Dissertation

Towards Automatic Compilation for Energy Efficient Iterative Stencil Computations

Submitted by

Yun Zou

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2016

Abstract

Towards Automatic Compilation for Energy Efficient Iterative Stencil Computations

Today, energy has become a critical concern in all aspects of computing. In this thesis, we address the energy efficiency of an important class of programs called "*Stencil Computations*", which occur frequently in a wide variety of scientific applications. We target the *compute intensive* stencil computations, and seek to automatically produce codes that minimize energy consumption. Two main energy consumption contributors are addressed in our work – *dynamic memory energy* and *static energy* – which are proportional to the number of off-chip memory accesses and execution time separately. We first target the dynamic energy consumption, and propose an energy-efficient tiling and parallelization strategy called *Flattened Multi-Pass Parallelization (FMPP)*, it seeks to *minimize the total number of off-chip memory accesses without sacrificing execution time.* Our strategy uses two-level tiling, which first partitions the iteration space into "passes", and then tiles the passes and executes the passes in a "non-synchronized" or overlapped fashion. Producing such codes are beyond the capability of current tiled code generators, because the schedules used are polynomials, thus are more general than multidimensional schedules. We present a parametric tiled code generation algorithm for FMPP strategy for the programs with parallelogram shaped iteration space. Then, we seek to reduce the static energy consumption by further improving the performance of generated code. We found that existing production compilers fail to vectorize the parametric tiled code efficiently, which is critical to the compiled program's performance. We propose a compilation method for parametrically tiled stencil computations that systematically vectorizes the loops with short vector intrinsics. Our method targets the non-boundary full tiles, trades register loads of register reorganization operations, enables vector register reuse within and across vectorized computations, and incorporates temporary buffering and memory padding to align memory accesses.

We developed a semi-automatic code generation framework to support our memory efficient strategy and compilation method for vectorization. Our framework allows a number of optimization choices to be configured (e.g., the trade-off of data reorganization instructions and the number of aligned loads, tiling and parallelization strategy etc). We evaluate our strategy on several modern Intel architectures with a set of stencil benchmarks. Our experimental results shown that our energy efficient tiling and parallelization strategy is able to significantly reduce the dynamic memory energy consumption on different platforms, by about a 74% (resp. 75% and 67%) reduction on an 8-core Xeon E5-2650 v2 (resp. 6-core Xeon E5-2620 v2 and 6-core Xeon E5-2620 v3). This leads to a reduction in the total energy consumption of the program by 2% to 14%. Our vectorized code also shows significant performance improvement over existing compilers. We get an average of 34% performance improvement for Jacobi 1D on all the platforms, and up to 40% performance improvement for some 2D stencils. With the savings in both static energy and dynamic memory energy, we are able to reduce the total energy consumption by 20% in average for 2D stencils on the Xeon E5-2620 v3 platform.

The tuning space for our experiment is fairly large (including both optimization choices and tile sizes), and exhaustively searching the whole space is extremely time-consuming.

In our work, we also take the first step for building an autotuner for our framework. We propose to use Artificial Neural Networks to assist the tuning process, and present a study of performance tuning with the assistance of neural networks. Our results show that the use of an Artificial Neural Network has a great potential to accurately predict the performance, and can help reduce the search space significantly.

Acknowledgements

My graduate school life has been a long journey, and now it finally comes to the end. During this time, I have received encouragement, help and support from many people. Without their support, it would have been impossible for me to finish this work.

First, I would like to express my very great gratitude to Professor Sanjay Rajopadhye, my research advisor, for his patient guidance and insightful critiques of this work. Professor Sanay Rajopadhye has brought me into the world of research, taught me how to do research, and provided me with great support during the entirety of my graduate school life. I wouldn't have been able to persist to the end without his support and encouragement.

I would like to thank all my committee members: Professor Michelle Strout, Professor Chuck Anderson, Professor Xingfeng Gao, and Professor Wim Bohm for their encouragement and insightful feedback. Especially Professor Chuck Anderson, who was willing to join my thesis committee at the last minute, and provideded me with great help on machine learning techniques.

I would also like to thank all the people in my group and all my friends in the Department. I am so lucky to have met all of you. I really enjoyed the time we have spent together for both work and entertainment. I would like to thank Martin Muggli and Matthew Malensek for their help on improving my English skills for both oral and writing communication.

Finally, I would like to give my special thanks to my parents and host family, who have been the strongest backbone for my life.

Table of Contents

Abstract ii

Acknowledgements v

List of Tables ix

List of Figures x

Chapter 1. Introduction 1

1.1. Thesis Organization 7

Chapter 2. Background 8

2.1. Architecture Model 8

2.2. Iterative Stencil Computations 11

2.3. Tiling for Stencil Computations 13

2.4. Polyhedral Model 16

Chapter 3. Related Work 19

3.1. Tiling Optimization 19

3.2. Tile Size Selection 23

3.3. Tiled Code Generation 26

3.4. Our Contributions 31

Chapter 4. Energy Efficient Tiling and Parallelization 33

4.1. Memory Behavior for Standard Wavefront Parallelization 33

4.2. Memory Efficient Tiling and Parallelization 34

4.3. Flattened Multi-Pass Parallelization (FMPP) 37

Chapter 5. Code Generation for FMPP 39

5.1. Code Generation Challenges 39

5.2. Code Generation for Sequential FMP with 2D Iteration Space 43

5.3. Sequential FMP for Higher Dimensional Iteration Space 52

5.4. Wavefront Parallelization for FMP Strategy 55

5.5. Experimental Evaluation 58

5.6. Conclusion 65

Chapter 6. Performance Optimization with Efficient Vectorization 67

6.1. SIMD Operations 68

6.2. Our SIMD Compilation Method Overview 70

6.3. Code Generation for Our Vectorization Strategy 78

6.4. Micro-benchmark for Achievable Stencil Ceiling 85

6.5. Experimental Evaluation 87

6.6. Conclusion 94

Chapter 7. Semi-automatic Code Generation Framework 96

7.1. Overview of Our Code Generation Framework 96

7.2. Experimental Evaluation for Overall Energy Savings 99

7.3. Conclusion 99

Chapter 8. Artificial Neural Network Assisted Autotuning for Performance 101

8.1. Our Approach 102

8.2. Experimental Validation 105

8.3. Conclusion 109

Chapter 9. Conclusion and Future Work 111

9.1. Conclusion 111

9.2. Future Work 112

Bibliography 114

List of Tables

5.1 Hardware specifications for Intel Xeon E5 2650 v2, Xeon E5-2620 v2 and Xeon E5-2620 v3. 59

5.2 Benchmark Details. Data D is the number of dimensions of a data grid, another time dimension is needed. NP stands for neighboring points, it means the number of neighboring points needed for the computation of each point. NV represents the number of variables that have to be computed during the computation. FPI is floating point operations per iteration. 59

5.3 Problem size used for each benchmark on each platform. 60

6.1 Hardware specifications of Intel Xeon E3-1230, E5-2620 v2 and E5-262 v3. 88

6.2 Benchmark Details. Data D is the number of dimensions of data grid, another time dimension is needed for computation. NP stands for neighboring points, it means the number of neighboring points needed for the computation of a point. FPI is floating point operations per iteration. 88

6.3 Problem size used for each benchmark on each platform. 88

8.1 Hardware features extracted for identifying the platform. 103

8.2 Program Features for characterizing the kernels. 103

8.3 Input and optimization features. 104

8.4 The number of code instances that occur in the predicted optimal solution space. 109

List of Figures

1.1 Normalized energy breakdown for stencil benchmarks on Intel Xeon E5-2620 v2, Intel Xeon E5-2650 v2 and Intel Xeon E5-2620 v3, based on a simple linear regression model (details are described in Section 5.5). Note that the dynamic memory energy consumption is only a small portion of the total energy consumption (purple). 4

2.1 The memory hierarchy of modern computer system. 9

2.2 (a) Adding two scalar variables *r1* and *r2*, and the final result is saved in *r3*. (b) Adding two vectors *V1* and *V2* to *V3*. The vector length is 4 in this example. 10

2.3 (a) The iteration space and for loop for the J1D example, where each dot represents one computation iteration. Arrows in the figure show dependencies, whose source is a consumer and destination is a producer.(b) The *C* loop structure for the J1D example (The code accesses a 2-dimensional array for simplicity of explanation. In general, Jacobi 1D can be implemented with two 1-dimensional arrays with pointer swapping. Also, the boundary copy statements can be optimized away with proper implementation). 12

2.4 Tiling for the J1D example. Arrows in the figure show dependencies, whose source is a consumer and destination is a producer. (a) Tiling on the original iteration space, which is illegal due to the cross dependence between tiles. (b) Rectangular tiling for J1D after time skewing. 14

2.5 Standard wavefront parallelization for J1D after time skewing. Each blue box represents one tile, and the orange lines are the wavefronts. 15

2.6 Two level tiling for Jacobi 1D. The iteration space is first tiled with 3×3 tiles (blue boxes), then each tile is further tiled with 2×2 tiles (red boxes). 16

4.1 The tiled iteration space and multi-pass parallelization for J1D. Each small rectangle represents one tile, and each tile depends on its left tile, bottom tile and lower left tile. The whole tiled iteration space is separated into two passes (blue and red). Each orange dashed line represents one wavefront. 35

4.2 The space-time diagram of the (a) standard, and (b) flattened multi-pass strategy for J1D. The horizontal axis represents the wavefront time step. Note that in the standard scheme, the time stamp is a *multi-dimensional vector*, and in the flattened strategy it is a *quadratic function* of the tile coordinates, pass number and row size. 38

5.1 The “space-time diagram” of traversing domain $\{i, j, k \mid 0 \leq (i, j, k) < N\}$ with a 2-D schedule $(i, j, k \rightarrow k, i + j)$” (top) is a sequence of rhombuses where one cannot start before the previous is completely done, leading to idle times between the rhombuses. The space-time diagram without the inefficiencies (bottom) may be desirable. However, it corresponds to a polynomial schedule $(i, j, k \rightarrow Nk + i + j)$. Current polyhedral code generators cannot produce code with this behavior. 40

5.2 The loop nest for the standard MPP strategy for J1D. The loop bounds are functions of the outer loop index values and problem parameters. For example,

$LB_t^{tile}(pt, time)$ represents the lower bound for the tile loop along t dimension for the pass pt at time step $time$. 42

5.3 The loop nest for the FMPP strategy for J1D. 42

5.4 Flattened iteration space for a parallelogram iteration space. The blue box is the first pass for both the original iteration space and flattened space. The red dashed box is the second pass in the original iteration space, and the red solid box is the second in the flattened space. Same for the orange boxes. 43

5.5 The loop nests for the multi-pass iteration space with the original iteration order (a) and permuted order(b) before flattening. The number of iteration points in each row is M and there are T iterations along the t dimension. h is the pass height. 45

5.6 The final sequential FMP code for the parallelogram with no second level tiling.Function `ceild`(a, b) returns $\lceil \frac{a}{b} \rceil$. 46

5.7 (a) The original loop nest (b) Loop nest after the first level tiling. (c) Loop nest after the second level tiling. h is the pass size, s_t and s_i are the tile size for the small tiles along t and i dimension. T and M are problem size parameters. 51

5.8 Tiled space for a parallelogram iteration space with the legal tile origin $\vec{0}$ (a) and the first iteration point (b). The first iteration point is $(2, 2)$. Each blue box represents one tile, and each red box is a small tile within a pass. 51

5.9 The loop structure for parallelogram iteration space with FMP: `lb_p` and `ub_p` are lower and upper bounds for the pass loop, `lb_ti`, `ub_ti`, `lb_tt` and `ub_tt` are lower and upper bounds for loop ti and tt, and `lb_i`, `ub_i`, `lb_t` and `ub_t` are lower and upper bounds for the original i and t loop. 53

5.10 Performance comparison for standard wavefront parallelization from our FMPP code generator with DTiler, Pluto and Pochoir. There is no performance for Pochoir for FDTD because Pochoir requires the neighboring points are all from the previous time step. 60

5.11 Validation result of our linear regression model for CPU energy. 64

5.12 Normalized energy consumption for the standard wavefront parallelization (Wave) and our flattened multi-pass parallelization strategy (FMPP) on Xeon E5-2620 v2, Xeon E5-2650 v2 and Xeon E5-2620 v3. 64

6.1 Achieving the shift operation of with shift distance 1, 2 and 3 with the Intel AVX instructions for double data precision. 69

6.2 A 4×14 full tile for the J1D example. The filled dots represents the computations in the tile, and the non-filled dots represents the data coming from other tiles that are required for the computations. 70

6.3 (a) Vectors required for one vector computation. (b) Assembly codes generated for one vector computation for the parametric tiled codes. *vmov* instruction moves data to a vector register, and the two *vadd* operation has one operands involves a memory address. 71

6.4 Vectorization without repeated data loads. 72

6.5 (a) The vectorized loop for Figure 6.4 when only one vector is computed within the loop body. (b) The vectorized loop with an unroll factor of two. In other words, two vectors are computed within the loop body. 72

6.6 (a) Data required (filled dots) from time step $t-1$ for the computation of one vector (non-filled dots) at time step t. The time dimension comes out of paper, and is omitted in the figure. (b) A register block with size 4×12, and the red dots, blue dots, and black dots represent one RVB separately. The filled black dots describe the data required for the RVB represented with red dots. 76

6.7 Code structure for the vectorized full tiles with tile size ts_i along the ith dimension, and register block size rs_i along the ith data dimension. 77

6.8 Register dependence graph for one UVB of J1D. 80

6.9 Code structure for the vectorized full tile for J1D. The data dimension i is unrolled by 2 vectors. Variable B is the data space that J1D operates on. It is a two dimensional array but with only two rows accessed modulo-2. 80

6.10 Memory padding for the temporary buffer for J1D. The blue dots represent padded extra memory, the non-filled dots are the memory used for the halo region. 82

6.11 code structure for the vectorized full tile for J1D. The i-dimension unroll factor is 8. 85

6.12 The design space of our vectorizer. 90

6.13 Normalized best performance of codes: SIMD NOBF, DTiler BF, SIMD BF-Aligned. The performance is normalized to the best performance achieved by DTiler. 90

6.14 Performance comparison of our SIMD code generator generated codes with the codes generated by existing compilers. The performance is normalized to the best performance achieved by DTiler. 93

6.15 Performance comparison of our SIMD code generator generated codes with the codes generated by existing compilers. The performance is also normalized to the best performance achieved by DTiler 94

7.1 Alpha representation for Jacobi 1D example. 97

7.2 semi-automatic code generation framework. 98

7.3 Flow graph for the code generation step. 98

7.4 Energy efficiency of the codes produced from our code generation framework 100

8.1 The flow graph of our autotuning approach. 105

8.2 Error trace for the square root mean during the training process. 106

8.3 The performance prediction result for both training data set and testing data set for 2D stencils. 107

8.4 The performance prediction result for both training data set and testing data set for 3D stencils. 107

8.5 Normalized best performance among the predicted optimal solution space to the actual best performance. 108

CHAPTER 1

Introduction

Stencil Computations constitute an important class of programs in scientific applications, such as environment modeling applications that involve Partial Differential Equation (PDE) solvers [88], computational electromagnetic applications using the Finite Difference Time Domain (FDTD) method [96], and computations based on neighboring pixels and multimedia/image-processing applications [36]. The importance of stencils has been noted by a number of researchers, and indicated by the recent surge of research projects and publications on this topic, ranging from optimization methods for implementing such computations on a range of target architectures [76, 46, 35], to Domain Specific Languages (DSLs) and compilation systems [97, 17, 40] for stencils. Naive implementations of many iterative stencils turn out to be several memory-bound. *Tiling/Blocking* is an important optimization technique that is used to improve data locality. Many authors have worked on applying tiling efficiently to stencil computations, and one successful technique is called *time tiling* [106, 12, 110, 108, 78, 31, 95, 4]. With time tiling, the whole computation space of a stencil, including the time loop, can be tiled and executed in a wavefront fashion. In this work, we target the compute intensive iterative stencil computations. We assume, like most of the work in the literature, that compute intensive iterative stencil computations are compute bound after time tiling. Due to the complexity of tiled code, automatic code generation for efficient tiled codes has been an active research topic for a long time.

When tile sizes are fixed, tiling can be described as a linear transformation, and can be handled perfectly by an existing technique called *polyhedral code generation* technique [6, 12].

There are two main disadvantages of fixed tile size polyhedral code generators. Many authors [91, 42, 20, 64] have pointed out that tile sizes have a large impact on the performance, and the optimal tile size depends on various aspects of the software and hardware, some of which are not known when the code is generated. As a result, the performance of code can be unpredictable. Second, even if these parameters were known early on, the process of tuning the tile sizes requires re-generation of the code for each candidate tile size, leading to a significant increase in the time for tuning. Therefore there is a growing need to delay the choice of tile sizes—possibly even until run-time. Parametrically tiled code generation addresses this need [5, 50, 37, 38] by allowing tile sizes to be *parameters*, that can be specified when the code is executed. It is also possible to adapt the code so that tile sizes are changed *dynamically* during the execution of a long running program [98]. However, when tile sizes are parametric, tiling becomes a non-linear transformation, which introduces challenges to the code generation problem. Although techniques [85, 51, 50, 37, 38, 5] have been developed to address the parametric tiled code generation problem, generating high performed parametric tiled codes still remains a challenge.

Despite the great achievements that have been made on the optimization and code generation techniques for stencil computations, past research mainly focused on performance (i.e., execution time). Nowadays, high performance computing (HPC) community is moving toward the era of exascale systems, and one of the key challenges raised is the *power wall* [8] problem. Therefore, energy efficiency becomes a critical concern during the hardware and software designs. Furthermore, widespread utilization of high performance computing (HPC) systems results in a dramatic increase in energy consumption [54, 66], which leads to costly energy bill, and also raises the operating temperature, hurting the reliability and stability of the machines. In this work, we address the parallelization and automatic compilation of stencil computations for energy efficiency.

Energy optimization for stencil computations needs careful consideration. Assuming that the application and its data set fit in main memory, the energy consumed consists of CPU energy and main memory system energy [45], and each of these can be further divided into static energy and dynamic energy. The static energy takes a significant fraction of the total energy consumption (see Figure 1.1), and since it is proportional to the execution time, the energy-optimal strategy usually also tends to finish the computation as soon as possible. As we stated before, the stencil computations that we target for become compute bound after time tiling. Therefore, after performance optimization, there is only limited "energy slack" to optimize for static energy, and any "optimization" that causes even a small slowdown is likely to lose its overall energy savings. Our **goal** is to *automatically* capture as much of this as possible—to reduce the energy consumption of tuned stencil computations. We target two main components of the energy consumption: dynamic memory energy and static energy, and seek to automatically produce codes that minimize the energy consumed by these two parts.

For static energy, the only way to reduce this part is to improve performance. In addition to time tiling, efficient utilization of available vector units is also one of the keys to improve performance on modern multi-core processors. Vector units, also called Single Instruction Multiple Data (SIMD) units, are used to exploit instruction level parallelism. They are now supported and keep improving on all major general purpose processors [75, 57, 22]. Programming models in the form of in-line assembly or intrinsic functions embedded in high-level language are introduced for explicit vector programming. Explicit vector programming

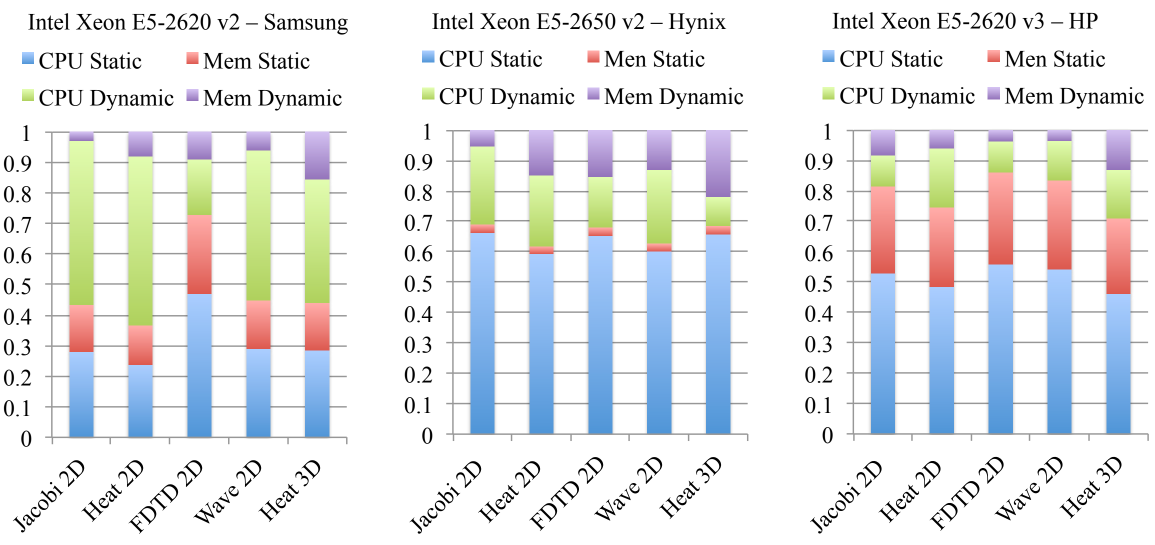


FIGURE 1.1. Normalized energy breakdown for stencil benchmarks on Intel Xeon E5-2620 v2, Intel Xeon E5-2650 v2 and Intel Xeon E5-2620 v3, based on a simple linear regression model (details are described in Section 5.5). Note that the dynamic memory energy consumption is only a small portion of the total energy consumption (purple).

requires a large amount of effort from the programmer side, which is time-consuming and error-prone. Therefore, an attractive alternative solution is to automatically generate SIMD codes. Over the past decades, automatic vectorization has received intensive attention in the research community. Techniques have been developed to automatically extract or generate the vectorized code [3, 56, 58]. Challenges that have been addressed are memory alignment [25], data reorganization overhead [115], interleaved data [71], irregular memory access [52, 16] etc. General production compilers like GCC [70] and ICC [11, 10] now successfully implement many of the vectorization techniques. The code produced by existing tiled code generator relies on the automatic vectorizers provided by production compilers. However, standard vectorizers fail to effectively vectorize parametrically tiled codes because of the complex structure of the code. In particular, the loop bounds involve complicated expressions involving max/min operations, and parameters that are not known at compile time. Our work seeks to further improve the performance by automatically producing codes with efficient vectorization strategies.

Another important component of energy consumption is dynamic energy. The dynamic energy consists primarily of the energy consumed by (on-chip and off-chip) memory accesses, and by arithmetic operations. After performance optimization, the number of computations cannot be reduced, but the energy consumed per cache access is much smaller than that by an off-chip memory access. Therefore, we seek to reduce energy consumption by reducing the off-chip memory accesses, i.e., further improving the cache hit rate. Figure 1.1 indicates that the dynamic memory energy consumption (purple) only takes up a small portion (3% to 20%) of the total energy. In this work, we seek to save this energy expenditure and furthermore, to do it "for free," i.e., with an automatic code generator. Our dynamic memory energy optimization strategy uses multi-level tiling associated with *multi-pass* execution [68], a technique that was originally introduced in the context of systolic arrays. We adapt this strategy to address the last level cache misses on general multi-core processors.

Most program optimizations introduce trade-offs in some aspects while gaining improvement in other aspects. The trade-offs can be very different on different platform for different applications. However, modeling the interactions between the software and hardware, and compressing them into a single analytical model is a challenge. Moreover, the hardware architectures today are becoming more and more complicated, which makes them even more difficult to model. Furthermore, tile sizes are important program parameters, which have large impact on performance. Although, many analytical models [34, 18, 14, 42, 30, 64] have been built to assist the tile size selection problem, none of them has been proven to be effective across platforms and kernels. In this thesis, we address the tile size and optimization strategy selection problem using Artificial Neural Networks (ANN). Our approach is based on an exhaustive search of the tuning parameter space using the neural networks trained performance model.

We develop a code generation framework that targets producing energy efficient codes for compute intensive stencil computations. Our contributions are as follows:

- an energy efficient tiling and parallelization strategy—flattened multi-pass parallelization (FMPP), focusing on reducing the number of off-chip memory accesses without sacrificing speed;
- a code generation algorithm for FMPP that supports parametric tile sizes for programs with hyper-parallelepiped shaped iteration space. It implements *polynomial* schedules instead of *multi-dimensional* schedules, which extends the state of the art in parallel code generation;
- an automatic compilation method for generating efficient vectorized parametrically tiled codes that trades register loads of register reorganization operations, and maximizes the register reuse;
- a semi-automatic code generation framework that supports the vectorization strategy and alignment optimizations in a configurable way;
- a micro-benchmark based evaluation method to obtain a ceiling on the achievable machine performance for stencils;
- an autotuning method for performance with the assistance of Artificial Neural Network (ANN) that targets for selection of both code generation parameters (e.g., register block size, temporary buffering) and tile size parameters.

1.1. Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 describes our target architecture model, definition for stencil computations and some necessary background for loop tiling. Chapter 3 discusses the related existing work about tiling and code generation for stencil computations. Chapter 4 presents our energy efficient tiling and parallelization strategy–FMPP. Chapter 5 shows the code generation algorithm for our FMPP strategy and the corresponding formula derivation. In Chapter 6, we describe our compilation method for vectorized code generation and the vectorization strategies supported. Chapter 7 gives an overview of our code generation framework. In Chapter 8, we present our autotuning approach that chooses the code generation parameters and tile sizes for performance with the assistance of ANN. Finally, we present our conclusion and possible future work in Chapter 9.

CHAPTER 2

Background

In this chapter, we describe our target architecture and the definition of iterative stencil computations. We also present the terminologies and examples used in the rest of this dissertation.

2.1. Architecture Model

The architecture we focus on is a modern multi-core processor with a memory hierarchy, and with Single Instruction Multiple Data (SIMD) units supported.

2.1.1. Memory Hierarchy. Figure 2.1 describes the general memory hierarchy, consisting of registers, caches, followed by the off-chip main memory and hard disk. From the register level to the hard disk level, the cost decreases and the capacity increases, but the latency also increases. The cache level consists of several levels of *N-way associative* caches. For an N-way associative cache, the whole cache is divided into sets, each set can hold N distinct cache lines, and one cache line is mapped to one set. On modern multi-core architecture, the inner levels of cache are private to each core, and the Last Level Cache (LLC) is shared among all the cores.

When some data is needed for a computation, a data request is issued to the memory hierarchy in a top-down manner. If the data is already in L1 cache, it will be transferred to the register (upper level memory), otherwise, a data request is going to be issued to L2 cache. The same process is repeated until the data is found at a certain level of memory.

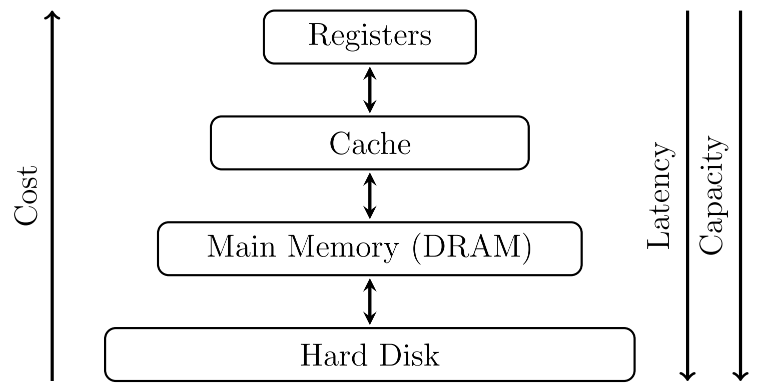


Figure 2.1. The memory hierarchy of modern computer system.

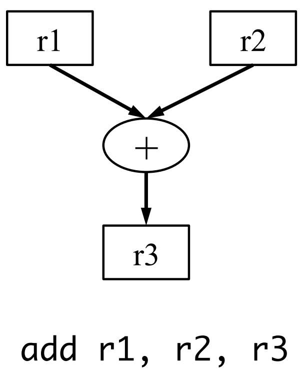
A failed request of read or write at a certain level of cache causes a cache miss, and a failed request of read/write to the LLC leads to a *LLC miss* and an *off-chip memory access*. In the rest of this paper, we will use off-chip memory access and LLC miss interchangeably. Cache misses are classified into three categories [41]: compulsory, capacity, and conflict misses.

Compulsory miss. A compulsory miss occurs on the first access to a memory location, since at the very beginning of execution, no data is in the cache. Such misses are also called *cold misses*, and are in general unavoidable.

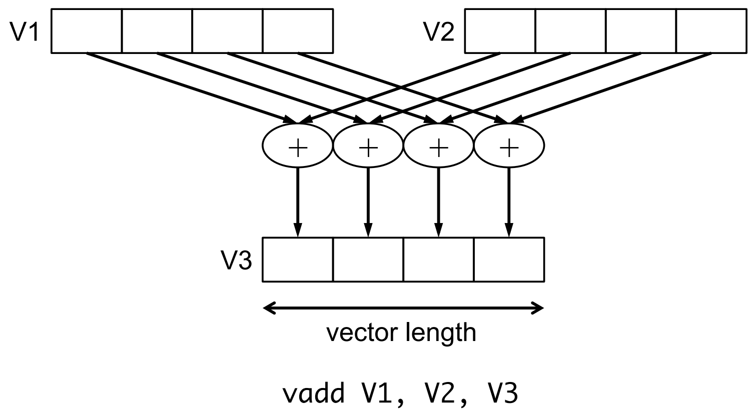
Capacity miss. A capacity miss occurs when the total amount of memory used by a program is larger than the cache capacity. In this situation, the cache is not large enough to hold all the data that is needed, Therefore, some data has to be evicted from the cache, and a subsequent request to this data will cause a miss. Capacity miss is the main focus of techniques that address data locality issue, and it is also the main focus of our work.

Conflict miss. A conflict miss occurs for an N-way associative cache when the program frequently accesses more than N distinct cache lines that are mapped into the same set. When the conflict misses are caused due to the memory references to the same variable, it is called ***self-interference misses***. Otherwise, if conflict misses are caused due to the memory references to multiple variable, it is called ***cross-interference misses***. Normally, conflict misses can be minimized using existing techniques [18, 14, 42, 64].

2.1.2. SIMD Units. **S**ingle **I**nstruction, **M**ultiple **D**ata (SIMD) or vector units are a powerful feature of modern multi-core processors. The SIMD units are used to exploit instruction level parallelism, and present opportunities to attain very high performance. Modern CPU supports special instructions to utilize the SIMD units, which performs operations on 1D arrays of same size, called ***vectors***, instead of scalar variables. The number of elements contained in a vector is called ***vector length***. The vector lengths supported on modern architectures are usually pretty short (e.g., 4 or 8). Figure 2.2a shows the regular addition operation with scalar variables. With SIMD instructions, the add operations can be performed on a vector of elements simultaneously as shown in Figure 2.2b.



(A)



(B)

Figure 2.2. (a) Adding two scalar variables *r1* and *r2*, and the final result is saved in *r3*. (b) Adding two vectors *V1* and *V2* to *V3*. The vector length is 4 in this example.

Almost all processors today have such "short SIMD" instructions, such as Intel Streaming SIMD Extensions (SSE, SSE2, SSE3, SSE4.1, SSE4.2), Intel Advanced Vector Extension (AVX, AVX2, AVX-512), AMD 3DNow!, Mortoroa AltiVec Extensions and SUN VIS. These

SIMD instructions support a rich set of operations for data reorganization in vector registers. Efficient utilization of the SIMD units with proper instructions is one of the key techniques for exploiting high performance on modern architectures.

2.2. Iterative Stencil Computations

A stencil computation is a computation that repeatedly updates each point in a d-dimensional data grid with size $(N_1 \times N_2 \times \ldots N_d)$ over T time steps. The d-dimensional data grid and the time dimension comprise a $(d+1)$-dimensional iteration space. In our work, we target for stencils with large values along all the dimensions, which leads to fairly large iteration spaces.

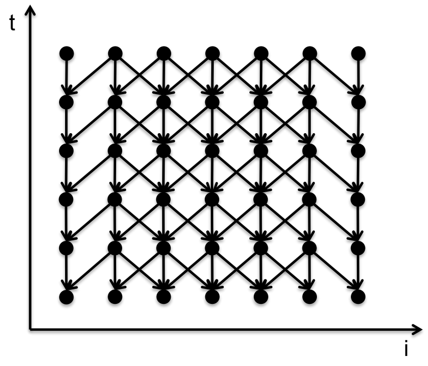
In a stencil computation, at a time step t, the computation for each point is defined as a function of its neighboring points at previous time steps and possibly current time step. When the computation of each point only depends on neighboring points at the *previous* time steps, we call it a ***Jacobi style*** stencil. Otherwise, the neighboring points used for the computation include points at the *current* time step, and such stencils are called ***Gauss-Seidel style*** stencils. A stencil computation is called an ***n-point stencil computation*** if each point is defined as a function of n neighboring points from the current and previous time steps. The ***order*** of a stencil computation is defined as the distance of the furthest grid point in the neighboring points. In the rest of this document, we use the classical Jacobi 1D and Jacobi 2D as illustrative examples to explain the main ideas.

Jacobi 1D (J1D) Example. J1D is a first-order 3-point stencil computation that repeatedly updates a 1-dimensional data space with size N over T time iterations, and the computation of each point depends on three neighboring points from the previous time step.

The detailed computation is described in Equation 1.

$$(1) \qquad B_{t,i} = \begin{cases} A_i, \quad t = 0 \\ B_{t-1,i}, \quad 0 < t \leq T \text{ and } (i = 0 \ or \ i = N) \\ 0.3333 \times (B_{t-1,i-1} + B_{t-1,i} + B_{t-1,i+1}), \quad 0 < t \leq T \ and \ 0 < i < N \end{cases}$$

In J1D, updating point i at time step t requires three values from the previous time step: $(i-1)$, i and $(i+1)$. Then we say iteration (t,i) depends on iteration $(t-1,i)$, $(t-1,i-1)$, and $(t-1,i+1)$. Figure 2.3a describes the iteration space and dependencies for J1D. The corresponding C loop is shown in Figure 2.3b, and the code presented accesses a 2-dimensional array for simplicity of explanation. In general, Jacobi 1D can be implemented with two 1-dimensional arrays with pointer swapping. Also, the boundary copy statements ($S2$ and $S3$) can be optimized away with proper implementation.



(A)

```
 //initialization
 for(i=0; i < N; i++)
S1:  B[0][i] = A[i];
 for(t = 1; t < T; t++) {
      //copy for boundary values
S2:  B[t][0] = B[t-1][0];
S3:  B[t][N-1] = B[t-1][N-1];
    for(i = 1; i < N-1; i++) {
S4:    B[t][i] =  0.3333*(B[t-1][i
   -1] + B[t-1][i] + B[t-1][i+1]);
    }
 }
```

(B)

FIGURE 2.3. (a) The iteration space and for loop for the J1D example, where each dot represents one computation iteration. Arrows in the figure show dependencies, whose source is a consumer and destination is a producer.(b) The C loop structure for the J1D example (The code accesses a 2-dimensional array for simplicity of explanation. In general, Jacobi 1D can be implemented with two 1-dimensional arrays with pointer swapping. Also, the boundary copy statements can be optimized away with proper implementation).

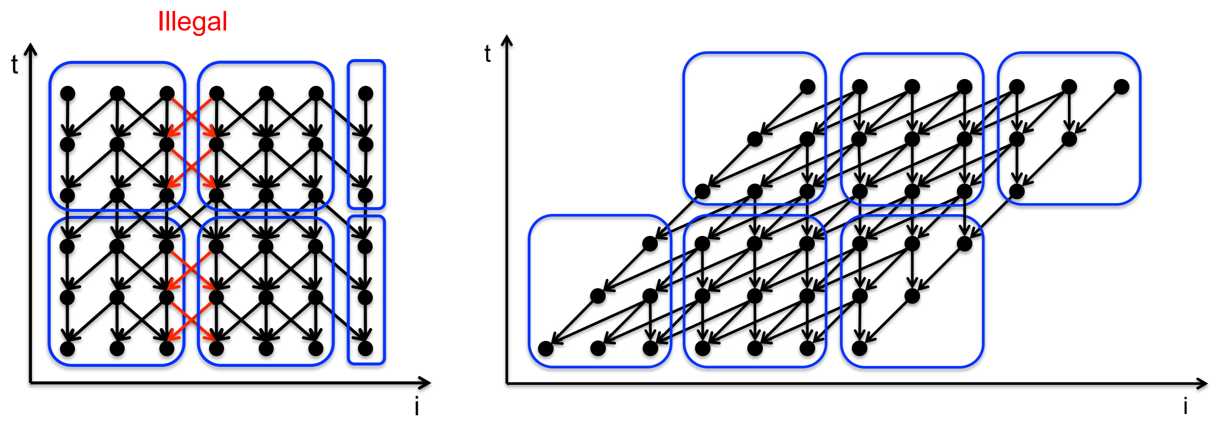
Jacobi 2D (J2D). This is a first-order 5-point stencil computation repeatedly updating a 2-dimensional data space with size $N \times M$ over T time steps. The computation is described below.

$$(2) \qquad B_{t,i,j} = \begin{cases} A_{i,j}, \quad t = 0 \\ B_{t-1,i,j}, \quad 0 < t \leq T \textit{ and } (i = 0 \textit{ or } i = N \textit{ or } j = 0 \textit{ or } j = M) \\ 0.2 \times (B_{t-1,i-1,j} + B_{t-1,i+1,j} + B_{t-1,i,j} + B_{t-1,i,j-1} \\ \qquad +B_{t-1,i,j+1}),\ 0 < t \leq T \textit{ and } 0 < i < N \textit{ and } 0 < j < M \end{cases}$$

In J2D, updating point (i, j) at time step t requires five values from the previous time step: $(i-1, j)$, $(i+1, j)$, (i, j), $(i, j-1)$, and $(i, j+1)$. Therefore, iteration (t, i, j) depends on iteration $(t-1, i-1, j)$, $(t-1, i+1, j)$, $(t-1, i, j)$, $(t, i, j-1)$, and $(t, i, j+1)$.

2.3. Tiling for Stencil Computations

For stencil computations, an important program optimization technique is called ***tiling*** or ***blocking***, which is used to improve data locality and parallelization granularity. We define the ***data footprint*** of a work unit as the distinct memory location touched in the work unit. Let us take the J1D (shown in Figure 2.3a) as an example, for which the computation is executed in lexicographical order and the data space is much larger than the LLC capacity. During time step t, every point in the data space is touched, which means the data footprint of time step t exceeds the LLC. Therefore, a large amount of capacity misses will occur when moving to time step $(t+1)$. Tiling addresses the capacity misses. It blocks the computation space and ensures the data footprint of each tile fits into certain level of cache. Conflict misses can also be minimized by proper choice of the tile sizes (blocksizes) [18, 14, 64].



(A) (B)

Figure 2.4. Tiling for the J1D example. Arrows in the figure show dependencies, whose source is a consumer and destination is a producer. (a) Tiling on the original iteration space, which is illegal due to the cross dependence between tiles. (b) Rectangular tiling for J1D after time skewing.

However, due to the dependencies between the successive time steps, normal rectangular blocking cannot be applied across the time dimension, because a cross dependence will be created between tiles as shown in Figure 2.4a. A technique called *time skewing* can be used to enable tiling across the time dimension [108].

2.3.1. Time Skewing and Wavefront Parallelization. Time skewing skews the data space with respect to the time dimension to make all the dependencies fall into the first quadrant, and then rectangular tiling can be applied to tile the whole iteration space. Figure 2.4b illustrates this for the J1D example, where the computation of i at time step t dependents on values produced at $i-2$, $i-1$ and i from the previous time step after time skewing. The rectangular tiles are further separated into two types: *full tile* and *partial tile*. A *full tile* is a rectangular tile such that every point within the tile actually represents a computation that happens in the application. On the other hand, a *partial tile* has some points in the rectangular space that do not represent any computation. After time skewing and rectangular tiling, the tiles that are along the same diagonal can be executed in the same time, and this is the standard wavefront parallelization for stencil computations. All tiles that can be executed at the same time step comprise a *wavefront*. Figure 2.5 describes this parallelization of the tiles of the J1D example. We see that with only 3 processors, there are not enough independent tiles to use all the available resources in the first 4 wavefronts. This is called the *pipeline fill* stage. Similarity, the last 4 wavefronts comprise the *pipeline flush* stage.

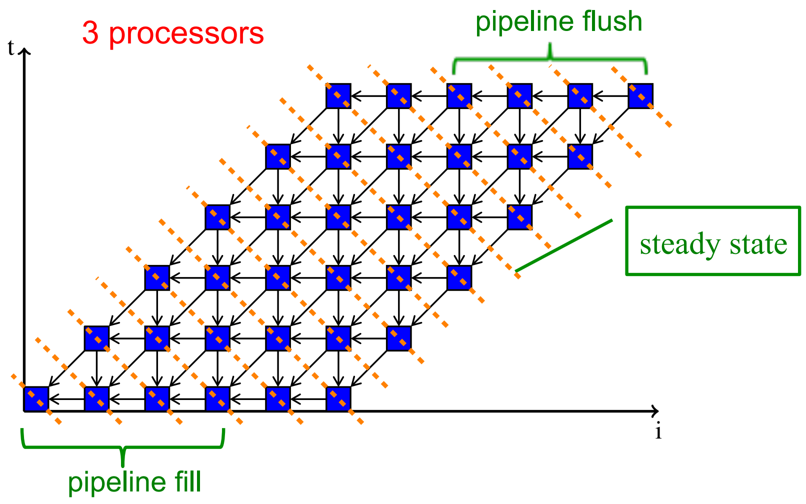


FIGURE 2.5. Standard wavefront parallelization for J1D after time skewing. Each blue box represents one tile, and the orange lines are the wavefronts.

Similar to J1D, time skewing has to be applied to enable rectangular tiling for J2D. After time skewing, the computation of (i, j) at time step t depends on the value produced at $(i-2, j-1)$, $(i, j-1)$, $(i-1, j-1)$, $(i-1, j-2)$, $(i-1, j)$ at time step $t-1$. Therefore, after time skewing, iteration (t, i, j) depends on iteration $(t-1, i-2, j-1)$, $(t-1, i, j-1)$, $(t-1, j-1, j-2)$ and $(t-1, i-1, j)$.

2.3.2. MULTI-LEVEL TILING. Multi-level tiling, also called *Hierarchical Tiling*, tiles the iteration space hierarchically. Muli-level tiling is usually applied to address two issues: 1) data reuse of the hierarchical memory structure, where multi-level tiling can be applied to address the capacity misses at different level of caches; 2) hierarchical organization for the target machines. One typical example for the second point is clusters with distributed memory model, where we can first tile the iteration space for mapping the computations to nodes of the cluster, and then further tile the iterations on each node to improve the data locality on each node. In this thesis, we utilize the multi-level tiling to address the data reuse issue. However, efficient mutli-level tiling on a single multi-core processor requires careful consideration, and will be illustrated later.

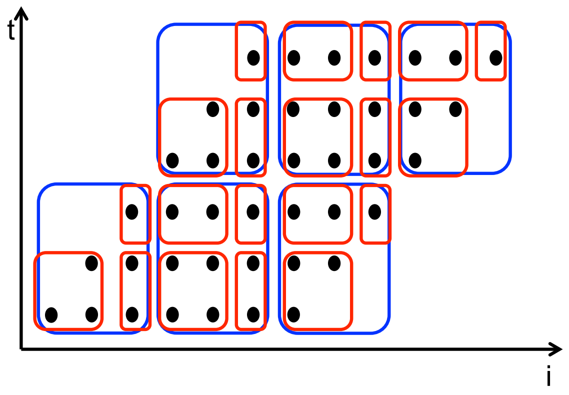


Figure 2.6. Two level tiling for Jacobi 1D. The iteration space is first tiled with 3×3 tiles (blue boxes), then each tile is further tiled with 2×2 tiles (red boxes).

A ***n-level tiling*** hierarchically tiles the iteration space n times. Figure 2.6 gives an example of two-level tiling using the Jacobi 1D example. The whole iteration space is first tiled with 3×3 tiles, and then each tile is further tiled with 2×2 tiles.

2.4. Polyhedral Model

The *polyhedral model* [79, 82, 26–28] is a mathematical formalism for analyzing, and transforming an important class of compute- and data-intensive programs, or program fragments, called *Affine Control Loops (ACL)*. The polyhedral model has also been proved to be very useful in automatic parallelization [29, 7, 6, 77]. Our target program – stencil computation – fits the polyhedral model perfectly, and therefore is represented and analyzed using the polyhedral model in this work. Furthermore, the computation dependencies occur in stencil computations also demonstrate a very nice pattern, which are called *Uniform Dependencies*. In the rest of this section, we will describe the definition for ACL programs, uniform dependencies, and some polyhedral terminologies that are used in the rest of description.

An ACL satisfies the following properties: (i) it only consists of (sequences of, possibly arbitrarily) nested loops, (ii) the statements in the loop are assignment statements, possibly with conditions; (iii) the loop lower (respectively, upper) bounds are the maxima (respectively, mimima) of a finite number of affine functions of outer loop indices and program parameters, and (iv) the conditions involved and array accesses are affine functions of outer loop indices and program parameters.

Iteration Vector. An iteration vector is used to represent an instance of a statement. It is defined as a d-dimensional vector $\vec{i} = (i_0, i_1, \ldots, i_{d-1})$, where d is the number of loops surrounding the statement, and i_k is the loop index of the kth loop (i_0 being the outermost).

Domain. Each computation statement in a program is surrounded by loops with affine bounds. The domain of a statement describes the iteration space in which the statement is defined. For a given statement that is surrounded by a d-dimensional loop nest, whose lower and upper bounds are $(lb_0, lb_1, \ldots, lb_{d-1})$ and $(ub_0, ub_1, \ldots, ub_{d-1})$, and the iteration vector is $\vec{i} = (i_0, i_1, \ldots, i_{d-1})$. The domain of the statement is represented as

$$\mathcal{D} = \{(i_0, i_1, \ldots, i_{d-1}) \mid lb_k \leq i_k \leq ub_k, 0 \leq k < d\}$$

Dependence. A statement instance $\langle S_1, \vec{i} \rangle$ is said to have a (***(flow) dependence***) on instance $\langle S_2, \vec{j} \rangle$, written as $\langle S_1, \vec{i} \rangle \rightarrow \langle S_2, \vec{j} \rangle$ if the value produced by $\langle S_2, \vec{j} \rangle$ is used by $\langle S_1, \vec{i} \rangle$. A shorthand for the dependence is written as $(\vec{i} \rightarrow \vec{j})$, which is called ***Dependence function***.

Uniform Dependence. A dependence is a uniform dependence if the dependence function is in the form of $(\vec{i} \rightarrow \vec{i} - \vec{c})$, where $\vec{c}$ is a constant vector and it is called ***dependence vector***.

Let us take the Jacobi 1D shown in Figure 2.3b as an illustration example. The domain for statement $S1$ is $\{i \mid 0 \leq i < N\}$, the domain for statement $S2$ and $S3$ is $\{t \mid 1 \leq t < T\}$, and the domain for the main computations statement $S4$ is $\{t, i \mid 1 \leq t < T, 1 \leq i < N-1\}$. The iteration vector for statement $S1$ is (i), and the iteration vector for statement $S4$ is (t, i). Each statement instance $\langle S4, (t, i) \rangle$ in domain $\{t, i \mid 1 \leq t < T, 2 \leq i < N-2\}$ depends on $\langle S4, (t-1, i-1) \rangle$, $\langle S4, (t-1, i) \rangle$ and $\langle S4, (t-1, i+1) \rangle$. The dependence functions are $(t, i \rightarrow t-1, i-1)$, $(t, i \rightarrow t-1, i)$ and $(t, i \rightarrow t-1, i+1)$ respectively. These dependencies are all uniform dependencies with dependence vector $(1, 1)$, $(1, 0)$, $(1, -1)$ respectively.

CHAPTER 3

Related Work

Stencil computations, as an important class of programs, have received a considerable amount of research attention during the past decades. In this section, we discuss the existing work of tiling optimizations and code generation techniques for stencil computations. Then we also discuss about how our work compares with existing work.

3.1. Tiling Optimization

Many authors have worked on optimizing stencil computations using tiling on both shared memory and distributed memory or hybrid platforms. The tiling optimization techniques for stencil computations can be classified into four categorizations: tiling optimizations within a single time step, cache aware tiling optimizations across multiple time steps, cache oblivious tiling optimizations across multiple time steps and multi-level tiling.

Within a single iteration, simple rectangular blocking can be directly applied on the data space for the Jacobi style stencil computations [86, 24, 76]. However, the amount of data reuse is very limited within a single time step, and it is important to exploit data reuse across multiple time steps.

Cache oblivious tiling is one technique that exploits data reuse across time steps. It starts with the original iteration space and recursively divides it into small tiles with trapezoidal surfaces [78, 31] or parallelotopes [94], and stops the recursion when the base tile is reached. The size for the base tile is decided without knowing the hardware details (usually based on some heuristic). Cache oblivious technique is an attractive technique and easy to implement. However, selecting tile size without knowing hardware detail is intuitively not optimal. Datta et al. [20] show in their work that the cache oblivious techniques suffer a performance degradation due to the sub-optimal compiler code generation and tile size selection. Yotov et. al [112] also pointed out that the recursive code style of cache oblivious tiling may prohibit the efficient exploration of processor pipelines and hardware prefetcher. Later, we also experimentally show the performance problem of the codes produced by a current cache oblivious code generator.

The tiling techniques that are most related to our work are the cache aware tiling and multi-level tiling, and are discussed below.

3.1.1. Cache Aware Tiling Across Multiple Time Steps. Time skewing [110, 108] is one of the most important approaches that exploit data locality across multiple time steps. It looks at the dependencies in the whole iteration domain and skews it with respect to the time dimension. Then all the dependencies fall into the first quadrant, and rectangular tiling can be applied to tile the whole iteration space. The same effect of time skewing can also be achieved by hyperplane tiling [44, 12, 4]. Instead of skewing the computation space to make rectangular tiling legal, they are trying to find a legal hyperplane to cut the whole computation space. Due to dependencies between tiles, most authors [108, 12] subsequently parallelize the tiled programs with the classic 45 degree wavefront parallelization.

While the time skewing technique enables the data reuse across the time steps, it also introduces inter-tile dependencies, and parallelization is enabled along the 45 degree diagonal, which introduces the pipeline fill-flush overhead that is described in section 2.3.1. Wonnacott and Strout [107] present a theoretic exploration of the impact on the scalability of various loop tiling strategies. Their result indicates that the pipelined parallelization strategy runs into scalability issue on platforms with high degree of parallelism for programs whose data set size grows linearly with the degree of parallelism.

In order to eliminate the pipeline overhead and improve the scalability, techniques like *overlapped tiling* [89, 23, 83, 55] and *split tiling* [109, 55] are developed. In overlapped tiling, each tile is enlarged to include the points that are needed for the computation in that tile. Split tiling partitions each tile into subtiles: one only includes computations that are independent of other tiles and others with points that depend on other tiles, and possibly others with points that other tiles need. During the execution, the independent subtiles can be started at the same time to explore concurrent start. Many authors also explore concurrent start based on different tile shapes, such as diamond tiling [74, 95, 4], and hexagonal tiling [35]. Orozco et al. [74] demonstrated the effectiveness of diamond tiling with a 1D and an 2D FDTD examples. Strzodka presents a technique called cache accurate time skewing (CATS) [95] for stencil computations, which explores the concurrent start through diamond shaped tiles. CATS is based on the reduction of higher dimensional problem into lower dimensional, non-hierarchical problem. In CATS, a subset of the dimensions are tied to form large tiles, and a sequential wavefront traversal is performed inside the tiles and the concurrent parallelization is explored among the diamond tiles.

3.1.2. Multi-level Tiling. Multi-level tiling is used to address data reuse at different levels of the memory hierarchy, for which further tiling is applied on the tiled space. Lakshminarayanan et al. [84] describe a multi-level tiling work on a 2D Gauss-Seidel stencil computations. They first tile the space for parallelism and further tile each tile into small tiles for data locality. They exploit two tiling choices for the outer level tile – tile the data space or tile the whole iteration space, and they also develop an execution time model for both strategy to guide the choice of the tile sizes.

Multi-level tiling is also used to explore multiple levels of parallelization. Dursun et al. [24] enables both the distributed memory parallelization and shared memory parallelization on a hybrid machine through two levels of tiling. Multi-level wavefront parallelization as a standard strategy has also been explored by many authors [12, 50, 5]. Shrestha et al. [90] point out the pipeline filling-flush overhead introduced by wavefront parallelization overhead at the inner level, which is also an overhead we will face later. They proposed a multi-level hyperplane tiling technique that explores concurrent start at the inner levels.

Malas et al. [61] proposed a multi-level tiling strategy to address the data reuse among threads on modern multi-core architectures. Their technique first applies the diamond tiling technique presented by Strzodka [95], and then applies the classic wavefront time tiling within each diamond prism. The available threads are first grouped into groups, and then assigned to different diamond prisms. The size of the diamond is chosen to be fit into the LLC, and the data reuse is enabled for the threads that are assigned to the same diamond prism. Their experimental results on a 10-core Intel Xeon processor show that the maximum data reuse can be achieved when all the available threads are assigned to the same diamond tile. However, a slow-down has to be paid for this, which is probably due to the wavefront overhead that is payed within each diamond prism, which is also pointed out by Shrestha et al. [90].

3.1.3. Energy Efficient Stencil Computations. Due to the growing importance of energy, some energy optimization work has also been done specifically for stencil computations or tiling technique. Kandemir et al. [48, 47] realize that the tile size that gives the best performance is not necessarily be the one that gives the best energy behavior. They experimentally evaluated the impact of different tile sizes on the energy behavior. They conclude that it is better to experiment with different tile sizes and pick up a suitable one for energy purposes. Tiwari et al. [100] explore the optimization space for stencil computations by tuning both the one level tiling parameters and the CPU clock frequency. Their experimental results confirm the popular belief that optimizing for performance often leads to better system-wide energy consumption. However, there is also about 5.8% of energy savings with about 4.1% slow-down with lowering the clock frequency. Although some "energy reduction" can be achieved with exploration of tile size parameters, it is not always guaranteed. Also, experimenting with CPU clock frequency is out of the scope of this work.

Garcia et al. [33] pointed out the importance of the dynamic memory energy consumption. They build an analytical energy model for the dynamic memory access, and choose the tile size and traversing order that minimizes the dynamic memory energy consumption. However, there is no guarantee that there is analytical solution for their model. Furthermore, the side effect to the performance is not considered in their work. The wavefront diamond blocking work proposed by Malas et al. [61, 60] addressed the data reuse among threads, which also helps to reduce the dynamic energy consumption. Their experimental results showed that they are able to save up to 6% dynamic memory energy consumption. However, there is also about 8% performance loss, and this may defeat the overall energy saving.

3.2. Tile Size Selection

It is well-known that the choice of suitable tile size has significant performance impact for the tiled codes. The past work for solving the tile size selection problem can be classified into two categories: 1) Static analytical model based approach, which analyzes the interaction of between the input kernel and hardware architectures and build analytical models to pick up the best tile size; 2) Prediction based approach, which relies on machine learning models to predict the solution space for searching.

A significant amount of effort has been put in to developing analytical models to guide the selection of tile sizes. Ghosh et al. [34] developed a Cache Miss Equation (CME) framework for quantifying the number of cache misses, and presented a tile size selection algorithm based on their CME framework that targets for eliminating the self-interference misses and capacity misses. Coleman and Mckinley [18] and Cahme and Moon [14] proposed tile size selection algorithms that take both self-interference and cross-interference misses into account. Hsu and Kremer [42] and Rivera and Tseng [87] point out that the interference misses can be reduced by array padding, and select a tile size that minimizes cache misses after the array padding optimization. However, these techniques usually only focus on a certain level of memory hierarchy. Nicholas Mitchell et al. [67] pointed out that level-specific cost function usually leads to sub-optimal choices, and a cost function that considers multiple levels of the memory hierarchy leads to better performance. Fraguela et al. [30] and Metha et al. [64] developed tile size selection algorithms based on the behavior of the whole memory hierarchy. The approach proposed by Metha et al. [64] also takes the interaction between tiling and SIMD units into account.

Despite the great achievement on the analytical model based approaches, none of them has proved to be effective across different kernels and different platforms. Designing analytical models that can accurately predict the performance is a complicated task, because many aspects from both the software side (i.e., arithmetic intensity, data reuse within and across tiles and problem size) and hardware side (i.e., memory hierarchy, available resources, SIMD unit and prefetching) have to be taken into account. Existing analytical models fail to compress the interaction of all these aspects into a single model. Nowadays, the hardware architectures are becoming more and more complicated, which makes the modeling task even more difficult.

Another popular approach is to use Artificial Neural Networks (ANNs) to automatically learn the tile size selection model based on different input features. Rahman et al. [81] collects the execution time for different tile sizes for a given kernel, and trains the ANN to predict the performance for the kernel. The input to the ANN is tile sizes, and the output is the execution time. Yuki et al. [114] targets for 3 dimensional loop programs (i.e., 2D data space) with square tiles, and uses the ANN to predict the best performance for kernels with a given architecture-compiler combination. They extract six input program features according to the memory references, and formulate the tile size selection model as a continuous function to predict the optimal tile sizes. Mialik [62] trained ANNs with dynamic program features (i.e., L1 load misses, L2 load misses etc), instead of static program features. All these work demonstrate fairly accurate predicted result, but all limited to a given architecture-compiler combination and kernels with 2D data space.

Luo et al. [59] developed a Fast Stencil Autotuning Framework (FAST) for predicting the optimal solution space for different kernels on different platforms. The optimal solution space predicted includes both tile sizes and optimizations should be applied (i.e., loop unrolling, register blocking etc.). In their approach, an instance of kernel-platform combination is represented with a set of hardware and software features. Based on the observation that two instances of kernel-platform combination with the most similar features have a large overlap in their optimal solution space, they return the optimal solution space for the kernel-platform combination instance with the most similar feature in the initialized database. The similarity function of the two given sets of features is learned with a polynomial regression.

Our observations indicate that the tile size selection problem is not only related to the hardware architecture and kernel parameters, but also strongly coupled with the style/structure of code that is generated. Therefore, the tile size selection problem cannot be separated from the code generation, and separate tile size selection models have to be developed/learned for different code generation frameworks. Furthermore, it is much more efficient to have a self-learned model use machine learning techniques with code structure independent features, rather than hand-crafting a model for each code generator.

3.3. Tiled Code Generation

Writing tiled codes is time consuming and error-prone, especially for parallel codes. Automatic code generation is an attractive solution since it requires little programmer effort.

3.3.1. Tiled Code Generation Based on Polyhedral Techniques.

Polyhedral compilation is one of the successful techniques for tiled code generation. Regarding to how tile sizes are represented in the generated code, existing polyhedral technique based code generators can be separated into two categories: fixed-size tile code generators and parametric tiled code generators.

When tile sizes are fixed, tiling can be described as a linear transformation, and purely polyhedral code generators — like ClooG [6], ISCC [103] and Omega+ [72] — are adequate. Based on existing polyhedral code generators, Bondhugula et al. [12] developed an automatic parallelizer that chooses tiling hyperplanes that minimize the communications between tiles and parallelizes the tiled program using classic 45 degree wavefront parallelization. The legality condition for the hyperplanes is a generalization of the classic condition proposed by Irigoin and Triolet [44] to imperfectly nested loops. Later, Bandishti et al. [4] extended the hyperplane tiling technique to find legal tiling hyperplanes that enables concurrent start, which leads to fully oblique (colloquially called ”diamond”) tiles.

Fixed size tiled code generation for different tile shapes has also been explored in other work. Grosser et al. [35] developed a hybrid hexagonal/classic tiled code generator for GPUs, for which hexagonal tiling is applied on the face constructed by the time dimension and one of the data dimensions and classic time tiling is applied on the other dimensions. Although the technique was developed for GPUs, it can also be generalized to CPUs. Trapezoid tiling is utilized in cache oblivious code generators like Pochoir [97].

Despite the great success of the fixed-size tiled code generators, there is a strong desire to delay the selection of tile size until launch time, and therefore parametric tiling is needed. However, parametric tiling is a non-linear transformation, which goes outside the scope of polyhedral model. This raises interesting challenges for the code generation problem. One simple solution to the parametric tiled code generation is to apply rectangular tiling to the bounding box of the parametric domain, and add guards for each point in a tile to check whether it belongs to the original iteration space [111]. However, this method ends up enumerating many empty tiles when the bounding box is much larger compared with the original iteration space. Therefore, many authors have explored parametric tiled code generation without enumerating empty tiles [85, 51, 50, 37, 38, 5].

PrimeTile [37], DynTile [38], PTile [5] and DTiler [50] are the most representative parametric tiled code generators developed recently. PrimeTile [37] is a sequential tiled code generator. It identifies the largest sub-rectangular iteration space from the original iteration space dynamically, and applies rectangular tiling on the sub-rectangular iteration space to avoid enumerating empty tiles. However, parallelization is essential to utilize the modern parallel architectures.

DynTile [38], DTiler [50] and PTile [5] are parametric tiled code generators that support wavefront parallelization. In these tools, the origin of the space $\vec{0}$ is usually assumed to be a legal tile origin, and a polyhedral set called *outset* [85] that contains the origin of all the non-empty tiles is constructed. A set of loops called *tiled loops* are generated to visit the tiles within the outset. The tiled loops generated either enumerates the first iteration point (called tile origin) in each tile or the coordinates of the tiles in the tiled space. Another set of loops called *point loops* are generated to visit each point for a given tile origin. DynTile generates tiled loops that enumerate the tile coordinates, and supports wavefront parallelization through a run-time scheduling approach. In DTiler and PTile, the tiled loops enumerate the tile origins, and wavefront parallelization is supported by computing the 45 degree wavefront schedule statically.

Muti-level tiling is supported in most of the fixed size tiled code generators and parametric tiled code generators [12, 50, 37, 38, 5] with regular shapes, and the 45 degree wavefront parallelization at each level. Shrestha et. al [90] extended the hyperplane technique used by Pluto [12, 4] to support concurrent start at the inner level.

The tile shape used in the above parametric tiled code generators is regular rectangular shape. Generating parametric tiled codes for other shapes like hexagonal or trapezoid shapes is known to be hard. Recently, Bertolacci et al. [9] presented an approach for producing parameterized diamond tiling with restrict to hyper-rectangular iteration spaces. However, the work is only done for one-level tiling.

3.3.2. Auto-tuning Frameworks and Domain Specific Compilers. During the past decades, many auto-tuning framework and domain specific compilers are developed for stencil computations. Kamil et al. [46] and Datta et al. [21] developed an auto-tuning framework for stencil computations that targets for different parallel platforms, including multi-core CPUs and GPUs. However, their framework only applies tiling within the same time step.

Pochoir [97] is a domain specific compiler developed for stencil computations, it takes a specified stencil computation, and generates tiled code with cache oblivious tiling. PATUS [17] is also a domain specific compiler for stencil computations, it takes a specification of stencil computation and parallelization strategy, then does code generation based on the specifications and the architecture characterization. However, both Pochoir and PATUS are restricted to Jacobi style stencils.

3.3.3. Vectorization. Due to the increasing prevalence of the SIMD architectures, there has been a spike of interest over the last decade on compiler techniques for automatically extracting SIMD parallelism and generating SIMD code. Two main categories of techniques have been explored: loop-based vectorizaton and the unroll-and-pack approach. The work we are looking at falls into the first category.

For loop-level vectorization, many authors vectorize the innermost loop with unit-stride or zero-stride memory access [3, 25]. Eichenberger et al. [25] proposed a method for vectorizing the loop with misaligned stride-one memory reference. Later, Nuzman et al. [71] extend loop-based vectorization to handle computations with power-of-two strides. Recently, work has also sought to address vectorization for arbitrary stride accesses [52, 16]. Techniques like data alignment adjustment and polyhedral transformations [102, 53] are also developed to increase the chance for vectorization.

For stencil computations, most of the loop-based vectorzation techniques fail to find an efficient parallelization strategy, because of the alignment conflict and reuse coming from multiple data streams. Especially for the tiled programs, only subsets of the data streams are operated, which causes most of the general alignment analysis to fail. Henertty et al. [39] address the alignment conflict issue for stencil computation by performing a non-linear data layout transformation. However, this work is not done in the context of tiling. They point out in their later work [40] that tiling on the transformed data layout imposes extra constraints on the legality of tiling, and proposed a technique to find legal tiling strategy based on formulation of a set of linear inequalities. This work is done for fixed-size tiling, because the nice linear property does not hold for parametric tiling.

The work presented by Kong et al. [53] is the only work we know of that addresses the vectorization problem for parametric tiling. Their method also targets vectoring the full tiles after applying parametric tiling techniques [5]. Since every tile separately still fits in polyhedral model (albeit with a few additional parameters), they first apply polyhedral analysis and transformations to tiles to make the innermost loop vectorizable. Then, extra time skewing and statement re-timing techniques are applied to minimize misaligned stores and loads—under the assumption that the first memory reference within a tile is aligned. Finally, SIMD code is generated to produce the so called prevect code, which further relies on a back-end compiler like SPIRAL to translate it to the final code with SIMD instructions. Optimizations like common sub-expression elimination, strength reduction, and replacing unaligned loads with aligned loads using data reorganization operation are also integrated in their work. The final experimental results on a Sandy Bridge machine show that significant performance improvement are achieved compared with the code generated from PTile [5] for 2D stencils, but a performance loss is exhibited for 3D cases. The alignment assumption made in their work is very optimistic, and generally does not hold for parametrically tiled iteration spaces. Also, different trade-offs are also introduced in the techniques, for example, the increased overhead of the pipeline fill-flush due to the extra time skewing, and the data reorganization overhead for replacing misaligned loads, but these trade-offs are not open for exploration in their framework.

3.4. Our Contributions

In our work, we target the energy optimization problem for compute intensive stencil computations for a single multi-core processor. Our goal is to develop a parametric tiled code generation framework for generating energy efficient code for stencil computations.

For the compute intensive stencil computations that we target, exploiting multi-level parallelization and maximum parallelism (concurrent start) on a single multi-core processor is not necessary. Due to the large value along all the dimensions and small number of resources available on a single multi-core processor (i.e., 4, 6, 8, 16, 32), there are generally enough independent tiles to exploit the available parallelism [101]. Let us take the J1D described in Equation 1 as an illustration example. Assume the tile size along the time dimension is tt , the tile size along the data dimension is ti, and there are P processors available on the platform. Then the total number of wavefronts can be approximated as $\frac{T}{tt} + \frac{N}{ti}$, and the number of wavefronts for the pipeline fill-flush is $2(P-1)$. Therefore, the steady state consists of $\frac{T}{tt} + \frac{N}{ti} - 2(P-1)$ wavefronts. When $\frac{T}{tt} \gg P$ and $\frac{N}{ti} \gg P$, the steady state dominates the whole execution, and the pipeline fill-flush overhead can almost be ignored. This is usually true for the stencils that we target.

Our energy efficient strategy is based on the classic time skewing and wavefront parallelization technique, combined with the multi-level tiling technique to explore data reuse among threads through the LLC. The work presented by Malas [61, 60] is the one that is the closest to our work. However, instead of exploring the trade-offs between the concurrent start and data reuse among threads, we seek to minimize the energy consumption by maximizing the data reuse among all the threads and still retain the high performance.

Moreover, we address the parametric tiled code generation support for our energy efficient parallelization strategy. As we will see later, our energy efficient parallelization strategy executes the tiles using a polynomial schedule, which is beyond the capability of the existing code generators. Polynomial scheduling was tackled by Achtziger and Zimmermann [1, 2], who also showed that they could provide better asymptotic execution times (under unbounded processor assumptions). However, the problem of exploiting such schedules remains open because of the code generation problem. In this research, we take the first step towards the code generation for polynomial schedules, and develop a code generation algorithm for one specific type of polynomial schedule for hyper-parallelepiped shaped iteration spaces.

Also, unlike most of the existing code generators that rely on the auto-vectorizers provided by production compiler (i.e., gcc, icc) to vectorize the generated code. We seek to automatically producing codes with explicit vectorization instructions that can better utilize the available SIMD (vector) units.

CHAPTER 4

Energy Efficient Tiling and Parallelization

We now present our energy-efficient tiling and parallelization strategy, as well as a quantitative analysis and justification of the reduction in off-chip memory accesses. First we describe the memory behavior of the standard wavefront parallelization, and show how execution in multiple passes (MPP) can reduce the off-chip memory accesses, but also note how it may lead to a potential slowdown due to the pipeline fill-flush overhead between every successive passes. Next we describe the intuition behind the method to regain this slowdown called flattened multi-pass parallelization (FMPP). We use the Jacobi 1D (described in Equation 1) as a running example to illustrate our approach and its memory behavior. Although the main ideas are simple, the challenge arises in retaining performance, and in automating the code generation.

4.1. Memory Behavior for Standard Wavefront Parallelization

In this section, we quantify the off-chip memory accesses for the standard wavefront parallelization of J1D based on a simple analytical model.

The standard wavefront parallelization for J1D is described in Figure 2.5. We assume, the memory accesses within a tile can all be made on-chip with proper choice of tile sizes. Also, the off-chip memory accesses only happen at the boundary of each tile, where accesses to the values produced by other tiles are made. Now, if the tile size is y along dimension i and x along dimension t, then the total number of tiles N_{tile} is approximately $\frac{T}{x} \times \frac{N+x-1}{y}$. When $N \gg x$, $N_{\text{tile}} \approx \frac{T}{x} \times \frac{N}{y}$. From the dependencies show in Figure 2.4b, we can see that one tile requires two columns of values from the left tile, one row from the bottom tile and a single value from the "south-west" tile (we ignore this latter one). Also, the left tile and bottom tile that it depends on are all executed in the previous wavefront. As a result, the data produced at the right and top boundaries of each tile of a wavefront has to be saved for the computation of next wavefront. When the data footprint of a wavefront is too large to fit into the last level cache, capacity misses will occur when memory accesses are made from the next wavefront. In the worst situation, every access made from the next wavefront can be a cache miss and lead to an off-chip memory access. Furthermore, the stencil computations that we are target for have large values along all dimensions, which are dominated by the steady state. Therefore, we assume that the boundary accesses made by every tile is a cache miss under the worst situation. The number of off-chip accesses of a tile is $(2x + y)$, then the total number of off-chip memory accesses V_{std} of the program is $N_{\text{tile}} \times (2x + y) = \frac{TN(2x+y)}{xy}$. Note that since $\frac{xy}{2x+y}$ is maximized when $2x = y$, and this choice of the tile aspect ratio minimizes V_{std} to $\frac{2TN}{y}$. This is further minimized by making y as large as possible, subject to the capacity constraints of the caches (private) on each core and the degree of parallelism.

4.2. Memory Efficient Tiling and Parallelization

As we described in the previous section, there can be no data locality for between the successive wavefronts when the data footprint of a wavefront is much larger than the LLC capacity for the standard wavefront parallelization. Therefore, The key idea of our energy efficient Multi-Pass Parallelization (MPP) is to improve the data locality between successive wavefronts.

4.2.1. Multi-Pass Parallelization (MPP) for Jacobi 1D.

To achieve reuse between successive wavefronts, we first partition the iteration space into passes (this can be

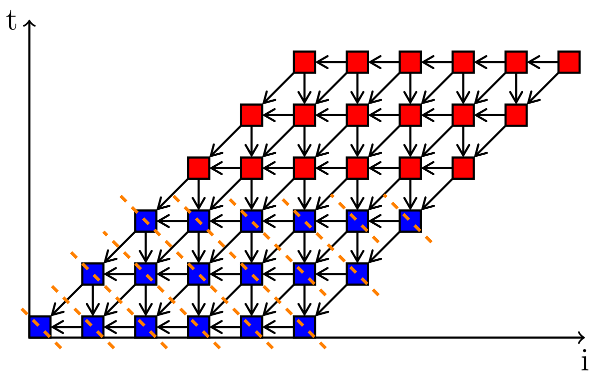


FIGURE 4.1. The tiled iteration space and multi-pass parallelization for J1D. Each small rectangle represents one tile, and each tile depends on its left tile, bottom tile and lower left tile. The whole tiled iteration space is separated into two passes (blue and red). Each orange dashed line represents one wavefront.

viewed as a special "outer" level of tiling) and then further tile each pass into tiles. The standard wavefront parallelization is applied to each pass, and passes are executed sequentially one after another. The pass size has to be chosen carefully, so that values produced by each wavefront in one pass fit in the LLC. Also note that the values needed from the previous pass will still be misses. This way, we optimize data reuse within one pass. Figure 4.1 illustrates multi-pass parallelization for the J1D example.

Let the pass height be H, and the tile sizes for the tiles within a pass be x' and y' (the optimal tile sizes for MPP may be different from the optimal ones for the standard wavefront parallelization). Since H is chosen so that there are no cache misses within one pass, the cache misses only occur between passes. Since the computation in one pass requires the last row from the previous pass, the volume, V_{multi}, of off-chip data transfer for J1D with MPP can be estimated as the product of the number of passes and the row size, $V_{\text{multi}} = \frac{T}{H} \times N = \frac{TN}{H}$. Comparing with $V_{\text{std}} = \frac{2TN}{y}$ for the standard wavefront parallelization, and noting that usually $H \gg y$, we expect the multi-pass parallelization to yield significant savings.

4.2.2. Higher Dimensional MPP. MPP can be generalized to higher dimensions. For a d-dimensional stencil (i.e., with $d+1$ dimensions in total), we first tile the outer d dimensions into passes, and then, for each pass, perform an inner level tiling on all the dimensions.

Let us analyze memory behavior for the higher dimensional case. As in J1D, when the data space is "large enough", every inter-tile access will cause a cache miss in the standard wavefront parallelization, but in MPP, only those inter-tile accesses that cross a *pass boundary* will cause cache misses.

Consider a 3D program with a regular $T \times N \times M$ iteration space, and assume that rectangular tiling can be applied without time skewing (this is an approximation to get asymptotic bounds). If the tile size for the standard wavefront parallelization is $t \times x \times y$, then the total number of tiles is $N_{\text{tile}} = \frac{T}{t} \times \frac{N}{x} \times \frac{M}{y}$. Assume that one tile requires a faces from its left tile and b faces from the bottom tile and c faces from its front tile, and let V_{tile} represent the number of off-chip memory accesses of one tile. Then $V_{\text{tile}} = a \times t \times y + b \times x \times y + c \times t \times x$.

The total number of off-chip memory accesses for the whole program with standard wavefront parallelization is

$$V_{\text{std}} = V_{\text{tile}} \times N_{\text{tile}} = \frac{TNM(aty + bxy + ctx)}{txy}. \tag{3}$$

If the pass sizes for the outer two dimensions are H_t and H_x, then the total number of passes N_{pass} for the whole iteration space is $\dfrac{T}{H_t} \times \dfrac{N}{H_x}$. Each pass requires a faces of values from its left pass, b faces from its bottom passes, and no pass is in front of a pass. Therefore,

$$\begin{aligned} V_{\text{multi}} &= N_{\text{pass}} \times (a \times M \times H_t + b \times M \times H_x) \\ &= \frac{TN(aMH_t + bMH_x)}{H_t H_x}. \end{aligned} \tag{4}$$

Then, our multi-pass parallelization can achieve better memory performance under the situation when $\frac{M(aty+bxy+ctx)}{txy} > \frac{(aMH_t+bMH_x)}{H_t H_x}$, and our experimental results show that this is usually achievable. Of course, our analysis is approximate, and only considers capacity misses. In practice many factors affect memory behavior, such as data access order, conflict misses etc. Our main goal was to motivate the MPP strategy in a simple quantitative way, and building a precise model for cache misses is out of the scope of this work.

4.3. Flattened Multi-Pass Parallelization (FMPP)

Recall that for our goal of energy-efficiency, it is critical to retain all speed optimizations of the original code, otherwise we risk losing more than the savings we gain from the off-chip accesses. In the standard parallelization of MPP, the computation in one pass starts after the previous pass is complete. As illustrated in Figure 4.2(a) this introduces idle time between passes due to pipeline flush-fill. This situation will also happen in the standard wavefront parallelization, but only once during the whole computation, whereas it happens for each pass for MPP. This effect will be exacerbated for higher dimensional cases. Indeed, our preliminary experiments showed that the MPP strategy as described above was up to 20% *slower* than the standard wavefront parallelization, for programs with higher dimensions. The overhead can be eliminated if we remove the "gap" between two successive passes by overlapping them as shown in Figure 4.2(b). We call this the *flattened multi-pass parallelization strategy* (FMPP). In this way, the fill-flush overhead will be incurred only once during the whole computation. Although this transformation is intuitively very simple, it raises significant challenges for code generation. We developed a code generation algorithm to support such overlapped execution order by integrating a specific polynomial schedule, and details are presented in Chapter 5.

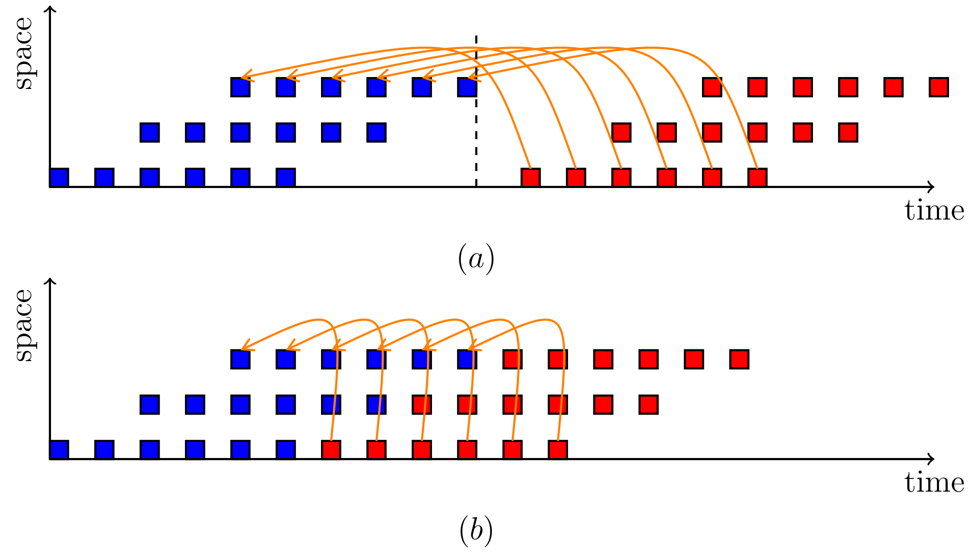


FIGURE 4.2. The space-time diagram of the (a) standard, and (b) flattened multi-pass strategy for J1D. The horizontal axis represents the wavefront time step. Note that in the standard scheme, the time stamp is a *multi-dimensional vector*, and in the flattened strategy it is a *quadratic function* of the tile coordinates, pass number and row size.

Also, note that there may be a limit on how much overlap is allowed, since the inter-pass dependencies (shown in Figure 4.2 as orange edges) must be satisfied. We ensure the legality of our code through constraints on tile sizes. As we will describe in Chapter 5, the wavefront time step for a tile depends on the tile size, and we can control the tile size so that the wavefront time step for the first tile in a pass starts after the tile it depends on (usually the first tile in the last row of the previous pass) from the previous wavefront.

CHAPTER 5

Code Generation for FMPP

We developed a parametric tiled code generation algorithm for our FMPP strategy described in Chapter 4 based on existing parametric tiled code generation techniques – D-Tiling [50]. Currently, our method is limited to d-dimensional hyper-parallelepiped shaped iteration spaces, whose bound expressions involve only the outermost loop index and parameters.

In this chapter, we first explain the limitations of the state of the art, and the challenges raised in the code generation for our FMPP strategy. Then, we present our detailed code generation algorithm for FMPP. For the simplicity of explanation, we first show how to generate sequential code for a 2D parallelogram iteration space with no tiling within each pass, and then generalize it to include tiling within each pass. Finally, we generalize it to higher dimensions, and wavefront parallelization.

5.1. Code Generation Challenges

Consider a standard cubic iteration space, $\mathcal{D} = \{i, j, k \mid 0 \leq (i, j, k) < N\}$. If we want the k dimension to be executed sequentially, while in each $\langle i, j \rangle$ plane, we want to parallelize using the $t = i + j$ wavefronts. The natural multidimensional schedule for this is $(i, j, k \to k, i + j)$. The "space-time diagram" of resulting program is shown in Fig 5.1(a) as a sequence of rhombuses. Consistent with lexicographic order, the earliest point in any rhombus comes *after* the last point in the previous rhombus. The multidimensional schedules are supported by existing code generators. However, now we wold like to avoid the "pipeline

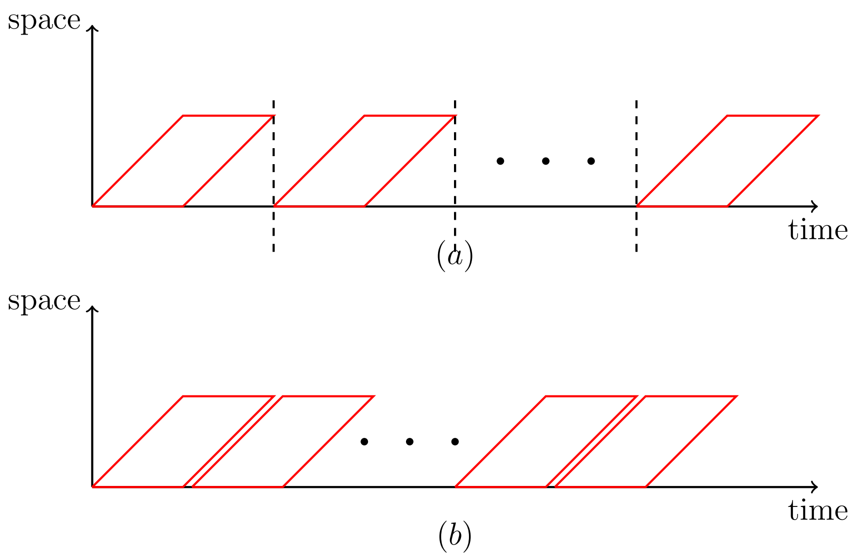


FIGURE 5.1. The "space-time diagram" of traversing domain $\{i, j, k \mid 0 \leq (i, j, k) < N\}$ with a 2-D schedule $(i, j, k \rightarrow k, i + j)$" (top) is a sequence of rhombuses where one cannot start before the previous is completely done, leading to idle times between the rhombuses. The space-time diagram without the inefficiencies (bottom) may be desirable. However, it corresponds to a polynomial schedule $(i, j, k \rightarrow Nk+i+j)$. Current polyhedral code generators cannot produce code with this behavior.

fill-flush overhead" as shown in the space-time diagram in Fig 5.1 (b), and it corresponds to a *polynomial schedule* $t = Nk + i + j$, not a multidimensional linear one. Handling the polynomial schedule raises interesting challenges.

First, note that the corresponding transformation for polynomial schedule is a nonlinear transformation, $\mathcal{Z}^d \rightarrow \mathcal{Z}$, that maps a d-dimensional point in the original space to a single integer in the new outer loop. From this, we need to deduce the original index point (actually, the original pass and tile coordinates). By restricting to hyper-parallelepiped shaped domains, we will show how to obtain simple functions to deduce these inverse mappings as closed form functions involving the tile size parameters.

Second, dealing with tile loops raises additional complications. For example, for hyper-parallelepiped shaped iteration spaces, although the number of *iterations* of any loop is independent of the values of surrounding loops, the same assertion cannot be made about the loops that visit the tile origins. Furthermore, as we shall see later, we also need to determine the number of tile origins visited in the very first instance of the innermost loop in any given pass, and ensure that this is independent of the specific pass.

Third, a pass is simply a tiling of the outer $d-1$ loops, leaving the innermost one untiled. Although the original loop nest is fully permutable, the multi-pass strategy introduces inter-pass dependencies that are not lexicographically positive, and this means that the untiled dimension is no longer permutable with the others, and it must be made the outermost loop traversing the tiles over the flattened space.

Finally, note that there may be a limit on how much overlap is allowed, since the inter-pass dependencies must be satisfied. We ensure the legality of our code through constraints on the tile sizes.

The code for the standard MPP strategy (as shown in Figure 4.1) can be generated with a simple extension to the existing multi-level parametric tiled code generation techniques. Figure 5.2 describes the loop structure for J1D example with the standard MPP strategy. The outermost loop iterates over the passes, followed by a time loop that enumerates the wavefront time steps for a given pass. Then, the tile loops enumerate tiles for a wavefront and finally a set of point loops are attached to scan each point within a given tile. The implicit synchronization that happens at the end of tt loop introduces the overhead as described in Figure 4.2(a). In order to eliminate this overhead, we need to generate code that supports our FMPP strategy.

Despite the challenges raised for supporting the polynomial schedule for our FMPP strategy, the final loop structure for FMPP still remains conceptually simple. The loop nest

```
//Outermost pass loop: iterates over the passes
for(pt =$LB_t^{pass}$; pt<=$UB_t^{pass}$; pt+=$P_t$)
 //time loop:enumerates the wavefront time steps within each pass
 for(time=start(pt); time<=end(pt); time++)
   //Tile loops: enumerate the tiles within each wavefront
   #pragma omp parallel
   for(tt=$LB_t^{tile}(pt,time)$; tt<=$UB_t^{tile}(pt,time)$; tt+=$S_t$)
     for(ti=$LB_i^{tile}(pt,time,tt)$; ti<=$UB_i^{tile}(pt,time,tt)$; ti+=$S_i$)
       //Point loops: scan the points within a tile
       for(t =$LB_t(tt,ti)$; t<=$UB_t(tt,ti)$; t++)
          for(i=$LB_i(tt,ti)$; t<=$UB_i(tt,ti)$; i++)
               Body(t, i);
```

FIGURE 5.2. The loop nest for the standard MPP strategy for J1D. The loop bounds are functions of the outer loop index values and problem parameters. For example, $LB_t^{tile}(pt, time)$ represents the lower bound for the tile loop along t dimension for the pass pt at time step $time$.

should have an outermost loop iterating over the wavefronts for the flattened space, followed by tile loops that enumerate the tiles that can be executed within a wavefront, and then point loops should be attached to scan the points in a visited tile. Due the non-uniform dependencies along the aligned dimension (dependencies between passes), this dimension is made the outer dimension while enumerating tiles. Figure 5.3 shows the loop structure for the J1D example with FMPP strategy: note that the tile loop ti is now an outer loop of the tiled loops instead of as an inner loop in Figure 5.2, which iterates over the tiles that are flattened along the innermost dimension.

```
//Time loop: iterates over the wavefront steps
for(time=start; time<=end; time++)
 //Tile loops: enumerate the tiles within each wavefront
 #pragma omp parallel
 for(ti=$LB_i^{tile}(time)$; ti<=$UB_t^{tile}(time)$; ti+=$S_i$)
   for(tt=$LB_t^{tile}(time,ti)$; tt<=$UB_t^{tile}(time,ti)$; tt+=$S_t$)
     //Point Loops: scan points within a tile
     for(t=$LB_t(tt,ti)$; t<=$UB_t(tt,ti)$; t++)
       for(i =$LB_i(tt,ti)$; t<=$UB_i(tt,ti)$; i++)
           Body(t, i);
```

FIGURE 5.3. The loop nest for the FMPP strategy for J1D.

We now describe the sequential FMP code generation algorithm for a 2D parallelogram iteration space. We first show how to generate sequential code for a 2D parallelogram iteration space with no tiling within each pass, and then generalize it to include tiling within each pass.

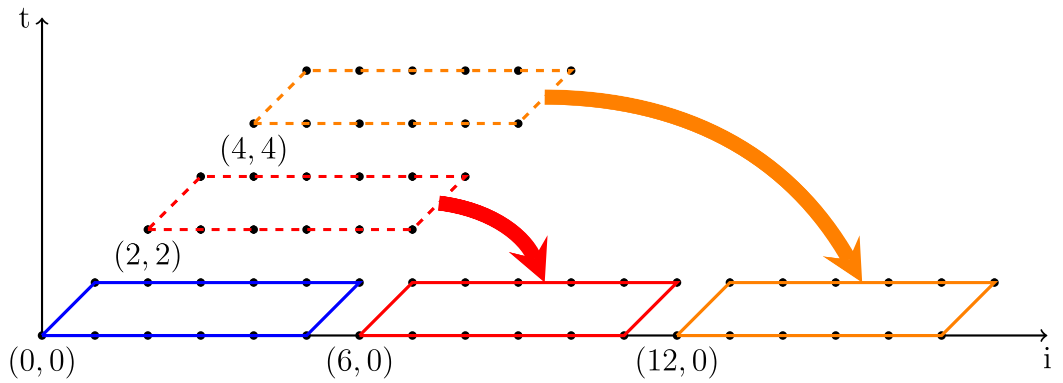


Figure 5.4. Flattened iteration space for a parallelogram iteration space. The blue box is the first pass for both the original iteration space and flattened space. The red dashed box is the second pass in the original iteration space, and the red solid box is the second in the flattened space. Same for the orange boxes.

5.2.1. Multiple Passes Without Tiling. Figure 5.4 describes the original multi-pass space and the flattened space for a parallelogram iteration space, and tiling is not applied for each pass. The code we seek to generate is a two dimensional loop nest that enumerates all the iteration points in the flattened space, with the outer one visiting all the points along the flattened *data* dimension i.

5.2.1.1. *Generating The Outer Loop.* As shown in Figure 5.4, the shape of each pass remains a parallelogram when no tiling is applied within passes, therefore, the number of iteration points in the first row is the same for all the passes, and is the *period* of a pass after the passes are laid out one after another, denoted by N_{period}. Let n_{pass} be the number of passes. Then, N_{fmp}, the total number of iteration points along the laid out dimension in the flattened space is the product of the two, plus pipeline fill-flush, which we count as part of the last pass ($N_{\texttt{last}}$). Then,

$$N_{\text{fmp}} = N_{\text{period}}(n_{\text{pass}} - 1) + N_{\text{last}}$$

For the example in Figure 5.4, $N_{\text{period}} = 6$, $n_{\text{pass}} = 3$, $N_{\text{last}} = 7$, $N_{\text{fmp}} = 18$. In general, these values can be determined with purely syntactic manipulation of the bounds expressions of the original loops. Figure 5.5(a) describes the loop nest for the non-flattened multi-pass space for a 2D parallelogram iteration space, where loop p iterates over the passes, and loop t and i iterates over the points within each pass. In the flattened space, the passes will be linearized along the i dimension. When the loop nest is fully permutable, we can even change the enumeration order for the points in each pass, and this is always true for a tilable program. Figure 5.5(b) shows a permuted loop nest that is equivalent to the one in Figure 5.5(a), but the points inside each pass are enumerated along the i dimension first. Since the p loop in Figure 5.5(a) iterates over the origins of each pass, n_{pass} is the number of iteration points in loop p. Also, the loop i in Figure 5.5(a) iterates over the points along dimension i for a given t iteration, then N_{period} is equivalent to the number of iteration points in the i loop within the first iteration of t loop. Similarly, the number of points along dimension i enumerated in the last pass is the number of iteration points for the i loop within the last pass iteration, and the corresponding loop nest is the i loop in Figure 5.5(b). For a given loop l, with lower bound lb, upper bound ub and stride s, the number of iteration points in loop l is iters $= \lceil \frac{ub-lb+1}{s} \rceil$, and the last iteration point in loop l is $(\text{iters}-1) \times s + lb$.

```
for(p=0;p<=T;p+=h)
 for(t=p;t<=min(p+h-1,T);t++)
  for(i=t; i<=t+M; i++)
      Body(t,i);
```

(A)

```
for(p=0;p<=T;p+=h)
 for(i=p;i<=min(p+h-1,T)+M;i++)
  for(t=max(p,i-M);t<=min(i,p+h-1,T);t++)
      Body(t,i);
```

(B)

FIGURE 5.5. The loop nests for the multi-pass iteration space with the original iteration order (a) and permuted order(b) before flattening. The number of iteration points in each row is M and there are T iterations along the t dimension. h is the pass height.

After N_{fmp} is computed, the outermost loop of the flattened space can be constructed as a loop that iterates from 0 to N_{fmp} with stride 1.

5.2.1.2. *Generating The Inner Loop.* The inner loop for the flattened space enumerates the corresponding points for a visited point along the linearized dimension in the flattened space. Note that the t loop in Figure 5.5(b) visits the points along t dimension for a given i iteration in the original space, which is exactly the information we need. However, in order to use this information, we need to convert the visited i in the flattened space back to the i values in the original space. For now, let us ignore the points in a pass that overlap with the next pass. Let the ti-th point along the i dimension in the flattened space be the n-th iteration along dimension i within the m-th pass. Then $ti = mN_{\text{period}} + n$, and $0 \leq m < n_{\text{pass}}, 0 \leq n < N_{\text{period}}$. Furthermore, given ti and N_{period}, the m and n can be computed as follows.

$$n = ti \mod N_{\text{period}}, \quad m = \left\lfloor \frac{ti}{N_{\text{period}}} \right\rfloor$$

Then the corresponding pass origin $\langle p, i \rangle$ in the original space is the m-th iteration value of the p loop and the n-th iteration value of the i loop in Fig 5.5(b) separately. The j-th iteration point for a given loop l with stride $s$is $t \times s + lb$. After the i value in the original multi-pass space is computed, the loop t in Figure 5.5(b) can be attached to visit the corresponding points along t.

5.2.1.3. *Handling The Overlapped Part.* We handle the overlapped part between passes by checking whether there are points from the current pass and also the previous pass for a visited point along the linearized dimension. Then, the points that are ignored at the current pass will be visited as the points in the previous pass during the execution of next pass. The previous pass information can be deduced easily from the current pass information, for example, the pass number is $m - 1$, and the iteration number along i is $n + N_{\text{period}}$. Our algorithm generates all those expressions by syntactic manipulations of bound expressions in the loop AST, and the resulting code is shown in Figure 5.6.

```
n_pass=ceild(T,h);
t=0;  p=t;  N_period=M+t-t;
p=(n_pass-1)*h;  N_last=min(p+h-1,T)+M-p;
N_fmp=(n_pass-1)*N_period+N_last;
for(ti=0; ti<N_fmp; ti++){
 p=(ti/N_period)*h;  i=(ti%N_period)+p;
 if((0<=p<=T) and (p<=i<=min(p+h-1,T)+M))
   for(t=max(p,i-M);t<=min(i,p+h-1,T);t++)
      Body(t,i);

 p=(ti/N_period-1)*h;
 i=ti%N_period+N_period+p;
 if((0<=p<=T) and (p<=i<=min(p+h-1,T)+M))
   for(t=max(p,i-M);t<=min(i,p+h-1,T);t++)
      Body(t,i);
}
```

FIGURE 5.6. The final sequential FMP code for the parallelogram with no second level tiling.Function ceild(a, b) returns $\lceil \frac{a}{b} \rceil$.

5.2.2. Sequential FMP with Tiling within Each Pass. When tiling is applied to each pass, instead of generating loops that visit each point within the flattened pass, we need to generate tiled loops that visit the tile origins in the flattened pass and point loops that visit each point within a tile. Our parametric tiled code generation algorithm is based on an existing parametric tiled generation technique — DTiler [50]. In the following description, we first provide a brief recap of DTiler algorithm, and then describe our extension to DTiler for generating parametric tiled code with our FMPP strategy.

5.2.2.1. *Recap of DTiler.* DTiler takes a d-dimensional loop nest, then generates two loop nests separately—the d-dimensional *tile loops* and the d-dimensional *point loops.* The tile loops are generated by modifying the upper and lower bound expressions of the input loop nest so that they visit a parametrically expanded set of points called the *outset* [85, 51] —a superset of the iteration space guaranteed to contain all the origins of non-empty tiles, and keeping tile size as a symbolic parameter used as the loop stride. The outset of the iteration space can be constructed by updating the lower and upper bound for each dimension using the `shift_down` and `shift_up` function. For the kind of iteration spaces that we are handling, the lower/upper bounds of a dimension t are an affine functions that involves the outer loop indexes and the program parameters, which can be represented as

$$lb_t = \sum_{c_i>0} c_i z_i + \sum_{c_j<0} c_j z_j + \sum_{b_h} b_h p_h + c$$

where c_i, c_j are the coefficients for the outer loop index, b_h is the coefficient for the program parameter. The `shift_down` function updates the lower bound with the following computation:

$$\texttt{shift\_down}(lb_t) = lb_t + (\sum_{c_j<0} c_j(s_j - 1)) + (s_t - 1) \tag{5}$$

where s_i is the tile size used for the corresponding dimension. The `shift_up` function updates the upper bound using Equation 6.

$$\texttt{shift\_up}(ub_t) = ub_t + (\sum_{c_i>0} c_i(s_i - 1)) \tag{6}$$

After the outset is constructed, the lower bounds of the tiled loops is shifted to align with the first tile origin within the outset. DTiler assumes $\vec{0}$ is a legal tile origin, and computes the shifted lower bound using Equation 7.

$$\left\lceil \frac{lb_i}{s_i} \right\rceil s_i \tag{7}$$

where lb_i is the lower bound of loop i and s_i is the stride (tile size) for loop i.

5.2.2.2. *Tiling within Passes.* Note that one important fact that we have used for generating code in Figure 5.6 is that N_{period} is the same for all passes. However, this may not be true when the iteration space is tiled. Figure 5.8 describes the tiled iteration space with $\vec{0}$ as a legal tile origin (an assumption made by most existing tiled code generators) for a parallelogram iteration space whose first iteration point is $(2, 2)$. As shown in Figure 5.8, although the number of iteration points are the same for each row in each pass, the number of tiles in the first row are different for the first pass and second pass. To resolve this, we align the first tile origin to the first iteration point of the space being tiled, instead of assuming that $\vec{0}$ is a tile origin. As described before, when the tile origin is $\vec{0}$, lower bounds of the tiled loops can be shifted to align with the first tile origin within the outset using Equation 7. Now, if v_i is a legal tile origin along i, we can first shift the whole iteration along dimension i by $-v_i$ to make 0 a legal tile origin along dimension i, then compute the shift in the same way as DTiler and finally shift the whole iteration space back, and the computation is the following:

$$\left\lceil \frac{lb_i - v_i}{s_i} \right\rceil s_i + v_i \tag{8}$$

Now, we will show the mathematical proof for the equivalence of N_{period}. Consider the 2D nested loop shown in Figure 5.7 (a), it describes a general 2D parallelogram iteration space, whose first iteration point is $(a, k \times a + b)$, and k is the tangent value for the angle of the parallelogram (k is fixed for a given parallelogram). The whole iteration space has to be first tiled into passes, and the pass loop can be obtained by two steps: 1) construct the outset of the iteration space; 2) align the pass origin to the first iteration point in the iteration space.

For the 2D parallelogram space, only the outer t dimension has to be tiled into passes, so we only need to update the lower and upper bound for the dimension t using Equation 5 and Equation 6. Moreover, dimension t is the outermost dimension in the original loop, whose lower/upper bounds do not involve any outer loop index, and therefore, the bounds for dimension t remain the same for the outset. Now, we align the lower bound of loop t to the first iteration point in the iteration space, which is $(a, k \times a + b)$. Since the offset along dimension t is a, the lower bound for the pass loop is $\left\lceil \frac{a+1-s_t-a}{s_t} \right\rceil s_t + a$, which can be simplified to a, and the result is shown in Figure 5.7 (b). Then the first iteration point for each pass is $(p, k \times p + b)$, and a similar process can be applied to generate the tiled loops for each pass, the final loop nest is shown in Figure 5.7 (c). The N_{period} for the tiled iteration space is the number of iteration points in the ti loop, which is the following:

$$
\begin{aligned}
N_{\text{period}} &= \left\lceil \frac{ub_{ti} - lb_{ti}}{s_i} \right\rceil \\
&= \left\lceil \frac{k(tt-p) + k(s_t - 1) + N - b}{s_i} - \left\lceil \frac{k(tt-p) + 1 - s_i}{s_i} \right\rceil \right\rceil
\end{aligned} \tag{9}
$$

Since N, s_t, s_i, and k are either a program parameter or a constant value for a given program, the only term that may involve the pass index value is $k(tt-p)$. Since the first iteration value of tt is p, $k(tt-p) = 0$. Then N_{period} is independent of the pass value. Similar to the flattened iteration space without second level tiling, the value for n_{pass} is $\lceil \frac{T-a+1}{h} \rceil$, and the tiles in the last row can be figured out from the permuted loop. After converting the tile visited along the i dimension back to the tile origin in the original iteration spaces, the corresponding tiles that have to be enumerated can all be obtained from the permuted loop nest.

Now, we show how to construct the loops. We call DTiler with the first iteration point in the space for tile as a legal tile origin instead of $\vec{0}$ *extended DTiler*. In order to get the loops for the two level tiling, we first apply the extended DTiler on the outer dimension to generate one tiled loop called *pass loop* and one point loop. Then the untiled inner dimension is attached inside the point loop to create a loop nest that iterates over the points in a pass. Since each pass is a polyhedron, we extract the polyhedral domain out of the created loop nest, and generate the permuted loop nest by calling CLooG [6] with the permuted domain. Finally, extended DTiler is applied on both original loop nest and the permuted loop nest that iterate over the points in a pass.

```
for(t=a; t<=T; t++)
  for(i=k*t+b; i<=k*t+M; i++)
      Body(t,i);
             (a)
```

```
for(p=a; p<=T; p += h)
  for(t=p; t<=min(p+h-1,T); t++)
    for(i=k*t+b; i<=k*t+M; i++)
        Body(t,i);
             (b)
```

```
for(p=a; p<=T; p+=h)
 for(tt=p; tt<=min(p+h-1,T); tt+=st)
  for(ti=ceild(k*t+b+1-si-(k*p+b),si)+(k*p+b);ti<=k*(tt+st-1)+M;ti+=si)
   for(t=tt;t<=min(tt+st-1,p+h-1,T);t++)
    for(i=max(ti,k*t+b);i<=min(ti+si-1,k*t+M;i++)
       Body(t,i);
             (c)
```

Figure 5.7. (a) The original loop nest (b) Loop nest after the first level tiling. (c) Loop nest after the second level tiling. h is the pass size, s_t and s_i are the tile size for the small tiles along t and i dimension. T and M are problem size parameters.

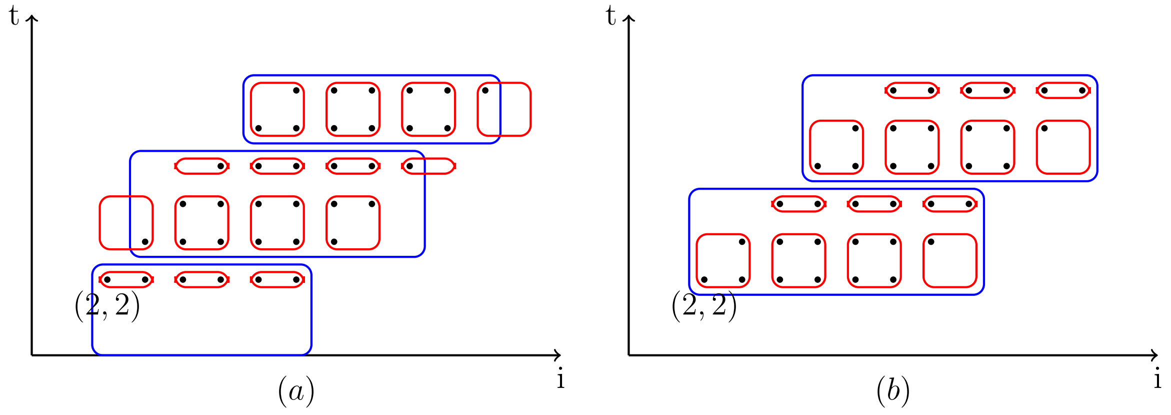


Figure 5.8. Tiled space for a parallelogram iteration space with the legal tile origin $\vec{0}$ (a) and the first iteration point (b). The first iteration point is $(2, 2)$. Each blue box represents one tile, and each red box is a small tile within a pass.

After obtaining the tiled loops, we can construct the tile loops that enumerate the tiles in the flattened space using the idea we described before when no tiling is applied within passes. Then, the point loops that iterate over the points inside each pass is needed. The point loops are generated in the same way as DTiler, where we take the original loop nest, and combine its original lower and upper bounds with the bounds of the tiles at all levels. For a loop at dimension i, its lower bound is replaced with $\max(lb_i, t_i)$, and the upper bound is replaced with $\min(ub_i, t_i + s_i - 1)$, where t_i is the tiled loop index for the corresponding dimension at the current tiling level, and s_i is the corresponding tile size. Figure 5.9 shows the final loop structure generated for the sequential FMP for a 2D parallelogram iteration space. Compare with the loop structure we described in Figure 5.3, the tile loops that visits the tiles within an outer loop iteration is now separated into two parts, one part visits the tiles come from current pass, and the other part visits the tiles from the previous pass. Corresponding guard test is performed to make sure the parts that are visited exists.

5.3. Sequential FMP for Higher Dimensional Iteration Space

Now that we have explained the basic ideas for the "easy-to-visualize" cases, we develop a number of extensions: first to higher dimensions, leading to the complete algorithm for sequential code. Next, we extend to standard wavefront parallelization of each pass.

We also note that although the development here deals with perfectly nested loops, most practical programs contain imperfectly nested loops. In our code generation algorithm, imperfectly nest loops are handled in the same way as DTiler. First, a preprocessing is used to create embedded imperfectly nested loops, for which each statement is surrounded by the same number of loops and the iteration spaces for the statements are disjoint. Then a perfectly nested loop with guards is derived from the embedded imperfectly nested loops. Then our code generation algorithm is applied on the perfectly nested loops to generate tiled loops. Because this follows directly from the earlier results, we do not describe it in detail here.

```
n_pass=...; N_period=...;
N_last=...; N_fmp=...;
for(ti = 0;ti<N_fmp;ti++){
   cur_pass =ti/N_period; p=...;
   cur_i = ti%N_period;  ti=...;
   /* check for current pass */
   if((lb_p<=p<=ub_p) and (lb_ti<=ti<=ub_ti))
     for(tt=lb_tt; tt<=ub_tt; tt+=st)
         //point loops
        for(t=max(tt,max(p,lb_t);t<=min(tt+st-1,p+h-1,ub_t);t++){
          for(i=max(ti,lb_i);i<=min(ti+si-1,ub_i);i++)
                 Body(t,i);

   /*check for the previous pass*/
   cur_pass=cur_pass-1;  p=...;
   cur_i=cur_i+N_period;  ti=...;
   if((a<=p<=T) and (lb_ti<=ti<=ub_ti))
     for(tt=lb_tt; tt<=ub_tt; tt+=st)
        //point loops
        for(t=max(tt,max(p,lb_t));t<=min(tt+st-1,p+h-1,ub_t);t++)
          for(i=max(ti,lb_i);i<=min(ti+si-1; ub_i);i++)
                 Body(t,i);
}
```

FIGURE 5.9. The loop structure for parallelogram iteration space with FMP: lb_p and ub_p are lower and upper bounds for the pass loop, lb_ti, ub_ti, lb_tt and ub_tt are lower and upper bounds for loop *ti* and *tt*, and lb_i, ub_i, lb_t and ub_t are lower and upper bounds for the original *i* and *t* loop.

5.3.1. SEQUENTIAL FMP FOR HIGHER DIMENSIONS. Above, we described a code generation algorithm for the 2D cases, and we now generalize it to higher dimensions. The main idea for handling the higher dimensional cases remains the same: if the number of tiles visited along the innermost dimension in the first row for each pass is the same, then the total number of tiles that have to be visited along the innermost dimension for the flattened space is $N_{\text{fmp}} = (n_{\text{pass}} - 1)N_{\text{period}} + N_{\text{last}}$. Then for each tile visited along the innermost dimension in the flattened space, it is transformed back to the original tiled iteration space, and the corresponding tiled loops and point loops can be obtained from the permuted loop nest.

First, let us check if $N_{\texttt{period}}$ is still the same for higher dimensional cases. Due to the regularity of the iteration spaces that we are handling, the inner most loop is always a loop that is skewed with respect to the outermost dimension, which has the same form as the inner loop in Figure 5.7 (a). Assume that the value along the outermost dimension for the first iteration point in a pass is bp, then the corresponding offset along the innermost dimension would be $k \times bp + b$, and the lower bound for the innermost tiled loop is $\left\lceil \frac{k \times tt + b + 1 - s_i - (k \times bp + b)}{s_i} \right\rceil \times s_i + (k \times bp + b)$, and the upper bound for the innermost tiled loop is $k \times (tt + s_t - 1) + M$, where tt is the loop indices for the outermost dimension, s_t is the tile size for the outermost dimension within a pass, and s_i is the tile size along the innermost dimension within a pass. With analysis similar to that for Equation 9, we can show that N_{period} is independent of the passes.

The main problem raised in the higher dimensional cases is how to compute n_{pass} and transform each visited tile iteration along the innermost dimension back to the original iteration space. Because more than one dimension is tiled into passes in the high dimensional cases, n_{pass} is not just the number of iterations in the outermost pass loop, and the pass number obtained from $\dfrac{ti}{N_{\text{period}}}$ is just the pass number in the unrolled space.

Now, let us see how to handle the passes in higher dimensions. Each pass here can be given a multi-dimensional pass number $(n_1, n_2, \ldots, n_{d-1})$, where n_l is the pass number for the passes along the l-th dimension, and d is the total number of dimensions. If the number of iterations along dimension l are the same for all the $(n_1, \ldots, n_{l-1})$, represented by n_{pass_l}, then the total number of passes is the product $\Pi_{l=1}^{d-1} n_{\text{pass}_l}$. Furthermore, pass number k in the unrolled space for $(n_1, n_2, \ldots, n_{d-1})$ is $k = n_1 \displaystyle\prod_{i=2}^{d-1} n_{\text{pass}_i} + n_2 \prod_{i=3}^{d-1} n_{\text{pass}_i} + \cdots + n_{d-1}$, where

$0 \le n_i < n_{\text{pass}_i}$. Given k and n_{pass_l} we compute n_l for each l as follows.

$$n_l = \frac{k - (n_1 \prod_{i=2}^{d-1} n_{\text{pass}_i} + \cdots + n_{l-1} \prod_{i=l}^{d-1} n_{\text{pass}_i})}{\prod_{i=l+1}^{d-1} n_{\text{pass}_i}}$$

In order to use the above formulas, the number of passes along dimension l has to be the same for all $(n_1, \ldots, n_{l-1})$. We achieve this with the same idea as how we obtain the same number of tiles in the first row. First, we only tile the outermost dimension into passes, which creates a set of *hyper-slabs.* Because of the assumption of hyperparallelepipedic iteration spaces, each hyper-slab will be a shift of the first hyper-slab, except for the last one, which is bounded by the original iteration space, and may be smaller. We pad this so that all hyper-slabs are identical (this causes us to redundantly visit some empty tiles, but can be avoided by choosing the right tile sizes). Because of the additional assumption that iteration space bounds are functions of only the outermost dimension, the remaining inner pass dimensions can be handled all at once, with the offset vector coming from the first iteration point of the hyper-slab. Then all the techniques we described above can be used. In Algorithm 1, we described the complete code generation algorithm for the sequential FMP strategy.

5.4. Wavefront Parallelization for FMP Strategy

After the tiled iteration space is flattened, the wavefront parallelization strategy is applied to parallelize the computations. For a tile whose tile coordinate is $(n_1, n_2, \ldots, n_d)$ in the flattened space, it is executed at time stamp $w = \sum_{k=1}^{d} n_k$. With the sequential FMP code generated, n_1 is the index value of the outermost loop ti, and n_k can be computed using $\frac{t_k - v_k}{s_k}$, where t_k is the tile origin value at the k-th dimension, v_k is the offset at the k-th

Algorithm 1 Code generation algorithm for the sequential FMP strategy.

Input: Tiled loop nest L and $L^{'}$ for the original iteration space and the permuted space, and the point loop nest $L_{\texttt{point}}$. The number of dimensions for the original iteration space is d, the loop depth for both L and $L^{'}$ is $2d-1$, the first $d-1$ loops are the same for both L and $L^{'}$, which are tiled loops for passes (represented by L_{pass}), the following d loops are tiled loops for each pass (represented by L_{tiled} and $L^{'}_{\text{tiled}}$ separately for the original iteration space and permuted space). For a given loop l, the lower bound is lb_l, the upper bound is ub_l, the stride is s_l and the loop index is t_l.

Output: Loop Nest for FMP.

```
 1: for each loop $l$ in $L_{\text{pass}}$ and $l$ is the $i$th loop in $L_{\texttt{pass}}$ do
 2:     print $\texttt{npass}_i = \lceil\frac{ub_l-lb_l+1}{s_l}\rceil$;
 3: end for
 4: print $\texttt{n\_pass} = \prod_{i=1}^{d-1}\texttt{npass}_i$
 5: for each loop $l$ in ($L_{\texttt{pass}}$ and the outer $d-1$ loops of $L_{\texttt{tiled}}$) do
 6:     print $t_l = lb_l$;
 7: end for
 8: Let loop $l$ be the $d$th loop in $L_{\texttt{tiled}}$
 9: print $\texttt{N\_period} = \lceil\frac{ub_l-lb_l+1}{s_l}\rceil$;
10: for each loop $l$ in $L_{\texttt{pass}}$ do
11:     print $t_l = (\lceil\frac{ub_l-lb_l+1}{s_l}\rceil - 1) \times s_l + lb_l$
12: end for
13: Let loop $l$ be the first loop in $L^{'}_{\texttt{tiled}}$
14: print $\texttt{N\_last} = \lceil\frac{ub_l-lb_l+1}{s_l}\rceil$
15: print $\texttt{N\_fmp} = \text{N\_period} \times (\text{n\_pass} - 1) + \text{N\_last}$;
16: print loop(ti, 0, N_fmp - 1, 1)
```

dimension for the corresponding pass, and s_k is the tile size for the k-th dimension. In general, the time stamp, w, is

$$w = ti + \sum_{k=1}^{d-1} \frac{(t_k - v_k)}{s_k} \tag{10}$$

Our code generation algorithm for the wavefront parallelization is a direct extension to DTiler, the outer $d-1$ tiled loops are kept and the last tile index is computed using the schedule in Equation 10. If h_k is the pass size along dimension k, the number of tiles along the innermost flattened direction is N_{fmp}, and the maximum number of tiles along dimension

Algorithm 1 (Continued) Code generation algorithm for the sequential FMP strategy.

```
17: print $\texttt{pass\_num} = \frac{\texttt{ti}}{\text{N\_period}}$;
18: print $\texttt{cur\_t} = \texttt{ti}\%\texttt{N\_period}$;
19: print $\texttt{res} = \texttt{pass\_num}$
20: print $\texttt{div} = \texttt{n\_pass}$
21: for $i$ from 1 to $d-1$ do
22:     print $\texttt{div} = \frac{\texttt{div}}{\texttt{npass}_i}$;
23:     print $\texttt{pass}_i = \frac{\texttt{res}}{\texttt{div}}$
24:     print $\texttt{res} = \texttt{res}\%\texttt{div}$
25:     Let loop $l$ be the $i$th loop in $L_{\texttt{pass}}$
26:     print $t_l = lb_l + \texttt{pass}_i \times s_l$;
27: end for
28: Loop $l$ is the first loop in $L'_{\texttt{tiled}}$
29: print $t_l = ub_l + \texttt{cur\_t} \times s_l$ ;
30: Loop $lp_i$ is the $i$th loop in $L_{\text{pass}}$, and loop $lt$ is the first loop of $L\_\text{tiled}'$
31: print a guard with condition that , for all $i$ in range 1 to $(d-1)$, $lb_{lp_i} \leq t_{lp_i} \leq ub_{lp_i}$ && $lb_{lt} \leq t_{lt} \leq ub_{lt}$
32: Attach the inner $(d-1)$ loops of the $L'_{tiled}$ within the guard, and attach $L_{\text{point}}$ inside the tiled loop
33: print pass_num = pass_num − 1;
34: print cur_t = N_period + cur_t;
35: REPEAT Line 19-32
```

k, for $k = 1 \ldots (d-1)$ is $\left\lceil \frac{h_k}{s_k} \right\rceil$. Therefore, the total number of wavefronts is bounded by $N_{\text{fmp}} + \sum_{k=1}^{d-1} \left\lceil \frac{h_k}{s_k} \right\rceil$.

However, if we simply keep the outer $d-1$ loops from the sequential FMP code, performance overhead will be introduced due to the large amount of empty wavefronts visited. Remember that the outermost loop in the sequential FMP code visits all the tile coordinates along the linearized dimension, whereas not all the tile coordinates along this dimension need to be visited. Note that at a given time step w, $ti = w - \sum_{k=1}^{d-1} n_k$. Since $0 \leq n_k < \left\lceil \frac{h_k}{s_k} \right\rceil$, then $w - \sum_{k=1}^{d-1} \left\lceil \frac{h_k}{s_k} \right\rceil < ti \leq w$. Therefore, for the first tile loop within the wavefront time loop, we combine the original lower bound of the loop with $\left(w - \sum_{k=1}^{d-1} \left\lceil \frac{h_k}{s_k} \right\rceil \right)$ and the original upper bound with w in the same way as the point loops. The code generation algorithm for FMPP is described in Algorithm 2.

Algorithm 2 Code generation algorithm for the wavefront parallelization for the flattened multi-pass strategy.

Input: The nested loops L for sequential flattened multi-pass strategy. The first d loops define the scanning order of the tile origins, and the inner d loops are point loops that enumerate the points in a tile. For a loop L_k in the first d loops, the iterator name is t_k, the lower bound is lb_k and the upper bound is ub_k.

Output: Nested loops for the wavefront parallelization.

```
1: print nWave = nCols + $\frac{h_1}{s_1} + \frac{h_2}{s_2} + ... + \frac{h_{d-1}}{s_{d-1}}$;
2: print loop(time, 0, nWave, 1);
3: print all the loops from $L_1$ up till $L_{d-1}$, and keep all the statements and guards;
4: for each $(d-1)$th loop do
5:     print $t_{d-1} = (\text{time} - (col + \frac{t_1 - lb_1}{s_1} + \frac{t_2 - lb_2}{s_2} + ... + \frac{t_{d-2} - lb_{d-2}}{s_{d-2}})) \times s_{d-1} + lb_{d-1}$;
6:     print a guard with condition $((lb_{d-1} \leq t_{d-1}) \wedge (t_{d-1} \leq ub_{d-1}))$
7:     print the body of $L_d$
8: end for
```

5.5. Experimental Evaluation

We evaluated our parallelization strategy using a set of stencil benchmarks. We first describe our experimental setup, then show the comparison between the standard wavefront parallelization and our FMPP strategy, and also evaluate the energy consumption of both strategies based on a linear regression energy model.

5.5.1. Experimental Setup. Our experiments are performed on three platforms: one with an 8-core Intel Xeon E5-2650 v2 with DRAM from Samsung, one with a 6-core Intel Xeon E5-2620 v2 with DRAM from Hynix Semiconductor, and the other is a 6-core intel Xeon E5-2620 v3 with DRAM from HP. The hardware characteristics of the three platforms are given in Table 5.1.

All platforms are running Linux operating systems. All the programs are compiled using `icc 16.0.2` with the optimization flags `-O3, -funroll-loops, -xHost` and `-ipo`. As an

TABLE 5.1. Hardware specifications for Intel Xeon E5 2650 v2, Xeon E5-2620 v2 and Xeon E5-2620 v3.

Processor	E5 2650 v2	E5-2620 v2	E5-2620 v3
Architecture	Ivy Bridge	Ivy Bridge	Haswell
Clock speed	2.6 GHz	2.1 GHz	2.4 GHz
Core number	8	6	6
LLC Capacity	20 MB	15 MB	15 MB
associativity of LLC	20-way	20-way	20-way
Machine Peak	166.4 GF/s	100.8 GF/s	115.2 GF/s

estimation of the off-chip memory accesses, the number of LLC misses is measured through PAPI 3.5 [99]. Table 5.2 gives descriptions about the main characteristics of each benchmark. Our benchmark suite includes stencil computations with different number of data dimensions, stencil orders, neighborhoods, number of floating point operations per iteration and also different numbers of variables involved. In Table 5.3, we give the problem size for each benchmark on each platform. The problem size is chosen to be large enough that the data footprint of one wavefront exceeds the LLC capacity.

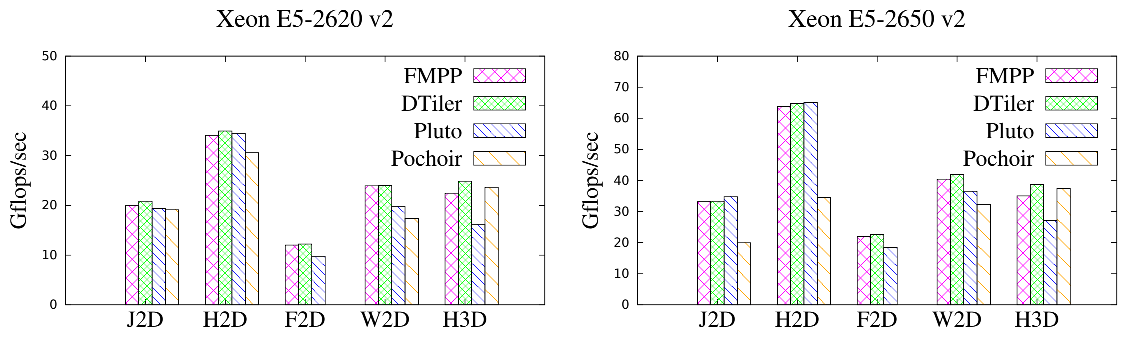
TABLE 5.2. Benchmark Details. Data D is the number of dimensions of a data grid, another time dimension is needed. NP stands for neighboring points, it means the number of neighboring points needed for the computation of each point. NV represents the number of variables that have to be computed during the computation. FPI is floating point operations per iteration.

Benchmark	Data D	Order	NP	NV	FPI
Jacobi 2D (J2D)	2	first	5	1	5
Heat 2D (H2D)	2	first	5	1	9
FDTD 2D (F2D)	2	first	5	3	11
Wave 2D (W2D)	2	third	13	1	13
Heat 3D (H3D)	3	first	7	1	15

The best tile size is selected by running an exhaustive search on all the tile sizes that fit into twice of the L2 cache. For our FMPP strategy, the exhaustive search is performed on all the legal tile sizes that fit into the L2 cache, and the pass size is also restricted that the values needed by each wavefront in the pass fit into LLC.

TABLE 5.3. Problem size used for each benchmark on each platform.

	E5-2650 v2		E5-2620 v2		E5-2620 v3	
	T	Data size	T	Data size	T	Data size
Jacobi 2D	8000	10000×4000	8000	8000×3000	8000	8000×4000
Heat 2D	8000	10000×3000	10000	8000×3000	9000	8000×4000
Wave 2D	6000	7000×4000	10000	8000×3000	9000	8000×4000
FDTD 2D	5000	6000×3000	5000	6000×3000	5000	6000×3000
Heat 3D	500	400×500×600	500	400×400×600	500	400×400×600



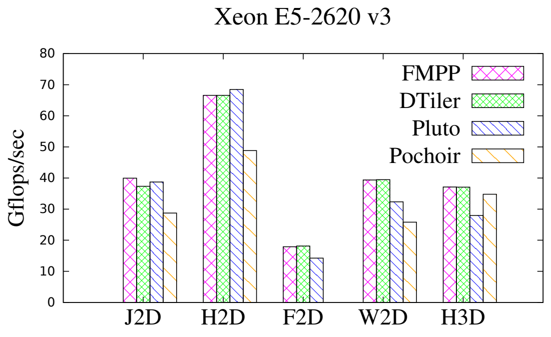


FIGURE 5.10. Performance comparison for standard wavefront parallelization from our FMPP code generator with DTiler, Pluto and Pochoir. There is no performance for Pochoir for FDTD because Pochoir requires the neighboring points are all from the previous time step.

5.5.2. PERFORMANCE EFFICIENCY OF GENERATED CODE. In this subsection, we demonstrate the efficiency of the codes generated by our FMPP code generator. We compare the performance of the standard wavefront parallelization codes generated by our code generator with the codes generated from DTiler [50], Pluto [12]. DTiler is a parametric tiled code generator, and Pluto is a fix sized code generator. We also compare the performance with the code generated by Pochoir [97], a fixed size code generator that implements a very different tiling and parallelization strategy called *cache obvious tiling* [78, 31, 32].

Figure 6.14 shows the performance achieved by the code generated from all the code generators for the benchmarks. We first see that all four code generators produce highly optimized codes with similar performance, provided we tune each one to choose the best tile size (except Pochoir, which is auto-tuned). We also see that the performance of the best parametric tiled code (generated by DTiler) is comparable to the best fixed size tiled code (generated by Pluto), often it is better, especially on higher-order and higher-dimensional cases. Similarly DTiler generated code performs better than that produced by Pochoir on all 2D cases. The performance of code generated by our FMPP strategy is nearly the same as that of DTiler, with a small (within 5%) loss of of performance.

On Jacobi 2D, the performance achieved by our FMPP code generator is actually better than DTiler on Xeon E5-2620 v3. Compared to other benchmarks, Jacobi 2D has the fewest floating point operations per iteration, and the Haswell architecture (i.e., Xeon E5-2620 v3) has a significantly improved on-chip memory bandwidth. Therefore, one possible reason for the improved performance is that Jacobi 2D on Haswell is still bound by the off-chip memory access, and some extra performance benefit is gained by reducing off-chip memory accesses.

5.5.3. Energy Efficiency Of Our FMPP. We evaluate the energy consumption of our benchmark on the three platforms based on a simple linear regression energy model.

5.5.3.1. *Energy Model Overview.* The energy consumed by an application consists of CPU energy and off-chip memory system energy, each of which can be separated into static and dynamic components.

$$E = E_{\text{cpu}}^{\text{static}} + E_{\text{cpu}}^{\text{dynamic}} + E_{\text{mem}}^{\text{static}} + E_{\text{mem}}^{\text{dynamic}} \tag{11}$$

where E represents the total energy consumption of the given application, $E_{\text{cpu}}^{\text{static}}$ and $E_{\text{cpu}}^{\text{dynamic}}$ are the static and dynamic energy consumed by CPU, $E_{\text{mem}}^{\text{static}}$ and $E_{\text{mem}}^{\text{dynamic}}$ are the static and dynamic energy consumed by the off-chip memory system.

We model the CPU energy and memory system energy separately. For the memory system energy, the static energy consumed can be computed as $P_{\text{mem}}^{\text{static}} T_{\text{exec}}$, where $P_{\text{mem}}^{\text{static}}$ is the static power of the memory system, and T_{exec} is the total execution time. The dynamic energy consumed by the memory system is proportional to the total number of memory access N_{acc}. Let e_{acc} be the energy consumed per memory access, then $E_{\text{mem}}^{\text{dynamic}} = e_{acc} N_{acc}$. Then the energy consumed by the off-chip memory system E_{mem} can be estimated as follows:

$$E_{\text{mem}} = P_{\text{mem}}^{\text{static}} T_{\text{exec}} + e_{\text{acc}} N_{\text{acc}}$$

The value of $P_{\text{mem}}^{\text{static}}$ and e_{acc} are derived using the DRAM power calculator [65] provided by Micron Technology, Inc. The three platforms used in our experiment have very different internal micro-architectures for DRAM, as indicated from the parameter values derived.

For the CPU energy consumption, we derive a simple energy model using linear regression. The static CPU energy consumption is the static CPU power multiplied by the execution time. Cong et al. [19] observe that dynamic CPU energy consumption is proportional to the number of completed instructions and cache accesses at each level. Since most of the instructions contained in uniform dependence programs are floating point operations (mostly translated to double precision vector instructions) and branch instructions, we further divide the completed instructions into the floating point instructions and branch instructions. In order to derive the linear regression model, we collected hundreds of training data points using a subset of the benchmarks with different problem sizes and tile sizes. Then we collected another hundreds of data points from the other benchmarks to validate the derived model.

The linear regression model is shown in Equation 12.

$$\begin{aligned} E_{\text{cpu}} &= \beta_0 + \beta_1 \times \text{DPVec} + \beta_2 \times \text{Branch} + \beta_3 \times \text{L1Acc} \\ &+ \beta_4 \times \text{L2Acc} + \beta_5 \times \text{L3Acc} + \beta_6 \times T_{\text{exec}} \end{aligned} \tag{12}$$

where the variables are the dynamic instruction/event counts of double precision vector operations, branch instructions, Access to caches at levels 1, 3 and 3, respectively, and these are measured using PAPI [99]. We computed the energy information by accessing the Running Average Power Limit (RAPL) sensors (the execution time of the program has to be controlled, so that the RAPL counter does not overflow). Then we build our linear regression model by using the statistical package R [80]. Figure 5.11 shows the validation result of the energy model. The Root Mean Square Error (RMSE) for all the validated points is only around 0.3%, this is probably due to the narrow class of benchmarks that we are using. Building an energy model that handles general programs is beyond our scope.

5.5.3.2. *Results.* Figures 5.12 show the energy consumption for all the benchmarks on the two platforms, normalized to the total energy consumed by the program with standard wavefront parallelization. On the average, dynamic memory energy consumption is about 14% of the total energy on the platform with Xeon E5-2650 v2, about 8.3% of the total energy consumption on the Xeon E5-2620 v2 based machine, and about 6.9% on the Xeon E5-2620 v3 based platform. Our FMPP is able to save a significant part of the dynamic

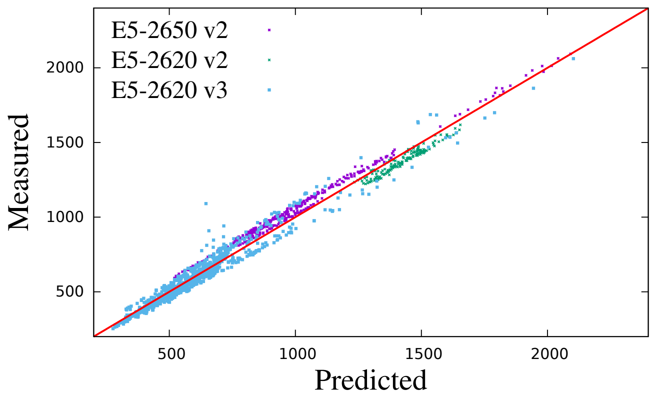


FIGURE 5.11. Validation result of our linear regression model for CPU energy.

memory energy consumption (red part): about 74% of dynamic memory energy consumption on the Xeon E5-2650 v2, 75% of the dynamic memory energy consumption on Xeon E5-2620 v2, and 67% on Xeon E5-2620 v3 based machine.

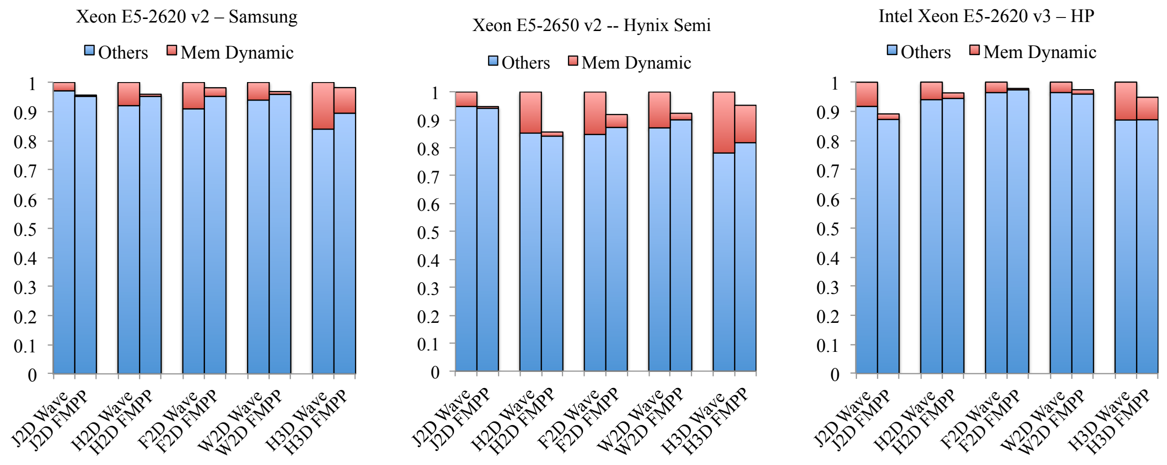


FIGURE 5.12. Normalized energy consumption for the standard wavefront parallelization (Wave) and our flattened multi-pass parallelization strategy (FMPP) on Xeon E5-2620 v2, Xeon E5-2650 v2 and Xeon E5-2620 v3.

In terms of savings of the total energy consumption, we got an average of 8% of total energy consumption saving on the Xeon E5-2650 v2 based machine, 3.1% on the Xeon E5-2620 v2 based machine, and 4.6% on the Xeon E5-2620 v3 based machine. We even got up to 14% of energy savings on the platform with Xeon E5-2650 v2 for Heat 2D. Also, due to the improved execution time on Xeon E5-2620 v3 for J2D, about 11% energy saving is achieved for J2D.

5.6. Conclusion

In this Chapter, we presented an energy-efficient parallelization strategy for dense stencil like programs, on which polyhedral analysis can be used and tiling can be applied. Energy optimization for tuned compute bound codes have very little "energy slack," and we targeted the main contributor—the dynamic memory energy consumption—by minimizing the number of off-chip memory transfers. On all three experimental platforms we were able to reduce this significantly—by over 65%.

We developed a parametric tiled code generation algorithm for our parallelization strategy, for which the key challenge was to incorporate schedules that are polynomial, rather than the standard multidimensional ones used in most code generators. This is the first code generator that implements polynomial schedules for parametric tile sizes.

Our experimental results showed that the generated code has performance comparable with, often better than, existing code generators. We achieve, on the average, overall energy savings of 7.9% on the Xeon E5-2650 v2, 3.4% the Xeon E5-2620 v2, and 4.6% on the Xeon E5-2620 v3. In the best case, we saved 14% of the energy on the Xeon E5-2650 v2.

Our work confirms that optimizing energy leads to good performance, but the converse is not necessarily true. Although our work is based on the rectangular tiling and wavefront parallelization, our method is in general independent of the tile shape or parallelization strategy that are chosen. Our main idea is dividing the iteration space into small passes, so that the data reuse within the whole pass is possible. Within a pass, any existing tiling technique and parallelization strategy for the tiled iteration space can be adapted. The key challenges here are how to schedule the computation so that no performance sacrifice will occur, and automatically generate the codes. Our strategy can also be used in many other programming models and architectures—distributed memory codes in MPI, hybrid MPI/OpenMP, accelerators like GPUs, and even FPGAs. It is also interesting and challenging to use different tile shapes within our multi-pass strategy.

CHAPTER 6

Performance Optimization with Efficient Vectorization

In Chapter 4 and Chapter 5, we presented our energy-efficient strategy that addresses the dynamic energy consumption and its code generation algorithm. In this chapter, we address another important energy contributor – static energy consumption.

As described in Chapter 1, static energy is proportional to execution time, the only way to improve the static energy is to improve the execution time. However, for the compute intensive stencil computations tackled in our work, they become *compute bound* after tiling. In other words, the execution time is mainly bounded by the number of computation. Although we are not targeting for reducing the number of computations in this work, it does not mean that the computation time can not be improved. For modern multi-core architectures, one key to obtain high performance is to make efficient use of the SIMD (vector) units.

Existing tiled code generators generate vectorization friendly code (i.e., make innermost loop vectorizable), and then rely on the automatic vectorization of existing sophisticated commercial compilers (i.e., gcc, icc). However, the vectorization strategy extract from the existing compiler may not be the most efficient strategy for stencil computations. Especially for parametric tiled codes, standard automatic vectorizers fail to efficiently vectorize the code because of the complex structure of the code. In particular, the loop bounds involve complicated expressions involving max/min operations, and parameters that are not known at compile time.

In this chapter, we present a compilation method for generating efficient parametrically tiled SIMD code for stencils, while retaining a simple code structure. We first introduce the basic SIMD operations provided on modern multi-core architectures. Then we give an overview of the vectorization strategy that is supported in our work, followed by the details of code generation.

6.1. SIMD Operations

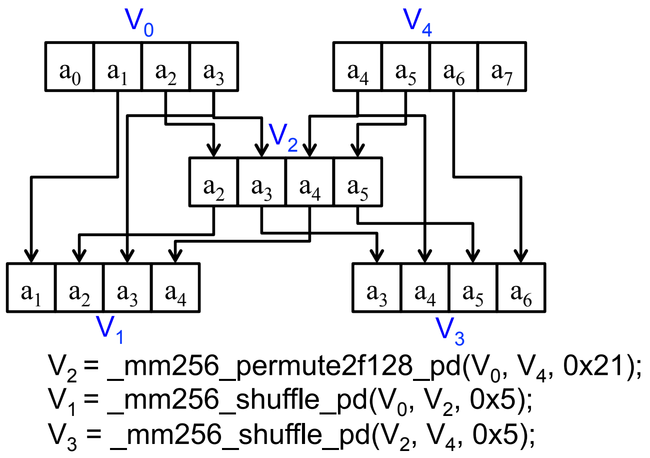
These SIMD instructions supported on modern processors provide a rich set of operations for data organization in vector registers. The main operations that are used in our work are the following:

vload(addr(i)). This operation loads a vector from a stride-1 memory reference *addr(i)*.

vstore(addr(i), src). This operation stores the data in vector register *src* to a stride-1 memory reference *addr(i)*.

vshift(V_i ***,*** V_{i+v} ***, l)***. This operation constructs vector register v_{i+l} with two given registers V_i and V_{i+v}, where V_i is a register that contains data that is aligned with the ith element in the data stream with stride-1 memory reference, $1 \leq l < v$, and v is the number of elements in a vector register. Fig 6.1 shows how the *vshift* operation is achieved with the Intel AVX instructions for double data type. The length of the vector provided by AVX is 256 bits, which is four double elements. Therefore AVX for double data is also called 4-way AVX. Similarly, AVX for single is called 8-way AVX. A more detailed description about the $vshift$ operation with different Intel instruction set can be found in [53].

vshift1(V_i ***,*** V_{i+2} ***)***. This operation takes two vectors V_i and V_{i+2}, and constructs the vector V_{i+1}.



```
V2 = _mm256_permute2f128_pd(V0, V4, 0x21);
V1 = _mm256_shuffle_pd(V0, V2, 0x5);
V3 = _mm256_shuffle_pd(V2, V4, 0x5);
```

FIGURE 6.1. Achieving the shift operation of with shift distance 1, 2 and 3 with the Intel AVX instructions for double data precision.

The data in memory are organized in cache lines (a cache line size is usually a multiple of vector length), and when a memory load/store is issued, the cache lines that contain the data will be fetched/written. Therefore, when a vector happens to cross a cache line boundary, loads/stores will cause a memory access to *two successive* cache lines. However, if the vector is only contained in one cache line, then only one cache line will be touched. We call the load/store of a vector that is aligned with the cache line boundary an ***aligned load/store***. Otherwise, it is called a ***load/store***. Aligning loads and stores with the cache line boundaries is a critical optimization on platforms with strict alignment constraints. In general, it is very difficult to align *all* loads and stores for parametric tiled stencil computations, since every tile only operates on a small subset of the global data space, especially for higher dimensional data and with skewing.

A loop (dimension) is ***vectorizable*** if the following conditions are satisfied:

- there is no loop carried dependence along the loop,
- all array references visited in the loop order have stride 1.

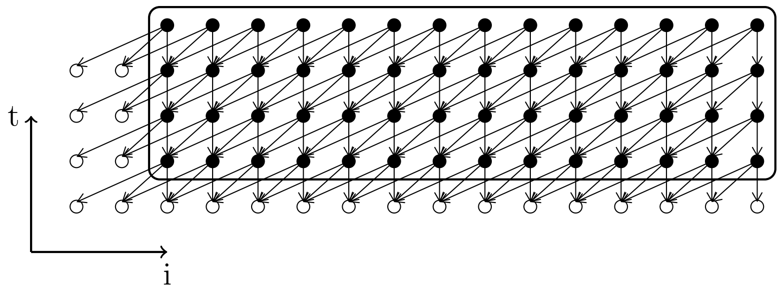


FIGURE 6.2. A 4×14 full tile for the J1D example. The filled dots represents the computations in the tile, and the non-filled dots represents the data coming from other tiles that are required for the computations.

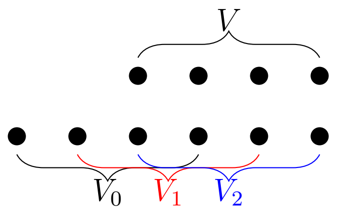
Stencil computation is a perfect candidate for vectorization. For the Jacobi style stencil computations, the loop that visits the innermost data dimension are always vectorizable. For the Gauss-Seidel style stencil computation, existing work [102, 53] can be used to find an affine program transformation to make the innermost loop vectorizable. Here, we focus on how to vectorize the loop after it is made vectorizable with proper pre-processing.

6.2. Our SIMD Compilation Method Overview

We now give an overview of our vectorization strategy. It builds on existing parametric tiled code generation techniques [85, 5, 50]. We target the full tiles—which can be separated out using known techniques—and seek a vectorization strategy that reduces the number of unnecessary memory operations, data reorganization instructions and improves the reuse of vector registers, while still keeping the program structure as simple as possible.

In Fig 6.2, we give an example of a 4×14 full tile for the J1D example, where the innermost loop iterating over the data dimension i is a vectorizable loop. Although a fixed size tile is shown for illustration, the sizes are parametric in the actually generated code. In the rest of paper, we also assume, for clarity of explanation, that the length of a vector is 4, which means one vector register contains 4 elements.

For the J1D example, computing one vector V requires three vectors from the previous time step, as shown in Fig 6.3a. The corresponding assembly code generated by Intel `icc 16.0.2` is shown in Fig 6.3b, where three memory operations are involved in the computation of one vector. However, these three vectors V_0, V_1, V_2 have overlapped data values. For example, the last three elements in V_0 are actually the first three elements of V_1. Fig 6.4 shows another strategy that avoids the loads with repeated data. Instead of loading all the registers required for the computation, we can load two vectors with contiguous non-overlapped data, which are V_0 and V_4 separately. Next, we construct register V_1 and V_2 using the register data reorganization operations like *vshift*. Furthermore, V_4 can be directly fed into the computation for the next vector V', and only one more vector is required to be loaded for the computation of V'. Therefore, this strategy only requires one vector register load and two *vshift* operations for each vector computation, in the steady state. The standard auto-vectorization strategy requires three memory operations. We can also peel out the initial load required for the very first vector out to make the operations uniform for all the vector computations within the vectorized loop.

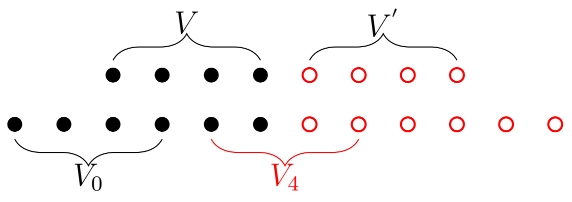


(A)

```
vmovupd    -8(%r13,%r14,8), %xmm2
vaddpd     (%r13,%r14,8), %xmm2, %xmm3
vaddpd     8(%r13,%r14,8), %xmm3, %xmm4
vmulpd     %xmm1, %xmm4, %xmm5
```

(B)

FIGURE 6.3. (a) Vectors required for one vector computation. (b) Assembly codes generated for one vector computation for the parametric tiled codes. *vmov* instruction moves data to a vector register, and the two *vadd* operation has one operands involves a memory address.



$$V_1 = vshift(V_0, V_4, 1) \qquad V_2 = vshift(V_0, V_4, 2)$$

FIGURE 6.4. Vectorization without repeated data loads.

```
register V0, V4;
V0 = vload(..);
for(i = lb; i < ub; i += 4) {
   //remaining loads
    V4 = vload(..);
    //vectorized computation
    V = compute(V0, V4);
    vstore(V, ..);
    //register copy
    V0 = V4;
}
```

(A)

```
register V0, V4;
V0 = vload(..);
for(i = lb; i < ub; i += 8) {
    /*first iteration*/
     //remaining loads
    V4 = vload(..);
    //vectorized computation
    V = compute(V0, V4);
    vstore(V, ..);

    /*second iteration*/
    //remaining loads
    V0 = vload(..);
    //vectorized computation
    V' = compute(V4, V0);
    vstore(V', ..)
}
```

(B)

FIGURE 6.5. (a) The vectorized loop for Figure 6.4 when only one vector is computed within the loop body. (b) The vectorized loop with an unroll factor of two. In other words, two vectors are computed within the loop body.

Figure 6.5a gives an example about the vectorized innermost loop for Figure 6.4, where function *compute* takes two registers as inputs and produces the final answer. Note that a register copy has to be performed at the end of loop to save the register that can be reused for the next iteration. However, under some situations, this register copy can also be saved. In Figure 6.5b, we show the vectorized innermost loop when it is unrolled twice. As we can see, the register $V4$ is directly fed as input to the computation for the second iteration. Also, since the values in register $V0$ are not useful anymore after moving to the second iteration, we can reuse $V0$ to store new values, and the new values are reused for the next iteration. Note that no register copy is required in the code shown in Figure 6.5b. When no register copy is required for saving the loaded values from an array, we say the loads for the array can be implemented in a ***perfectly rotated manner***. The main idea of our vectorization strategy is to load non-overlapped vector and construct the intermediate registers when it is necessary. Clearly the cost for this is the additional instructions to build the needed vectors, and indirectly, the register pressure that this may introduce. Before we describe the details of our vectorization algorithm, we formalize some of the patterns of stencil accesses.

We characterize the ***data stream*** that is touched by a dependence using the dependence vector of the dependence, but with the innermost vectorized dimension removed. For example, the dependence vectors for the J1D example are $(1, 2)$, $(1, 1)$ and $(1, 0)$, and after projecting out the innermost dimension, the value along the outer dimension is 1, for all three dependences. Therefore, only one data stream is read by the computation. For the J2D example described in Equation 2, the dependence vectors after time skewing for one computation are $(1, 2, 1)$, $(1, 0, 1)$, $(1, 1, 1)$, $(1, 1, 2)$ and $(1, 1, 0)$. After projecting out the vectorized dimension we get $(1, 2)$, $(1, 0)$, $(1, 1)$, $(1, 1)$ and $(1, 1)$, and there are thus three distinct data streams touched for one computation.

The *relative distance* of the data touched in the data stream by a dependence is defined as the negation of the value of the innermost (vectorized) dimension of dependence vector. For example the relative distance touched by the three dependence in J1D is -2, -1 and 0 respectively. With the knowledge of the relative distances in a data stream for a computation, we know the *relative data range* that is covered for the computation in the data stream, which is defined as the range from the minimum of the relative distances to the maximum of the relative distances. For example, the relative data range covered for the J1D example is $[-2, 0]$. For the J2D example, the relative data range covered in the data stream $(1, 1)$ is $[-2, 0]$, but $[1, 1]$ for both data stream $(1, 2)$ and $(1, 0)$. We simply add the vector length to this get the relative data ranges for an entire *vector computation*.

Property 6.2.1. *For a variable with only uniform dependencies, if K vectors of continuous data in a data stream are needed for one vector computation, then the last $K - 1$ vectors in the stream can be reused for the next computation.*

Proof. Assume that the relative data range for the data stream is $[a, b]$, and the vector length is v. Then the relative data range covered by one vector computation is $[a, b+v]$. For a vector computation that is aligned with the ith element, the required data in the stream is from $i + a$ to $i + b + v$, and the number of vector loads K is $\lceil \frac{v+b-a}{v} \rceil$, and each vector starts with element $i + a + v \times k$ in the data stream, where $0 \leq k \leq K$. For the next vector computation, it will be aligned with the $(i + v)$th element. Due to uniformity of the dependencies, the same data range pattern will carry over, and the data has to be loaded from the same data stream, starting from $i + v + a$, which is aligned with the second vector loaded in the previous vector computation. □

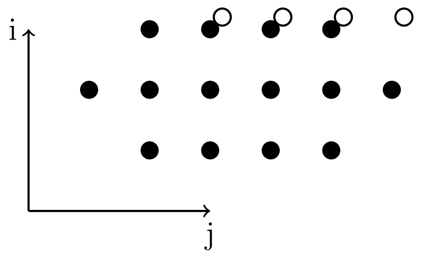
Property 6.2.2. *The vector loads for a data stream can be implemented in a perfect register rotation manner if the vectorizable loop is unrolled at least K vector times. Otherwise, register copies are needed.*

Proof. As described in Theorem 6.2.1, there is only one vector difference between two continuous vector computations. This means that vector registers can only shifted by one vector once. Therefore, at least K vector iterations will be needed to rotate back to the first register. $\square$

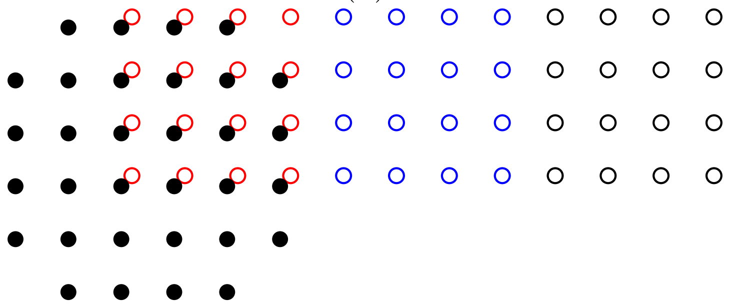
Properties 6.2.1 and 6.2.2 can both be applied for stencil computations, since the dependence involved in stencil computations are all uniform dependences. According to Property 6.2.1, only one more extra load is required from a data stream when we move from one vector computation to the next vector computation. Then we can peel off the first $K-1$ loads in the very first vector computation, and keep the code uniform for all the vector computations within the vectorized loop body.

In order to exploit vector register reuse along the outer dimensions, we support register blocking for the data dimensions and the register block size along the vectorized dimension is required to be a multiple of vector length. We further block the register block along the vectorized dimension with the vector length, and each sub-register block is called a *Register Vector Block* (RVB), which is our analysis unit. Let us take J2D as an example, the innermost j dimension is the vectorizable dimension for J2D. Fig 6.6 shows the data required (filled dots) for the computation of one vector (non-filled dots). Figure 6.6b describes a register block with size 4×12 and the RVBs within the register block (successively as red, blue and black circles).

We consider a tiled d-dimensional stencil computation with $ts_0 \times ts_1 \times \ldots \times ts_d$ tiles, and $rs_0 \times rs_1 \times \ldots \times rs_d$ register blocks, where ts_i is the tile size for the ith dimension and rs_i is the register block size for the ith data dimension. The register block size has to be fixed during code generation time, but tile sizes are parametric. The code structure generated for a full tile is shown in Figure 6.7, where the last register copy code block is not needed when perfect rotation (Property 6.2.2) holds.



(A)



(B)

FIGURE 6.6. (a) Data required (filled dots) from time step $t-1$ for the computation of one vector (non-filled dots) at time step t. The time dimension comes out of paper, and is omitted in the figure. (b) A register block with size 4×12, and the red dots, blue dots, and black dots represent one RVB separately. The filled black dots describe the data required for the RVB represented with red dots.

Our vectorization algorithm consists of four steps, and are described as follows:

- Compute relative ranges of the data to be loaded for one RVB.
- Construct peeled loads: the first $K-1$ loads for each data stream are peeled off before the vectorized loop.
- Construct the vectorized loop body for the $\frac{rs_{d-1}}{v}$ RVBs.
- Check whether perfect register rotation is applicable, otherwise, generate register copies to align the registers.

The second and last steps are not complicated, so we describe how we compute the data range for one RVB and how we construct the vectorized loop body.

Note that, in order for the code described in Fig 6.7 to produce correct answers, the tile size along the i dimension ts_i must be a multiple of the corresponding register block size

```
register V0, V1, ...;    //register for loads
for(t = tt; t < tt + st; t++) {
 for($i_0$=$ti_0$; i<$ti_0$ + $ts_0$; $i_0$+= $rs_0$){
  for($i_1$=$ti_1$; i<$ti_1$ + $ts_1$; $i_1$+= $rs_1$){
       .  .  .  .        //other data dimension loops
    for($i_{d-2}$=$ti_{d-2}$; i<$ti_{d-2}$+$ts_{d-2}$; $i_{d-2}$+=$rs_{d-2}$){
      peeled_loads();
      for($i_{d-1}$=$ti_{d-1}$; i<$ti_{d-1}$+$ts_{d-1}$; $i_{d-1}$+=$rs_{d-1}$){
       /* Code block for first RVB */
       remaining_loads($RVB_0$);
       compute_body($RVB_0$);

       /* Code block for second RVB */
       remaining_loads($RVB_1$);
       compute_body($RVB_{01}$);

       .  .  .  .        //other code blocks

       /* Code block for $n = \frac{rs_{d-1}}{v}$ th RVB */
       remaining_loads($RVB_{n-1}$);
       compute_body($RVB_{n-1}$);

       /*NOT Needed when perfect rotation can be applied */
       register_copy();
      }
     }
   }
 }
}
```

FIGURE 6.7. Code structure for the vectorized full tiles with tile size ts_i along the ith dimension, and register block size rs_i along the ith data dimension.

along this dimension. Otherwise, the values at the boundary will be redundantly updated by different tiles. In section 6.3.3, we will describe how temporary buffering can help to ensure the correctness for any tile sizes.

Moreover, aligning loads and stores with cache line boundaries is an important optimization with a big impact on performance for machines with strong alignment constraints. When the first reference in a tile is aligned with the cache line boundary, existing techniques can be utilized to align all the loads and stores. However, with parametric tile sizes, the first reference within a tile changes with the tile sizes, and the alignment assumption usually does not hold. In section 6.3.3 we will also show how the load and store alignment can be achieved easily with temporary buffering.

6.3. Code Generation for Our Vectorization Strategy

In this section, we describe how we compute the data range that has to be loaded for each UVB, and how the vectorized loop body is constructed during code generation. Also, we illustrate how temporary buffering can be utilized to ensure the correctness of arbitrary tile sizes and guarantee the alignment of loads and stores.

6.3.1. Data Range Computation.

In section 6.2, we described how to compute the relative data range for a data stream. Here, we will describe how to compute the relative data range required for the whole RVB that is relative to the first computation point of the register block.

The way we compute the data range for one RVB is straightforward. For a RVB, we extract all the data streams that are touched in the whole RVB, and then compute the relative data range in each data stream. In order to do this, we enumerate all the dependences in the first RVB with respect to the first point in the register block. In the J2D example, the first point in a register block is (t, i, j), and it depends on $(t-1, i-2, j-1)$, $(t-1, i, j-1)$, $(t-1, i-1, j-1)$, $(t-1, i-1, j-2)$ and $(t-1, i-1, j)$. The next point in the RVB along the i dimension is $(t, i+1, j)$, which depends on points $(t-1, i-1, j-1)$, $(t-1, i+1, j-1)$, $(t-1, i, j-1)$, $(t-1, i, j-2)$ and $(t-1, i, j)$, and the corresponding dependence vectors are $(1, 1, 1)$, $(1, -1, 1)$, $(1, 0, 1)$ and $(1, 0, 2)$. Similarly, we can extract dependence vectors for the rest points in the RVB. Then the data streams touched in one RVB and the data range covered in each data stream can be computed using the previous description. According to Property 6.2.1, when we move from one RVB to the next RVB along the vectorized dimension, we just need to shift the data range in each stream with the vector length.

6.3.2. Loop Body Construction. After the loads are constructed, we need to construct the vectorized loop body. As described in Fig 6.7, the unrolled vectorized loop body consists of n unrolled code blocks, where n is the number of RVBs within a register block. Each code block also includes two parts: the loads for the additional vectors needed, and the vectorized computation itself. The vectorized computation is a direct SIMD instruction translation from the original computation, so the key problem here is how to construct all the registers needed for the computation with the loaded registers.

In order to solve this problem, we first construct a register dependence graph with all the registers needed in the computations and the loaded registers. A register V_a depends on register V_b and V_c if V_a can be constructed with V_b and V_c with a vector register data reorganization operation like *vshift*. When there are multiple candidates for construction of V_a, we pick up the one with the least latency. The loaded registers do not depend on any other registers. After the register dependence graph is constructed, a topological sort is performed to decided the order of the register construction. For the J1D example shown in Fig 6.4, the loaded registers are V_0 and V_4, and the other needed registers are V_1 and V_2. Register V_1 can be constructed either using $vshift(V_0, V_4, 1)$ or $vshift1(V_0, V_2)$. The latency for *vshift1* is usually shorter than the *vshift*, therefore there is an edge from V_1 to V_0 and V_2. Fig 6.8 shows the register dependence graph for the computation of one UVB of J1D, and Fig 6.9, shows the vectorized code for the full tiles in the J1D example.

$$\begin{tikzcd}
V_1 \arrow[r] \arrow[d] & V_2 \arrow[dl] \arrow[d] \\
V_0 & V_4
\end{tikzcd}$$

FIGURE 6.8. Register dependence graph for one UVB of J1D.

```
register V0, V4;
for(t = tt; t < tt + st; t++) {
  //peeled loads
  V0 = vload(&B(t, ti));
  for(i = ti; i < ti + si; i += 8) {
    /* Code block for first UVB */
    V4 = vload(&B(t, i+4));   //remaining loads
    //vectorized computation
    compute_body(t, i, B, V0, V4);

    /* Code block for second UVB */
    //remaining loads, using rotated register
    V0 = vload(&B(t, i+8));
    //vectorized computation
    compute_body(t, i+4, B, V4, V0);
  }
}

void compute_body(int t, int i, double **B, register V0, register V4) {
  //construct needed registers
  register V2 = vshift(V0, V4, 2);
  register V1 = vshift1(V0, V2);
  //vectorized computation
  register out = vmul(vadd(vadd(V0,V1),V2), v(0.33333 ) );
  //write back
  vstore( &B(t,i),out);
}
```

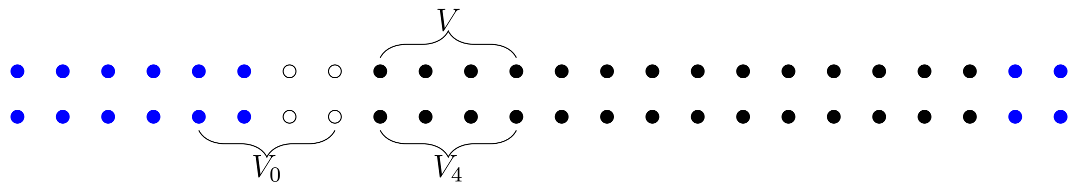
FIGURE 6.9. Code structure for the vectorized full tile for J1D. The data dimension i is unrolled by 2 vectors. Variable B is the data space that J1D operates on. It is a two dimensional array but with only two rows accessed modulo-2.

6.3.3. TEMPORARY BUFFERING. We described our vectorization strategy and the algorithm to generate the vectorized code. However, there are still two problems that remain: 1) ensuring the correctness for arbitrary tile sizes, 2) aligning loads and stores.

The common technique to ensure correctness for arbitrary tile sizes is by peeling out the initial and final few iterations to make sure that the rest of the loop iterates over a multiple of the register block size. The peeled iterations are executed in scalar mode. However, this complicates the code structure, which may prohibit some compiler optimizations. There is also extensive work on aligning loads and stores [25, 39, 53], most of which shifts the loads to align with a cache boundary, and then reorganizes the data with vector operations like *vshift* used in this paper. However, this does not work when the memory is allocated dynamically. Especially, in the parametric tiled code, every tile only operates on a small subset of the whole data space, and the relative address of first reference in a tile is not known until run-time.

We resolve both problems in a simple way by allocating and using a temporary buffer. Furthermore, this also retains the simplicity of the code structure. The main idea is straight forward: allocate perfectly cache-aligned temporary buffers (memory) to hold the data required by the tile, and have the tile code access these buffers rather than arrays of the original code. For every full tile, we allocate n buffers whose size matches the data accessed by the tile, where n is one more than the dependence in the time dimension. This storage mapping is known to be a *schedule-independent* memory allocation for stencil computations [93]. For our J1D example, $n = 2$, because the furthest dependence along the time dimension is 1. Also the data space (along i dimension) touched by the tile shown in Fig 6.2 is $14 + 2 = 16$, where 14 is the tile size along the time dimension, and two additional values are required from a neighboring tile. Therefore, for the J1D example with tile size t_i along the data dimension, two one dimensional array with size $t_i + 2$ are allocated. Similarly, for the J2D example with tile sizes $t_i \times t_j$ along the two data dimensions, two 2-dimensional arrays of size

$(t_i + 2) \times (t_j + 2)$ are allocated. This array could be allocated as either a multi-dimensional array, or as a linearized (1-D) data structure. On the machines we experimented with, the one-dimensional linearized allocation performed better.



$V_1 = vshift(V_0, V_4, 2)$ $V_2 = vshift(V_1, V_4, 1)$

FIGURE 6.10. Memory padding for the temporary buffer for J1D. The blue dots represent padded extra memory, the non-filled dots are the memory used for the halo region.

With temporary buffering, the computations within a tile read and update temporary buffers instead of the original data space. Only the data that is consumed by a tile in the *next* wavefront, or output data are written back to the global memory. Data that is reused between successive tiles is repeatedly updated directly in the temporary buffers. The scheme works even if the tile sizes are not multiples of the register block size. However, we have to be careful to avoid memory references to non-allocated memory, because the buffer is allocated according to the tile sizes. One way to solve this is to use loop peeling, but this complicates the code structure, introduces conditional codes with parametric sizes, and also executes some computations in a scalar mode. Our solution is to pad the temporary buffers to make its size correspond to the next larger tile size that is a multiple of the register block size. For example, if the unroll factor for J2D is 4×8 along the i and j data dimensions (note that the j dimension is the vectorizable dimension, and the innermost unroll factor is 2 vectors long), the buffer size with padding is $(\lceil \frac{t_i}{4} \rceil \times 4 + 2) \times (\lceil \frac{t_j}{8} \rceil \times 8 + 2)$. The loop and its body remains unchanged. This approach results in some dummy computations, but no peeling or checking is required, no computation is done in scalar manner, and the code structure remains simple.

We now show how the temporary buffers allow us to align all the loads and stores. Let us return to the vectorized J1D in Fig 6.4, and assume that all the operations are performed on the temporary buffer. Let the starting address of each buffer be aligned with the cache line boundary (this can be done with special `malloc` calls) and let each cache line be 8 data elements, then the load of vector V_0 and V_4 are aligned with the cache line boundary, as are all the contiguous vectors that are loaded. However, the store of V is not aligned with the cache line boundary, and the next vector V' actually crosses the cache line boundary. As we can see, if the buffer allocation is aligned with the beginning of the loads, all the loads will be aligned, but all stores will be misaligned. In order to align the stores too, we pad extra memory in front each buffer to make the first store align with the beginning of a cache line. Now, all the stores are aligned with the cache line boundary, but the loads may become misaligned depending on the amount extra memory padded in-front. Like most previous work on aligning loads, we shift the first load in each data stream to align with the closet cache line boundary, and then construct the registers with data reorganization operations. Our previous properties for register reuse and rotation still hold with these shifted loads, since it does not break the uniform property of the dependence, which is equivalent to adding one more dummy dependence that is aligned with the boundary of the cache line.

With memory padding and load shifting, we are able to guarantee the alignment of all the loads and stores. However, this also comes with a trade-off of some extra data reorganization overhead. If we look at the last row of data stream that is required for the computation of a RVB for the J2D example in Fig 6.6b. Without shifting for alignment, only one load is enough, but with shifting, data reorganization is needed to construct the register that is needed. If we look at the second row of data stream, data reorganization is required even without shifting due to the reuse of the data. Now, with shifting, the first reference in the data stream will also have to be constructed. However, when every point in the relative data range of the data stream is used in the computation, there must be one reference in the range that will become aligned after shifting. Therefore, the number of data reorganization operations still remains the same, and this is usually true for stencil computation. In order to explore this trade-off, we separate the data streams into *reusable* and *non-reusable* data streams according to Property 6.2.1. If the number of vectors K required in a data stream for one computation is equal to 1, the data stream is called a non-reusable data stream, otherwise it is a reusable data stream. For a reusable data stream, we still do the shifting to align all the loads in the data stream, but the original misaligned loads are still performed for a non-reusable data stream, and we call this *mostly aligned strategy.* In order to support the exploration of the trade-offs here, both strategies (all aligned and mostly aligned) are supported in our framework.

Fig 6.10 describes the memory padding for the buffers and also the vectorization strategy with shifted loads for the J1D example. As shown in Fig 6.10, there are extra memory (blue dots) padded in front for the alignment of loads and stores, and there are also extra memory (blue dots) padded at the end to ensure the memory size is a multiple of the register block size.The (pseudo) code for the J1D example with padded memory and shifted vector loads is shown in Fig 6.11.

```
allocate_padded_buffer(buffer_0);
allocate_padded_buffer(buffer_1);
init_buffer(buffer_0);

register V0, V4;
for(t = tt; t < tt + st; t++) {
  update_halo(buffer_0);
  //peeled loads, shifted to align with the cache line boundary
  V0 = vload(&buffer_0(ti-2));
  for(i = ti; i < ti + si; i += 8) {
    /* Code block for first UVB */
    V4 = vload(&buffer_0(i+2));   //remaining loads
    //vectorized computation
    compute_body(t, i, V0, V4);
    /* Code block for second UVB */
    V0 = vload(&buffer_0(i+6));   //remaining loads, using rotated
       register
    compute_body(t, i+4, V4, V0);   //vectorized computation
  }
  write_halo_back(buffer_1);
  swap(buffer_0, buffer_1);
}
free_buffer(buffer_0);
free_buffer(buffer_1);
```

FIGURE 6.11. code structure for the vectorized full tile for J1D. The i-dimension unroll factor is 8.

6.4. Micro-benchmark for Achievable Stencil Ceiling

There are many situations that effect the performance of the parallelized stencil computations. These include the pipeline fill-flush overhead for wavefront parallelization, load imbalance, communications between wavefronts etc. Unfortunately, these effects are hard to model analytically. Approaches like roofline model [13, 105, 73, 92] are developed to provide upper bounds on the achievable performance for a given kernel. These approaches make very ideal assumption such as unbounded register file size, and the upper bound derived are usually far away from the actual achievable performance. In order to get a better understanding about the achievable performance for stencil computation, we develop micro-benchmarks to measure the achievable performance with simulation of reasonable ideal situations.

Our benchmarks are designed to simulate the steady state of wavefront parallelization, for which the following properties are satisfied:

- The tiles executed are all full tiles;
- Every thread executes the same number of tiles of same size within each wavefront.
- SIMD optimization, loop unrolling and register reuse are all applied.

Benchmarks with the above properties have no tile level pipeline fill-flush overhead and are perfectly load-balanced. Also, the full tile kernel executed is developed based on each stencil kernel, which contains exactly the same computations as the stencil kernel that is simulated. We developed two sets of micro benchmarks for each stencil kernel used — one with no communications (MicroBench NC) between wavefronts and another set with a controlled amount of communications (MicroBench WC) between the successive wavefronts.

The implementation for the micro-benchmarks with no communication is straightforward, where every thread executes a series of independent full tiles of the same size, and synchronizes at the end of each wavefront. The final results of each tile are accumulated into a single scalar value to prevent the compiler from optimizing it away as dead code (especially since aggressive optimization flags are usually required to achieve good performance). For the micro-benchmarks with communications, every thread still executes a series of independent full tiles within each wavefront, but the results produced in one wavefront have to be saved as inputs to the next wavefront. This output-input communication is achieved through a shared global buffer, each thread operating on a thread-specific, private portion of the buffer at a time. In order to observe and quantify thread-data affinity, we introduce a parameter called *thread data distance*, dis, which specifies that thread i reads data written by thread $((i + dis) \bmod P)$, where P is the total number of threads used. Combined with the environment variable $KMP\_AFFINITY$ for icc (a different variable is used when gcc is used — $GOMP\_CPU\_AFFINITY$) —which specifies the thread-to-core affinity—we can indirectly control the data-to-core affinity. When 0 is specified for parameter dis, a thread fetches the data from the previous wavefront that is written by itself, which is the most ideal thread-data affinity.

With the micro-benchmarks with no communication, we get an optimistic ceiling on an ideal execution environment. This is in general not achievable for the whole stencil program. The micro-benchmark with communication provides a more realistic ceiling that is closer to the whole stencil program, and also gives us a quantitative idea of how much each stencil is affected by communication, pipeline fill-flush and load imbalance.

6.5. Experimental Evaluation

In order to demonstrate the efficiency of our generated vectorized codes, we experiment with a set of stencil benchmarks on three modern Intel architectures. In addition to carefully exploring the effect of each aspect of our strategy, we also compare the performance of our generated code with the code produced from three state-to-art code generators, namely DTiler [50, 85], which is our baseline implementation, and Pochoir [12] and Pluto [97].

6.5.1. Experimental Setup. We conduct experiments on a set of three modern Intel CPUs, and the hardware characteristics of the three platforms are given in Table 6.1. All platforms have three levels of cache, with L1 and L2 being private, and the last level cache (LLC) shared among all the cores. Hyper-threading is also supported on all architectures. All platforms are running Linux, and programs are compiled using `icc 16.0.2` with the optimization flags `-O3, -funroll-loops, -xHost` and `-ipo`.

TABLE 6.1. Hardware specifications of Intel Xeon E3-1230, E5-2620 v2 and E5-262 v3.

Processor	E3-1230	E5-2620 v2	E5-2620 v3
Architecture	Sandy Bridge	Ivy Bridge	Haswell
Clock speed	3.2 GHz	2.1 GHz	2.4 GHz
Core number	4	6	6
LLC Capacity	8 MB	15 MB	15 MB
Machine Peak	102.4 Gflops/sec	100.8 Gflops/sec	115.2 Gflops/sec

TABLE 6.2. Benchmark Details. Data D is the number of dimensions of data grid, another time dimension is needed for computation. NP stands for neighboring points, it means the number of neighboring points needed for the computation of a point. FPI is floating point operations per iteration.

Benchmark	Data D	Order	NP	FPI
Jacobi 1D (J1D)	1	first	3	3
Jacobi 2D (J2D)	2	first	5	5
Heat 2D (H2D)	2	first	5	10
Wave 2D (W2D)	2	third	13	13
Blur 2D (B2D)	2	second	25	31
Heat 3D (H3D)	3	first	7	15
Jacobi 3D (J3D)	3	first	27	30

TABLE 6.3. Problem size used for each benchmark on each platform.

	Sandy Bridge		Ivy Bridge		Haswell	
	T	Data size	T	Data size	T	Data size
Jacobi 1D	120000	1000000	160000	1500000	160000	1500000
Jacobi 2D	1000	6000×6000	4000	5000×5000	1000	7000×8000
Heat 2D	1000	5000×6000	1000	6000×6000	1000	8000×8000
Wave 2D	1000	4000×4000	2000	3000×3000	2000	4000×4000
Blur 2D	800	4000×4000	1000	3000×4000	1000	5000×5000
Heat 3D	300	300×400×400	200	400×400×500	300	400×400×500
Jacobi 3D	200	300×300×400	100	400×400×400	300	400×400×400

Our benchmark suite includes stencil kernels with different number of data dimensions, stencil orders, neighboring points, and number of operations per iteration. Details are in Table 6.2. Also, in Table 6.3, we give the problem size used for each benchmark on each platform. The problem size is chosen such that the data grid size is much larger than the LLC capacity, and the time step is large enough to make the application compute intensive.

6.5.2. Vectorization Efficiency. Here, we demonstrate the efficiency of our generated vectorized code. Figure 6.12 gives the design space of our vectorizer. When temporary buffering and vectorization are both not applied to the tiled code, our code generator basically produces the same code as DTiler, and thus the DTiler performance is used as the baseline in our experiments. We can also choose to only apply vectorization to the tiled codes (SIMD NOBF). For this version of code, the loads/stores within a tile are not guaranteed to be aligned, and the tile size has to be a multiple of register block size in order to ensure the correctness. DTiler BF represents the non-vectorized code with temporary buffering. When both temporary buffering and vectorization are applied to the tiled codes, there are two choices for the vectorization strategy. We can either choose the “all aligned” strategy to align all the loads/stores within a tile, or choose the “mostly aligned” strategy to disable the alignment for the data streams with no reuse between successive iterations. The preliminary results of our experiments showed that the “mostly aligned” strategy is better than “all aligned” strategy on all three platforms. Therefore, we only report the “mostly aligned” strategy in our experiments (SIMD BF-Aligned). We generate the four versions of code — DTiler, SIMD NOBF, DTiler BF, SIMD BF-Aligned — and discuss their performance in this section.

In order to obtain the best performance, an exhaustive search is performed for all the tile sizes whose data footprint is less than twice the L2 cache size. Our experience shows that the best tile sizes are almost always in this range. We also explored different register block sizes for our vectorized code. For the J1D example, we explored register block sizes up to

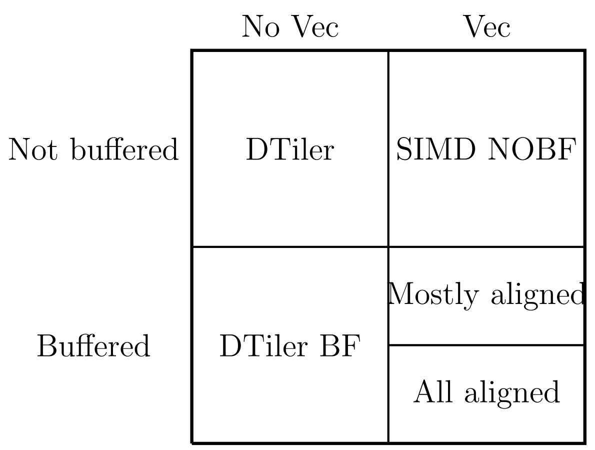


FIGURE 6.12. The design space of our vectorizer.

10 along the data dimension. For 2D (resp. 3D) cases, it we explored sizes up to 6 (resp. 4) along each data dimension. In Figure 6.13, we show the best performance in Gflops/sec of the three versions of code, normalized to the best performance achieved by DTiler.

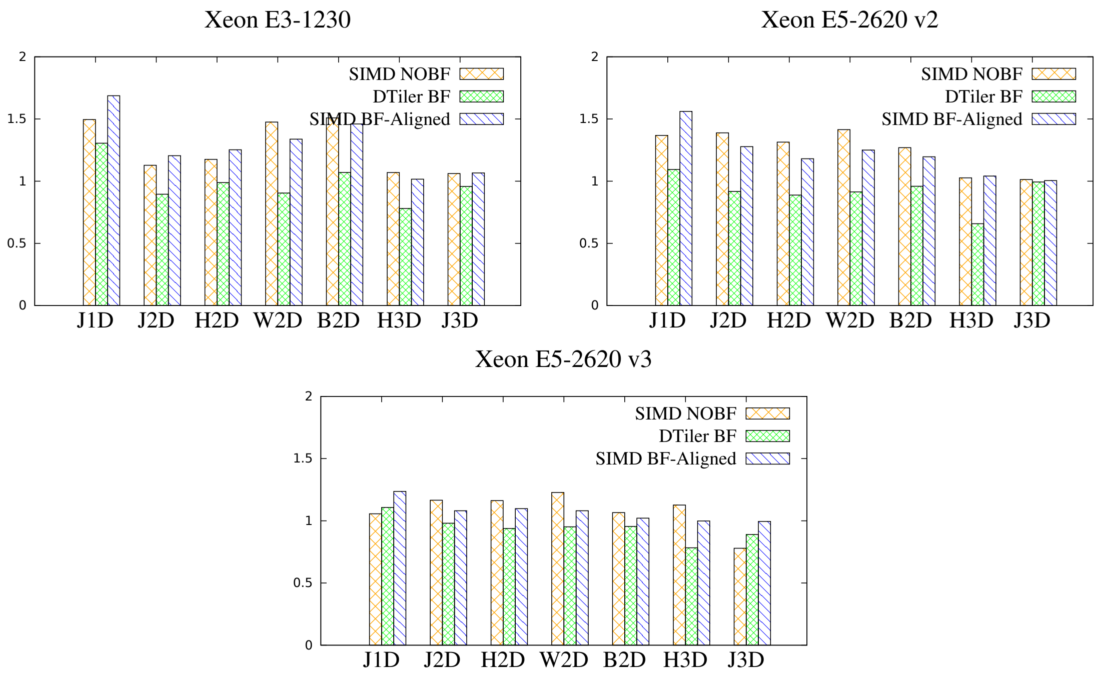


FIGURE 6.13. Normalized best performance of codes: SIMD NOBF, DTiler BF, SIMD BF-Aligned. The performance is normalized to the best performance achieved by DTiler.

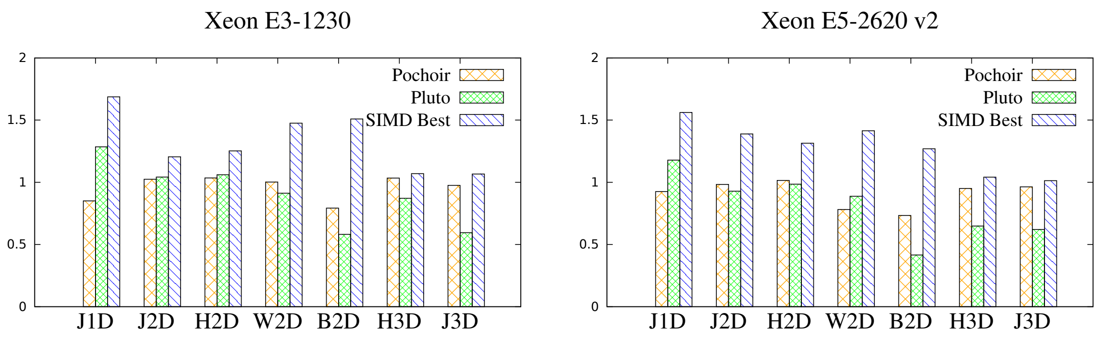
We first look at the performance achieved with only vectorization (SIMD NOBF). On both SandyBridge and IvyBridge platform, an average of 35% improvement is achieved for the 1D and 2D stencils. About 13.5% performance improvement is achieved on the Haswell platform for the 1D and 2D stencils. Haswell is one of the most recent Intel platform, which has a significantly improved on-chip memory bandwidth, and trading off the memory operations for data reorganization operations is not as beneficial as other platforms. For 3D stencils, we are not able to achieve significant performance improvement on all three platforms – about 2% on SandyBridge and 6.5% on IvyBridge, and on Haswell we are even observing small performance degeneration for the Jacobi 3D example. We hypothesize that the reason is the register pressure once we start to increase the register block size, larger number of live values need to be maintained in registers.

Next, we see what happens if we start adding the temporary buffering. As mentioned in section 6.3.3, extra communications are introduced at the boundary for each tile when temporary buffering is applied. However, this overhead is not very significant for compute intensive stencils. As indicated from the results of DTiler BF, the performance lost is in general less than 10% on all the platforms (except for H3D, which is about 30%). For the Jacobi 1D example, we are even gaining performance improvement by accessing a small array that usually fits in the L1 cache. After vectorization is applied, we are able to compensate this loss for all the cases, and even gaining much better performance compare with DTiler. However, for most cases, the overall performance gain for vectorization with temporary buffering is not better than that without temporary buffering. There might be two reasons for this: 1) with proper choice of tile sizes, most of the loads and stores can already be aligned; 2) on modern Intel architectures, the load/store alignment issue is not so severe.

6.5.3. Performance Comparison with Existing Tools. We compare the best performance achieved by our generated SIMD code with two fixed size tiled code generators: Pochoir [97], and Pluto [12]. Pluto supports both standard wavefront parallelization and parallelization with concurrent start. We use the codes with standard wavefront parallelization that are generated from Pluto, because concurrent start is not necessary (indeed, its overall performance is marginally lower than standard wavefront parallelization) for the compute intensive stencils handled in our work. Pochoir implements a completely different strategy called *cache oblivious tiling* [78, 31]. The codes generated from Pluto are "prevectorized," in the sense that vectorization pragmas are attached for auto-vectorization of `icc`. Pochoir generated code uses pointer accesses for arrays, which also improves the chances for auto-vectorization. Therefore, for the codes generated by existing code generators, auto-vectorization from *icc* is enabled by default.

We also did an exhaustive search with Pluto to explore the best performance, but the tile size is fixed for the codes generated by Pochoir. Figure 6.14 shows the performance, normalized to that of DTiler. The performance achieved by our code generator is better than existing code generator. If the same efficiency issue of automatic vectorization also occurs in the codes generated by existing code generator, our vectorization code generation technique may also help to improve the performance of the generated code.

6.5.4. Performance Comparison with Micro-Benchmarks. Now, we compare the best performance achieved by our generated SIMD codes with the best achievable performance or "pseudo ceilings" suggested by the micro-benchmarks. The thread data distance parameter for the micro-benchmarks with communications is set to 0 for the experiments,



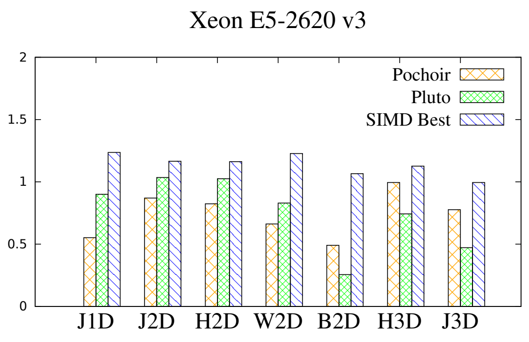


FIGURE 6.14. Performance comparison of our SIMD code generator generated codes with the codes generated by existing compilers. The performance is normalized to the best performance achieved by DTiler.

which simulates the best communication situation. For J1D, the amount of communication is negligible, so we did not run the micro-benchmark with communication,.

Figure 6.15 shows that on Haswell for all benchmarks except Wave 2D, we achieve performance within 10% of the micro-benchmark with communication. Wave 2D is a 3rd-order stencil, and needs a larger skewing factor before tiling. This may cause a larger overhead from pipeline fill-flush and load imbalance. We saw similar results on Ivy Bridge. On Sandy Bridge, we are also getting close to the micro-benchmarks with communication. On all the platforms, the J1D performance nearly matches that of the communication free micro-benchmarks.

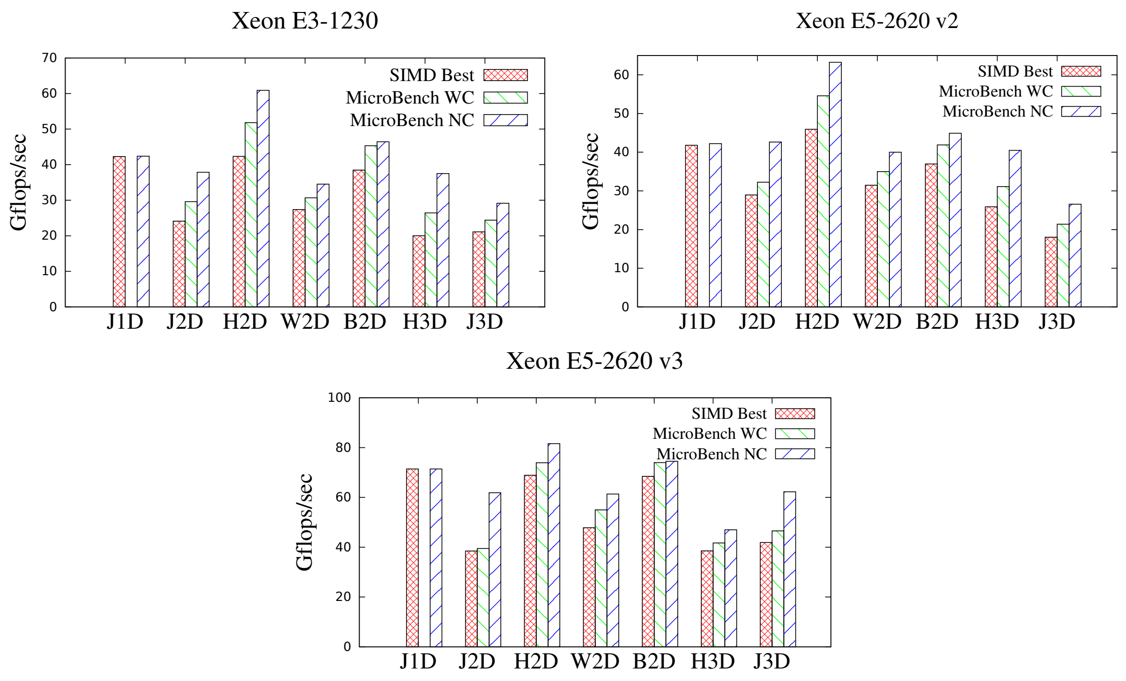


FIGURE 6.15. Performance comparison of our SIMD code generator generated codes with the codes generated by existing compilers. The performance is also normalized to the best performance achieved by DTiler

Now, if we compare the performance of the micro-benchmarks with communication (MicroBench WC) and the communication free micro-benchmarks, the overhead for the high-order stencil computations on all the platforms is within 10%. However, the overhead of the communication for the J2D and Heat 2D are pretty high on the platforms, which is due to the relatively small amount of computations that is not enough for overlapping the communications. We are observing some phenomena that we cannot currently explain, where the overhead of Heat 3D on Sandy Bridge is much higher than on Haswell, but the other way around for J3D.

6.6. Conclusion

In this chapter, we presented an SIMD compilation method for stencil computations after parametric tiling. Our vectorizer applies vectorization to rectangular non-boundary full tiles, and targets reducing the number of unnecessary loads and maximize the vector register reuse. Temporary buffering and padding techniques are integrated to guarantee the alignment of loads and stores within a tile, and also the correctness for arbitrary tile sizes with arbitrary register block size.

We perform experiments on a set of stencil benchmarks on Intel Sandy Bridge, Ivy Bridge and Haswell machines. Our experimental results demonstrate significant performance improvement compare with relying on the vectorizers of product compilers – average 35% for the 1D and 2D stencils on the SandyBridge and IvyBridge platforms, about 13.5% for the 1D and 2D stencils on the Haswell platform. Comparing with the performance achieved by the code produced by existing tools, we are also able to achieve comparable, or even better performance for all the test cases.

We also developed sets of micro-benchmarks for providing more reasonable ceilings than the ones derived from existing analytical approaches. Our experimental results indicate that the best performance achieved by our generate code is approaching close to the measured performance of the micro-benchmarks, around 10% performance margin for many cases.

Although the vectorization strategy generated by our code generator can significantly improve the performance for 1D and 2D stencils, it does not show similar effect for the 3D stencils. For 3D stencils, the performance gain is very low or no improvement for some cases. There are many phenomena occur in 3D stencils whose reason remains unknown, and performance optimization for 3D stencils is still an open research problem.

CHAPTER 7

Semi-automatic Code Generation Framework

In Chapter 4, we introduced our energy efficient strategy – FMPP – that targets reducing the dynamic memory energy by seeking to reduce off-chime memory accesses, without sacrificing execution time. In Chapter 6, we showed that the performance can be significantly improved through better utilization of vector resources which reduces the static energy consumption. We develop a semi-automatic code generator to support these optimizations in a configurable way. In this chapter, we give an overview of our code generation framework, and also evaluate the overall energy saving with both memory and vectorization optimization on one of the most recent Intel architectures.

7.1. Overview of Our Code Generation Framework

We integrated the FMPP and vectorization technique into our polyhedral program transformation and code generation system, AlphaZ [113]. The structure of our framework is shown in Figure 7.2. Due to the fast evolution of existing architectures, automatic choice of optimization strategies is a challenge and remains as an open research problem. Many existing tools [113, 15] are developed in a semi-automatic way, where the optimization strategies are left as configuration parameters, and we also adapt this strategy. Our compilation framework takes a program representation and a configuration file called *Target Mapping* as inputs, and generates the corresponding C+OpenMP+AVX code. Currently, our code generator only supports the generation of Intel AVX instructions (which is supported on most modern architectures), but it would be easy to extend it to support other SIMD instructions like SSE.

```
affine jacobi_1d {TSTEPS,N | TSTEPS > 2 && N > 5}
input
output
        double B {t,i|0<=i<N && t==TSTEPS};
local
        double temp_B {t,i|0<=i<N && 0<=t<TSTEPS};
let
        temp_B[t,i] = case
              {|t == 0} : [i]*[i-1];
              {|t > 0 && 1<=i<N-1} : (temp_B[t-1,i-1] + temp_B[t-1,i] +
                                      temp_B[t-1,i+1])*0.33333;
              {|t > 0 && 0==i } || {|t > 0 && i==N-1 } : temp_B[t-1,i];
              esac;

        B[t,i] = temp_B[t-1,i];
.
```

FIGURE 7.1. Alpha representation for Jacobi 1D example.

The input program to our code generation framework is written in a language called ALPHA [104], which is used to represent the programs that fit in polyhedral model. Stencil computations fit well in polyhedral model, and Figure 7.1 illustrates the input for Jacobi 1D. The target mapping specifies the following components:

- polyhedral program transformations (i.e., skewing), represented as affine functions;
- memory mappings that are used to specify the memory storage locations for the values produced by each statement;
- apply our vectorization strategy or not;
- apply temporary buffering or not;
- if temporary buffering is applied, whether to use mostly aligned or all aligned strategy;
- register block size.

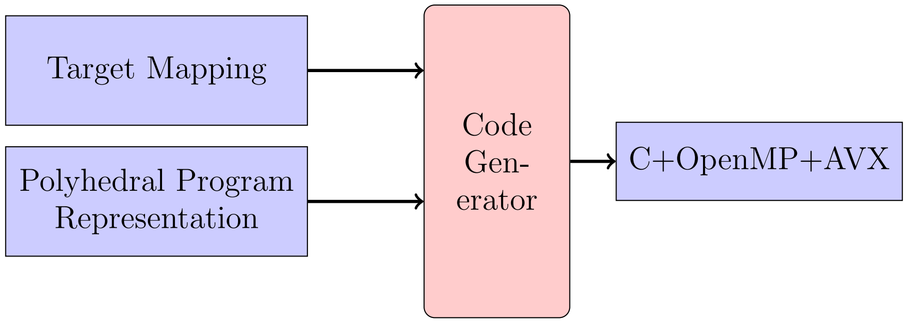


Figure 7.2. semi-automatic code generation framework.

Code generation is the most important component in our framework, and the code generation flow is described in Figure 7.3. The code generator first applies the specified polyhedral transformation as a preprocessing, then generates two sets of loops: original loops and the optimized loops. The original loops corresponds to the input polyhedral program after transformation, and are generated using existing polyhedral code generation technique [6]. The optimized loops represent the point loops for the full tiles with vectorized computation statements, and are generated with techniques described in Chapter 6. Finally, a FMPP code generator takes these two sets of loops, and produces the final desired code with techniques described in chapter 5. The point loops of the separated full tile are substituted with the optimized loops.

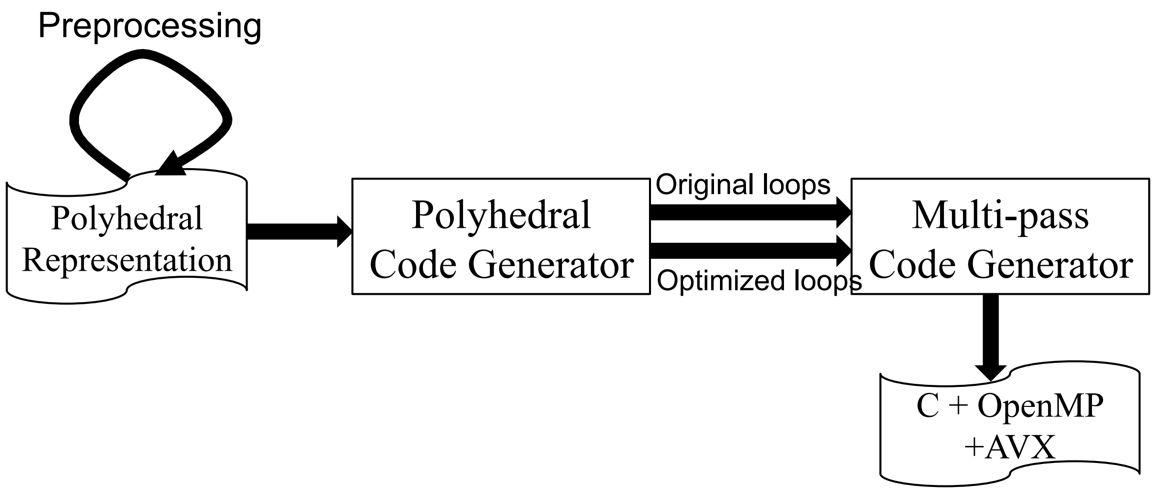

Preprocessing
Polyhedral Representation
Polyhedral Code Generator
Original loops
Optimized loops
Multi-pass Code Generator
C + OpenMP +AVX

Figure 7.3. Flow graph for the code generation step.

7.2. Experimental Evaluation for Overall Energy Savings

Our code generation framework is now able to produce FMPP codes with vectorization. In this section, we evaluate the overall energy improvement of the vectorized FMPP compares with the original code generated for standard wavefront parallelization. Our experiments are performed on the Haswell architecture and all the benchmarks used in section 6.5. The problem size is adjusted so that the data footprint in one wavefront is much larger than the LLC capacity.

In Figure 7.4, we show the over all energy efficiency of the codes produced by our code generation framework. Compared with the original codes produced with standard wavefront parallelization, we are able to reduce both the static energy consumption and dynamic energy consumption for most test cases. Over 20% total energy saving is achieved for the Jacobi 2D, Heat 2D and Wave 2D, and about 15% energy reduction for Blur 2D. For the 3D stencils, we are not able to achieve significant improvement, only about 5% for Heat 3D, and almost no gain for Jacobi 3D.

7.3. Conclusion

We developed a code generation framework that supports both our energy-efficient tiling and parallelization strategy and the compilation method for vectorization. Our framework is developed as a semi-automatic tool, which allows many optimizations to be applied in an optional way, and this helps exploration of trade-offs of different optimizations.

With both improvement in dynamic memory energy consumption and static energy, we are able to reduce the overall energy consumption significantly. Our experimental results on one of the most recent Intel architecture show that over 20% of overall energy improvement can be achieved for most 2D test cases. For the Heat 3D, we are able to reduce the overall

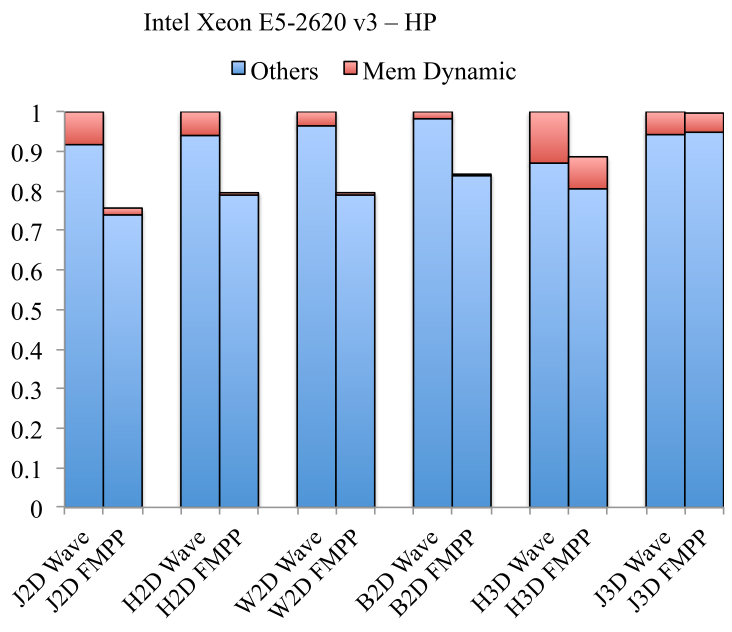


FIGURE 7.4. Energy efficiency of the codes produced from our code generation framework

energy consumption by 10%, but almost no energy saving is achieved for the Jacobi 3D benchmark.

One important problem raised here is the time spent for searching the tuning space – which usually takes weeks to finish one set of experiments. Since our code generation framework allows many optimizations to be applied in an optional way, this significantly increased the dimension of the tuning space. Therefore, it is very important for our code generation framework to have an autotuning approach that can help reduce the searching space. In the next chapter, we take the first step of building an autotuner for our framework.

CHAPTER 8

Artificial Neural Network Assisted Autotuning for Performance

Performance tuning with exhaustive search is a very time-consuming step, which can take up to weeks or months to finish. The performance tuning problem is also severe for our code generation framework. One example is that the experiments in section 6.5 took over two weeks to finish. This is clearly unacceptable in a production setting. Therefore, it is important to develop techniques to help tune the performance quickly.

Existing performance tuning techniques are separated into two categories: analytical model based approaches and prediction-based approaches. During the past decades, many analytical models [18, 14, 42, 30, 64] have been developed to assist the tile size selection problem. However, abstracting the interaction between kernels and hardware architectures into a single analytical model is always a challenge. Especially with the increased complexity of today's hardware architecture, developing analytical model to guide the performance tuning across kernels and platforms becomes almost impossible. This is the position taken by a large community of researchers that has promoted iterative compilation and auto-tuning. One of the most successful techniques is utilizing machine learning techniques to assist the choice of available performance tuning parameters. In this chapter, we describe our use of Artificial Neural Networks to assist the choice of tile sizes and the optimizations supported in our framework. Currently, this study is only performed for performance tuning with optimizations described in Chapter 6, but it can be generated to support the tuning for energy.

8.1. Our Approach

Artificial Neural Network (ANN) is a supervised learning method, which takes pairs of inputs and desired outputs, and learns some function that minimizes the error between the function outputs and the desired outputs. Our ***goal*** is to utilize ANN to help find the tile sizes and optimizations that yield optimal or near optimal performance. There are two ways to achieve this: 1) train the ANN to predict the desired tile sizes and optimizations; 2) train the ANN to predict the performance with tile sizes and optimizations as part of the inputs, and then search the solution space using the trained model to find the one that gives the best performance. As the first step, we use the second approach to achieve our goal. We observe that the solution with the *best* predicted performance usually does not give the actual optimal or near optimal performance. However, the optimal/near optimal solution usually occurs in the top K predicted solutions, and K is generally a small number (around 30 based on our observation). Therefore, we return an optimal solution space that contains the top K solutions in our approach, instead of just one solution.

In this section, we first describe the inputs and configurations for the ANN, and then give an overview of our approach.

8.1.1. Input Features.

In order for the neural network trained model to identify the platform, kernels, and optimizations applied, we extract four types of features as inputs to the neural network . The four types of features are *hardware features*, *kernel features*, *program input features* and *optimization features*.

The hardware features extracted are summarized in Table 8.1, and we also describe how the value for each hardware feature is obtained in the description column. Table 8.2 gives the program features extracted for identifying the kernels. For the computations that happen within each iteration, only addition, subtraction and multiplication are extracted in the features in our work. This is because most of the kernels we handle only have these three types of operations, but it can be extended to include other operation types.

TABLE 8.1. Hardware features extracted for identifying the platform.

Feature name	description
Number of cores	architecture specification
Clock frequency	architecture specification
L1/L2/L3 cache size	architecture specification
L1/L2/L3 cache memory bandwidth	Intel optimization reference manual [43]
Off-chip memory bandwidth	measured by STREAM benchmark [63]

TABLE 8.2. Program Features for characterizing the kernels.

Feature name	description
space order	longest dependence length along the space dimensions
time order	longest dependence length along the time dimension
number of points	number of neighboring points
number of adds	number of addition operations in each iteration
number of subs	number of subtraction operations in each iteration
number of multiplies	number of multiplication operations in each iteration

Table 8.3 summarizes the kernel input feature and the optimization features. The kernel input feature is the input problem size to the kernel, since the performance optimization for stencils is related to the problem size. The optimization feature includes parameters that can be used for performance tuning.

8.1.2. NEURAL NETWORK CONFIGURATION. We use a fully connected, multi-layer, feed-forward neural network [69], and the neural network is trained with the scaled conjugate gradient method [49], and targets minimizing the mean square error. The input

TABLE 8.3. Input and optimization features.

Kernel Input Feature	
feature name	description
Problem size	2D: 3 values, 3D: 4 values
Optimization Feature	
feature name	description
register block size	2D: 2 values, 3D: 3 values.
temporary buffering	apply temporary buffering or not (0/1).
tile size	2D: 3 values, 3D: 4 values
hyper-threading	run the program with hyper-threading or not (0/1)

parameters to the input layer of the neural network consist of all the features described in Table 8.1, Table 8.2 and Table 8.3. The output of the neural network is the performance (measured in Gflops/sec since this allows us to normalize the execution time for a range in input problem sizes). Note that the number of elements for the input feature and optimization feature is different for 2D and 3D stencils. It is possible to train the 2D stencils and 3D stencils together by treating 2D stencils as a special case of 3D stencils whose outermost dimension's size is 1. Here, we choose to train the 2D stencils and 3D stencils separately.

Two most important parameters for a multi-layer neural network are the number of hidden layers and the number of units in each layer. The previous work [114, 81, 62] on neural network assisted autotuning uses either one or two hidden layers, and at least 20 units per layer. We tried several configurations of neural network with one or two layers, and number of units from 20 to 50 within each layer. We found that a two-layer neural network with 50 units within each layer gives the best prediction accuracy, and this is used as our final neural network configuration.

8.1.3. OVERVIEW OF OUR APPROACH. After the neural network is trained for predicting the performance, we can use it to find the optimal solution space. Figure 8.1 shows the flow of our autotuning approach. For a given kernel with certain input size and a platform, we first generate the solution space by enumeration of all the possible combinations of the values of the optimization features. Then, we use the trained neural network to predict the performance for each solution in the solution space. Finally, an output filter is used to select the optimal solution space that contains the top K solutions.

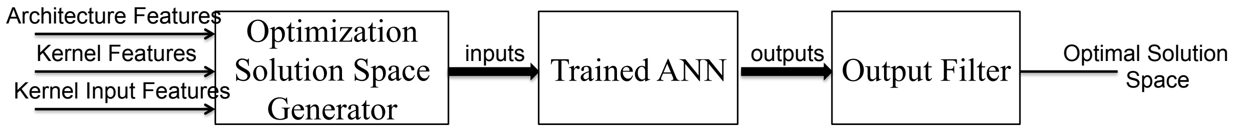


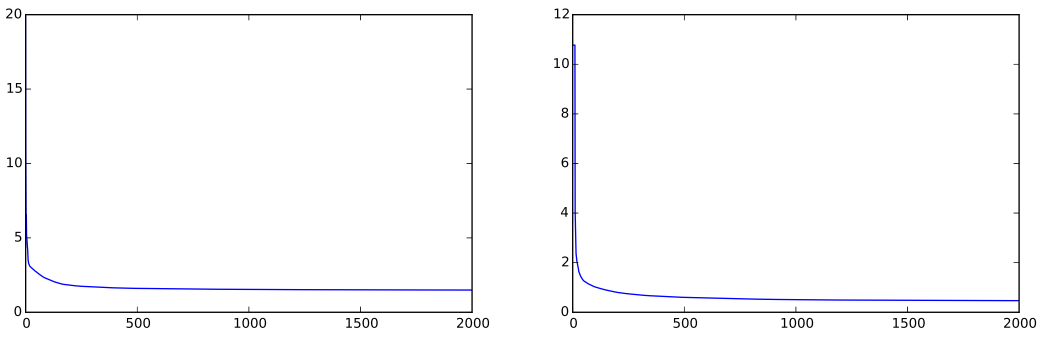
FIGURE 8.1. The flow graph of our autotuning approach.

8.2. EXPERIMENTAL VALIDATION

In this section, we validate our approach for predicting the optimal solution space with a set of kernel-platform combinations.

8.2.1. EXPERIMENTAL SETUP. We target the three Intel platforms used in Section 6.5 – SandyBridge, IvyBridge and Haswell. The data collected in Section 6.5 for all the 2D and 3D kernels are used to initialize the database for training. Since there are only two 3D kernels used in Section 6.5, we added one more 3D kernel to increase the database size for 3D stencils. The 3D kernel added is **Wave 3D (W3D)**, a 14-point stencil with second order along both time and space dimension, and with 16 computations per iteration. About 300K data points are collected for the 2D/3D stencils, and the neural network is trained for 2000 iterations. The training time for the 2D / 3D stencils took about 5 hours, which although pretty time-consuming, needs to be performed only once.

8.2.2. PERFORMANCE PREDICTION. We separate the database into two sets: training data set and testing data set. The training data set randomly picks up 80% data form the database, and the rest are used as testing data. Figure 8.2 shows the error trace (square root mean) during the training process for the 2D and 3D stencils. As shown in Figure 8.2, the error for both 2D and 3D stencils drops off very quickly and flattens after 1000 iterations.



(A) Error trace for 2D stencils (B) Error trace for 3D stencils.

FIGURE 8.2. Error trace for the square root mean during the training process.

Figure 8.3 is the scatter plot for the performance prediction result for the 2D stencils. After training for 2000 iterations, the root mean square percentage error (RMSPE) is about 8.6% for the training data set, and about 8.8% for the testing data set. The performance prediction result for 3D stencils is shown in Figure 8.4. The RMSPE for the training data set is only 4.8%, and 4.9% for the testing data set.

8.2.3. OPTIMAL SOLUTION SPACE PREDICTION. Above, we showed that the trained ANN can predict the performance pretty accurately, whereas our final goal is to predict the optimal solution space for any platform-kernel combination. We pick up nine kernel-platform combinations, and use a different problem size from the ones already in the database (usually different along one or two dimensions). The value K for the optimal solution space size is set to 30 in our experiments (this is chosen based on observations).

We run all the 30 optimization configurations in the predicted optimal solution space, and choose the one with the best performance. Then, we normalize the best performance to

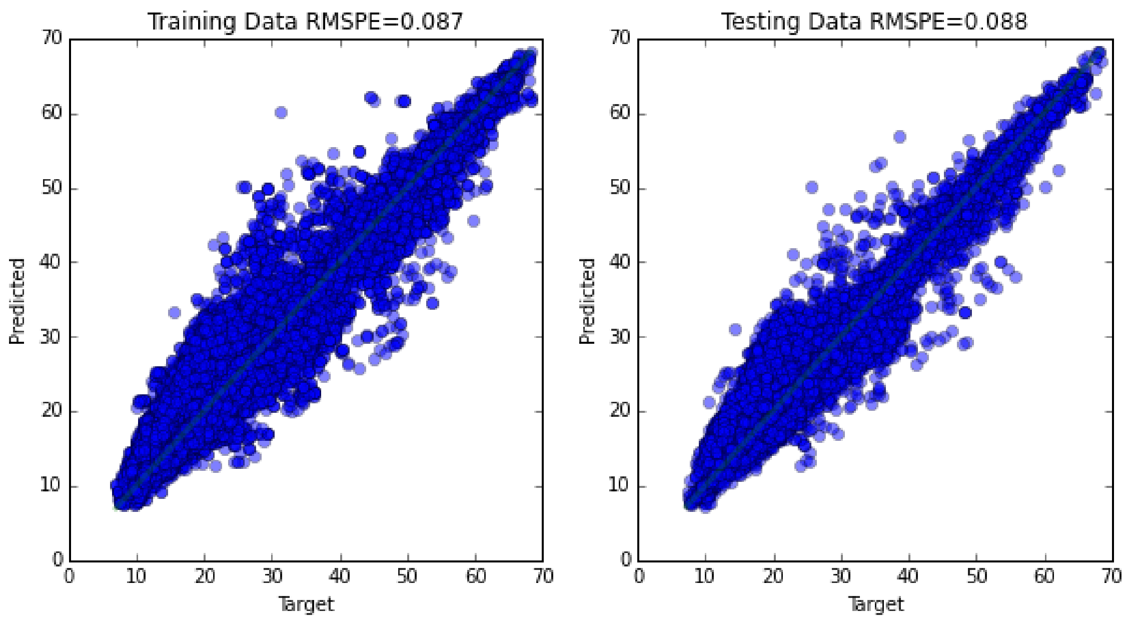


FIGURE 8.3. The performance prediction result for both training data set and testing data set for 2D stencils.

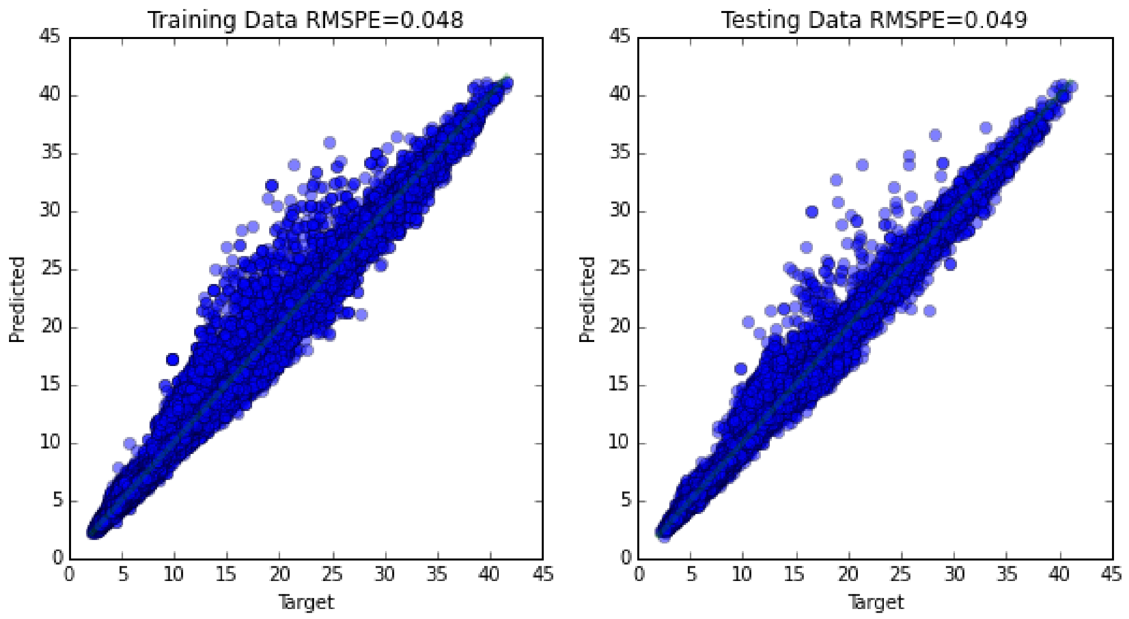


FIGURE 8.4. The performance prediction result for both training data set and testing data set for 3D stencils.

the actual optimal performance, and the result is shown in Figure 8.5. The actual optimal performance is obtained by exhaustive search. As we can see in Figure 8.5, eight out of nine test cases achieve performance within 10% of the actual optimal performance. The achieved performance of Wave 3D on Haswell is about 20% off the actual optimal performance. We notice that the amount of 3D data used for training on Haswell is relatively smaller compare to other platform, and more data might help improving the accuracy of predicting the optimal solution space.

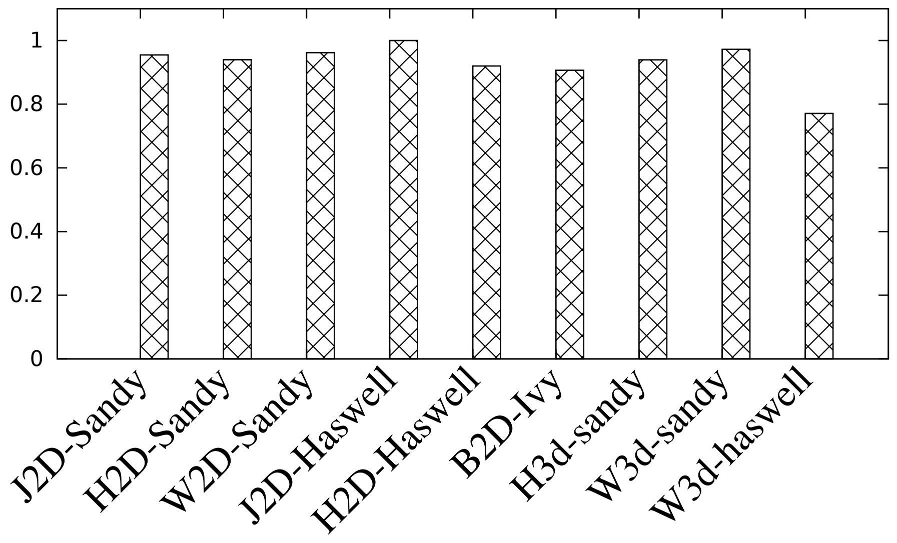


FIGURE 8.5. Normalized best performance among the predicted optimal solution space to the actual best performance.

Although the optimal solution space in our experiment contains 30 configurations for the optimization feature, the actual number of code instances that have to be generated is much smaller. As described in Table 8.3, the register block size and temporary buffering optimization need to be decided for code generation, but others like tile sizes are not needed for code generation, because we are generating parametric tiled codes. Therefore, the number of different combinations of register block size and temporary buffering is the number of code instances that have to be generated. Table 8.4 summaries the number of code instances that have to be generated for the predicted optimal solution space. The number of code instances is pretty low for all the cases, and therefore reduces the code generation and compilation time comparing with using fixed-size tiling.

TABLE 8.4. The number of code instances that occur in the predicted optimal solution space.

kernel-platform	number of code instances
J2D-Sandy	6
H2D-Sandy	1
W2D-Sandy	1
J2D-Haswell	2
H2D-Haswell	1
B2D-Ivy	4
H3D-Sandy	6
W3D-Sandy	1
W3D-Haswell	4

8.3. CONCLUSION

In this chapter, we take the first step towards building an autotuner for our code generation framework. We proposed an autotuning approach based on an exhaustive search of the tuning space with a performance model that is trained using ANN. Our experimental results showed that the ANN trained performance model can predict the performance pretty accurately. The optimal solution space predicted with the assistant of the trained model is able to capture the near optimal/optimal solution for most test cases. We also would like to thank Professor Chuck Anderson for the help of getting results using his implementation of ANN.

Although our autotuning approach is able to predict the optimal solution space with optimal/near optimal solution for most test cases, it still takes up to 30 solutions to predict the optimal/near optimal solution. Since the points in the solution space that we are interested in are the points that give optimal/near optimal performance, it might be better to initialize the database with these points, so that the model can be trained to fit the points give optimal/near optimal performance better. And this may help improving the accuracy of the optimal solution space prediction. Also, our final goal is to develop an autotuner for energy consumption, but in this work we only targeted for performance.

CHAPTER 9

Conclusion and Future Work

9.1. Conclusion

Previous work on stencil computations mainly focuses on performance optimizations. With the growing importance of energy, saving energy at the application level has become a popular research topic. In this work, we target the compute intensive stencil computations, and seek to automatically produce codes that minimize the energy consumption.

We proposed an energy-efficient tiling and parallelization strategy that seeks to reducing the dynamic memory energy consumption by minimizing the number of off-chip memory transfer, but without sacrificing performance. Experimental results on all the platforms show that we can significantly reduce the dynamic memory energy consumption – by over 65% on all platforms. We also target the main energy consumption contributor — static energy consumption — by further improving the performance of the generated codes. We presented a compilation method for vectorization that automatically produce vectorized code that can make efficient use of the available vector units. The vectorization strategies supported in our vectorizer seek to improve performance by trading memory operations with data reorganization operations, and also maximizing the register reuse. Comparing with relying on the automatic vectorizers provided by exiting commercial compilers, our vectorized code can significantly improve the performance for 2D stencils – about 35% on the SandyBridge and IvyBridge platform, and 13.5% on the Haswell platform. With reduction in both dynamic energy and static energy, we are able to get over 20% total energy savings for most 2D cases on the Haswell Platform. Our results also confirm that optimizing energy leads to good performance, but the converse is not necessarily true.

We developed a semi-automatic code generation framework to support our energy-efficient strategy and compilation method for vectorization. In our framework, many optimizations can be applied in an optional way, which is useful in helping the exploration of trade-offs of different optimizations. For example, we can specify whether to apply temporary buffering or not. Although our experimental results showed that applying temporary buffering is not beneficial for performance for many case on our test platforms, there are still situations that it helps (Jacobi 2D on IvyBridge). The trade-offs of these optimizations depend on both software and hardware features, which are hard to model. Therefore, it is useful to have a semi-automatic code generator to support different optimizations in a configurable way.

In addition to the above, we also proposed an autotuning approach to assist the choice of optimization strategies and tile sizes for performance. Our approach predicts the optimal solution space based on a performance model that is trained using an Artificial Neural Network. The tuning space of our code generation framework is very large, and performing exhaustive search over the whole space is extremely time-consuming. With our autotuning approach, we are able to significantly reduce the search space for optimal/near optimal solution, and therefore, improve the practical impact of our framework.

9.2. Future Work

There are several directions in future research raised in this work:

- adapting our energy efficient strategy and vectorization strategy for other tiling and parallelization strategy. Our work is based on the classic time skewing and wavefront parallelization. However, many authors have pointed out that for stencil computations with very small number of time iterations (1 to 8), other techniques that can be used explore concurrent start are more beneficial, such as diamond tiling, trapezoid tiling and etc. Adapting our energy efficient idea and vectorization to those techniques may raise different challenges.

- applying our energy efficient strategy on other platforms. We have shown that our strategy is beneficial on modern multi-core CPUs, and it will be interesting to see how our strategy work on other platforms like GPU and FPGA.
- optimizations for 3D cases. As shown in our result, as well as the past research, optimizations for 3D cases still remains as a challenge. Many optimization strategies, that work great for 1D and 2D cases, do not show significant improvement once they are applied to 3D cases, and sometime may even hurt the performance.
- machine learning assisted autotuning for energy efficiency. In this work, our machine learning assisted autotuning is only for performance, and it can be extended for energy prediction.

Bibliography

[1] W. Achtziger and K.-H. Zimmermann. Optimal Schedules for Recurrence Equations Via Continuous Optimization Methods. Technical Report Reports and Scripts No. 5, University of Erlangen-Nuremberg, Erlangen,German, 1999.

[2] W. Achtziger and K.-H. Zimmermann. Finding Quadratic Schedules for Affine Recurrence Equations Via Nonsmooth Optimization. *Journal of VLSI signal processing systems for signal, image and video technology*, 25(3):235–260, 2000.

[3] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

[4] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling Stencil Computations to Maximize Parallelism. In *the International conference on high performance computing, networking, storage and analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[5] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized Tiling Revisited. In *in Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM.

[6] Cedric Bastoul. 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, 2004. IEEE Computer Society.

[7] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In

ETAPS International Conference on Compiler Construction (CC'2010), pages 283–303, Paphos, Cyprus, March 2010. Springer Verlag.

[8] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, and et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. September 2008.

[9] Ian J. Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L. Chamberlain, David G. Wonnacott, and Michelle Mills Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 197–206, New York, NY, USA, 2015. ACM.

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

[11] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *Int. J. Parallel Program.*, 30(2):65–98, April 2002.

[12] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, jun 2008. ACM.

[13] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *J. Parallel Distrib. Comput.*, 5(4):334–358, August 1988.

[14] Jacqueline Chame and Sungdo Moon. A Tile Selection Algorithm for Data Locality and Cache Interference. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 492–499, New York, NY, USA, 1999. ACM.

[15] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A Framework for Composing High-Level Loop Transformations. Technical report, Department of Computer Science, University of Southern California, 2008.

[16] Linchuan Chen, Peng Jiang, and Gagan Agrawal. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 47–58, New York, NY, USA, 2016. ACM.

[17] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Proceedings of the 2011 IEEE International Parallel Distributed Processing Symposium*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society.

[18] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[19] Jason Cong and Bo Yuan. Energy-efficient Scheduling on Heterogeneous Multi-core Architectures. In *in Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 345–350, New York, NY, USA, 2012. ACM.

[20] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Rev.*, 51(1):129–159, February 2009.

[21] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[22] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, March 2000.

[23] Chris Ding and Yun He. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 50–50, New York, NY, USA, 2001. ACM.

[24] Hikmet Dursun, Ken-ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng, Richard Seymour, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. In-Core Optimization of High-Order Stencil Computations. In Hamid R. Arabnia, editor, *PDPTA*, page 533538. CSREA Press, July 2009.

[25] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM.

[26] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–53", issn="1573–7640, 1991.

[27] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *Int. J. Parallel Program.*, 21(5):313–348, October 1992.

[28] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[29] Paul Feautrier. Automatic Parallelization in the Polytope Model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, UK, 1996. Springer-Verlag.

[30] Basilio B. Fraguela, Martn G. Carmueja, and Diego Andrade. Optimal Tile Size Selection Guided by Analytical Models. In *Proceedings of the International Conference ParCo*, 2005.

[31] M. Frigo and V. Strumpen. Cache Oblivious Stencil Computations. In *International Conference on Supercomputing (ICS)*, ICS'05, pages 361–366, New York, NY, USA, 2005. ACM.

[32] Matteo Frigo and Volker Strumpen. The Memory Behavior of Cache Oblivious Stencil Computations. *The Journal of Supercomputing*, 39(2):93–112, feb 2007.

[33] Elkin Garcia, Daniel Orozco, and G Gao. Energy efficient tiling on a Many-Core Architecture. *Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2011)*, pages 53–66, 2011.

[34] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, jul 1999.

[35] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[36] Robert M. Haralick and Linda G. Shapiro. *Computer and Robot Vision*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1992.

[37] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric Multi-level Tiling of Imperfectly Nested Loops. In *in Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[38] A. Hartono, M.M. Baskaran, J. Ramanujam, and P. Sadayappan. DynTile: Parametric Tiled Loop Generation for Parallel Execution on Multicore Processors. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12. IEEE, April 2010.

[39] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data Layout Transformation for Stencil Computations on Short-vector SIMD Architectures. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[40] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A Stencil Compiler for Short-vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[41] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12):1612–1630, December 1989.

[42] Chung-hsing Hsu and Ulrich Kremer. A Quantitative Analysis of Tile Size Selection Algorithms. *J. Supercomput.*, 27(3):279–294, March 2004.

[43] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Number 248966-032. January 2016.

[44] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[45] Abhishek Jaiantilal, Yifei Jiang, and Shivakant Mishra. Modeling CPU Energy Consumption for Energy Efficient Scheduling. In *Proceedings of the 1st Workshop on Green Computing*, GCM '10, pages 10–15, New York, NY, USA, 2010. ACM.

[46] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, IPDPS '10, pages 1–12, April 2010.

[47] Mahmut Kandemir, Narayanan Vijaykrishnan, MaryJane Irwin, and HyunSuk Kim. Experimental Evaluation of Energy Behavior of Iteration Space Tiling. In SamuelP. Midkiff, JosE. Moreira, Manish Gupta, Siddhartha Chatterjee, Jeanne Ferrante, Jan Prins, William Pugh, and Chau-Wen Tseng, editors, *Languages and Compilers for*

Parallel Computing, volume 2017 of *Lecture Notes in Computer Science*, pages 142–157. Springer Berlin Heidelberg, 2001.

[48] Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and H. S. Kim. Towards Energy-Aware Iteration Space Tiling. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '00, pages 211–215, London, UK, UK, 2001. Springer-Verlag.

[49] Kristian Kersting and Niels Landwehr. *Scaled Conjugate Gradients for Maximum Likelihood: An Empirical Comparison with the EM Algorithm*, pages 235–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[50] Daegon Kim. *Parameterized and Multi-Level Tiled Loop Generation.* PhD thesis, Department of Computer Science, Colorado State University, Fort Collins, CO, USA, 2010.

[51] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level Tiling: M for the Price of One. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 51:1–51:12, New York, NY, USA, 2007. ACM.

[52] Seonggun Kim and Hwansoo Han. Efficient SIMD Code Generation for Irregular Kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 55–64, New York, NY, USA, 2012. ACM.

[53] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 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, 2013. ACM.

[54] Jonathan G. Koomey. Estimating Total Power Consumption by Servers in the U.S. and the World. Technical report, Lawrence Derkley National Laboratory, feb 2007.

[55] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[56] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[57] Ruby B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[58] Rainer Leupers. Code Selection for Media Processors with Simd Instructions. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '00, pages 4–8. ACM, 2000.

[59] Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS'15, pages 187–196, Newport Beach, CA, USA, 2015. ACM.

[60] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. Towards energy efficiency and maximum computational intensity for stencil algorithms using wavefront diamond temporal blocking. *CoRR*, abs/1410.5561, 2014.

[61] Tareq M. Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein, and David E. Keyes. Multicore-Optimized Wavefront Diamond Blocking for Optimizing Stencil Updates. *CoRR*, abs/1410.3060, 2014.

[62] Abid M. Malik. Optimal Tile Size Selection Problem Using Machine Learning. In *Proceedings of the 2012 11th International Conference on Machine Learning and Applications - Volume 02*, ICMLA '12, pages 275–280, Washington, DC, USA, 2012. IEEE Computer Society.

[63] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture(TCCA) Newsletter*, pages 19–25, dec 1995.

[64] Sanyam Mehta, Gautham Beeraka, and Pen-Chung Yew. Tile Size Selection Revisited. *ACM Trans. Archit. Code Optim.*, 10(4):35:1–35:27, December 2013.

[65] Inc Micron Technology. DDR3 SDRAM System-Power Calculator. `http://www.micron.com/products/support/power-calc/`.

[66] Lauri Minas and Brad Ellison. The Problem of Power Consumption in Servers. *Intel Press*, 2009.

[67] Nicholas Mitchell, Karin Högstedt, Larry Carter, and Jeanne Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[68] D.I. Moldovan and J. Fortes. Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays. *IEEE Transactions on Computers*, C-35(1):1–12, Jan 1986.

[69] Matin Moller. *Efficient Training of Feed-Forward Neural Networks.* PhD thesis, Computer Science Department, Aarhus University, Aarhus, Denmark, Nov 1997.

[70] D. Nuzman and A. Zaks. Autovectorization in GCC – two years later. In *Proceedings of the GCC Developers Summit*, Jun 2006.

[71] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of Interleaved Data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 132–143, New York, NY, USA, 2006. ACM.

[72] University of Utah. Omega+: Pressburger engine and code generator. `http://ctop.cs.utah.edu/ctop/?page_id=21`.

[73] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Pschel. Applying the Roofline Model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, March 2014.

[74] Daniel Orozco, Elkin Garcia, and Guang Gao. Locality Optimization of Stencil Applications Using Data Dependency Graphs. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 77–91, Berlin, Heidelberg, 2011. Springer-Verlag.

[75] A. Peleg and U. Weiser. MMX Technology Extension to The Intel Architecture. *IEEE Micro*, 16(4):42–50, Aug 1996.

[76] Liu Peng, R. Seymour, K. Nomura, R.K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W.R. Volz, and C.C. Wong. High-Order Stencil Computations on Multicore Clusters. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, IPDPS '09, pages 1–11, may 2009.

[77] Louis-Noël Pouchet, Uday Bondhugula, Cdric Bastoul, Albert Cohen, and J. Ramanujam. Combined iterative and model-driven optimization in an automatic parallelization

framework. In *In SC '10: International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, New Orleans, LA, 2010. IEEE Computer Society Press.

[78] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.

[79] Patrice Quinton and Vincent Dongen. The Mapping of Linear Recurrence Equations on Regular Arrays. *J. VLSI Signal Process. Syst.*, 1(2):95–113, October 1989.

[80] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2013.

[81] Mohammed Rahman, Louis-Noël Pouchet, and P. Sadayappan. Neural Network Assisted Tile Size Selection. In *International Workshop on Automatic Performance Tuning*, IWAPT'2010, Berkeley, CA, June 2010. Springer Verlag.

[82] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. *On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies*, pages 488–503. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986.

[83] Fabrice Rastello and Thierry Dauxois. Efficient Tiling for an ODE Discrete Integration Program: Redundant Tasks Instead of Trapezoidal Shaped-Tiles. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 138–, Washington, DC, USA, 2002. IEEE Computer Society.

[84] L. Renganarayana, M. Harthikote-Matha, R. Dewri, and S. Rajopadhye. Towards Optimal Multi-level Tiling for Stencil Computations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, March 2007.

[85] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized Tiled Loops for Free. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 405–414, New York, NY, USA, 2007. ACM.

[86] Gabriel Rivera and Chau-Wen Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, pages 32–32, Washington, DC, USA, Nov 2000. IEEE Computer Society.

[87] Gabriel Rivera and Chau-Wen Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[88] A. Sawdey, M. O'Keefe, R. Bleck, and R. W. Numrich. The Design, Implementation, and Performance of a Parallel Ocean Circulation Model. In *in Proceedings of the 6th ECMWF Workshop on the Use of Parallel Processors in Meteorology*, pages 523–550, November 1994.

[89] Aaron Sawdey and Matthew T. O'Keefe. Program Analysis of Overlap Area Usage in Self-Similar Parallel Programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '97, pages 79–93, London, UK, UK, 1998. Springer-Verlag.

[90] S. Shrestha, J. Manzano, A. Marquez, J. Feo, and G. Gao. Jagged Tiling for Intra-tile Parallelism and Fine-Grain Multithreading. In *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '14, pages 161–175, Cham, September 2014. Springer International Publishing.

[91] Yonghong Song and Zhiyuan Li. *Impact of Tile-Size Selection for Skewed Tiling*, volume 613 of *The Springer International Series in Engineering and Computer Science*. Springer US, 2001.

[92] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. ACM.

[93] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent Storage Mapping for Loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 24–33, New York, NY, USA, 1998. ACM.

[94] R. Strzodka, M. Shaheen, D. Pajak, and H-P. Seidel. Cache Oblivious Parallelograms in Iterative Stencil Computations. In *Proceedings of the 24th ACM/SIGARCH International Conference on Supercomputing (ICS)*, ICS'10, pages 49–59, New York, NY, USA, June 2010. ACM.

[95] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache Accurate Time Skewing in Iterative Stencil Computations. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, ICPP'11, pages 571–581, Sept 2011.

[96] A. Taflove and S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech Hourse Publishers, third edition, 2005.

[97] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the 23rd ACM*

symposium on Parallelism in algorithms and architectures, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[98] S. Tavarageri, L. N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Dynamic Selection of Tile Sizes. In *the 18th International Conference on High Performance Computing*, HPC '11, pages 1–10. IEEE, Dec 2011.

[99] Knoxville The University of Tennessee. Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[100] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. Auto-tuning for Energy Usage in Scientific Applications. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP'11, pages 178–187, Berlin, Heidelberg, 2011. Springer-Verlag.

[101] Yuki Tomofumi. *Beyond Shared Memory Loop Parallelism In The Polyhedral Model*. PhD thesis, Department of Computer Science, Colorado State University, Fort Collins, CO, USA, 2013.

[102] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 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, 2009. IEEE Computer Society.

[103] Sven Verdoolaege. Isl: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[104] Doran K. Wilde. The ALPHA Language. Technical report, Unite de recherche Inria Roquencourt et Sophia-Antipolis, 1994.

[105] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[106] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[107] Dave G. Wonnacott and Michelle Mills Strout. On the scalability of loop tiling techniques. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, January 2013.

[108] David Wonnacott. Time Skewing for Parallel Computers. In *the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 477–480, Berlin, Heidelberg, 2000. Springer-Verlag.

[109] David Wonnacott. Using Time Skewing to Eliminate Idle Time Due to Memory Bandwidth and Network Limitations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, IPDPS '00, pages 171–, Washington, DC, USA, 2000. IEEE Computer Society.

[110] David Wonnacott. Achieving Scalable Locality with Time Skewing. *Int. J. Parallel Program.*, 30(3):181–221, Jun 2002.

[111] Jingling Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[112] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An Experimental Comparison of Cache-oblivious and Cache-conscious Programs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.

[113] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In Hironori Kasahara and Keiji Kimura, editors, *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, LCPC'12, pages 17–31, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[114] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 190–199, New York, NY, USA, 2010. ACM.

[115] Hao Zhou and Jingling Xue. Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 59–69, New York, NY, USA, 2016. ACM.