int l_j1_strip, int seq_l_i1;
9  __shared__ float _sh_tmp0[1024], _sh_tmp1[128];
10 i=_gtid; seq_l_i1=(l_i1_start+l_i1_lane_id);
11 if (seq_l_i1<l_i1_end){
12 float red_var_sum=0.0F;
13 for (l_j1_strip=(0+(slave_vect_id_l1*32)); l_j1_strip<SIZE;
       l_j1_strip=(l_j1_strip+(32*8))){
14   for (j=l_j1_strip; j<(l_j1_strip+32); j ++ ){
15     red_var_sum+=(M[j][seq_l_i1]*V[j]);
16 }}
17 _sh_tmp0[...]=red_var_sum;
18 __syncthreads();
19
20 inter-vect reduction across slave vectors
21 intra-vect reduction
22
23 __syncthreads();
24 if ((slave_vect_id_l1==0))
25 _sh_tmp1[...]= _sh_tmp0[...];
26 }
27 __syncthreads();
28 if (i<block_guard){
29   sum=_sh_tmp1[((threadIdx.y*32)+threadIdx.x)];C[i]=sum; }
```

Figure 6-9: The generated CUDA code by outer-loop vectorization in full mode. The vectorization factor is 32 , block size (TB) is 128, and the max block size (TB_{max}) is 1024.

Step 6: Schedule the vector operations. When inner-loop vectorization is applied on the target loop, it indicates the dominant parallel loop is serialized so that the GPU threads occupied are given to the vectorized target loop. In the partial execution mode, the vectorized target loop in each iteration of the serialized dominant parallel loop is executed one after another by the reclaimed vectors. Sometimes there exists data reuse across iterations of the serialized dominant parallel loop, such as `V[j]` at line 6 in Fig. 6-2. Loop unroll-and-jam can be applied on the dominant parallel loop to take advantage of this kind of data reuse. Line 11-27 in Fig. 6-8 presents the results of loop unroll-and-jam with a loop unrolling factor 4. The loop unrolling factor is an important performance factor for the GPU kernel because loop unroll-and-jam may introduce more memory usage and in turn affects the occupancy of the GPU kernel.

When the thread-reuse execution model is running in the full mode (in Section 6.2.3), there are extra free vectors for executing the vectorized loop. In either inner-loop or outer-loop vectorization, the strip loop from loop strip-mining on the target loop is used to distribute the vector operations over the slave vectors. For example, the strip-loop at line 13 in Fig. 6-9 is distributing the work of the target loop in the size of 32, the vectorization factor, to 8 slave vectors.

Step 7: Transform reduction operations. The target loop sometimes has reduction operations on scalars. We use interleaved log-step reduction [Wilt, 2013] to handle reduction. In the partial execution mode, intra-vector reduction is performed such as the one on `_sh_tmp0` in Fig. 6-8. On the other hand, in the full execution mode, inter-vector reductions across the slave vectors are followed by intra-vector reductions, as depicted in Fig. 6-9. The intermediate reduction results are kept in shared memory.

Step 8: Generate vectorized code. The generation of vectorized code from the resulting loops – the serialized dominant parallel loop, the strip-loop, and the element loop – is simply performed by changing the relevant loop into a vectorized form according to the vectorization type. In the case of inner-loop vectorization, the element loop is transformed into a vectorized form, as shown in line 17-23 in Fig. 6-8. For the outer-loop vectorization, the serialized dominant parallel loop is transformed back to a parallel loop.

6.4 Evaluation

6.4.1 Experimental Methodology

Our proposed vectorization approach is implemented in the Cetus source-to-source compiler [Lee et al., 2009]. The Cetus compiler can generate OpenMP code with automatic loop parallelization and CUDA code from OpenMP programs. The front-end of the Cetus compiler is modified to take C code with OpenACC pragmas. We implemented new vectorization for nested TLP as optimization passes and reused the existing optimization passes to work with the new passes. The generated CUDA code from our vectorization is compiled by Nvidia CUDA compiler of version 6.5. The same compiler is used for building other versions of benchmarks in CUDA.

The performance evaluation is carried out on a Nvidia GTX 645 with 1G memory. Most of the benchmarks used are from Rodina benchmark suite [Che et al., 2009] and NPB benchmark [Bailey et al., 1991a]. Among these benchmarks, Streamcluster (SC), K-means (KM) and Backprop (BP) are from Rodina. CG is from NPB benchmark. For simplicity, we only evaluate the performance of the hot-spot loops with nested TLP in these benchmarks. In addition, three synthesized benchmarks, matrix vector multiplication (MV), transposed vector multiplication (TMV) and a mix of these two (TMV-MV) are the typical examples of nested TLP in scientific computing.

We also compared the performance against an industrial compiler, PGI compiler 15.5 64 bit, with options `-O3 -acc -ta=nvidia -Minfo=accel`. The PGI compiler supports OpenACC directives and is highly optimized for GPUs and accelerators. Because PGI compiler cannot output CUDA code, we use the Nvidia profiler (nvprof) to collect the kernel execution time for the performance comparison. We also use the information from `-Minfo=accel` to inspect the mapping strategies adopted by PGI compiler.

6.4.2 Experimental Results

Performance Comparison between Execution Models for Nested TLP

In this section, we compare the performance between different execution models for nested TLP on CUDA GPUs. As mentioned in Sec. 6.2.3, there exist two execution

models for nested TLP, inter-TB master-slave (Fig. 6-4(b)) and intra-TB master slave (Fig. 6-4(c)).

In Fig. 6-10, we report the speedups over the Yang and Zhou's method [Yang and Zhou, 2014]. Yang and Zhou's work adopts the intra-TB master-slave execution model (in Fig. 6-4(c)) to deal with nested TLP in the CUDA programs. They use GPU threads aligned along one dimension in a thread block (e.g., `threadIdx.x`) as master threads and ones aligned along another dimension (e.g., `threadIdx.y`) as slave threads. The choices of thread dimensions for master and slave threads lead to two mapping strategies, inter-warp and intra-warp. The inter-warp strategy uses threads in different warps to work collaboratively for the workload of a master thread. The intra-warp uses threads within a warp to distribute the nested loop.

The drawback of Yang and Zhou's method is that the number of threads for nested TLP is limited by the max thread block size TB_{max}. In order to allocate more slave threads, programmers have to carefully balance the number of master and slave threads. In contrast, our method reuses the threads used by the dominant parallel loop. The number of threads for nested TLP is reclaimed according to the vectorization factor, and thus it can be as large as the thread block size.

The performance of TMV, MV and TMV-MV shows another disadvantage of Yang and Zhou's method. The master-slave thread configuration tailored to one nested parallel region may not fit another. On the other hand, our vectorization approach is flexible to adapt to different regions. In summary, our vectorization approach to nested TLP can achieve the same or even better performance than Yang and Zhou's method without tweaking the structure of thread blocks.

We also compared our vectorization approach to the inter-TB master-slave execution model (in Fig. 6-4(b)). The performance comparison is depicted in Fig. 6-11. Without specific execution configuration through the OpenACC `gang`, `worker`, and `vector` clauses, PGI compiler automatically uses the inter-TB master-slave execution model to handle nested TLP. A block size of 128 is used by default for nested loops. In general, if the nested parallel loop is a candidate for inner-loop vectorization, MV, for example, PGI compiler is also able to deliver better performance than without optimizing for nested TLP. However, if outer-loop vectorization is more suitable for the nested parallel loop,

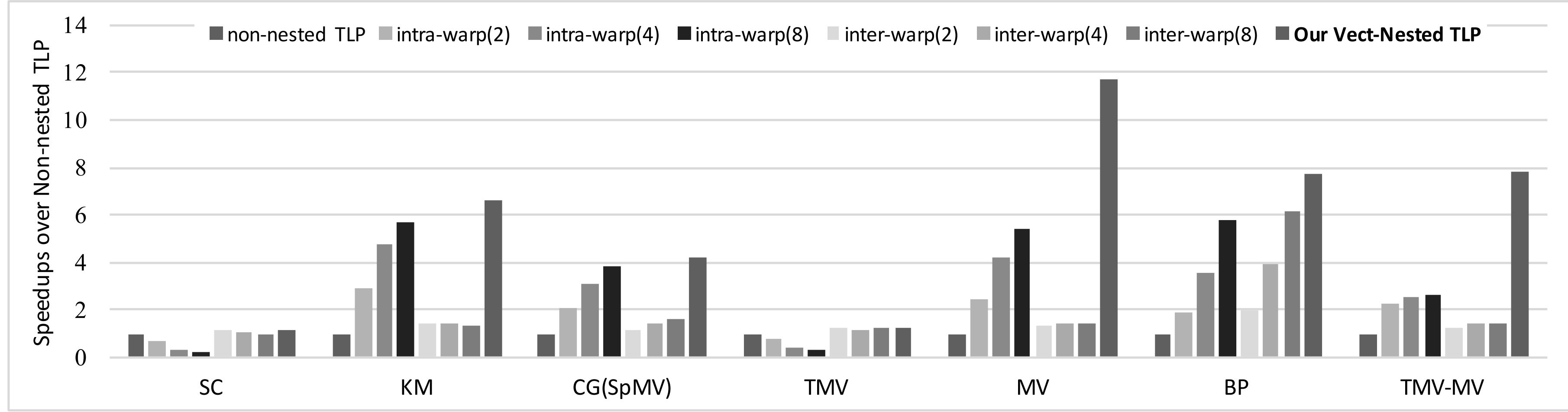


Figure 6-10: Performance comparison between our vectorization approach and Yang and Zhou's method [Yang and Zhou, 2014] (TB = 128)

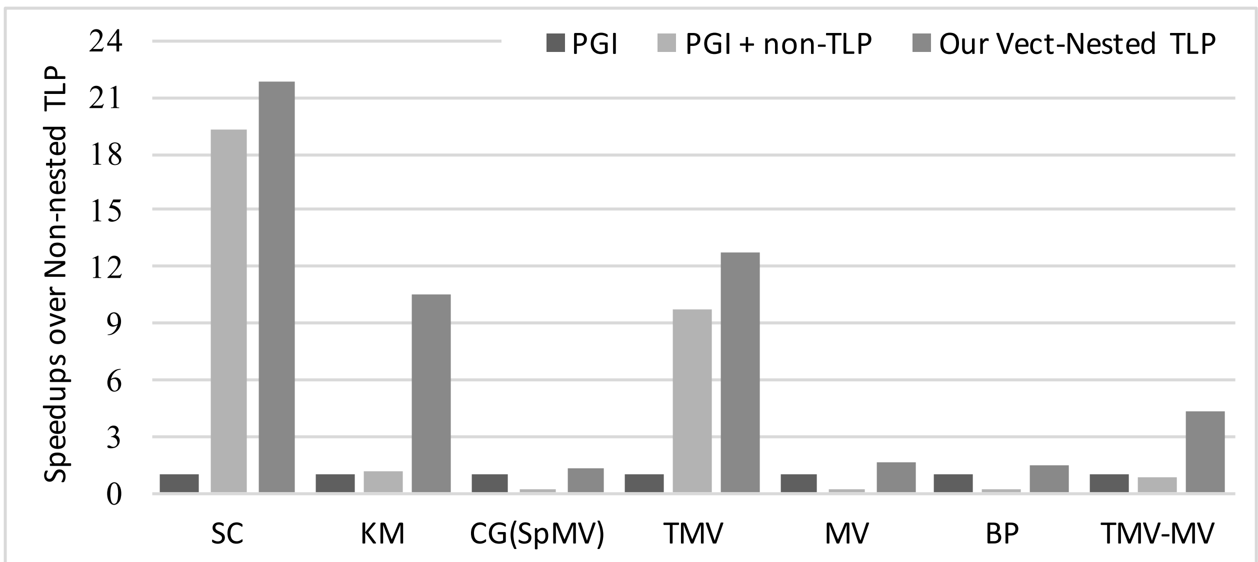


Figure 6-11: Performance comparison between our approach and PGI compiler. The `loop seq` directive is used to disable the default handling of nested TLP in PGI compiler.

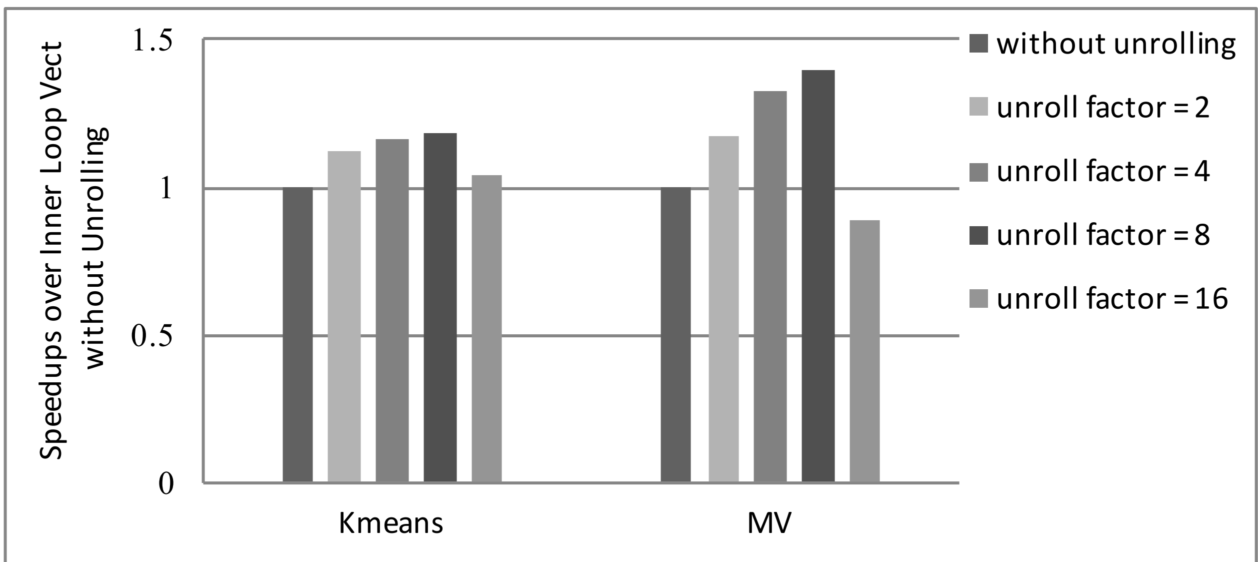


Figure 6-12: Performance impact of loop unrolling factor in partial mode. VF for Kmeans and MV is 16 and 32.

the same optimization strategy would lead to significant performance degradation, for instance, TMV. Compared to the default approach in PGI compiler, our vectorization method is able to give better performance in all the benchmarks.

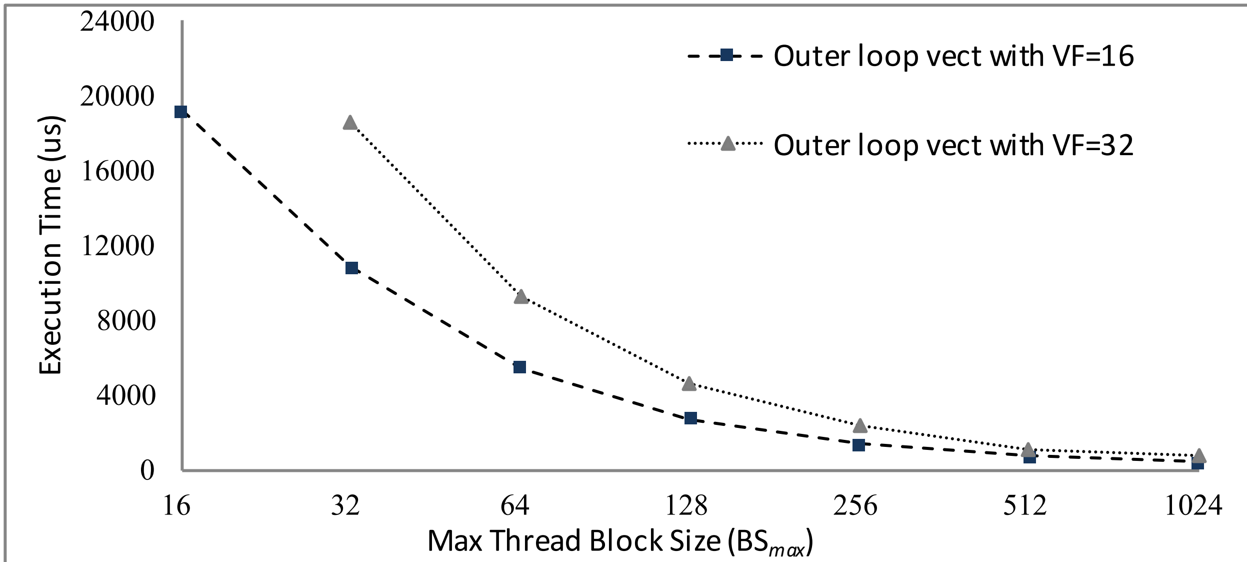


Figure 6-13: Performance impact of vectorization factor and TB_{max} in full execution mode in Backprop. The block size TB is set to vectorization factor VF.

Performance Factors

Loop Unrolling Factor As mentioned in Sec.6.3.3, for inner-loop vectorization, we also use loop unroll-and-jam to schedule vector operations over the salve vectors. Fig. 6-12 shows the performance impact of loop unrolling factors. Both Kmeans and MV have data reuse across the dominant parallel loop. Therefore, unrolling and jamming the serialized dominant parallel loop will help reduce the number of memory accesses due to the reused data. However, as the unrolling factor increases, the shared memory usage for the reduction operation goes up as well, which affects the overall thread occupancy and in turn leads to performance degradation.

Vectorization Factor and Execution Configuration We use Backprop (BP) to study the performance impact of the vectorization factor and the max thread block size for outer-loop vectorization. The performance results shown in Fig. 6-13 indicate that the vectorization factor should be decided according to the ratio of the iteration count of the dominant parallel loop and the target loop. The dominant parallel loop in BP only has 16 iterations so that a larger vectorization factor (32) makes half of the threads in a vector inactive. In addition, for outer-loop vectorization, the execution model in full mode with a proper max thread block size gives more slave vectors to distribute the work of the target loop.

The hand-tuned CUDA version of BP in the Rodina benchmark changed the computation structure and introduced global reduction running on the CPU side. But by only comparing to the GPU part alone, our method already obtains a speed-up of 1.52 $\times$.

6.5 Related Work

There has been extensive work on vectorization for short SIMD vectors on CPUs. Different vectorization techniques exploit SIMD parallelism from different scopes of the program, such as basic blocks [Larsen and Amarasinghe, 2000], loop nests [Nuzman and Zaks, 2008][Kennedy and Allen, 2002], and functions [Karrenberg and Hack, 2011]. Some work [Wu et al., 2005] also attempted to build a unified framework to integrate above mentioned techniques together. Finding contiguous memory access and transforming data access into contiguous forms underlies these vectorization techniques.

On the other hand, automatic vectorization in C-to-CUDA compilation seems unnecessary for CUDA GPUs because of the underlying SIMT execution model and lack of SIMD execution units in the CUDA steaming processors. Jang et al. introduced vectorization via data transformation to benefit vector-based architectures (e.g., AMD GPUs) [Jang et al., 2011b]. Our work in Chapter 4 extended the SLP vectorization [Larsen and Amarasinghe, 2000] to support semi-isomorphic instructions [Xu and Gregg, 2014a] and applied it in the C-to-CUDA compilation to improve the memory performance of GPU programs [Xu and Gregg, 2015]. However, our work in Chapter 4 and 5 does not consider using vectorization to deal with nested TLP. Nested TLP is also an important source of contiguous memory access. Our loop vectorization approach to nested TLP in this Chapter is demonstrated as an efficient way of exploiting contiguous memory access in nested TLP for better performance.

Yang and Zhou put forward a compiler framework CUDA-NP based on OpenMP-like pragmas for nested TLP in CUDA code [Yang and Zhou, 2014]. They adopt the intra-TB master-slave execution model, which we found not to be as effective as our thread-reuse execution model. Our work uses loop vectorization technique to deal with nested TLP in the C code with pragmas and automatically generates CUDA code. Lee et al. studied nested TLP in the presence of their high-level language [Lee et al., 2014].

Their high-level language utilizes computation patterns (e.g., map and reduce) to enable automatic compilation to GPU kernels. A multi-dimensional analysis is presented to map nested TLP to the GPU. However, their solution as well as Yang and Zhou's method is not suitable for the cases where a parallel loop has several nested parallel loops with varying features. Because some of these loops may require a different mapping strategy from others. In contrast, our loop vectorization method is flexible to adapt to parallel loops with different features. The decision on which type of loop vectorization should be applied is determined by data access patterns, in particular, contiguous memory access.

Kim et al. put forward a locality-centric thread scheduling to schedule work-items in OpenCL for CPUs [Kim et al., 2015]. The problem they solved is essentially the same as how to efficiently map nested parallel loops but in a different scenario. The choices of scheduling strategies are decided by the data locality, which is also the foundation of vectorization. Therefore, our vectorization approach to nested TLP can be used as an another solution to their problem.

Other approaches proposed efficient code generation strategies for GPU from compiler directive based programming models [Lee and Vetter, 2014]. Bertolli et al. introduced inspector-executor schemes to coordinate threads for mapping OpenMP 4.0 to CUDA [Bertolli et al., 2014]. Tian et al. discussed how to map the three levels of parallelism expressed in OpenACC directives – `gang`, `worker` and `vector` – to the GPU [Tian et al., 2013]. Our vectorization method can be adopted to derive an efficient configuration of these directives for nested TLP.

Hong et al. put forward a warp-centric programming method to improve the performance of graph algorithms [Hong et al., 2011]. In addition to using warps as basic execution units, their work introduced dynamic work scheduling to resolve load imbalance issues. Bauer et al. presented a domain specific language for combustion chemistry [Bauer et al., 2014]. Their method partitions computations into sub-computations and assigns them to different warps within a thread block. In contrast, we use loop vectorization technique to off-load sub-computations contained in nested TLP to GPU threads.

6.6 Summary

In this chapter, we present a loop vectorization approach to exploit nested TLP in C-to-CUDA compilation for CUDA GPUs. Our vectorization approach is designed to use the GPU threads for outer parallel loop(s) to execute nested TLP. To this end, we introduced a thread-reuse execution model, which characterizes the deep hierarchy of execution model of CUDA GPUs from the vector point of view. Due to the imprecision of cost models in automatic vectorization, we extended the vector clause in OpenACC to allow users to specify the vectorization factor, vectorization type – inner-loop vectorization or outer-loop vectorization, and the mode of execution model. We implemented our method in the Cetus compiler and compared performance against both existing research work and an industrial compiler. The experimental evaluation shows that our method is feasible and highly effective.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Fine-grained AoS-to-SoA for Customizable Precision Arithmetic

7.1 Introduction

One of the most important developments over the last decade has been the move from desktop computing to battery-powered computing in hand-held, wearable and mobile devices. This move from the desktop to the wider world is also reflected in the growth of applications that operate on real world data such as images, video, sound and motion. These applications are highly computationally intensive, and pose huge challenges both for mobile devices and for cloud-based services that receive and process large amounts of such data.

One popular approach to improving the performance of these types of applications is to use reduced precision data. The inputs and outputs of media applications are approximations, and introducing additional imprecision in the computation may have little or no effect on the final result. For example, ARM embedded processors support half precision *binary-16* floating-point (FP), as compared with the more common single precision (FP) *binary-32*.

When designing custom hardware such as FPGA and ASIC designs, data precision can be customized precisely to the needs of an application. Reducing precision can reduce the size of the hardware, but crucially it can also allow less data to be transfered between the processor and memory. Reduced precision data may allow faster compu-

tation, and crucially it reduces the amount of memory traffic. The energy required to fetch a word of data from main memory is a large multiple of that required to perform an arithmetic operation on that data.

On general-purpose processors it is much more difficult to customize the precision of data to the application. Most general-purpose processors provide only two floating-point sizes — single and double precision — and a limited range of integer data sizes, typically 8, 16, 32, and 64-bit. For example, if 9 bits of integer precision are required for an application, the programmer will normally use a 16-bit type. Similarly, if one needs 13-bit floating-point (FP), one might use a half precision *binary-16* FP type if it is available, or single precision *binary-32* if not.

When the data consists of a large array of values, the cost of using more precision than necessary can become large. The obvious problem is that the larger data size requires more space in memory. But the larger data size also requires more memory bandwidth when transferring between processor and memory, and more energy to drive external pins, wires and buses when transferring unnecessarily large data.

In this chapter, we propose a new approach to supporting arrays of irregular precision floating-point and integer data types on general purpose architectures with SIMD extensions. We use *software bitslice* format to represent arrays (vectors) of data. Using bitwise logical instructions we build arithmetic operators that perform addition, multiplication or division on an entire vector of 32, 64, or 128 values at once. By building operators out of fundamental logical operators, we achieve enormous flexibility in the precision and type of operators that we support.

7.2 Software Bitslice Representations

In the standard representation of simple types, such as integer and floating-point values, a single value fits inside an 8, 16, 32 or 64-bit word. In a bitslice representation, the different bits of a single number are spread across multiple machine words.

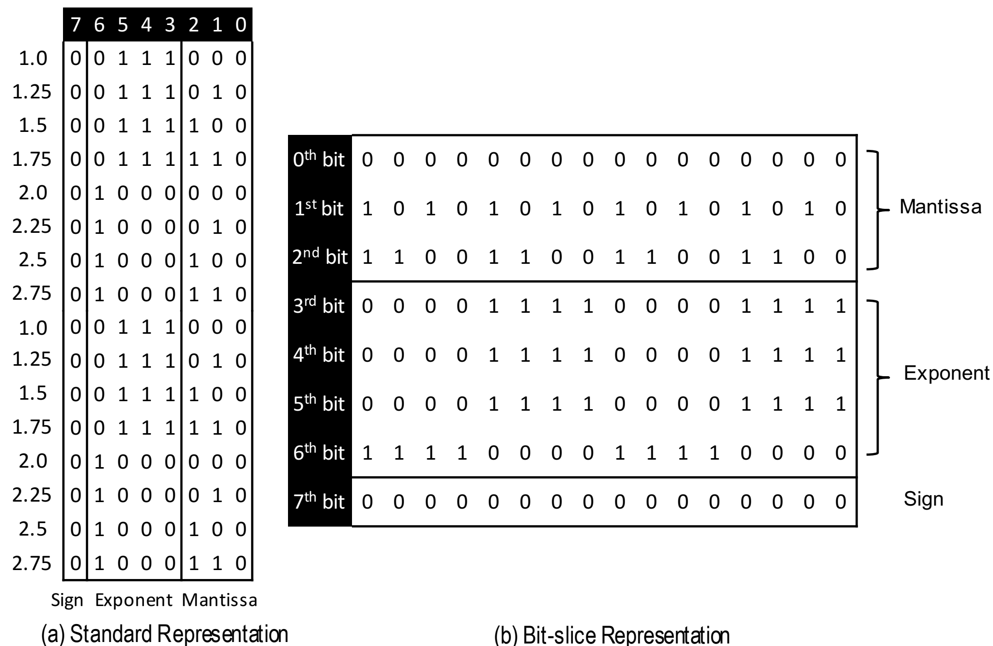


Figure 7-1: Standard and bitslice representation of an array of sixteen 8-bit floating-point numbers.

Figure 7-1 shows an example of standard and bitslice representations of arrays. Both the standard and bitslice representations show an array of sixteen 8-bit floating-point numbers. Each number has 1 sign bit, 4 exponent bits and 3 mantissa bits. However, the physical representation of the data in memory is quite different. Instead of using sixteen 8-bit words, the bitslice representation uses eight 16-bit machine words. The first bit of each of these eight 16-bit words corresponds to one of the eight bits of the first array element. Similarly, the other 8-bit values are represented by a bit from each of the eight 16-bit words.

Bitslice representations are sometimes used for highly efficient implementations of symmetric cryptography algorithms such as the Data Encryption Standard (DES) or Advanced Encryption Standard (AES) [Biham, 1997]. These cryptography algorithms perform large numbers of bit-level operations, which can be very fast on bitslice repre-

sentations. Bit-serial parallel systems built using one-bit processors (e.g. MPP [Batcher, 1980], DAP [Active Memory Technology Inc., 1988] and CM-2 [Thinking Machine Corp., 1989][Hillis, 1992]) can natively support operations on data in bitslice representation. But to our knowledge they have not been applied to more general purpose computation on general-purpose processors with SIMD extensions.

7.3 Bitslice Vector Computing

In this chapter, we propose using bitslice representations of arrays as the basis of a new approach to vector SIMD computing on general-purpose processors with SIMD extensions. We show how to construct vector SIMD operations in software that operate on these types. This allows us to construct SIMD vector types of a fixed number of elements but with arbitrary bit precision per element. For example, we can construct a vector of thirty-two 8-bit elements, but equally we can construct a vector of thirty-two 17-bit elements.

```
1  #define ADD_BITS 13
2  uint32_t ary1[ADD_BITS];
3  uint32_t ary2[ADD_BITS];
4  uint32_t result[ADD_BITS];
5
6  uint32_t carry = ALLZEROS;
7  for (int i = 0; i < ADD_BITS; i++) {
8    t1 = ary1[i];
9    t2 = ary2[i];
10   xxor = t1 ^ t2;
11   aand = t1 & t2;
12   // add
13   result[i] = xxor ^ carry;
14   // update carry
15   carry = (carry & xxor) | aand;
16 }
```

Figure 7-2: Bitslice adder for two arrays of unsigned integers. Each integer has ADD_-BITS bits. The size of uint32_t decides the number of array elements are being processed.

To operate on the elements of bitslice vector types, we propose building arithmetic and other operators from native integer bitwise instructions. Figure 7-2 shows a simple integer adder routine for bitslice vectors with thirty-two elements, each of 13 bits. Note that the addition is performed by a sequence of bitwise operations that are the software

equivalent of a hardware adder. Thus the sum of two bits $t1$ and $t2$ is $t1$ XOR $t2$ and the carry from the addition is $t1$ AND $t2$. By applying a sequence of these bitwise operations, an entire k-bit addition can be performed.

In a hardware adder, each logic gate operates on one binary value. However, the bitwise logical operators in the adder in Fig. 7-2 operate on an entire 32-bit register of values at once. Thus, the addition is performed sequentially by a sequence of bitwise operations. But each bitwise instruction operates on thirty-two separate 13-bit values at a time and exploits thirty-two way bitwise parallelism. So our adder operates in vector SIMD style, requiring a number of steps that is proportional to the number of bits in each value, but operating on a vector of different values that is equal to the word-size of the underlying native machine type.

A big advantage of our proposal for bitslice vector types is that they allow vectors of values with an arbitrary number of bits. One can easily support vectors of numbers with 5, 9, or 13 bits. Operating on bitslice vector types is laborious from a sequential point of view, but exploits large amounts of bit-level parallelism within the conventional machine word. The major downside of operating on bitslice vector types is that each operation requires large numbers of bitwise operations. As the number of bits in each value grows, the execution time of the arithmetic operators increases rapidly. However, as we show in the following sections, it can work well for vectors of short, irregularly-sized types.

It has been demonstrated that not all programs need the precision provided by the generic FP hardware and different sections of a program can benefit from different bitwidths for the sake of overall accuracy and power consumption [Tong et al., 2000]. The balance between accuracy and performance makes our solution perfectly suited to the needs of approximate computing.

7.4 Operating on Bitslice Vectors

Bitslice vectors have previously been used for the implementation of cryptography algorithms. These algorithms perform a large number of operations on individual bits. The bitslice representation provides direct access to the individual bits within a vector of numbers. Figure 7-3 shows two versions of code that negates the n'th bit of each number

in an array of thirty-two 16-bit numbers. In the standard representation, negating every n'th bit requires a loop with separate instructions to operate on each number. In contrast, the bitslice representation gives direct access to the n'th bit of all thirty-two numbers in the vector with just a few instructions.

```
1   void negate_bit_standard(uint16_t a[32], int n)
2   {
3     for ( int i = 0; i < 32; i++ ) {
4       a[i] = a[i] ^ (1 << n);
5     }
6   }
7
8   void negate_bit_bitslice(uint32_t a[16], int n)
9   {
10    a[n] = a[n] ^ 0xFFFFFFFF;
11  }
```

Figure 7-3: Two versions of code that negate the n'th bit of each element of a vector of thirty-two 16-bit integers. The first code fragment operates on the standard representation of arrays of numbers. The second operates on bitslice vectors.

Bitslice representations can be highly-efficient for bit manipulation operations where the same operation is applied to the same bit of a vector of numbers as in Figure 7-3. Similarly bit shift, bit rotate and bit permutation can all be implemented efficiently in bitslice format. This makes bitslice formats ideal for symmetric cryptography algorithms which perform large numbers of bitwise operations.

However, there is another property of bitslice representations that to our knowledge has not been previously been exploited for approximate operators on general-purpose processors with SIMD extensions. Bitslice formats allow us to represent vectors of numbers with non-standard numbers of bits. We can create vectors of 9, 13 or 17 bit integers. Equally, bitslice formats allow the creation of floating-point vector types with arbitrary mantissa and exponent bit widths.

This flexibility in bit-widths of integer and floating-point numbers raises the possibility of vectors of numbers with just enough precision. One of the main strengths of custom hardware designs with FPGAs or ASICs is the possibility to customize the data precision to the needs of the application, which simplifies circuits and reduces the size of data in memory. Our bitslice vector types offer the same sort of bit-level customized precision in software when operating on vectors.

A nice feature of our approach is that we implement arithmetic operators using Boolean logic in the form of bitwise logical software instructions. We adapt existing circuit design techniques that were developed for Boolean logic in hardware and use them to build arithmetic circuits in software.

Thus, our bitslice vector approach offers a new model of vector computing that exploits the bitwise parallelism in existing bitwise logical instructions of general purpose processors with SIMD extensions. However, our bitslice approach can also exploit techniques that arose out of decades of research on implementing efficient hardware Boolean circuits. Our approach breaks down the traditional boundaries between hardware circuits and software instructions.

7.4.1 Basic operations

For our bitslice vector types to be useful, we need to be able to perform basic arithmetic operations, such as vector add. We focus on "vertical" vector operations, such as adding the values in the corresponding lanes of a pair of vectors to produce a result vector. These vertical vector operations benefit from significant bitwise parallelism, because the same bitwise operation is applied to a given bit of all element of the vector. In all cases we consider vectors where the number of bits of precision, n, is known statically in advance.

- Integer **addition/subtraction** can be performed by building full adders from `xor` and `and` operations, as shown in Figure 7-2. Integer addition requires $O(n)$ operations to add a pair of vectors, where each element of the vector has n bits of precision.

- A wide range of techniques exist for optimizing integer **multiplication** circuits, especially for combining partial sums [Ercegovac and Lang, 2004]. We implemented a relatively simple shift-and-add multiplier which requires $O(n^2)$ operations to multiply two vectors with n digits.

- A range of **division** circuits has been proposed, with various trade-offs in area and latency. We implement a *restoring division* [Muller et al., 2010] circuit that requires $O(n^2)$ operations to divide the elements of one vector by the corresponding elements of another.

- A constant bit **shift/rotate** that is applied to all lanes of a vector can be implemented in a simple loop to permute the bits, that requires $O(n)$ operations to shift/rotate a vector of n digits.

- A **variable bit shift** where each lane may be shifted by a different amount is a more expensive operation. We have implemented a logarithmic shifter, which can perform a variable bitwise shift independently on each lane using $O(nlog_2(n))$ bitwise logical operations. Logarithmic shifters have been shown to be an effective way of saving power in hardware [Pillai et al., 1997][Acken et al., 1996].

The variable bitwise shift is particularly instructive. When implementing variable shifts in hardware, single-cycle barrel shifters are popular. However, single-cycle hardware shifters are typically designed to minimize latency rather than area.

When operating on software bitslice vectors, each gate in the circuit performing the operation must be implemented with a bitwise logical instruction. A circuit with fewer gates requires fewer instructions to execute and is therefore usually faster to implement in software. This is an important difference between circuit design for software bitslice vectors and for hardware. In hardware, the primary goal is usually to minimize latency, whereas for software bitslice vectors the aim is to minimize gate count.

7.5 Bitslice Floating Point Vector Operations

The basic operators from Section 7.4.1 operate on vectors of integer types. However, the great flexibility provided by bitslice types allows us to implement vectors of any numeric type. In this section we describe how to build vectors of floating-point types with customized mantissa and exponent. We implement the floating-point operators using circuit techniques that were originally developed for hardware. We start with a brief description of IEEE 754 floating-point, and then describe the operators in more detail.

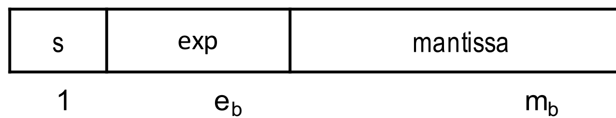


Figure 7-4: Floating-point word.

7.5.1 IEEE-754 Floating Point Format

The IEEE Floating-point Standard 754 was developed so that all machines could provide consistent floating-point behaviour. The standard is designed for several floating-point types, the most common of which are 32 bit words (single precision) or 64 bit words (double precision). Each word consists of a sign bit(s, 1 bit), a mantissa (mnt, m_b bits) and an exponent (exp, e_b bits) as shown in Fig. 7-4, and presents the value of a number:

$$s \times mnt' \times 2^{exp'} = s \times h \cdot mnt \times 2^{exp-bias} \tag{7.1}$$

where h is an implicit bit known as the hidden bit; $bias$ is a constant depending on e_b and has the value $2^{e_b-1} - 1$. The floating-point format in this number presentation can represent zeros, infinites, exceptions and two number types, normal ones (normalized) and numbers very close to zero (denormalized). These five types are differentiated by the exponent and mantissa values. All possible combinations of exponent and mantissa values are depicted in Table 7.1.

Table 7.1: Types of floating-point numbers.

Type	Exponent	Mantissa	h	value
Zero	0	0	-	± 0
Denormalized	0	$\neq 0$	0	Eq.7.1
Normalized	1 to $2^{e_b} - 2$	-	1	Eq.7.1
Infinites	$2^{e_b} - 1$	0	-	$\pm\infty$
NaN	$2^{e_b} - 1$	$\neq 0$	-	-

Note that the IEEE 754 standard also supports rounding of intermediate results when performing operations such as addition and multiplication. The intermediate result must normally be computed to a greater precision than the final result, after which rounding is applied. The IEEE standard provides four different rounding methods:

- Nearest: rounding to the nearest value, to even when tie

- Up: towards $+\infty$
- Down: towards $-\infty$
- Zero: towards 0, truncate

7.5.2 Bitslice Floating-point Operators

We present three bitslice floating-point (BFP) operators – addition/subtraction, multiplication and division. BFP operators can be constructed from logic gates in the same way as the integer operations in Section 7.4.1. In fact, the bitslice integer operations form building blocks that we use to construct BFP operators.

```
1  typedef uint16_t BFP_ELEM_TYPE;
2  #define SIGN_BIT 1
3  #define EXPO_BIT 8
4  #define SIG_BIT 23
5  typedef struct{
6  BFP_ELEM_TYPE data[SIGN_BIT + EXPO_BIT + SIG_BIT];
7  } BFP_FP_VEC_TYPE;
```

Figure 7-5: Bitslice floating-point vector types for FP32.

Figure 7-5 shows an example of a bitslice type that is used to implement a vector of sixteen 32-bit floating-point numbers. Our BFP operators follow the classic implementation of floating arithmetic circuits in hardware [Ercegovac and Lang, 2004] but with the aim of minimizing the number of gates rather than the overall latency.

The IEEE-754 floating-point standard handles five number types in a common format while maximizing the total set of numbers that are represented. This design increases the complexities of the arithmetic units because apart from the calculation, a preprocessing of input numbers (i.e., pre-normalization) and a post-processing of output numbers (i.e., post-normalization) are required to deal with different types of floating-point numbers. Therefore, when implementing a floating-point operator either in hardware or software, additional logic is required to handle the complexity of the format. In order to reduce the number of logic operations for BFP operations, we assume the input values are normalized numbers so that the logic for pre-normalization can be eliminated. In addition, we do not take care of subnormals and special values (except zero) as results and exceptions arising from the operations.

To get the exactly-rounded result of any operation, up to three extra mantissa bits – guard bit, round bit and sticky bit – are needed in addition to the bits provided by the standard format. At the stage where the result is rounded, these extra bits of the result and the sign of the result (in the case of round-towards-up/down) are analyzed to perform the rounding. Of the four rounding modes, round-to-zero is the simplest to implement. It is almost equivalent to truncating the result rather than rounding it. A major complication of rounding modes such as round-to-nearest is that it may result in an extra 1 being added to the least significant digit of the result. Propagating the carry from this increment potentially requires $O(e_b + m_b)$ additional gates. Although our implementation supports both round-round-to-nearest and round-to-zero, we confine ourselves to round-to-zero throughout this chapter.

For each of the following operators, we present a brief high-level description of the operation, followed by a more detailed description of the algorithm we use to implement it. Note that our goal is to minimize the gate count rather than minimizing hardware circuit latency, and we therefore favour smaller software circuits with fewer gates. Let x and y be the floating-point operands represented by (S_x, E_x, M_x) and (S_y, E_y, M_y) respectively. In IEEE 754 format, the first bit of the mantissa of a normalized number is always 1, and is therefore not stored. However, we need to consider both the number of stored montissa bits m_b and the total number of mantissa bits $m = m_b + 1$. Both x and y are normalized numbers.

Bitslice Floating-point Addition

Our bitslice floating-point addition/subtraction is implemented as the following steps:

1. Find the operand with maximum absolute value, and if necessary swap the operands so the maximum is first. We do both steps in a single pass in $O(e_b + m_b)$ time.

2. Produce sign of result in time $O(1)$.

$$S_r = \begin{cases} S_x & \text{if } |x| > |y| \\ S_y & \text{if } |x| < |y| \\ S_x = S_y & \text{if } |x| = |y| \text{ and } S_z = 0 \\ 0 & \text{if } |x| = |y| \text{ and } S_z = 1 \end{cases}$$

   where $S_z = S_x \bigoplus S_y$.

3. Subtract exponents $d = E_x - E_y$ in time $O(e_b)$.

4. Align the mantissa. Shift the mantissa of the operand with the smaller exponent right by d positions. Each lane of the vector may be shifted by a different amount, so a variable distance shifter is used, which takes $O(mlog(m))$ time.

5. Add/subtract mantissas. Some vector lanes may require addition, while others require subtraction. We implement a fused add/subtract software circuit which exploits common sub-expressions between the two, and takes $O(m)$ time.

6. Normalization of result. There are three situations:

   (a) The results may already be normalized.

   (b) For addition, overflow can occur, in which case normalization increments the exponent while shifting the mantissa right by one in $O(m)$ time.

   (c) In the case of subtraction, the result might have several leading zeros. In this case, normalization shifts the mantissa left by a number of the positions correspondingly to the number of leading zeros, and decrements the exponent by the number of leading zeros. We count the leading zeros in $O(mlogm)$ time, and use the logarithmic shifter described in Section 7.4.1, which takes $O(mlogm)$ time.

7. Pack result with truncation towards zero in time $O(1 + e_b + m_b)$.

Extra bits for rounding Our implementation supports the rounding mode round-to-zero, which is almost the same as truncation. However, it is not quite the same. As

noted above, subtraction can create a result with leading zeros, which must be shifted left so that the result is a normalized number. It is not sufficient to simply shift in zeros on the left when normalizing the number. Instead, the subtraction must take account of the lower bits of the number with the smaller exponent. We must track two extra bits of mantissa in the result beyond those bits that are stored in the normal format, to take proper account of these lower bits [Brumley and Page, 2011].

Bitslice Floating-point Multiplication

Our bitslice floating-point multiplication is implemented as the following steps:

1. Compute the sign of the result $S_z = S_x \bigoplus S_y$ in $O(1)$ time.

2. Multiply mantissas. Multiplication of mantissas produces a magnitude P of $2m$ bits. We adopt the unsigned integer multiplier described in Section 7.4.1. This multiplier finishes in time $O(m^2)$.

3. Add exponents. In the biased representation, the addition of exponents is performed as $E_{B,z} = E_{B,x} + E_{B_y} - B$, where B is the bias $(2^{e_b-1} - 1)$. For customizable precision, we compute the bias for a given specification of precision and directly put the value into bitslice vectors for computation. The time complexity is $O(e_b)$.

4. Normalization. Since $1 \leq M_x, M_y < 2$, the result of multiplication is in the range $[1, 4)$. Therefore, it might be necessary to normalize the result by shifting right by one position and incrementing the exponent by one. The cost of normalization is $O(m + e_b)$.

5. Pack result with truncation towards zero in time $O(1 + e_b + m_b)$.

Bitslice Floating-point Division

Our bitslice floating-point division is implemented as the following stages:

1. Compute the sign of the result $S_z = S_x \bigoplus S_y$ in $O(1) time$.

2. Compare and shift. The normalization step depends on the range in the representation of the mantissas. Since $1 \leq M_x, M_y < 2$, the result of division is in the range $(1/2, 2)$. If the the value is less than 1, normalization is required, and is implemented by a left shift of one position and decrement of the exponent. Given $2M_x/M_y \in (1,4) \subset [1,4]$, $2M_x/M_y \cdot 2^{-c} \in [1,2)$ provided c satisfies

$$c = \begin{cases} 0 & \text{if } M_x < M_y \\ 1 & \text{if } M_x \geq M_y \end{cases}$$

   Therefore, to avoid a separate normalization after mantissa division, we first check whether $M_x < M_y$. If $M_x < M_y$, we directly shift M_x left by one position, and set c to 1, which will be used in the exponent subtraction. The cost of comparison is $O(e_b + m_b)$ and a conditional shift by the constant 1 is $O(m)$.

3. Subtract exponents. In the biased representation, the subtraction of exponents is performed as $E_{B,z} = E_{B,x} - E_{B_y} + B$, where B is the bias $2^{e_b - 1} - 1$. When c in Step 2 is 1, it means we need to decrement exponent by one [Muller et al., 2010]. The cost is $O(e_b)$.

4. Divide mantissas. We adopt the *restoring division algorithm* with a complexity of $O(m^2)$, which is the simplest digit-recurrence algorithm [Muller et al., 2010].

5. Pack result with truncation towards zero in time $O(1 + e_b + m_b)$.

7.6 Code Generator and Optimization

7.6.1 Code Generation Framework

Software floating-point libraries (e.g., Berkeley Softfp [Hauser, 2017]) are usually directly implemented in the C/C++ language. This approach works well for the standard floating-point types (FP32 or FP64) because there are only a couple of different types to support. In contrast our BFP vector types can use a fully-customized number of exponent and mantissa bits, so we need to support an unbounded number of BFP precision levels. In addition, we need minimize the number of logic operations required by each

floating-point operation in the specified precision.

Ideally, we would be able to write our floating-point operators in simple C, such as the bitslice adder in Figure 7-2 and let the compiler generate highly optimized code for the particular data size. However, we found that standard compilers do not optimize these operators as much as we need. There are a number of problems:

- Pointer aliasing. Arrays are often passed around with pointers and in general pointer aliasing is a hard problem for compilers.

- Loop unrolling. Most of our operators benefit from full loop unrolling, but compilers will not consistently unroll them.

- Elimination of memory access. We represent our bitslice vector types as arrays, but when operating on them we would like to keep intermediate values in registers. In principle compilers can promote array elements to scalar variables using array scalarization [Bacon et al., 1994][Gao et al., 1993], but current compilers such as GCC and LLVM do not perform this optimization on our code.

- Boolean Expression Optimization. Compilers optimize boolean expressions with boolean identities, such as De Morgan's laws. However, we found that existing compilers do not optimize logical expressions as effectively as specific logic optimization tools.

To overcome these limitations we have constructed a program generator that emits customizable precision BFP operators. Our code generator is a very simple one, but it allows us to generate efficient code for our operators. Our code generation framework takes the implementation of floating-point operations on bitslice vectors with our predefined bitslice vector library. However, when executing the simple C/C++ code we overload the bitwise operators to emit to file a record of each operation as it executes. This gives us a trace of all bitwise operations executed when performing the operator, with loops fully unrolled, calls to sub-operators inlined, and all accesses to intermediate results converted to scalar variables.

This simple tracing approach works because the code to implement our bitslice operators contains no conditional statements. The only control flow is on the number of

loop iterations, which is constant for a given BFP type. The generated file consists of a pure combinational circuit, built from simple boolean operators such as `and`, `or` and `xor`. The resulting logic operations are passed into a logic optimizer to apply advanced logic optimization used in hardware synthesis.

7.6.2 Logic Optimization

For logic optimization, we embed a popular and widely-used synthesis and verification tool – Berkeley ABC [Berkeley Logic Synthesis and Verification Group, 2017] [Brayton and Mishchenko, 2010] – in our code generation framework. ABC is software system for synthesis and verification of binary sequential logic circuits in synchronous hardware designs. ABC includes scalable logic optimization based on and-inverter graphs (AIGs), optimal-delay DAG-based technology mapping for look-up tables and standard cells, and some innovative algorithms for sequential synthesis and verification. Compared to other systems for logic synthesis and optimization (e.g., SIS [Sentovich et al., 1992], VIS – Verification Interacting with Synthesis [Brayton et al., 1996], MVSIS– multivalued SIS), the ABC synthesis tool uses simple data structure — two input ANDs and Inverters, and transforms logic network by rewriting AIGs.

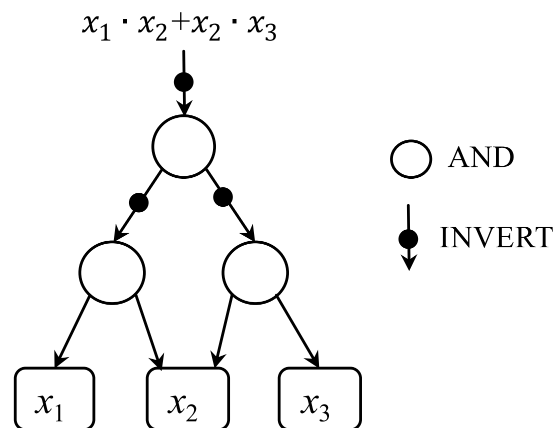


Figure 7-6: AIG representation of boolean network

AIG provides a number of optimization and verification layers, and we use its circuit optimizations for combinatorial logic. For a given boolean network, ABC converts it into an AIG. For example, for the given boolean expression $f = x1 \cdot x2 + x2 \cdot x3$, the equivalent logic expression in terms of ANDs and Inverters is $f = ((x1 \cdot x2)' \cdot (x2 \cdot x3)')'$ and

the corresponding AIG is shown in Fig. 7-6. Using AIG representation, ABC reduces the AIG size by choosing AIG sub-graphs rooted at a node and replacing with pre-computed smaller subgraphs, preserving functionality at root node. AIG rewriting selectively *collapse*s, *refactor*s and *balance*s AIGs with the aim of reducing the number of AIG nodes (logic operations) and number of logic levels (delay) [Mishchenko et al., 2006a]. When AIG rewriting is complete, ABC can map the resulting AIG to a set of hardware "standard cells". These standard cells can be the lookup tables found in FPGAs, or simple boolean gates.

ABC can take a logic network in simple gate-level Verilog as input, and output the mapped results in Verilog. Verilog is a hardware description language (HDL) used to model electronic systems [Thomas and Moorby, 1996]. To integrate ABC into our code generation framework, we need to transform our bitwise bector implementation in C language to gate-level Verilog, and convert the output of ABC in Verilog to C code. Our code generator first executes the implementation of floating-point operations on bitslice vectors to collect all the logical operations. These resulting logic operations are rewritten in Verilog. The inputs and outputs of a floating-point operation are mapped to input and output parameters of a Verilog function. All other intermediate variables are mapped to *wire* variables.

To generate C code from ABC, we define all the logic operations supported by the target processor as standard cells, and put them in a combinational gate specification .genlib file. We use ABC to map the optimized logic network in the form of AIGs to our defined standard cells. We modified the code generation of ABC so that for each logic operation, it emits a C assignment statement with the computation in the form of macros which represents the standard cells. For example, for our 4-bit unsigned integer multiplication, the output of ABC is shown in Fig. 7-7.

In addition to ABC's powerful logic optimization, the specification of standard cells provides an efficient way to utilize more complex bitwise logic instructions provided by the target processor. For the example in Fig. 7-7, `ANDNOT2X1` represents the unique bitwise logical "andnot" instruction in the Intel processor. Given this instruction as a standard cell, with the optimal-delay DAG mapping based on k-feasible cuts, ABC is able to select and generate corresponding C macros. Suppose we use SIMD integer `__m256i`

```
1  BFP_ELEM_TYPE n14, n15, n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27,
      n28,
2     n30, n31, n32, n33, n34, n35, n36, n37, n38, n39, n40, n41, n42, n43,
3     n44;
4    result_0 = AND2X1  (b_0, a_0 );
5    n14 = AND2X1  (b_0, a_1 );
6    n15 = AND2X1  (b_1, a_0 );
7    result_1 = XOR2X1  (n15, n14 );
8    n17 = INVX1   (b_1 );
9    n18 = INVX1   (a_0 );
10   n19 = ANDNOT2X1(n18, n14 );
11   n20 = INVX1   (a_1 );
12   n21 = AND2X1  (b_0, a_2 );
13   n22 = XOR2X1  (n21, n20 );
14   n23 = XOR2X1  (n22, n19 );
15   n24 = OR2X1   (n23, n17 );
16   n25 = ANDNOT2X1(b_1, n21 );
17   n26 = ANDNOT2X1(n25, n24 );
18   n27 = INVX1   (b_2 );
19   n28 = OR2X1   (n27, n18 );
20   result_2 = XOR2X1  (n28, n26 );
21   n30 = ANDNOT2X1(n26, a_0 );
22   n31 = OR2X1   (n30, n20 );
23   n32 = ANDNOT2X1(n31, b_2 );
24   n33 = OR2X1   (n28, n26 );
25   n34 = ANDNOT2X1(n33, n20 );
26   n35 = OR2X1   (n34, n32 );
27   n36 = AND2X1  (b_0, a_3 );
28   n37 = ANDNOT2X1(n14, a_2 );
29   n38 = ANDNOT2X1(a_2, a_1 );
30   n39 = AND2X1  (n38, result_0 );
31   n40 = OR2X1   (n39, n37 );
32   n41 = ANDNOT2X1(n17, n40 );
33   n42 = XOR2X1  (n41, n36 );
34   n43 = XOR2X1  (n42, n35 );
35   n44 = AND2X1  (b_3, a_0 );
36   result_3 = XOR2X1  (n44, n43 );
```

Figure 7-7: Output of ABC logic optimization for a 4-bit unsigned integer multiplication. AND2X1, OR2X1, XOR2X1, ANDNOT2X1, INVX1 are C macros for logic instructions supported by the processor.

in Intel AVX-2 as the BFP_ELEM_TYPE, the code generated by the compiler contains 45 instructions. Without ANDNOT2X1, the generated code will have 49 instructions. The saving of four instructions due to ANDNOT2X1 already can reduce the number of operations for the given simple operation on bitslice vectors by 8%. When it comes to large operations such as floating-point multiplication and division, as demonstrated in Section 7.7, this architecture aware logic optimization and mapping in ABC can yield significant performance improvement.

The output of the logic optimization is a single C function without any control flow statements (e.g., if, for loops), and compiled by the compiler for the target architecture to generate a library. All intermediate values in the C code generated by our modified ABC are stored in scalar local variables, and we rely on the C compiler to allocate these variables to registers or memory locations. Register pressure in our generated code is high, because an intermediate bitslice vector with n bits of precision requires n separate scalar variables. Our code generation framework in this chapter depends on the chosen compiler for efficient register allocation and instruction scheduling.

7.7 Experimental Evaluation

7.7.1 Experiment Methodology

We evaluated the performance of our BFP operations on a Linux platform with an Intel(R) Core(TM) i7-4770 (Haswell) CPU, which supports AVX-2 SIMD instructions. For each test, we compared the performance of operations on bitslice vectors generated from our code generator with and without advanced logic optimization against the equivalent operation on data in standard representations – unsigned integer and single precision floating-point. For both we use Clang/LLVM 3.8 as the back-end compiler with compilation options -std=c99 -march=core-avx2 -O3 -fno-vectorize -fno-slp-vectorize.

7.7.2 Performance of Building Blocks

We first evaluated the performance of three building blocks – unsigned integer addition, subtraction, and multiplication that multiplies two n bit numbers producing a n bit num-

ber. For each building block, we present the performance with input data in a range of bit sizes from 4 to 32. We also evaluate four different underlying machine-word types that are used to construct our software bitslice vectors: `unit32_t`, `unit64_t`, `__m128i` (Intel SSE 128-bit word), and `__m256i` (Intel AVX 256-bit word). The number of lanes in each of our bitslice vectors is determined by the number of bits in the underlying machine word. For example, we use `unit64_t` to construct vectors with sixty-four lanes. To construct a vector of sixty-four lanes with bit size of 4-bits, we need four `unit64_t` values.

To evaluate the relative performance of the different types, we use a loop with a power-of-two number of iterations which performs the operation, such as addition or multiplication, on two input arrays and produces a result array. Our baseline is the amount of time needed by a loop which performs the operation in the scalar 32-bit integer or floating-point implemented natively by the hardware. The performance of each of the bitslice vector types is based on them performing the operation on arrays in bitslice vector format. The bitslice vector types that are based on wider underlying machine types (e.g., `unit64_t` as compared to `unit32_t`) have more bitwise-parallel vector lanes, and so complete the benchmark loop more quickly.

As shown in Figure 7-8 and Figure 7-9, for all bit sizes, unsigned integer addition and subtraction on bitslice vectors with `__m256i` SIMD integer types (256-bit wide) are able to outperform hardware scalar integer addition/subtraction on data in `unit32_t`. The main reason is the large amount of bitwise parallelism available in the `__m256i` type. However, for addition and subtraction even bitslice vectors based on `unit32_t` provide enough bitwise parallelism to be competitive up to 15 bits of precision. For unsigned integer subtraction, the logic optimization in our code generator can deliver slight performance improvement depending on the integer types of bitslice vectors. In contrast, logic optimization significantly improves the performance of unsigned integer multiplication, as depicted in Figure 7-10. However, as the bit size increases, the costs of multiplication on bitslice vectors become prohibitive.

7.7.3 Performance of BFP Operations

We evaluated the performance of each BFP operation with varying precision up to 28 bits. Floating-point formats with 8 bits to 15 bits use an exponent of four bits. For

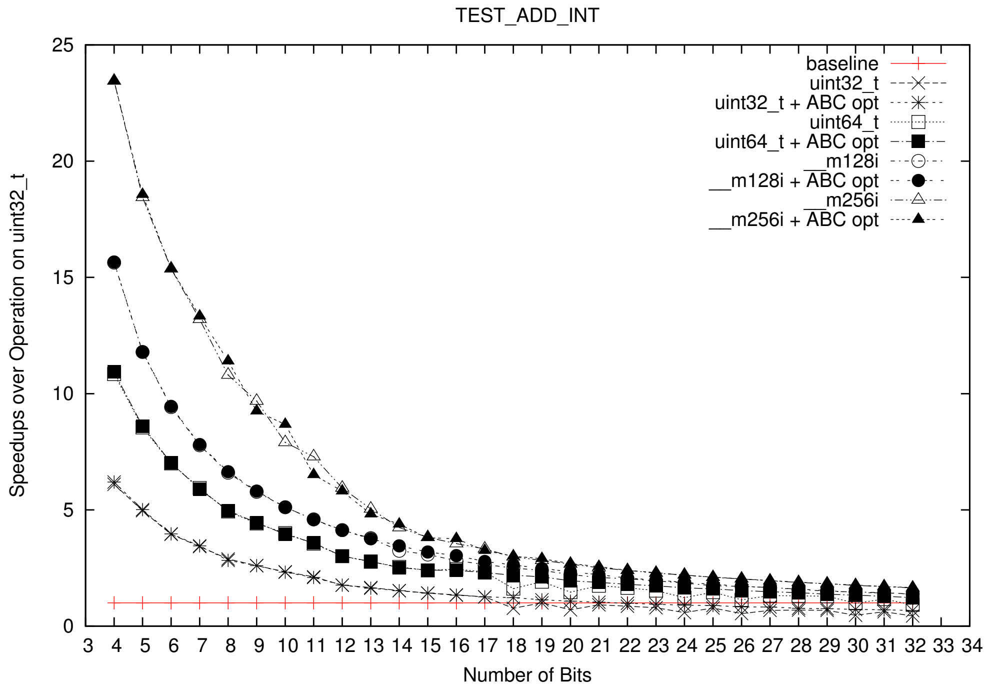


Figure 7-8: Performance of unsigned integer addition with bit sizes from 4 to 32.

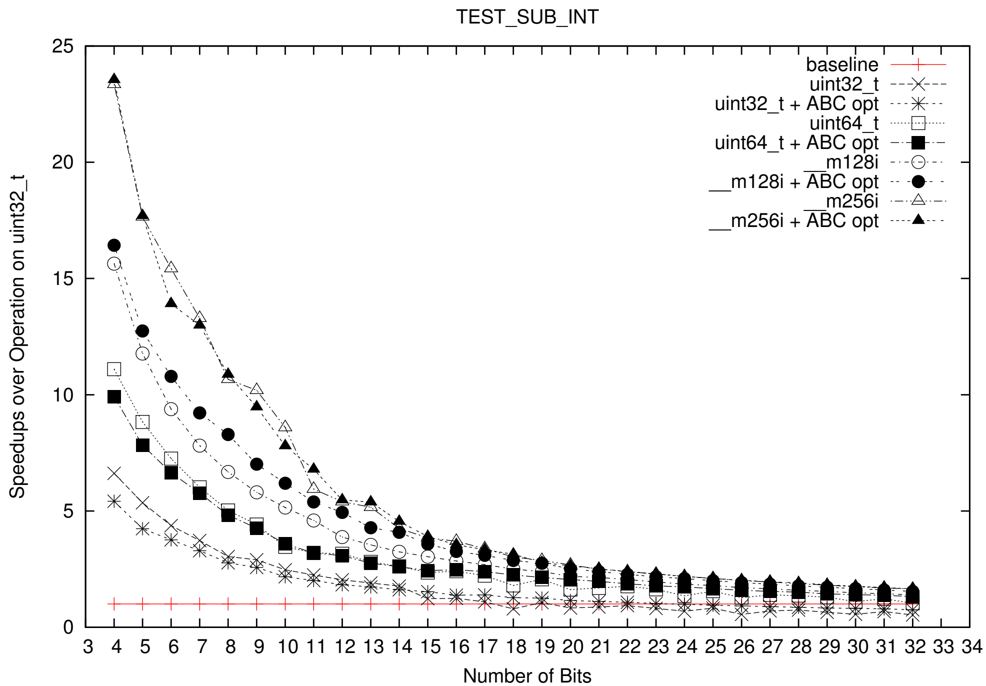


Figure 7-9: Performance of unsigned integer subtraction with bit sizes from 4 to 32.

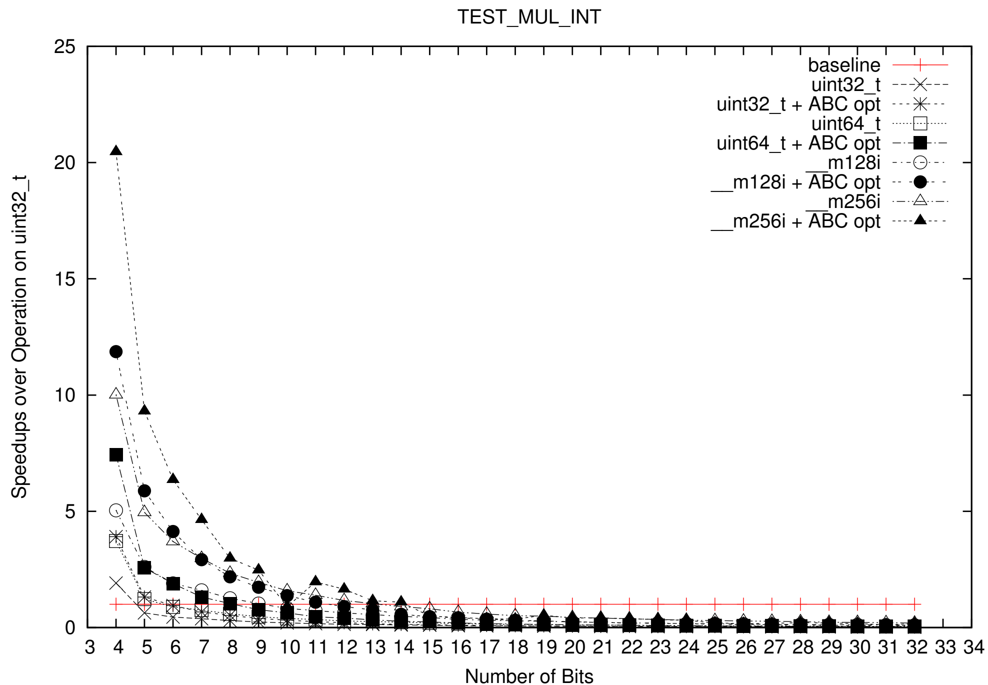


Figure 7-10: Performance of unsigned integer multiplication with bit sizes from 4 to 32.

example, 15-bit floating-point has 1 sign bit, 4 bit exponent bits and 10 mantissa bits. Similarly, from 17-bit to 28-bit floating-point, they all have the same 5-bit exponent as 16-bit floating-point, which is also known as half-precision in the IEEE 754 floating-point standard.

As discussed in Section 7.5, to reduce the cost of handling five types numbers and different round methods defined in IEEE 754 standard, our BFP operations adopt round-to-zero as the rounding method. Figure 7-11 depicts the performance of BFP addition, and Figure 7-12 and Figure 7-13 present the performance of BFP multiplication and division, respectively. As we can see, the performance of our BFP operations is competitive with the scalar single precision floating-point in hardware when the precision is below around 16 bits, and when a sufficiently wide underlying data type is available.

Our results show that bitslice vectors are not competitive when the precision becomes higher. Nonetheless, we believe that bitslice vectors fill an important niche in the representation of numeric data. The standard types provided by general-purpose processors provide a very limited range of sizes and precisions. In contrast, bitslice vectors provide a solution that is effective for a case that standard representations do not deal well with: customized numeric types with low precision. In particular, our BFP operations

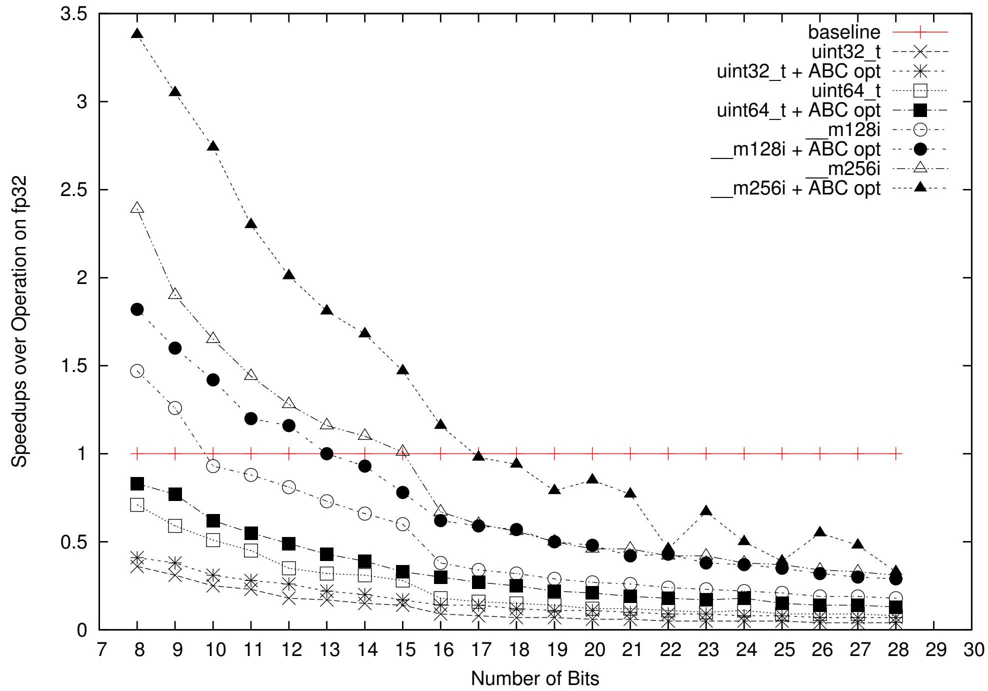


Figure 7-11: Performance of bitslice floating-point addition/subtraction with bit sizes from 8 to 28.

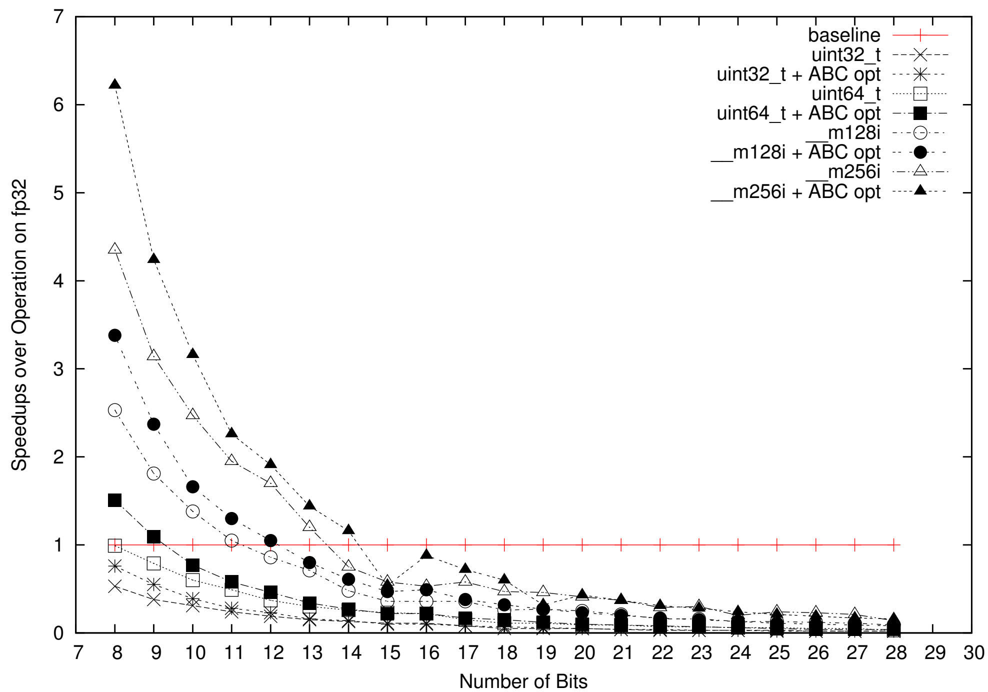


Figure 7-12: Performance of bitslice floating-point multiplication with bit sizes from 8 to 28.

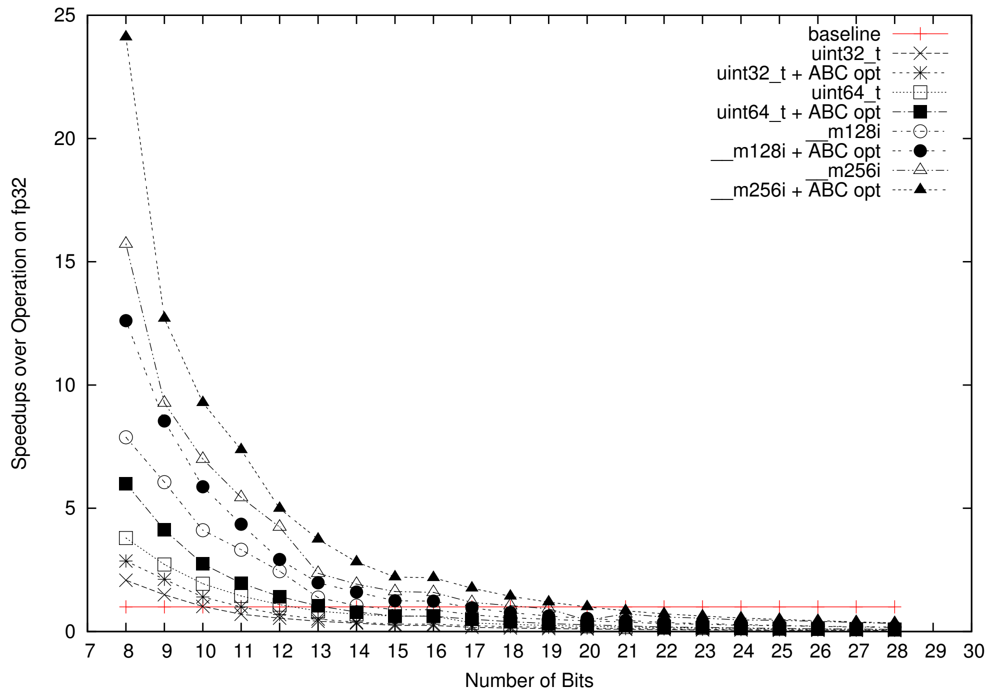


Figure 7-13: Performance of bitslice floating-point division with bit sizes from 8 to 28.

can be used in approximate computing to explore the precision required by the application without resorting to any dedicated hardware. Indeed one important use case for bitslice vectors may be emulating non-standard precisions for the design of approximate computing accelerators.

The width of integer types used for bitslice vectors plays a critical role in the performance of BFP operations. Because a wider integer data type such as __m256i in Intel AVX-2 with 256 bits allows more data items being processed in parallel by a BFP operation. Processors from supercomputers to embedded systems are inclined to have wider and wider SIMD vectors. For example, Intel Xeon PHI coprocessors give support for 64 way 8-bit SIMD vectors. This architecture design trend of wider SIMD vectors will translate into improved bitslice vector performance with customizable precision, alongside the performance improvements to standard precision operations with wider vectors.

7.7.4 Performance of Real-world Applications

We evaluated the combinations of BFP operations in four real-world computational kernels as follows:

- BLAS-1 operations:
  - **xSCALE**, which computes $x \leftarrow \alpha x$;
  - **xAXPY**, which computes $y \leftarrow \alpha x + y$;
- BLAS-2 operation: **xGEMV**, the matrix $\times$ vector multiplication, which calculates $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{y}$;
- Stencil Computation: **One-dimensional (1D) Blur**, which performs a 3-point stencil computation, $Y[i] = (X[i] + X[i+1] + X[x+2])/3$.

We present the performance of each computational kernel in BFP operations with bit sizes between 8 and 16 bits. The BFP operations used in each application are optimized by the advanced logic optimization in our code generator.

Figure 7-14 depicts the performance of xSCALE. This kernel performs a BFP multiplication but with a loop invariant scalar coefficient. Compared to a single BFP multiplication, the performance of xSCALE includes the cost of broadcasting a scalar value into a bitslice vector. Despite the cost of on-the-fly data reorganization, xSCALE in BFP operations still can achieve a speedup of up to 4.8 $\times$.

The performance of BLAS-1 xAXPY is illustrated in Figure 7-15. xAXPY combines a BFP multiplication and a addition. Unlike xSCALE, we transformed the scalar coefficient into a bitslice vector ahead of execution so that there is no extra overhead due to on-the-fly data transformation.

Figure 7-16 presents the performance of BLAS-2 xGEMV. The input $n \times n$ matrix $\mathbf{A}$ in column-major is represented by n bitslice vectors, where n is the size of the underlying machine type used in the bitslice vectors; the input vector $\mathbf{x}$ has n elements organized in a bitslice vector. xGEMV involves a addition reduction operation. Since we do not support BFP horizontal addition, each element of $\mathbf{x}$ is first duplicated across a bitslice vector and then is multiplied by a row of $\mathbf{A}$, which is also a bitslice vector. The result of multiplication is accumulated to a bitslice vector $\mathbf{y}$. The performance shown contains the cost of broadcasting a single data item in a bitslice vector.

The performance of 1D Blur is shown in Figure 7-17. It is not trivial to perform stencil computation on bitslice vectors because the computation is not a pure "vertical" vector

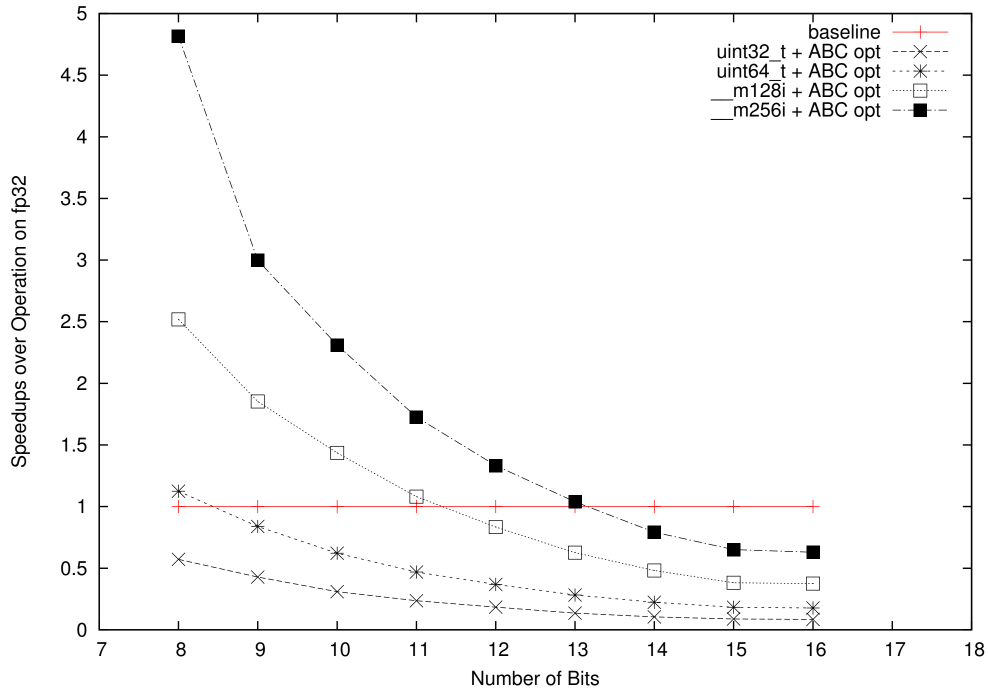


Figure 7-14: Performance of BLAS-1 xSCALE in bitslice floating-point operations with bit sizes from 8 to 16.

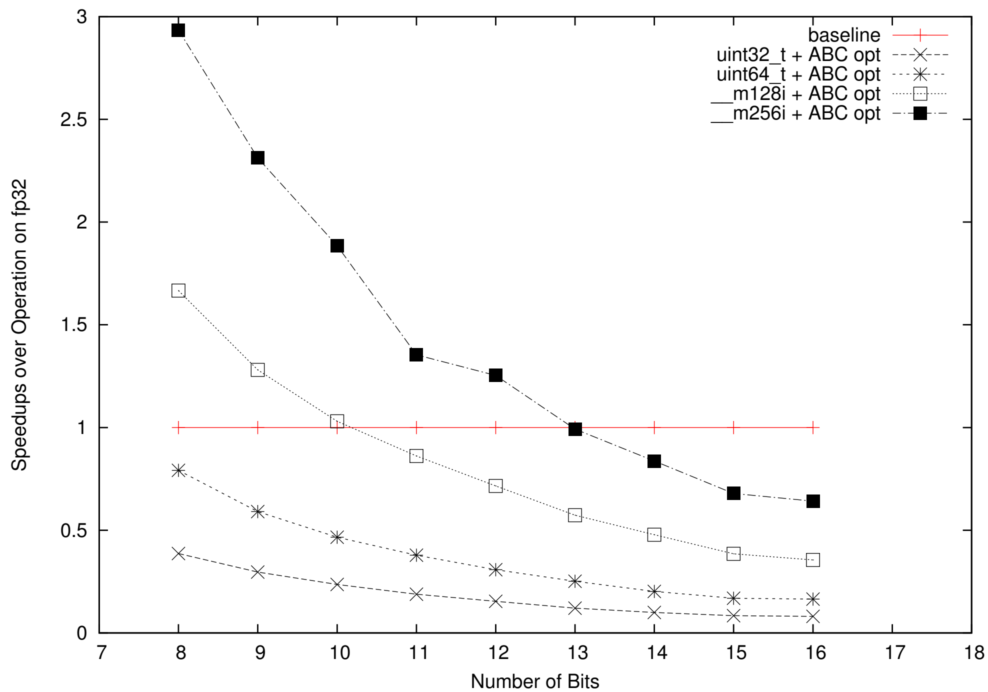


Figure 7-15: Performance of BLAS-1 xAXPY in bitslice floating-point operations with bit sizes from 8 to 16.

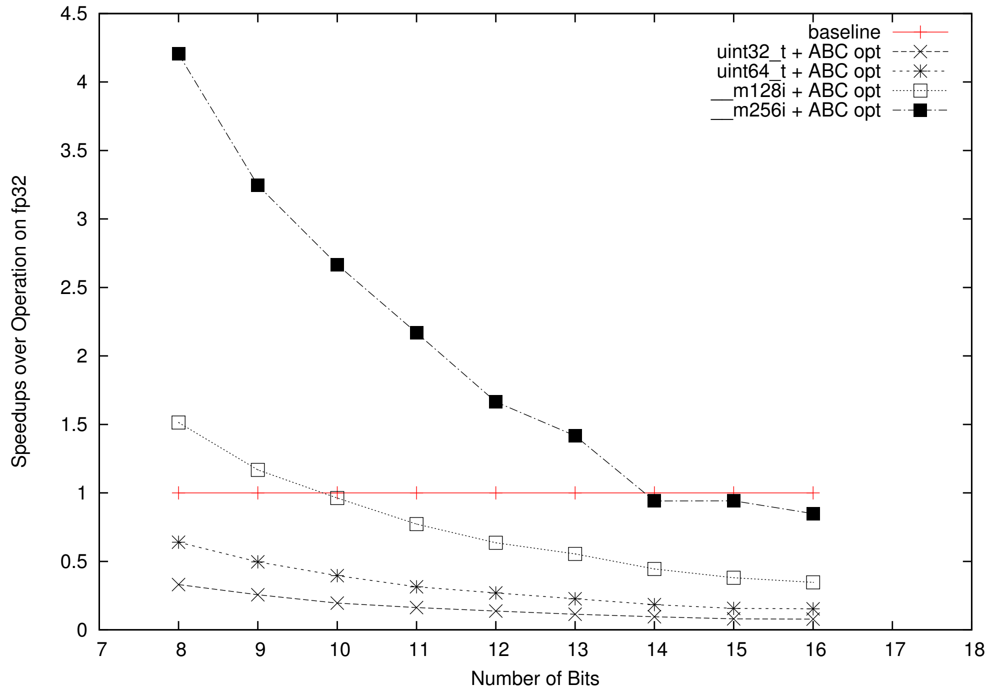


Figure 7-16: Performance of BLAS-2 xGEMV where $y = 0$ in bitslice floating-point operations with bit sizes from 8 to 16.

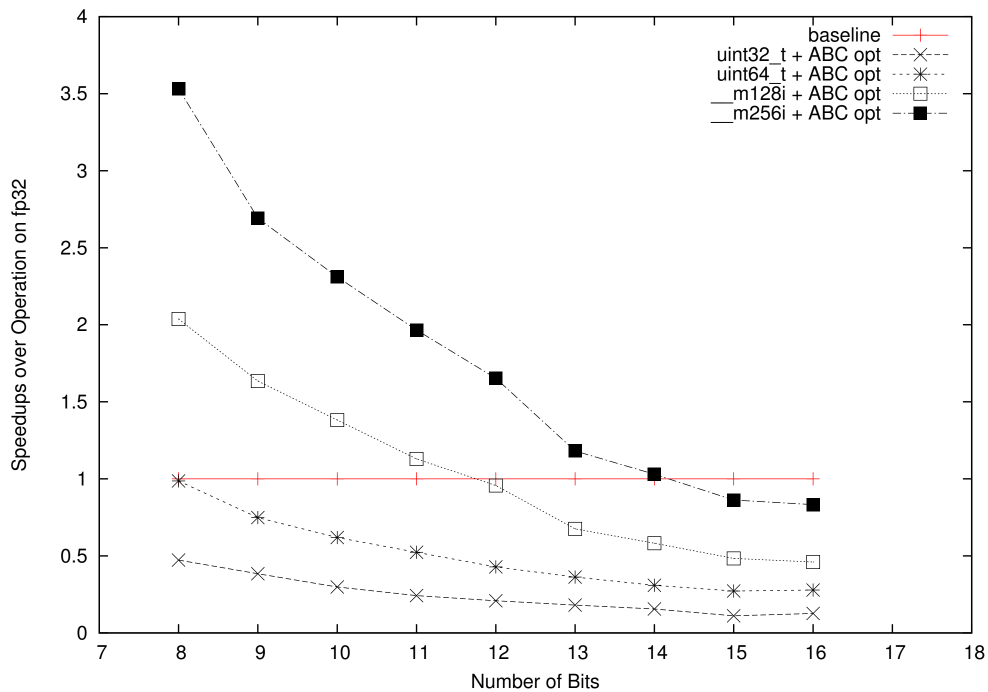


Figure 7-17: Performance of 1D Blur in bitslice floating-point operations with bit sizes from 8 to 16.

operation and instead requires values in the neighborhood. To solve this problem, we apply *data realignment* optimization on bitslice vectors. Data realignment is widely adopted by automatic vectorization in compilers [Nuzman and Zaks, 2008]. To form a bitslice vector for $X[j+1{:}j+1+\text{sizeof(BFP_ELEM_TYPE)}]$, where j is a multiple of sizeof(BFP_ELEM_TYPE), we shift the bitslice vector B_0 representing $X[j{:}j+\text{sizeof(BFP_ELEM_TYPE)}]$ right by one bit, and then blend the first bit of the bitslice vector B_1 for $X[j+\text{sizeof(BFP_ELEM_TYPE)}{:}j+2\times\text{sizeof(BFP_ELEM_TYPE)}]$ at the last bit of B_0. The performance shown includes the cost of data realignment. Despite this kind of costly data transformation, our BFP operations are still able to deliver significant performance for small precision.

7.8 Related Work

The closest work to our bitslice vector computing is bit-serial parallel processing systems. Examples of such systems include the MPP [Batcher, 1980], DAP [Active Memory Technology Inc., 1988] and CM-2 [Thinking Machine Corp., 1989][Hillis, 1992]. These SIMD architectures are built using one-bit processors, which is designed for processing data stream bit by bit [Batcher, 1982]. Smitely and Iobst investigated bit-serial SIMD programming and related optimization techniques on both CM-2 and the vector parallel processor Cray-2 [Smitley and Iobst, 1991]. In contrast, our work takes advantage of the wide machine words available on modern general-purpose processors with SIMD extensions. To the best of our knowledge, we are the first proposing a software approach to customizable data precision using bitslice representations and related compiler optimizations on general-purpose processors with SIMD extensions.

Fisher et al. propose a model of sub-word bitwise parallelism which they refer to as SIMD within a register (SWAR) [Fisher and Dietz, 1998] . For example, three 10 bit integers can be packed into a single 32-bit integer. With appropriate use of explicit spacer bits, or with additional operations, it is possible to simultaneously operate on all three of the 10-bit subword values using 32-bit operations. This approach works well for integer addition and simple bitwise operations, but less well for more complex integer operations and cannot be used to implementing floating-point operatoions. An

advantage of the SWAR approach is that it can make use of existing hardware circuits, at least for addition and subtraction. In contrast, we build our circuits entirely from scratch using bitwise logical operators, which allows us to implement arbitrary circuits but makes less use of existing hardware.

There are various software implementations of the IEEE 754 standard for binary floating-point arithmetic, such as SoftFloat [Hauser, 2017] and FLIP [Jeannerod et al., 2010]. FLIP is a C library that provides a software support for binary-32 floating-point arithmetic on integer processors. Compared to SoftFloat, FLIP is particularly targeted at DSP processors, and has been validated on VLIW integer processors (e.g., the ST200 family from STMicroelectronics). However, both libraries are only capable of performing fixed-precision floating-point arithmetic such as 32-bit single-precision and 64-bit double-precision. In contrast, our BFP operations support customizable bit precision. FloPoCo is a generator of customizable precision floating-point arithmetic cores for FPGAs [de Dinechin and Pasca, 2011]. Our BFP operations provide arbitrary precision but using a software approach.

Some programs may not need the dynamic range or the precision of FP arithmetic. For these programs, it is a general design practice to translate the floating-point arithmetic into a suitable finite fixed point presentation [Tong et al., 2000]. However, some programs may still require 6 bits or more in the exponent to preserve a reasonable degree of accuracy. In other words, these applications need more than the typical 32 bits of precision that fixed point arithmetic offers. Therefore, support for small, irregularly sized floating-point makes our bitslice vector types a perfect fit for this kind of application.

We support just enough precision by providing vector types with exactly the right number of bits. Existing research on FPGAs shows that even greater benefits from data type customization are possible. For example, when implementing a multiply-accumulate unit, performing the multiply in floating-point and the addition in a wide fixed-point type can improve accuracy while reducing hardware cost [de Dinechin et al., 2008]. Exactly the same approach can be implemented with our custom bitslice vector types.

Lowering energy consumption is one of the major benefits of approximate computing. Disciplined approximate programming asks programmers to specify which parts of a

program can be computed approximately. The approximate computation thus reduces the energy cost. An ISA extension is put forward to provide approximate operations and storage [Esmaeilzadeh et al., 2012]. With this extension, hardware has freedom to save energy at the cost of accuracy. Our customizable precision BFP vector types and related operations can serve as a software ISA for approximate computation.

Existing research on approximate computing has proposed a large variety of circuits for approximate operators, which compute an approximation of an operator such as multiplication [Shao and Li, 2014]. These approximate operators significantly reduce the number of gates needed for multiplication or addition [Liu et al., 2014]. A promising area of future research is to investigate the implementation of approximate arithmetic operators in bitslice vector format.

The *bit plane slicing* [Cho et al., 2005] image format stores images in bitslice format, where the most significant bit of each pixel is stored in one group, followed by the next most significant bit of each pixel. When an image is transferred over a communications link using bit plane format, the most significant bits are sent first. This allows an increasingly detailed approximation of the image to be displayed, as successive bits of each pixel are transferred. Our bitslice vector operators offer a mechanism for computing directly on bit plane format images without converting back to a standard format.

7.9 Summary

In this chapter, we propose an new approach to vector computing based on bitslice vector formats and building arithmetic operators from bitwise instructions on general-purpose processors with SIMD extensions. This approach allows us to support a vector processing model that can operate on data with an arbitrary number of bits. Thus, we can create vectors of integer or floating-point types of five, nine, eleven or any number of bits. This ability to customize the precision of vector data exactly to the application creates new opportunities for optimization. In particular, it allows data precision optimizations on general-purpose processors with SIMD extensions that were previously available primarily on custom hardware. In addition, matching precision to the application may reduce the memory footprint of applications, which may in turn reduce memory traffic and the

energy required for data movement [Dally et al., 2008].

The complexity of the arithmetic operators is related to the number of bits of precision in the data types. On the other hand, all our bitslice vector types benefit from significant bit-level parallelism, as the same bitwise operation is applied in parallel to all 32, 64 or 128 lanes of the vector. Our experiments show that for larger precision, the costs of arithmetic operators become prohibitive. However, for smaller data types the benefits of exploiting bitwise parallelism across a vector of values can outweigh the costs of bitwise arithmetic. To our knowledge we are the first to propose and evaluate general-purpose bitslice vector representations on general-purpose processors with SIMD extensions. We believe that it is a promising approach for approximate computing using just enough precision.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Conclusion and Final Thoughts

In this thesis, we have demonstrated several data-layout oriented compilation techniques for efficient vectorization. Our semi-automatic data layout transformation in Chapter 3 can help users to easily change their program, and exploit the best possible data layout in terms of vectorization. The vectorization based on hyper loop parallelism in Chapter 4 provides a way to take advantage the relationship between data layout and computation structure. The experimental results demonstrated that this vectorization technique can yield significant performance gain. In addition, this technique is of great use to boost the memory performance on CUDA GPUs as demonstrated in Chapter 5.

We showed pioneering work that uses classic loop vectorization techniques to deal with nested thread-level parallelism on CUDA GPUs in Chapter 6. As loop vectorization prioritizes vectorizing loops with contiguous memory access patterns, it is of great help to achieve efficient mapping between loop parallelism and the deep execution hierarchy of CUDA GPUs.

The last but not the least work is our new bitslice vector computing for customizable arithmetic precision on general-purpose processors with SIMD extensions in Chapter 7. Our proposed bitslice vector computing not only breaks the limit of hardware arithmetic precision but also achieves great performance. It demonstrates the great power of logic optimization widely used in hardware synthesis in optimizing C/C++ code with a large amount of logic operations.

8.1 Future Work

Despite the results achieved by our proposed data layout oriented compilation techniques, there are several directions for future work.

8.1.1 Integrate Semi-automatic Data Layout Transformation into Performance Auto-tuning Systems

One possible extension to our semi-automatic data layout transformation is to integrate our compiler directive based method into an existing program transformation framework (e.g., POET [Nesterenko et al., 2015]) . Despite a large variety of advanced compiler optimizations in modern compilers, it is of great difficulty to have a single sequence of optimizations that can achieve good performance for any kinds of applications. This is because compiler can only perform conservative optimizations without sufficient understanding of the input code.

To alleviate the difficulty, two general approaches are widely adopted: 1) we can keep the sequence of optimizations fixed in the compiler, but try to figure out a combination of optimizations that yield the best performance for the given program with the help of a performance auto-tuning system; 2) we can annotate the program with some directives to direct the compiler to perform the optimizations as we specified. For example, POET is a program transform system allowing user to write scripts to control the desired sequence of loop transformations. The compiler directives used in our semi-automatic data layout transformation can be easily integrated into such program transformation systems. This gives another alternative when search for the best-performing solutions.

8.1.2 Seamless Data Layout Transformations for C++ Code

When writing C++ code with the object-oriented design, it is more intuitive to model a real world entity as an object that are represented by user defined data types like struct or class. A collection of such entities will become an array of structures that are not friendly to vectorization. Efficient vectorization requires changes not only to the data structures, which breaks the object-oriented design, but also to the existing C++ algorithms. Intel SIMD data layout template is a possible solution to this problem [Nimisha R., 2016]. It

uses a special C++ template class as a replacement for C++ arrays, and the template class allows different possible data layouts. However, it can only handle a small set of data layout transformations. Our data layout transformation directives are capable of expressing different forms of data layout. It would be a nice extension to our semi-automatic data layout transformation if we can support seamless data layout transformation for C++ code.

8.1.3 A Source-to-source Vectorizing Compiler for Bitslice Vectors

In this thesis, we put forward bitslice vector types and compose arithmetic operators with primitive logic operations on bitslice vectors. But it is tedious and error-prone for users to explicitly transform their existing code into bitslice vector operations. One possible extension to our current work is to provide a source-to-source vectorizing compiler. This compiler would not only vectorize loops with arithmetic operators composed of bitslice vector operations, but would also perform necessary data layout transformation from conventional array layout to bitslice vectors. This would greatly simplify the programming of bitsclice vectors. Thus, it would allow researchers to more easily evaluate bitslice vectors on a variety of approximate computing problems.

8.1.4 Exploit More Logic Optimization for Bitslice Vector Operators

With the advance of instruction sets in modern processors, more advanced logic operations have become available to perform complex computations. For example, Intel AVX512 supports user-defined ternary bitwise operators based on look-up tables (LUT) [Intel, 2016]. As the loop-up table can be used to describe different patterns of ternary logic operations, it is not obvious how to automatically take advantage of such instructions in compilers. In order to use such instructions for our bitslice vector operations, we can define all the possible cells with ternary bitwise logical operators, and use ABC's standard cell mapping techniques to map an optimized logic graph built with and-inverter graphs to such cells.

We can also utilize the results in the form of LUTs from ABC. For instance, for the 4-bit integer multiplication, with a 3-LUT mapping strategy, the output of ABC logic

```
1   BFP_DATA_AVX_TYPE n16, n18, n19, n20, n21, n22, n24, n25, n26, n27, n28, n29,
        n30, n31;
2   assign n31 = a_0 & b_3;
3   assign n30 = a_0 & (n18 | n22);
4   assign n29 = (~b_2 | ~n30 | a_1) & (~b_2 | n30 | ~a_1);
5   assign n28 = a_3 & b_0;
6   assign n27 = a_1 & a_2 & b_0;
7   assign n26 = ~n27 & (n19 | ~n20);
8   assign n25 = ~n26 ^ (n28 ^ ~a_2);
9   assign n24 = b_1 ? n25 : ~n28;
10  assign result_3 = n31 ^ (n24 ^ n29);
11  assign n22 = a_2 & b_0 & ~b_1;
12  assign n21 = a_0 & b_2;
13  assign n20 = a_0 & a_1 & b_0;
14  assign n19 = a_1 ^ (~a_2 | ~b_0);
15  assign n18 = b_1 & (n19 | ~n20) & (~n19 | n20);
16  assign result_2 = n21 ^ (n18 | n22);
17  assign n16 = a_1 & b_0;
18  assign result_1 = ~n16 ^ (~a_0 | ~b_1);
19  assign result_0 = a_0 & b_0;
```

Figure 8-1: The customized output from ABC logic optimizer for the 4-bit integer multiplication with a 3-LUT mapping strategy.

optimizer is illustrated in Fig. 8-1. In this case, we do not need define all the possible cells in advance. We can instead simply translate each LUT to a ternary logic instruction. As there are several LUT mapping techniques [Mishchenko et al., 2006b], we need evaluate and choose the best mapping method suitable for our bitslice vector operations.

8.2 Final Thoughts

In this thesis, we discussed several compilation techniques and a programming method bitslice vectors to take advantage of the characteristics of different data layouts for better utilization of the underlying SIMD instructions. But data layout is only one of the factors that affects the efficiency of automatic vectorization techniques in compilers.

There has been extensive work on improving both loop vectorization and super-word level parallelism vectorization to accommodate new SIMD instructions in modern processors. Maleki et al. reviewed the capabilities of several vectorizing compilers , and summarized the solutions to the difficulties they encountered [Maleki et al., 2011]. However, the advance of SIMD instructions imposes more challenges on vectorization techniques.

One of the great challenges comes from the increasingly wider SIMD widths. It is not trivial to utilize wider SIMD instructions because it demands more SIMD parallelism to feed the instruction. Take loop vectorization as an example; when the SIMD width gets wider, if the old SIMD width is not available anymore, existing loops with small loop trip counts may not be efficiently vectorized or not vectorized at all. Masked SIMD instructions are getting more and more popular in modern processors. Such instructions in fact can give us a better way to vectorize the small loops with wider SIMD instructions. However, none of the existing open-source compiler can so far perform such vectorization.

Masked SIMD instructions can also be of great use to vectorize scalar wind-down loops from loop vectorization, because sometimes the loop trip count is only 1 or 2 smaller than the SIMD width, still revealing enough SIMD parallelism. For example, if the SIMD width is 2048, the scalar wind-down loop may have 2047 iterations. In this case, vectorizing the loop with masked instructions should be better than either executing it as a scalar loop or vectorizing it with a smaller SIMD width plus a scalar wind-down loop.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[Acken et al., 1996] Acken, K., Irwin, M., and Owens, R. (1996). Power Comparisons for Barrel Shifters. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, ISLPED '96, pages 209–212.

[Active Memory Technology Inc., 1988] Active Memory Technology Inc. (1988). AMT DAP series technical overview.

[Anderson et al., 2015] Anderson, A., Malik, A., and Gregg, D. (2015). Automatic vectorization of interleaved data revisited. *ACM Trans. Archit. Code Optim.*, 12(4):50:1–50:25.

[Bacon et al., 1994] Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler Transformations for High-performance Computing. *ACM Comput. Surv.*, 26(4):345–420.

[Bae et al., 2013] Bae, H., Mustafa, D., Lee, J.-W., Aurangzeb, Lin, H., Dave, C., Eigenmann, R., and Midkiff, S. P. (2013). The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *Int. J. Parallel Program.*, 41(6):753–767.

[Bailey et al., 1991a] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991a). The NAS Parallel Benchmarks — Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA. ACM.

[Bailey et al., 1991b] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991b). The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications.

[Barik et al., 2010] Barik, R., Zhao, J., and Sarkar, V. (2010). Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 201–212, Washington, DC, USA. IEEE Computer Society.

[Bastoul, 2004] Bastoul, C. (2004). Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA. IEEE Computer Society.

[Batcher, 1982] Batcher, K. (1982). Bit-serial parallel processing systems. *IEEE Transactions on Computers*, 31(5):377–384.

[Batcher, 1980] Batcher, K. E. (1980). Design of a massively parallel processor. *IEEE Transactions on Computers*, C-29(9):836–840.

[Bauer et al., 2014] Bauer, M., Treichler, S., and Aiken, A. (2014). Singe: Leveraging Warp Specialization for High Performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14.

[Berkeley Logic Synthesis and Verification Group, 2017] Berkeley Logic Synthesis and Verification Group (2017). ABC: A System for Sequential Synthesis and Verification.

[Bertolli et al., 2014] Bertolli, C., Antao, S. F., Eichenberger, A. E., O'Brien, K., Sura, Z., Jacob, A. C., Chen, T., and Sallenave, O. (2014). Coordinating gpu threads for openmp 4.0 in llvm. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21.

[Biham, 1997] Biham, E. (1997). A fast new DES implementation in software. In Biham, E., editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer Berlin Heidelberg.

[Bocchino and Adve, 2006] Bocchino, Jr., R. L. and Adve, V. S. (2006). Vector LLVA: A Virtual Vector Instruction Set for Media Processing. In *the 2006 International Conference on Virtual Execution Environments*.

[Brayton and Mishchenko, 2010] Brayton, R. and Mishchenko, A. (2010). *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, chapter ABC: An Academic Industrial-Strength Verification Tool, pages 24–40. Springer Berlin Heidelberg.

[Brayton et al., 1996] Brayton, R. K., Hachtel, G. D., Sangiovanni-Vincentelli, A. L., Somenzi, F., Aziz, A., Cheng, S., Edwards, S. A., Khatri, S. P., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R. K., Sarwary, S., Shiple, T. R., Swamy, G., and Villa, T. (1996). VIS: A system for verification and synthesis. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, pages 428–432.

[Brumley and Page, 2011] Brumley, B. and Page, D. (2011). Bit-sliced binary normal basis multiplication. In *2011 20th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 205–212.

[Che et al., 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54.

[Che et al., 2011] Che, S., Sheaffer, J. W., and Skadron, K. (2011). Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11.

[Cho et al., 2005] Cho, C.-Y., Chen, H.-S., and Wang, J.-S. (2005). Smooth quality streaming with bit-plane labeling. In Li, S., Pereira, F., Shum, H.-Y., and Tescher, A. G., editors, *Visual Communications and Image Processing 2005*, volume 5960, pages 2184–2195.

[Cypher and Sanz, 1989] Cypher, R. and Sanz, J. L. C. (1989). SIMD architectures and algorithms for image processing and computer vision. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(12):2158–2174.

[Dally et al., 2008] Dally, W. J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R. C., Parikh, V., Park, J., and Sheffield, D. (2008). Efficient embedded computing. *Computer*, 41(7):27–32.

[de Dinechin et al., 2008] de Dinechin, F., Detrey, J., Cret, O., and Tudoran, R. (2008). When FPGAs Are Better at Floating-point Than Microprocessors. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 260–260.

[de Dinechin and Pasca, 2011] de Dinechin, F. and Pasca, B. (2011). Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27.

[Dibyendu Das, 2012] Dibyendu Das, Soham Sundar Chakraborty, M. L. (2012). Experience with Partial SIMDization in Open64 Compiler Using Dynamic Programming. In *Open64 Workshop*.

[Duan and Yang, 2016] Duan, J. and Yang, Y. (2016). Efficient virtual network embedding for variable size virtual machines in fat-tree data centers. In *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*, pages 1–10.

[Duncan, 1990] Duncan, R. (1990). A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16.

[Eichenberger et al., 2004] Eichenberger, A. E., Wu, P., and O'Brien, K. (2004). Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04.

[Ercegovac and Lang, 2004] Ercegovac, M. D. and Lang, T. (2004). *Digital Arithmetic*. Morgan Kaufmann Oxford, San Francisco (Calif.).

[Esmaeilzadeh et al., 2012] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. (2012). Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*.

[Fisher and Dietz, 1998] Fisher, R. J. and Dietz, H. G. (1998). Compiling for SIMD Within a Register. In *11th International Languages and Compilers for Parallel Computing Workshop (LCPC'08)*.

[Flynn, 1972] Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, 21(9):948–960.

[Gao et al., 1993] Gao, G., Olsen, R., Sarkar, V., and Thekkath, R. (1993). *Languages and Compilers for Parallel Computing: 5th International Workshop New Haven, Connecticut, USA, August 3–5, 1992 Proceedings*, chapter Collective loop fusion for array contraction, pages 281–295. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Geva, 2011] Geva, R. (2011). Elemental functions: Writing data-parallel code in C/C++ using Intel Cilk Plus.

[Girbal et al., 2006] Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., and Temam, O. (2006). Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Int. J. Parallel Program.*, 34(3):261–317.

[Grosser et al., 2012] Grosser, T., Größlinger, A., and Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4).

[Han and Abdelrahman, 2009] Han, T. D. and Abdelrahman, T. S. (2009). hiCUDA: A High-level Directive-based Language for GPU Programming. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2.

[Hauser, 2017] Hauser, J. (2017). The SoftFloat and TestFloat packages.

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

[Henretty et al., 2011] Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., and Sadayappan, P. (2011). *Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures*, pages 225–245. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Hillis, 1992] Hillis, W. D. (1992). *The Connection Machine*. MIT Press.

[Hong et al., 2011] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. (2011). Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11.

[Intel, 2016] Intel (2016). Intel Architecture Instruction Set Extensions Programming Reference.

[Ira Rosen and Zaks, 2007] Ira Rosen, D. N. and Zaks, A. (2007). Loop-aware SLP in GCC. In *Proceedings of GCC Developers Summit (GCC Developers Summit07)*.

[Jang et al., 2010] Jang, B., Mistry, P., Schaa, D., Dominguez, R., and Kaeli, D. (2010). Data Transformations Enabling Loop Vectorization on Multithreaded Data Parallel Architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 353–354, New York, NY, USA. ACM.

[Jang et al., 2011a] Jang, B., Schaa, D., Mistry, P., and Kaeli, D. (2011a). Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1).

[Jang et al., 2011b] Jang, B., Schaa, D., Mistry, P., and Kaeli, D. (2011b). Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118.

[Jeannerod et al., 2010] Jeannerod, C.-P., Mouilleron, C., Muller, J.-M., Revy, G., Bertin, C., Jourdan-Lu, J., Knochel, H., and Monat, C. (2010). Techniques and Tools for Implementing IEEE 754 Floating-point Arithmetic on VLIW Integer Processors. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 1–9.

[Karrenberg and Hack, 2011] Karrenberg, R. and Hack, S. (2011). Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 141–150, Washington, DC, USA. IEEE Computer Society.

[Kennedy and Allen, 2002] Kennedy, K. and Allen, J. R. (2002). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc.

[Kennedy and Kremer, 1998] Kennedy, K. and Kremer, U. (1998). Automatic Data Layout for Distributed-memory Machines. *ACM Trans. Program. Lang. Syst.*, 20(4):869–916.

[Kerr et al., 2012] Kerr, A., Diamos, G., and Yalamanchili, S. (2012). Dynamic Compilation of Data-parallel Kernels for Vector Processors. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12.

[Kim et al., 2015] Kim, H.-S., El Hajj, I., Stratton, J., Lumetta, S., and Hwu, W.-M. (2015). Locality-centric Thread Scheduling for Bulk-synchronous Programming Models on CPU Architectures. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 257–268.

[Kim and Han, 2012] Kim, S. and Han, H. (2012). Efficient SIMD Code Generation for Irregular Kernels. In *the 2012 Symposium on Principles and Practice of Parallel Programming*, PPoPP '12.

[Kohn et al., 1995] Kohn, L., Maturana, G., Tremblay, M., Prabhu, A., and Zyner, G. (1995). The visual instruction set (VIS) in UltraSPARC. In *Compcon '95.'Technologies for the Information Superhighway', Digest of Papers.*, pages 462–469.

[Kong et al., 2013] Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.-N., and Sadayappan, P. (2013). When Polyhedral Transformations Meet SIMD Code Generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA. ACM.

[Larsen and Amarasinghe, 2000] Larsen, S. and Amarasinghe, S. (2000). Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *the 2000 Conference on Programming Language Design and Implementation*, PLDI '00.

[Larsen et al., 2005] Larsen, S., Rabbah, R., and Amarasinghe, S. (2005). Exploiting vector parallelism in software pipelined loops. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 119–129, Washington, DC, USA. IEEE Computer Society.

[Lee et al., 2014] Lee, H., Brown, K. J., Sujeeth, A. K., Rompf, T., and Olukotun, K. (2014). Locality-Aware Mapping of Nested Parallel Patterns on GPUs. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 63–74.

[Lee, 1995] Lee, R. B. (1995). Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32.

[Lee and Eigenmann, 2010] Lee, S. and Eigenmann, R. (2010). OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA. IEEE Computer Society.

[Lee et al., 2009] Lee, S., Min, S.-J., and Eigenmann, R. (2009). OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110.

[Lee and Vetter, 2014] Lee, S. and Vetter, J. S. (2014). OpenARC: Extensible OpenACC Compiler Framework for Directive-based Accelerator Programming Study. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 1–11.

[Liu et al., 2014] Liu, C., Han, J., and Lombardi, F. (2014). A Low-power, High-performance Approximate Multiplier with Configurable Partial Error Recovery. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 95:1–95:4.

[Liu et al., 2012] Liu, J., Zhang, Y., Jang, O., Ding, W., and Kandemir, M. (2012). A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 347–358, New York, NY, USA. ACM.

[Majeti et al., 2014] Majeti, D., Barik, R., Zhao, J., Grossman, M., and Sarkar, V. (2014). Compiler-Driven Data Layout Transformation for Heterogeneous Platforms. In *Euro-Par 2013: Parallel Processing Workshops*. Springer Berlin Heidelberg.

[Maleki et al., 2011] Maleki, S., Gao, Y., Garzarán, M. J., Wong, T., and Padua, D. A. (2011). An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA. IEEE Computer Society.

[Melax, 2012] Melax, S. (2012). 3D Vector Normalization Using 256-Bit Intel® Advanced Vector Extensions. Intel Developer Zone.

[Mishchenko et al., 2006a] Mishchenko, A., Chatterjee, S., and Brayton, R. (2006a). DAG-aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 532–535.

[Mishchenko et al., 2006b] Mishchenko, A., Chatterjee, S., and Brayton, R. (2006b). Improvements to Technology Mapping for LUT-based FPGAs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 41–49, New York, NY, USA. ACM.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[Muller et al., 2010] Muller, J.-M., Brisebarre, N., de Dinechin, F., Jeannerod, C.-P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., and Torres, S. (2010). *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston.

[Nesterenko et al., 2015] Nesterenko, B., Wang, W., and Yi, Q. (2015). Interactive Composition of Compiler Optimizations. In *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers*, pages 91–105.

[Nimisha R., 2016] Nimisha R., Alex Wells, G. R. (2016). Data Layout Optimization Using SIMD Data Layout Templates.

[Novack and Nicolau, 1995] Novack, S. and Nicolau, A. (1995). Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pages 16–30, London, UK, UK. Springer-Verlag.

[Nuzman et al., 2011] Nuzman, D., Dyshel, S., Rohou, E., Rosen, I., Williams, K., Yuste, D., Cohen, A., and Zaks, A. (2011). Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *the 2011 International Symposium on Code Generation and Optimization*.

[Nuzman et al., 2006] Nuzman, D., Rosen, I., and Zaks, A. (2006). Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06.

[Nuzman and Zaks, 2008] Nuzman, D. and Zaks, A. (2008). Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *the 2008 Conference on Parallel Architectures and Compilation Techniques*.

[NVIDIA, 2014] NVIDIA (2014). CUDA Programming Guide 6.5.

[O'Boyle and Knijnenburg, 1997] O'Boyle, M. F. P. and Knijnenburg, P. M. W. (1997). Non-singular Data Transformations: Definition, Validity and Applications. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97.

[OpenACC, 2011] OpenACC (2011). OpenACC: Directives for Accelerators.

[OpenMP, 2013] OpenMP (2013). OpenMP: Version 4.0.

[Padua and Wolfe, 1986] Padua, D. A. and Wolfe, M. J. (1986). Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201.

[Park et al., 2012] Park, Y., Seo, S., Park, H., Cho, H. K., and Mahlke, S. (2012). Simd defragmenter: Efficient ilp realization on data-parallel architectures. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 363–374, New York, NY, USA. ACM.

[Pennycook et al., 2013] Pennycook, S. J., Hughes, C. J., Smelyanskiy, M., and Jarvis, S. A. (2013). In *the 27th International Symposium on Parallel and Distributed Processing*.

[Pillai et al., 1997] Pillai, R. V. K., Al-Khalili, D., and Al-Khalili, A. J. (1997). Energy Delay Measures of Barrel Switch Architectures for Pre-alignment of Floating Point Operands for Addition. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 235–238.

[Pop et al., 2006] Pop, S., Cohen, A., Bastoul, C., Girbal, S., andr Silber, G., and Vasilache, N. (2006). Graphite: Polyhedral analyses and optimizations for gcc. In *In Proceedings of the 2006 GCC Developers Summit*, page 2006.

[Porpodas et al., 2015] Porpodas, V., Magni, A., and Jones, T. M. (2015). PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 190–201, Washington, DC, USA. IEEE Computer Society.

[Ragan-Kelley et al., 2013] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. (2013). Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13.

[Ramachandran et al., 2013] Ramachandran, A., Vienne, J., Wijngaart, R. V. D., Koesterke, L., and Sharapov, I. (2013). Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 736–743, Washington, DC, USA. IEEE Computer Society.

[Ren et al., 2006] Ren, G., Wu, P., and Padua, D. (2006). Optimizing data permutations for simd devices. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 118–131, New York, NY, USA. ACM.

[Rice University, 1993] Rice University, C. (1993). High Performance Fortran Language Specification. *SIGPLAN Fortran Forum*, 12(4).

[Sentovich et al., 1992] Sentovich, E., Singh, K., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R., and Sangiovanni-Vincentelli, A. (1992). SIS: A system for sequential circuit synthesis. In *Technical report*, U.C. Berkley.

[Shao and Li, 2014] Shao, B. and Li, P. (2014). A Model for Array-based Approximate Arithmetic Computing with Application to Multiplier and Squarer Design. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED ’14, pages 9–14.

[Shin, 2007] Shin, J. (2007). Introducing Control Flow into Vectorized Code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT ’07, pages 280–291.

[Shirako et al., 2014] Shirako, J., Pouchet, L.-N., and Sarkar, V. (2014). Oil and water can mix: An integration of polyhedral and ast-based transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’14, pages 287–298, Piscataway, NJ, USA. IEEE Press.

[Sinkarovs and Scholz, 2013] Sinkarovs, A. and Scholz, S.-B. (2013). Semantics-Preserving Data Layout Transformations for Improved Vectorisation. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC ’13.

[Smitley and Iobst, 1991] Smitley, D. and Iobst, K. (1991). Bit-serial simd on the cm-2 and the cray-2. *Journal of Parallel and Distributed Computing*, 11(2):135 – 145.

[Sung et al., 2010] Sung, I.-J., Stratton, J. A., and Hwu, W.-M. W. (2010). Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’10.

[Tenllado et al., 2005] Tenllado, C., Piñuel, L., Prieto, M., Tirado, F., and Catthoor, F. (2005). Improving Superword Level Parallelism Support in Modern Compilers. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS ’05.

[Thinking Machine Corp., 1989] Thinking Machine Corp. (1989). Connection machine technical summary – Version 5.1.

[Thomas and Moorby, 1996] Thomas, D. E. and Moorby, P. R. (1996). *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 3rd edition.

[Tian et al., 2014] Tian, X., Xu, R., Yan, Y., Yun, Z., Chandrasekaran, S., and Chapman, B. (2014). Compiling a High-Level Directive-based Programming Model for GPGPUs. In *Languages and Compilers for Parallel Computing*, pages 105–120. Springer International Publishing.

[Tian et al., 2013] Tian, X., Xu, R., Yan, Y., Yun, Z., Chandrasekaran, S., and Chapman, B. M. (2013). Compiling a high-level directive-based programming model for gpgpus. In *26th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2013. Revised Selected Papers*, pages 105–120.

[Tong et al., 2000] Tong, J. Y. F., Nagle, D., and Rutenbar, R. A. (2000). Reducing Power by Optimizing the Necessary Precision/Range of Floating-point Arithmetic. *IEEE Trans. Very Large Scale Integr. Syst.*, 8(3):273–285.

[Trifunovic et al., 2009] Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., and Rosen, I. (2009). Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA. IEEE Computer Society.

[Verdoolaege et al., 2013] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. (2013). Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23.

[Wilt, 2013] Wilt, N. (2013). *The CUDA Handbook: A Comprehensive Guide to GPU Programming*.

[Wu et al., 2005] Wu, P., Eichenberger, A. E., Wang, A., and Zhao, P. (2005). An Integrated Simdization Framework Using Virtual Vectors. In *the 2005 Annual International Conference on Supercomputing*, SC' 2005.

[Xu and Gregg, 2014a] Xu, S. and Gregg, D. (2014a). *Efficient Exploitation of Hyper Loop Parallelism in Vectorization*. LCPC '2014.

[Xu and Gregg, 2014b] Xu, S. and Gregg, D. (2014b). *Semi-automatic Composition of Data Layout Transformations for Loop Vectorization*, pages 485–496. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Xu and Gregg, 2015] Xu, S. and Gregg, D. (2015). Exploiting Hyper-Loop Parallelism in Vectorization to Improve Memory Performance on CUDA GPGPU. In *2015 IEEE International Symposium on Parallel and Distributed Processing with Applications*, ISPA '2015.

[Yang and Zhou, 2014] Yang, Y. and Zhou, H. (2014). CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 93–106.

[Zima and Chapman, 1991] Zima, H. and Chapman, B. (1991). *Supercompilers for Parallel and Vector Computers*. ACM, New York, NY, USA.