Automatic Code Generation for Stencil Computations on GPU Architectures

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Justin Andrew Holewinski, M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University
2012

Dissertation Committee:
P. S. Sadayappan, Advisor
Atanas Rountev
Radu Teodorescu
ABSTRACT

The development of parallel architectures is now nearly ubiquitous in not only the high-performance computing field, but also the commodity electronics market. Even embedded processors found in cell phones and tablet computers are starting to incorporate parallel architectures. These architectures are exploiting both SIMD (Single-Instruction Multiple-Data) and SIMT (Simple-Instruction Multiple-Thread) parallelism to achieve higher levels of performance that was previously possible. Additionally, multiprocessors are becoming increasingly heterogeneous by incorporating different architectures into the same die, such as NVIDIA’s Tegra and AMD’s Fusion APU. As the computer hardware industry moves to increasingly parallel and heterogeneous architectures, the computer software industry has been forced to make drastic changes to the way software is designed and developed. This has led to an increasing burden on software developers to not only write bug-free software, but also to scale software performance with the diverse architectures.

Multi-processors are increasingly exposing SIMD parallelism as a way to improve the per-core performance of the hardware without requiring significant clock speed increases. Vector instruction sets are using larger vector widths to increase the parallelism available. Intel’s SSE instruction set on x86 uses 128-bit vector and instructions that can operate on 4 single-precision or 2 double-precision floating-point numbers at a time. Recently, the AVX instruction set on x86 extends to 256-bit
vectors and instructions that can operate on 8 single-precision or 4 double-precision floating-point numbers at a time. Exploiting the SIMD parallelism available in modern hardware can be a difficult process for software developers. Vector instruction sets often impose limitations such as alignment restrictions on vector loads/stores and a lack of scatter/gather operations that make it a non-trivial process to convert scalar code into higher-performance vector code. In the first part of this dissertation, we present a method for automatically finding sections of scalar application code that would likely benefit from SIMD vectorization.

Many-core architectures such as those found in Graphics Processing Units (GPUs) have become good targets for high-performance applications such as those found in scientific computing. Modern high-end GPUs have a theoretical floating-point throughput of over 2 TFlop/s, making them prime targets for scientific computing. Programming environments such as NVIDIA’s CUDA [45], Khronos’ OpenCL [22], and Microsoft’s DirectCompute allow application developers to write software that executes directly on GPU hardware, but their abstractions are very close to the actual hardware and are complex for developers to use. To take advantage of GPU devices, application developers must first determine which parts of their applications would benefit from GPU acceleration, then port those parts of their application to CUDA, OpenCL, or DirectCompute. After porting their code, significant optimization effort is then often required to maximize performance of the code, separate from any previous effort spent on optimization of the GPU versions. This makes GPU programming very complex for computational scientists and other software writers that do not have a background in computer architecture and GPU programming models. In the second part of this dissertation, we present
an automatic code generation framework for stencil computations on GPU devices. Stencil computations are important parts of many scientific applications, including PDE solvers, grid-based simulations, and image processing. Our code generation framework takes a high-level description of the stencil problem and generates high-performance code for a variety of GPU architectures.

The performance of GPU programs is often highly dependent on the choice of thread block and tile size. The optimal choice of block and tile size can be different based on the characteristics of the GPU program. The code generation framework for stencil programs proposed in this work is parameterized on the choice of block and tile size, and the choice of block and tile size can have a large impact on performance. In the third part of this dissertation, we explore the effects of the choice of block and tile size on the performance of stencil programs generated by our code generation framework and propose a performance model that uses a description of the stencil program and the target GPU hardware to automatically select an optimal block and tile size.
To Jeana, who never stopped believing;

To my parents, who always pushed me to do my best;

And to Casper and Mary; to whom I am forever grateful.
ACKNOWLEDGMENTS

My journey through college has been a long and often trying experience, but it has taught me much about the educational process, computer science research, and even life itself. After nine years at Ohio State I feel very fortunate to have been part of something truly amazing and meaningful, and to have been given the opportunity to work with some of the best people in the field.

First, I want to thank my advisor, Saday, and the rest of my dissertation committee, Nasko and Radu. Without their support, this work would not have been possible. Without their guidance, I would never have found my way to completing this work. It can be very easy to feel overwhelmed and even depressed in grad school, but they always led me back to the path to enlightenment. They believed in me during some of the most trying times of grad school, and I cannot thank them enough for that.

I feel very fortunate to have worked with the current and past students of Saday’s research group. When I was a new grad student, Jim Dinan, Brian Larkins, Muthu Baskaran and Uday Bondhugula showed me the ropes. They helped me to start my research career and I always enjoyed conversations with them. I could always look up to them for advice in situations that they themselves were in just a few years earlier. I thank them for making my first couple of years more enjoyable and
productive. After they left, Tom Henretty, Kevin Stock, and I became the senior students in the lab. We experienced the trials and tribulations of grad school together, and I always felt like they had my back if and when I needed it. Now, as I prepare to leave the lab, I leave it in the hands of Mahesh Ravishankar, Md Humayun Arafat, Naznin Fauzia, Naser Sedaghati, and everyone else in Dreese 474/574. They were always willing to help with paper and project deadlines, and I can only hope that I have made their experiences slightly less painful.

None of this work would have been possible without the love and support of my parents, Matthew, Melissa, and Ann; my grandparents Larry, Mary Ann, Jack, and Dorothy; my great-uncle Casper and great-aunt Mary; my friends Adam Hinzey, Matt Spayde, Matt Emmelheinz, and Michael McGrath; and my fiancée Jeana. In the time that I have been in grad school, I have gained new friends and met my future wife. I have broadened my horizons and found what I want to do with my life. Whenever I ask myself if it was worth it, I always remind myself of that and it all becomes clear.

I want to add a special thanks to Mark Arnold, the administrator of our RI cluster. I cannot even begin to count how many times I had to ask him to restart a machine after an experiment of mine had crashed it. He was always willing to work with me to make sure everything I needed was available, and a lot of the results in this dissertation would have been a lot harder to obtain without his help.

Justin Holewinski
Columbus, Ohio
December 3, 2012
VITA

February 18, 1985 ........................................... Born: Toledo, OH, USA

Summer 2004—Winter 2007 ......................... Software Integration Intern,
American Electric Power,
Columbus, OH, USA

June 2007 .................................................. B.S.C.S.E.,
The Ohio State University,
Columbus, OH, USA

Autumn 2008—Present ............................... Graduate Research Associate,
The Ohio State University

December 2011 ........................................... M.S.,
The Ohio State University,
Columbus, OH, USA

March 2012—June 2012 ............................. Software Engineering Intern,
NVIDIA Corporation,
Santa Clara, CA, USA

PUBLICATIONS

Research Publications

Justin Holewinski, Louis-Noël Pouchet, P. Sadayappan,

Justin Holewinski, Naznin Fauzia, Mahesh Ravishankar, Louis-Noël Pouchet, Atanas Rountev, P. Sadayappan
Jeswin Godwin, Justin Holewinski, P. Sadayappan

**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan
TABLE OF CONTENTS

Abstract ................................................................. ii
Dedication ............................................................... v
Acknowledgments ......................................................... vi
Vita .................................................................. viii
List of Figures .......................................................... xiv
List of Tables ............................................................. xvii
List of Algorithms ......................................................... xviii

Chapters:

1. Introduction ............................................................ 1
   1.1 Dynamic Analysis for SIMD/SIMT Parallelism ................. 1
   1.2 GPU Code Generation for Overlapped Tiling ................. 2
   1.3 Performance Modeling for GPU Stencil Code ............... 4
   1.4 Outline ................................................................ 4

2. Background .............................................................. 6
   2.1 SIMD/SIMT Parallelism .............................................. 6
      2.1.1 Writing Vector Code ........................................... 8
      2.1.2 Performance Implications ................................... 9
   2.2 GPU Code Generation ............................................... 11
      2.2.1 GPUs: Past, Present, and Future ......................... 12
      2.2.2 Programming Models ........................................ 14

Page

x
3. Dynamic Analysis for SIMD/SIMT Parallelism .................................. 19
   3.1 Motivation ................................................................. 20
      3.1.1 Finding Independent Operations .......................... 22
      3.1.2 Finding Operations that Access Contiguously Located Data Elements .................................................................. 28
   3.2 Dynamic Data-Dependence Graphs .......................................... 29
      3.2.1 DDG Generation ....................................................... 29
      3.2.2 Candidate Instructions ............................................ 31
      3.2.3 Generation of Parallel Partitions .............................. 31
      3.2.4 Partitioning for Contiguous Access .......................... 33
      3.2.5 Non-Unit Constant Stride Access .............................. 34
   3.3 Experimental Evaluation .................................................. 36
      3.3.1 Characterization of Applications .............................. 37
      3.3.2 Assisting Vectorization Experts ............................... 41
      3.3.3 Array-Based vs. Pointer-Based Code ....................... 42
      3.3.4 Case Studies ........................................................... 43
   3.4 Conclusion ........................................................................ 52

4. Stencil Grids ........................................................................... 58
   4.1 Computation Grid ........................................................... 58
   4.2 Data Grids ....................................................................... 60
   4.3 Stencil Functions ............................................................. 60
   4.4 Program Specification ...................................................... 61
   4.5 Example ......................................................................... 62
   4.6 OverTile: Stencil-DSL Compiler for GPGPUs ...................... 63
      4.6.1 Overview ................................................................. 63
      4.6.2 Inputs ....................................................................... 65
      4.6.3 Example Stencil Programs ....................................... 68
      4.6.4 Code Generation ...................................................... 70
   4.7 Conclusion ........................................................................ 70

5. GPU Code Generation for Overlapped Tiling ............................... 72
   5.1 Overview ......................................................................... 72
   5.2 Code Generation ............................................................. 74
   5.3 Generation of GPU Kernel Code ....................................... 80
   5.4 Generation of Host Code .................................................. 85
   5.5 Experimental Evaluation ................................................ 87
      5.5.1 OverTile Performance Results ................................. 88
5.5.2 Comparison with Automatically Optimized CPU Code . . . 90
5.5.3 Parameter Sensitivity ........................................ 91
5.5.4 CDSC Medical Imaging Benchmarks .......................... 95
5.6 Conclusion .......................................................... 98

6. Performance Modeling for GPU Stencil Code ..................... 102
6.1 Motivation ......................................................... 102
6.2 Kernel Execution Estimation ..................................... 104
   6.2.1 Phase 2 Modeling .......................................... 106
   6.2.2 Phase 3 Modeling .......................................... 108
   6.2.3 Phase 4 Modeling .......................................... 109
   6.2.4 All-Phase Modeling ........................................ 109
6.3 Model Calibration .................................................. 112
6.4 Experimental Evaluation ......................................... 113
   6.4.1 Parameter Sensitivity ...................................... 116
   6.4.2 Block/Tile Size Selection Accuracy ....................... 117
6.5 Conclusion .......................................................... 119

7. Related Work ...................................................... 120
7.1 Dynamic Analysis for Parallelism ............................... 120
7.2 GPU Code Generation and Optimization ......................... 122
   7.2.1 Polyhedral Code Optimization ............................. 123
   7.2.2 Synchronization .......................................... 123
   7.2.3 Data-Layout Transformations ............................. 124
   7.2.4 Multi-GPU Systems ....................................... 124
   7.2.5 Memory Hierarchy Optimizations .......................... 125
7.3 High-Level Languages for GPUs ................................ 125
7.4 Stencil DSL Compilers .......................................... 126
7.5 GPU Performance Modeling ....................................... 127

8. Future Work ....................................................... 129
8.1 Dynamic Analysis for Vectorization Potential .................. 129
8.2 GPU Code Generation from High-Level Specifications .......... 130
8.3 Hybridization of Dynamic Analysis and GPU Code Optimization . 131

9. Conclusion .......................................................... 132

Appendices:
A. Generated Code .......................... 133
   A.1 Jacobi 2D 5-Point ...................... 133
   A.2 Rician 3D ............................ 138

Bibliography ............................. 150
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 SIMD vector processing.</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Loading a vector register with elements with stride-6.</td>
<td>10</td>
</tr>
<tr>
<td>2.3 NVidia Architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Dynamic Analysis Architecture</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Sample dynamic data-dependence graphs</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Dependences for Example 1.</td>
<td>24</td>
</tr>
<tr>
<td>3.4 Dependences for Example 2. Part (a) shows the actual data dependencies for the code, part (b) shows the Larus-style partitioning, and path (c) shows the desired partitioning for vectorization.</td>
<td>26</td>
</tr>
<tr>
<td>4.1 Example of a 2-D grid</td>
<td>59</td>
</tr>
<tr>
<td>4.2 OverTile compiler workflow</td>
<td>64</td>
</tr>
<tr>
<td>4.3 Grammar for OverTile Stencil Specification</td>
<td>71</td>
</tr>
<tr>
<td>5.1 Jacobi 5-Point Stencil</td>
<td>74</td>
</tr>
<tr>
<td>5.2 Overlapped Jacobi 5-Point Stencil for T = 2.</td>
<td>74</td>
</tr>
<tr>
<td>5.3 Tile region computation for 1-D stencil</td>
<td>78</td>
</tr>
<tr>
<td>5.4 Tile region computation for 1-D multi-stencil</td>
<td>80</td>
</tr>
</tbody>
</table>
5.5 Performance evaluation of OverTile-generated GPU kernels on Tesla C2050 and Tesla K10 (single-precision). The numbers above the bars show the achieved GFlop/s. .................................................. 88

5.6 Performance evaluation of OverTile-generated GPU kernels on Tesla C2050 and Tesla K10 (double-precision). The numbers above the bars show the achieved GFlop/s. .................................................. 89

5.7 Performance evaluation of OverTile-generated GPU kernels vs. Pochoir, a state-of-the-art stencil compiler for multi-core CPUs. The CPU benchmarks were executed on a dual-socket Xeon E5630 with 8 threads. (single-precision) ............................................................... 91

5.8 Performance evaluation of OverTile-generated GPU kernels vs. Pochoir, a state-of-the-art stencil compiler for multi-core CPUs. The CPU benchmarks were executed on a dual-socket Xeon E5630 with 8 threads. (double-precision) ............................................................... 92

5.9 Performance variation with respect to changes in time tile size for Jacobi 2D and Heat 2D stencil programs. Problem size: $4000^2$ ........................................ 93

5.10 Performance variation with respect to changes in spatial tile size (Y) for Jacobi 2D and Heat 2D stencil programs. Problem size: $4000^2$ ........................................ 94

5.11 Performance counter data for Jacobi 2D stencil on Tesla C2050 for a range of time tile sizes. ................................................................. 95

5.12 Performance results for CDSC medical imaging benchmarks. ............... 96

6.1 Elapsed time and occupancy for a few configurations of a Jacobi 2D kernel. ......................................................................................... 103

6.2 Control flow through a generated stencil program kernel. ................. 105

6.3 Model fit for a Jacobi 2D stencil on Tesla C2050. ............................ 113

6.4 Model fit for a Poisson 2D stencil on Tesla C2050. ............................ 114

6.5 Model fit for a Total Variation stencil on Tesla C2050. .................... 115

6.6 Model fit as a function of spatial tile size. ....................................... 116
6.7  Model fit as a function of time tile size. . . . . . . . . . . . . . . . . . . 117
6.8  Model fit as a function of block size. . . . . . . . . . . . . . . . . . . 118
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Analysis results for analyzed benchmark loops</td>
</tr>
<tr>
<td>3.2</td>
<td>Analysis results for computation kernels</td>
</tr>
<tr>
<td>3.3</td>
<td>Analysis results for UTDSP benchmark suite</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance measurements for the case studies</td>
</tr>
<tr>
<td>5.1</td>
<td>Characteristics of stencil programs used for performance evaluation</td>
</tr>
<tr>
<td>5.2</td>
<td>Rician Denoise 3D performance results before and after the elimination of field g</td>
</tr>
<tr>
<td>6.1</td>
<td>Hardware constraints for NVIDIA GPUs</td>
</tr>
<tr>
<td>6.2</td>
<td>Accuracy of GPU performance model</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Timestamp computation</td>
<td>32</td>
</tr>
<tr>
<td>5.1 Overlapped-tiling code generation algorithm</td>
<td>75</td>
</tr>
<tr>
<td>5.2 Finding the tile size</td>
<td>77</td>
</tr>
<tr>
<td>5.3 Generating shared-memory definitions</td>
<td>82</td>
</tr>
<tr>
<td>5.4 Thread code generation</td>
<td>84</td>
</tr>
<tr>
<td>5.5 Host code generation</td>
<td>86</td>
</tr>
</tbody>
</table>
LIST OF LISTINGS

2.1 A simple scalar loop. ...................................................... 7
2.2 A simple vector loop. ...................................................... 8
2.3 A simple vector loop in SSE intrinsics. ............................... 8
2.4 A simple CUDA kernel: $A = A + B$ .................................. 17
2.5 A simple OpenCL kernel: $A = A + B$ ................................. 17
2.6 A simple DirectCompute kernel: $A = A + B$ ...................... 18
3.1 Example 1 for dynamic parallelism analysis. ....................... 23
3.2 Example 2 for dynamic parallelism analysis. ....................... 25
3.3 Vectorization benefits of data layout transformations: stride-N column access and array-of-structures access. .................. 35
3.4 Loop transformations and data layout transformations applied to Listing 3.3 ......................................................... 36
3.5 Original and transformed Gauss-Seidel code. ....................... 44
3.6 Original and transformed 2D PDF Solver. ............................ 46
3.7 Original and transformed bwaves code. ............................. 55
3.8 Original and transformed milc code. .................................. 56
3.9 Original and transformed gromacs code. ............................. 57
4.1 A Jacobi 2D stencil program written in SSP. ....................... 68
4.2 A Finite-Difference 2D stencil program written in SSP. ........... 69
5.1 Original Rician Denoise 3D stencil program. . . . . . . . . . . . . . . 100
5.2 Transformed Rician Denoise 3D stencil program. . . . . . . . . . . . . 101
CHAPTER 1

Introduction

Parallelism has become a primary way of improving the performance of micro-processors by increasing (1) the number of cores (multi-core), and (2) the width of vector datapaths within cores (vectorization). Current high-performance Intel Xeon CPUs have up to 8 cores, each with 128-bit vector data paths; accelerators such as Intel Xeon Phi [14] have more than 50 cores, each with 512-bit vector data paths; and current NVIDIA GPGPU accelerators have 1,536 cores with scalar data paths. It is therefore increasingly important for applications to take advantage of both multi-core and vector parallelism. In this work, we propose two methods for achieving high performance on modern parallel architectures. First, we propose a dynamic analysis technique that works on arbitrary serial code and guides developers to sections of code that would benefit from manual vectorization. Then, we propose a code generation and performance modeling technique for stencil computations on GPU accelerators.

1.1 Dynamic Analysis for SIMD/SIMT Parallelism

Modern multi-processors are using increasingly wide vector instruction sets as a means of improving the performance of time-critical code. These instruction sets
allow for a single instruction to be applied to multiple data elements at the same
time, not only increasing the parallelism in the system but also cutting down on
the time required for instruction issue and dispatch by reducing the total number
of instructions in the program. This increased efficiency comes at a cost, however.
These instruction sets often impose strict restrictions on their use, such as enforc-
ing aligned access for vector loads/stores and only allowing operations to act on
contiguous elements in vector registers.

This work presents an automatic dynamic analysis technique that characterizes
the vectorization potential for arbitrary sequential code. This analysis leads devel-
opers to sections of code that exhibit access patterns that could benefit from vector-
ization. Developers can use this analysis to concentrate their efforts on portions of
code that are amenable to vectorization but auto-vectorizing compilers fail to vec-
torize. Compilers writers can also use this analysis to identify vectorizable sections
of code that their compilers are missing.

1.2 GPU Code Generation for Overlapped Tiling

As special-purpose graphics accelerators become increasingly general-purpose
in terms of the types of applications that can be efficiently executed, it is becom-
ing increasingly important for programming models and compiler technology to
abstract the hardware from the general user. Current Graphics Processing Units
(GPUs) are capable of over 2 teraflops of floating-point through-put, but program-
mimg these devices is often done at a low level, requiring intimate knowledge of
the hardware architecture.
This makes general-purpose computing on such devices possible, but most users such as computational scientists and application engineers do not have the time to properly understand the architecture, just to accelerate portions of their code. Therefore, it is becoming increasingly important to develop higher level abstractions that such users can employ to write a specification of their program and let the compiler automatically generate efficient low level code for any target architecture, be it present and future GPU architectures, or CPU architectures that implement high-throughput SIMD vector units.

This work presents code analysis and optimization techniques that aim to improve the run-time performance of scientific applications without requiring application writers to have intimate knowledge of the target hardware. We present methods for both the characterization of program performance on vector-enabled Instruction Set Architectures (ISAs) as well as automatic code generation for GPU architectures from high-level stencil specifications.

We focus on stencil computations for several reasons. First, stencil computations are very common in scientific applications and are used to implement many types of grid-based solvers that iterate over time to reach a numeric solution of a given precision. Second, stencil computations make interesting targets for GPU architectures where memory throughput is high but direct implementation of such code is often very inefficient due to peculiarities of current GPU architectures.

Our goal is thus to create a code analysis and generation framework that allows potential users, such as computation scientists, to write their programs using higher-level abstractions than those found in current GPU-based programming
models/languages while automatically taking advantage of GPUs and other accelerator architectures through a robust and effective compiler framework. For the purposes of this work, Graphics Processing Units (GPUs) are the primary architectural target for code optimization. Such hardware is becoming ubiquitous in nearly all forms of devices, from high-end servers to cheap desktops and smartphones. It is also a very promising target for computational workloads due its high floating-point peak throughput.

1.3 Performance Modeling for GPU Stencil Code

The code generation framework proposed in this work is parametric on the choice of block and tile size for the GPU kernel. Auto-tuning can be used to select an optimal block/tile size, but is expensive and must be performed for every program variant being compiled. Instead, we propose a performance model that uses characteristics of the stencil program being compiled and the target machine to automatically determine an optimal block/tile size. This model can be applied in a fraction of the time required by auto-tuning approaches.

1.4 Outline

The rest of this document is organized as follows. In Chapter 2 we present background material pertaining to SIMD vectorization and existing dynamic approaches to identifying parallelism, and GPU hardware and their current programming models and languages. Related work in the fields of dynamic analysis for vectorizability, GPU code generation, and GPU performance modeling are presented
in Chapter 7. In Chapter 3 we present a dynamic analysis technique that characterizes the potential performance of arbitrary sequential applications on abstract SIMD architectures. This work can be used by application developers to identify sections of code that would likely benefit from SIMD-enabling transformations, as well as compiler writers to identify vectorizable sections of code that static compiler techniques currently miss. An experimental evaluation of this work is also provided in this chapter. We introduce the concept of stencil computations and their definition in Chapter 4. In Chapter 5 we present a set of code generation algorithms that produce high-performance GPU code from high-level stencil specifications, and an experimental evaluation of these algorithms. This algorithm produces both GPU kernel code and CPU host code, providing a complete solution that can be used by users to accelerate computations. We build a GPU performance model for this code generation framework in Chapter 6 along with an analysis of its effectiveness. Finally, we provide an overview of future work in 8 and conclude in Chapter 9.
CHAPTER 2

Background

2.1 SIMD/SIMT Parallelism

Modern multi-processors incorporate vector instruction sets that use the Single-Instruction Multiple-Data (SIMD) paradigm to accelerate application performance. These instruction sets expose vector registers that can hold multiple scalar values, and vector instructions that act on vector registers and apply an operation to each operand in the vector. Current implementations such as Intel SSE and ARM NEON use 128-bit vector registers that can hold sixteen 8-bit values, eight 16-bit values, four 32-bit values, or two 64-bit values. A single vector arithmetic operation on these instruction sets will perform the operation on every scalar value held in the input operand registers. A simple example of this process is shown in Figure 2.1.

There are several advantages to such vector instruction sets. First, the hardware can be designed to perform vector operations concurrently. The Arithmetic Logic Units (ALUs) can be replicated to handle all vector components at once, greatly increasing the number of scalar operations per second the chip can perform. Second, vector instructions can decrease the total number of instructions in the application.
by grouping instructions that perform the same operation on different data elements, which decreases the time required to decode and issue instructions. Both of these have implications for both performance and power consumption. By completing the task faster and decoding and issuing fewer instructions, the chip can decrease overall power consumption.

The current trend is to continue increasing the vector width of SIMD instruction sets. Newer AMD and Intel x86 chips such as Bulldozer and Sandy Bridge support 256-bit vectors, doubling the number of scalar values that can be processed at a time. Future architectures such as Intel’s Many Integrated Cores (MIC) in the Xeon Phi accelerators support 512-bit vectors. Taking advantage of the performance offered by current and future multi-processors requires exploiting these vector instruction sets to increase the parallelism of applications.
Listing 2.2 A simple vector loop.

```c
for (i = 0; i < N; i += 4) {
    A[i:i+3] = B[i:i+3]*C[i:i+3];
}
```

Listing 2.3 A simple vector loop in SSE intrinsics.

```c
for (i = 0; i < N; i += 4) {
    __m128 VecB = _mm_load_ps(B+i);
    __m128 VecC = _mm_load_ps(C+i);
    __m128 VecA = _mm_mul_ps(VecB, VecC);
    _mm_store_ps(A+i, VecA);
}
```

2.1.1 Writing Vector Code

The basic use of vector instructions is outlined in C-style pseudo-code in Listing 2.1 and Listing 2.2. Listing 2.1 shows a simple C-style loop that multiplies the elements of two arrays element-wise to produce a third array. This is a typical implementation strategy for array-based computations in C-like languages. Listing 2.2 shows the same computation being performed using 4-element vectors. The for-loop is still present, but only iterates one-quarter of the number of times as the previous loop. The multiplication occurs by forming 4-vectors from arrays \(B\) and \(C\), and performing an element-wise multiplication, which is then stored into array \(A\). The \(A[i:i+3]\) syntax represents the elements of \(A\) from index \(i\) to \(i+3\), inclusive. Note that this is not valid C syntax and is only meant to show vector pseudo-code.

To use vector instructions in the C and C++ languages, compilers expose *intrinsics* that represent individual vector instructions in the target ISA and are used like standard C functions, but compile down to a specific instruction in the compiled
binary. Special data types are also exposed which represent vector types in the target hardware. The previous example can be coded using SSE intrinsics as shown in Listing 2.3. In this example, the load, multiply, and store intrinsic functions will compile down to single x86 SSE instructions. While intrinsics allow programmers to write high-performance code, the use of intrinsics is platform-dependent. All hardware instruction sets use a different set of intrinsics, so such code must be specially written for each target platform.

Some compilers will attempt to automatically transform user code to make use of vector instructions. Such auto-vectorizing compilers use loop transformations and cost models to determine how and when to transform scalar input code into vectorized binary code. This is a very active area of research.

2.1.2 Performance Implications

Vector instruction sets are often difficult to use efficiently for real-world applications due to (1) limitations in the instruction set, and (2) the potential for performance bottlenecks if not used carefully. Most current instruction sets require vectors to be loaded as contiguous ranges of memory. For example, vector loads are often of the form load %V1, [ADDR] where %V1 is a vector register and [ADDR] is a virtual memory address. Strided access can be particularly troublesome in this regard. Loading strided data into a single vector register on current vector instruction sets requires the compiler to load all needed elements into as many vector registers are needed to satisfy the contiguous access constraint and then use vector shuffles to derive a single vector register that contains all of the needed elements. Loading four elements from memory into a vector register with stride-6 access is shown
in Figure 2.2. In this case, four vector loads and three vector shuffles are needed just to form the vector register. Depending on how the vector is used, the cost of vectorization might outweigh the benefits.

Memory alignment is another issue that may inhibit the performance of vector code. Many instruction sets require vector loads and stores to use addresses which are aligned to the vector width. For 128-bit vectors, this corresponds to a 16-byte alignment. On some vector instruction sets, unaligned loads/stores are not supported and will trigger hardware faults if encountered, while on others unaligned loads/stores are supported but are significantly slower than aligned loads/stores. Compilers must always be conservative in their analyses, so if a vector load/store is not guaranteed to be aligned on architectures where alignment is mandatory, the compiler must emit code to test the alignment at run-time and use a combination of aligned loads and shuffles. If manual intrinsics are being used, it is the job of the programmer to ensure addresses are aligned if needed. Otherwise, seemingly-random application crashes can occur if an address is (erroneously) assumed to be aligned.
2.2 GPU Code Generation

Code generation for GPU hardware is a relatively new field. Hardware shader compilers for OpenGL [21] and Direct3D [40] have only been around for about a decade, and general-purpose compilers have been around for less than that. However, GPUs have been shown to provide major speed-ups for many classes of scientific applications and other programs that require high-through floating-point performance. Compiler technology for GPU hardware is a rapidly evolving field that faces many challenges in the upcoming years as GPU hardware becomes not only increasingly complex, but also ubiquitous in the consumer computer markets.

In this chapter, modern, low-level GPU programming models such as CUDA and OpenCL will be explained. An understanding of these models is extremely important for the automatic analysis and generation of code that will ultimately use these models. For the purposes of this discussion, the CUDA, OpenCL, and DirectCompute programming models will be introduced. These models all share the same general level of abstraction but differ on various implementation details and applicability. For example, the CUDA programming model is only available for NVidia hardware, while the DirectCompute programming model is only available on Microsoft Windows. Both are proprietary, closed-source products. In contrast, OpenCL is an open standard with implementations available for nearly all GPU devices on all platforms.
2.2.1 GPUs: Past, Present, and Future

The first GPUs were fixed-function devices whose sole purpose was to accelerate rasterization, shading, and texture mapping for computer graphics. Such devices functioned primarily as accelerators for floating-point matrix-multiplication, dot-product and scalar operations, which form the basis of the original rasterization and lighting algorithms. Hardware support for z-buffering, stencil testing, and texture sampling were also present. These devices provided a large performance benefit for graphics applications, but lacked any general-purpose programmability.

Eventually, GPUs began to adopt general-purpose programmability to support the wide range of graphics algorithms that were being developed. Hardware vendors could not implement every possible algorithm in hardware, so they developed specialized processors that could execute a limited instruction set that allowed programmers to implement diverse graphics algorithms. At first, GPU programs were limited to vertex and fragment (pixel) processing, but have since been extended to include geometry processing and tesselation. These small programs were often called shaders and were restricted to very specific types of input and output.

With some basic programming models in place, computational scientists began to be able to use graphics hardware for general-purpose computation outside of the world of computer graphics [17]. By phrasing their computations in the form of vertex and pixel transformations, the graphics hardware could be leveraged to greatly accelerate some computations. However, the programming models were severely limited and forced scientists to become intimately familiar with computer graphics algorithms to solve any type of problem.
Soon, graphics hardware manufacturers began to see the need for more general-purpose programmability for their devices. The shader hardware in the graphics chips were already fully programmable scalar/vector floating-point units, and the floating-point throughput for their devices were much greater than general-purpose processors and would likely be very successful at accelerating a wide range of computational science applications. This gave birth to NVidia’s Compute Unified Device Architecture (CUDA) [45] which permitted general-purpose computations to be executed directly on a GPU device without involving the graphics layer. Soon, open standards were developed and adopted for general-purpose GPU computing, such as Khronos’ OpenCL [22]. Microsoft has even jumped on this with their new DirectCompute [40] API.

**Modern GPUs**

Today, GPU devices can be found in nearly every computer market from high-end supercomputer centers to basic smartphones. These devices offer enormous compute power that far surpass CPUs in terms of FLOP/$. However, a trade-off exists when considering the programming models and the types of workloads that can be offloaded to GPU devices. Even with software abstractions like CUDA and OpenCL, writing general-purpose computation code on GPU architectures is a low-level task that makes it impractical for non-experts, including many computational scientists. Furthermore, GPU devices are optimized for high-throughput computing and often fall short of their CPU counterparts in low-latency scenarios. Thus, harnessing the raw computational power of GPUs for a particular application still requires an expert that not only understands the programming models but also
understands how best to balance the workloads between CPU and GPU compute units.

Hybrid Architectures

With GPU devices becoming increasingly commonplace in nearly all computing markets, architects have begun to develop hybrid architectures that tie CPU and GPU components together into a single die. While traditional discrete GPU configurations require the CPU and GPU to communicate through a PCI Express bus and maintain separate memory spaces, some new designs have started to tie both the CPU and GPU into the same memory space and even link CPU and GPU cores together into a common interconnect. Such designs have the potential to greatly increase the speed at which CPUs and GPUs can share data, while making it easier for the two devices to synchronize.

One example of such an architecture is AMD’s Fusion [1] line of APU chips. These chips contain both AMD x86 cores and AMD Radeon GPU shader cores on the same die. The GPU cores access the same global memory space as the CPU, greatly reducing the cost of memory transfers between the two devices. While these devices do not yet offer the same level of GPU performance as their discrete Radeon counterparts, they do offer a glimpse into the future of GPU accelerators.

2.2.2 Programming Models

In this section, we look at the programming models/languages that are available today for general-purpose computing on GPU architectures.
CUDA

The first general-purpose programming model for GPU hardware was NVidia’s Compute Unified Device Architecture (CUDA) [45]. This programming model provides an abstraction that is very close to the actual hardware, where computations are divided into logical blocks of work, each containing multiple threads of execution. These blocks and threads directly map to the GPU device resources.

The GPU device contains multiple Streaming Multiprocessors (SMs), each with multiple Streaming Processors (SPs), as shown in Figure 2.3. Each SP is a scalar processing element that is capable of performing a scalar floating-point multiply-add or integer operation, each LS is a load-store unit capable of dispatching a memory read/write operation, and each SFU is a special-function unit capable of executing complex floating-point operations such as reciprocal square-root. Each SM contains a local shared memory that is program-controlled, however newer architectures have added the ability to use parts of the local shared memory as a device-controlled cache. These two memory spaces are shown as Scratch and Cache in the figure, respectively. In general, a thread executes on an SP and a block of threads concurrently executes on the same SM.

A primary consideration for this programming model is that each SP in a SM must execute the same instruction. Hence, each SM operates in a Single-Instruction Multiple-Thread (SIMT) fashion. It is therefore critical that each thread in the GPU program execute the same instructions whenever possible, otherwise the device resources may become under-utilized and performance will suffer. A consequence of this is that branch divergence comes with a huge performance penalty. If threads in a thread block branch to different instructions in the executing program, they
can no longer execute concurrently on the SM and hence must be split into smaller blocks. This can lead to under-utilization of the device.

A simple example kernel is provided in Figure 2.4. In this example, vector addition is implemented for an N-dimensional vector. In CUDA kernels, the code is written from the perspective of a single thread of execution. For this example, each thread processes one vector element, reading the appropriate value from the two input vectors, and writing the result back to the first vector. The threadIdx vector allows the thread code to identify itself. Similar variables exist for identifying the thread’s block (blockIdx) as well as the block and grid dimensions (blockDim, gridDim).
Listing 2.4 A simple CUDA kernel: \( A = A + B \)

\begin{verbatim}
__global__
void vector_add(float* A, float* B, int N) {
    if (threadIdx.x < N) {
        A[threadIdx.x] += B[threadIdx.x];
    }
}
\end{verbatim}

Listing 2.5 A simple OpenCL kernel: \( A = A + B \)

\begin{verbatim}
__kernel
void vector_add(__global float* A, __global float* B, int N) {
    if (get_local_id(0) < N) {
        A[get_local_id(0)] += B[get_local_id(0)];
    }
}
\end{verbatim}

OpenCL

The OpenCL programming model is fundamentally very similar to the CUDA programming model. The same abstraction is used with the primary difference being kernel syntax and the API. OpenCL is an open standard that has implementations for many graphics devices on many platforms, including Linux, Mac OS X, and Windows. The OpenCL equivalent of the vector addition kernel is shown in Figure 2.5.

DirectCompute

Like OpenCL, the DirectCompute model is very similar to the CUDA model. However, DirectCompute is designed exclusively for the Microsoft Windows platform and integrates very well with the DirectX graphics API, unlike OpenCL. The DirectCompute equivalent of the vector addition kernel is shown in Figure 2.6.
Listing 2.6 A simple DirectCompute kernel: $A = A + B$

```c
RWTexture1D<float> A;
RWTexture1D<float> B;

[numthreads(N,1,1)]
void vector_add(uint3 Tid : SV_ThreadID) {
    if (Tid.x < N) {
        A[Tid.x] += B[Tid.x];
    }
}
```
CHAPTER 3

Dynamic Analysis for SIMD/SIMT Parallelism

Accelerator architectures, such as graphics processing units (GPUs) and SIMD vector units often the potential for high-performance execution for specific classes of problems. For instance, GPU accelerators offer very high floating-point throughput for regular computations but do not perform well with applications that require low-latency memory access or random memory access. Therefore, one important consideration when optimizing code for an accelerator architecture is to first determine if the application would benefit from the properties of the accelerator.

In this chapter, a technique is presented to characterize the performance of programs on Single-Instruction Multiple-Data (SIMD) and Single-Instruction Multiple-Thread (SIMT) architectures. This technique uses the run-time access patterns of a program to determine how well the program could utilize vector compute resources like those found in many SIMD/SIMT architectures. Program instrumentation is used to collect execution traces and an off-line, post-processing step is used to determine the overall vectorizability of a particular execution of a program. The overall architecture of the analysis is provided in Figure 3.1. The primary focus of this technique is SIMD architectures like those found in modern CPUs such
Figure 3.1: Dynamic analysis architecture.

as SSE, Altivec/VSX, and NEON/VFP. However, a future research direction is to extend the presented analysis to SIMT GPU architectures.

3.1 Motivation

Taking advantage of vector instruction sets is becoming a mandatory requirement for any application wishing to achieve a high level of performance on nearly any architecture, from high-end x86/Power systems to smartphones with ARM chips. Vectorizing compilers are able to automatically transform some scalar code and generate efficient vector code for their target architecture, but they still miss many opportunities. Additionally, there are many cases where language semantics interfere with the ability of vectorizing compilers to generate efficient code. A classic example of this is pointer aliasing in the C/C++ languages.

The analysis presented in this chapter provides valuable insight to application developers that identifies portions of code that exhibits data access and computation patterns that can take advantage of vector instructions for higher levels of
Figure 3.2: Sample dynamic data-dependence graphs showing two iterations of a loop. The example in (a) shows a parallel loop while the example in (b) shows a serial loop.

performance. Additionally, such insights are useful for compiler writers to identify code patterns that should be vectorizable but are currently not handled by their compiler. Such insight helps compiler writers to direct their effort to real-world use cases.

Our proposed approach is based on the following key observation: to identify and quantify the vectorization potential of a given program, the dynamic analysis
needs to uncover independent operations that could be executed concurrently. Furthermore, these independent operations should exhibit a pattern of contiguous access to memory locations. The rest of this section provides high-level overview and examples to illustrate these two key issues. The specific details of the approach are elaborated later in the chapter.

3.1.1 Finding Independent Operations

Prior work on using dynamic analysis to characterize potential parallelism in sequential programs falls broadly under one of two general categories. One approach, exemplified by the early work of Kumar [26], performs timestamp-based analysis of instrumented statement-level execution of the sequential program, using shadow variables to maintain last-modify times for each variable. Each runtime instance of a statement is associated with a timestamp that is one greater than the largest of the last-modify times of all input operands of the statement. A histogram of the number of operations at each time value provides a fine-grained parallelism profile of the computation, and the maximal timestamp represents the critical path length for the entire computation.

In contrast to the above fine-grained approach, an alternate technique by Larus [29] performs analysis of loop-level parallelism at different levels of nested loops. Loop-level parallelism is measured by forcing a sequential order of execution of statements within each iteration of a loop being characterized, so that the only available concurrency is across different iterations of that loop.

To illustrate Kumar’s approach, consider the code example in Listing 3.1. For explanation purposes, this example is extremely simple and is used only to highlight
the parallelism characterization from prior work. The run-time instances of the first statement form a chain of dependences of length $N - 1$. The second statement has $N(N - 1)$ run-time instances, with dependences as shown in the figure. The run-time statement instances and their dependences define a dynamic data-dependence graph (DDG), as shown in Fig. 3.3(a). The top row represents instances of statement S1 and the other nodes represent instances of statement S2. For ease of comprehension, a node may be labeled with the array element that is written by that node.

The analysis of potential parallelism computes a timestamp for each DDG node, representing the earliest time this node could be executed; these timestamps are also shown in Fig. 3.3(a). The largest timestamp, compared to the number of nodes, provides a characterization of the inherent fine-grain parallelism in the program; this largest timestamp gives the length of the critical path in the DDG. In essence, the timestamps implicitly model the best parallel execution of all possible dependence-preserving reorderings of the operations performed by the program. In the example from above, the critical path has length $2(N - 1)$, and the overall parallelism is characterized by $(N + 1)/2$, which is the ratio between the number $(N + 1)(N - 1)$ of DDG nodes and the length of the critical path.
All nodes with the same timestamp are independent and can be executed in parallel. However, this method for partitioning of DDG nodes cannot be used to uncover the groups of independent operations needed to characterize the vectorizability of the computation. Consider the example in Listing 3.1. Statement S2 has large vectorization potential: for a particular fixed value of \( j \), all \( N \) run-time statement instances for various values of \( i \) are independent (and, as discussed later, they
Listing 3.2 Example 2 for dynamic parallelism analysis.

```c
for(i = 1; i < N; ++i) {
    A[i] = 2.0 * B[i-1];  // S1
    B[i] = 0.5 * C[i];    // S2
}
```

exhibit a pattern of contiguous access. However, if one were to consider the instances of S2 that are partitioned based on the same timestamp in Fig. 3.3(a), those partitions uncover less parallelism for S2 than what is truly available in the DDG: rather than having \( N - 1 \) partitions of size \( N \), we have a total of \( 2(N - 1) \) partitions. Further, the operations in each partition do not access contiguous memory locations, i.e., are not potentially vectorizable.

As described later, we propose a new form of timestamp computation and critical path analysis that focuses on all instances of a specific statement (e.g., S2 in this example). This analysis considers whether two instances of the statement of interest are connected by a path in the DDG (with any instances of other statements along the path). If such a path exists, our algorithm guarantees that the timestamp of the first node is smaller than the timestamp of the second node (i.e., the two nodes will be placed in different partitions). Furthermore, each node is guaranteed to have the earliest possible timestamp. For the DDG from Fig. 3.3(a), our timestamps are shown in Fig. 3.3(b). Note that all instances of S2 for \( j=1 \) are now given timestamp 1, because they do not depend on any other instances of S2. In general, all instances of S2 for a particular value of \( j \) have the same timestamp and form a partition. As described later, these partitions (containing independent operations) are then subjected to an analysis for contiguous memory access patterns.
Figure 3.4: Dependences for Example 2. Part (a) shows the actual data dependencies for the code, part (b) shows the Larus-style partitioning, and path (c) shows the desired partitioning for vectorization.

The key problem in this example is that the parallelism analysis interleaves the instances of $S_1$ and $S_2$. An alternative approach could be to separately consider the loops in Listing 3.1, and perform loop-level parallelism analysis using an approach based on work by Larus [29]. This technique tracks inter-iteration dependences and computes timestamps for all statement instances in all loop iterations. Inside a loop iteration, the statement instances are executed sequentially. When a statement instance $s$ in loop iteration $i$ depends on a statement instance $s'$ in another iteration, the execution of $i$ stops upon reaching $s$, until $s'$ is executed. With this approach, the second loop in Listing 3.1 will be considered independently, and the analysis will uncover that any two iterations of the $i$ loop at line 4 are independent. In essence, this will create the parallel partitions shown in Fig. 3.3(b).
However, this approach for uncovering loop-level parallelism is also inappropriate for our target goal to discover independent operations that can be vectorized. The code in Listing 3.2 illustrates this point. There is a loop-carried dependence from S2 to S1, as illustrated in Fig. 3.4(a). As a result, the loop-level parallelism identified by the analysis would be of the form shown in Fig. 3.4(b). The resulting partitions do not expose the high vectorization potential of S1 (or S2). However, it is easy to see that the computation can be split into two separate loops: first, a loop that iterates over all instances of S2, followed by another loop that iterates over all instances of S1. The loop-level parallelism analysis can easily uncover that each loop (in this hypothetical transformed version) is fully parallel; in fact, each loop is fully vectorizable. However, since the unit of analysis is the original loop code, the potential for parallelism/vectorization is not discovered. Our approach, when analyzing the DDG in Fig. 3.4(a), will first consider all instances of S1, will discover that they are all independent, and will form a partition containing all of them. Similarly, all instances of S2 will be put in a single partition. The result of our technique is illustrated in Fig. 3.4(c). Comparing with Fig. 3.4(b), it is clear that we uncover more parallelism, which in turn leads to finding more potential vectorization.

To summarize, this technique characterizes the parallelism of an entire loop. However, this characterization is constrained by the order of operations within the loop body that is imposed by the sequential execution of the original program. Thus the potential for increased parallelism via dependence-preserving reordering of operations may be missed. Our approach considers all possible dependence-preserving reorderings of all run-time instances of a specific statement of interest, which exposes the necessary parallelism for the purposes of vectorization.
3.1.2 Finding Operations that Access Contiguously Located Data Elements

Consider again Example 1 from above, and specifically the timestamps shown in Fig. 3.3(b). Each timestamp defines a partition containing $N$ instances of $S_2$ that are independent of each other. Furthermore, all these instances access contiguous regions of memory. For a fixed $j$ and varying $i$, the triples of memory addresses corresponding to the triple of expressions $(B[j][i], B[j - 1][i], A[i])$ exhibit a pattern of contiguous memory access with the row-major data layout used for arrays in C. This makes the statement instances within each partition viable candidates for vectorization.

We propose the first analysis to analyze contiguous memory accesses of independent operations in order to characterize vectorization potential. As described later, the analysis considers all statement instances within a single parallel partition, and defines subpartitions such that within a subpartition, the tuples of memory accesses follow the same pattern of contiguous memory accesses. For example, tuple $(B[j][i], B[j - 1][i], A[i])$ for a fixed $j$ and varying $i$ will produce tuples of run-time addresses of the form $(c_1 + i \times d, c_2 + i \times d, c_3 + i \times d)$. Here $d$ is the size an array element and $c_{1,2,3}$ are base addresses. These tuples represent accesses to contiguous memory, and together form one single subpartition (which covers the entire parallel partition defined by this particular value of $j$). One can imagine all statement instances in the subpartition being combined into a single vector operation $[c_1 : c_1 + (N - 1) \times d] = [c_2 : c_2 + (N - 1) \times d] \oplus [c_3 : c_3 + (N - 1) \times d]$ where $\oplus$ represents a vector operation on vectors of size $N$. Our analysis computes
such subpartitions and uses them to characterize the vectorization potential of the analyzed statement.

3.2 Dynamic Data-Dependence Graphs

The dynamic analysis technique presented in this chapter works on a Dynamic Data-Dependence Graph (DDG) data structure, which captures the instruction-level data dependencies that were observed during a run-time execution of the program. In this graph structure, each node represents an instruction that was executed and each edge represents a data dependency that was observed, either through memory or registers. Two examples of this graph structure are provided in Figure 3.2. In Figure 3.2 (a), the loop is completely parallel, while the loop in Figure 3.2 (b) is serial due to the loop-carried dependency. Red nodes represent memory access instructions and turquoise nodes represent computation instructions, and black edges represent register-based dependencies and red edges represent memory-based dependencies. In the following sections, the properties of and operations on the DDG data structure are presented.

3.2.1 DDG Generation

Generating a dynamic data-dependence graph (DDG) requires an execution trace of the program (or a contiguous subtrace), containing run-time instances of static instructions, including any relevant run-time data such as memory addresses for loads/stores, procedure calls, etc. Our implementation uses LLVM [30] to instrument arbitrary C/C++/Fortran code. The Clang [12] front-end is used to compile C/C++ code into LLVM IR, and the DragonEgg [16] GCC plugin is used to
compile Fortran 77/90 code into LLVM IR. The LLVM IR is instrumented to generate a run-time trace to disk, and the instrumented code is compiled to native code.

Once an execution trace is available, the construction of the DDG creates a graph node for each dynamic instruction instance. Edges are created between pairs of dependent nodes (i.e., one instruction instance consumes a value produced by the other). In our implementation each graph node represents a dynamic instance of an LLVM IR instruction, and dependences are tracked through memory and LLVM virtual registers. To construct the graph edges, bookkeeping information for each memory/register remembers the graph node that performed the last write to this location. Note that the graph represents only flow dependences. Anti-dependences and output dependences are not considered, since they do not represent essential features of the computation, and could potentially be eliminated via transformations such as scalar/array expansion. Control dependences are also not considered, since our goal is to focus on the data flow and the optimization potential implied by it. It is straightforward to augment the DDG with additional categories of dependences, without having to modify in any way the subsequent graph analyses (described below).

One interesting case that arises in practice is due to reductions, for example, because of a statement \( s += a[i] \) in an \( i \)-loop. The instances of such a statement would form a chain of dependences in the DDG. However, sometimes it is possible to vectorize reductions (e.g., by updating a vector \( sv \) instead of a scalar \( s \)). Our analysis currently does not consider the potential for such vectorization. A possible enhancement is to identify and remove dependence edges that are due to updates of
reduction variables; detection of such dependences has already been used by prior work [51] in a different context.

3.2.2 Candidate Instructions

The execution trace may contain many instructions that should not be analyzed for SIMD parallelism, such as integer operations for loop book-keeping. Hence, the analysis is restricted to instructions that involve floating-point addition, subtraction, multiplication, and division. These instructions correspond to the set of floating-point instructions that have vector counterparts in SIMD architectures. They are also of particular importance for optimization of certain computationally-intensive applications (as exemplified by the SPEC floating-point benchmarks). Of course, all other instructions that participate in dependences are taken into account by the analysis, but their potential SIMD parallelism is not characterized.

3.2.3 Generation of Parallel Partitions

To benefit from SIMD parallelism, an instruction must exhibit fine-grained parallelism. For a static instruction $s$ that is being characterized, the potential parallelism of $\{s_1, s_2, \ldots\}$ (where $s_k$ is the $k$-th run-time instance of $s$) can be uncovered by observing the data dependences from any $s_i$ to any $s_j$ for $j > i$. This is equivalent to identifying whether the DDG contains a path from the node for $s_i$ to the node for $s_j$. Such a path exists if and only if the data produced by $s_i$ is directly or indirectly used by $s_j$ (i.e., $s_i$ and $s_j$ cannot be executed concurrently). The converse of this observation also holds: if two dynamic instances cannot be executed concurrently, then there must exist a path between them.
Each candidate static instruction $s$ (as described earlier) is analyzed independently, using Algorithm 3.1. A unique ID for $s$ is assigned at instrumentation time. A topological sort traversal of the DDG is performed and a timestamp is assigned to each node. For each visited node, the largest predecessor timestamp is determined. If the node is an instance of the static instruction $s$ being analyzed, the timestamp is incremented by one; otherwise, it is not altered.

The first key property of the algorithm is that if there exists a run-time dependence from $s_i$ to $s_j$ (either directly, or indirectly through instances of instructions other than $s$, or through other instances of $s$ itself), then the timestamp of $s_i$ is strictly smaller than the timestamp of $s_j$. The second property is that each $s_i$ is assigned the smallest possible timestamp that satisfies the first property.

The generated timestamps are then used to construct partitions within the graph, by putting all instances of $s$ with the same timestamp into the same partition. Consider the DDG shown in Fig. 3.3(a). The nodes at the top of the DDG represent the
run-time instances of S1, while the rest of the DDG nodes are instances of S2. Suppose we wanted to evaluate the potential vectorizability of S2. For this static instruction, the analysis will compute timestamps for S2 instances as shown in Fig. 3.3(b). Each timestamp value defines one partition of S2 instances.

**Property 1.** Consider any DDG node $s_i$ and a DDG path $p$ ending at $s_i$. Let $s(p)$ be the number of nodes on $p$ (excluding $s_i$) that are instances of the static instruction $s$ being analyzed. The timestamp computed for $s_i$ by Algorithm 3.1 is the largest value of $s(p)$ for all $p$ leading to $s_i$.

The proof is by induction on the length of $p$. This property has two implications. First, consider two instances $s_i$ and $s_j$ of $s$. If there exists a run-time dependence from $s_i$ to $s_j$ (either directly, or indirectly through instances of instructions other than $s$, or through other instances of $s$ itself), then the timestamp of $s_i$ is strictly smaller than the timestamp of $s_j$. Thus, all instances of $s$ with the same timestamp (i.e., in the same partition) are independent of each other. Second, each $s_i$ is assigned the smallest possible timestamp—that is, it is scheduled at the earliest possible time. Considering the average partition size as a metric of available parallelism for the instances of $s$, the following property can be proven easily.

**Property 2.** Algorithm 3.1 finds the maximum available parallelism for each static instruction $s$.

### 3.2.4 Partitioning for Contiguous Access

Algorithm 3.1 ensures that the instances of $s$ within a partition are independent, but efficient SIMD parallelism also requires contiguous memory access. On most
SIMD architectures, the cost of loading vector elements individually and shuffling them into a vector register offsets the benefit of exploiting the vector hardware.

Thus, the partitions must be further subdivided into units that exhibit parallelism and contiguous memory access. In general, we want to ensure unit-stride accesses—the distance in memory between consecutive memory accesses is equal to the size of the data type. We also allow zero-stride (i.e., the distance in memory between consecutive accesses is zero), since vector splats (copying a scalar value into all elements of a vector) are cheap for most SIMD architectures. Note that the zero-stride case also covers operations with constant operands.

To ensure unit-stride, the instruction instances within a parallel partition are sorted according to the memory addresses of their operands. For constants or values produced by other instructions but not saved to memory, an artificial address of zero is used. The sorted list is then scanned, ending the current subpartition (and starting a new one) when the stride is (1) non-zero and non-unit, or (2) different from the previously observed stride. The result is a (now potentially larger) set of subpartitions such that the dynamic instruction instances within a subpartition are independent, and exhibit uniform zero-stride or unit-stride accesses to memory. The average size of these subpartitions is a metric of the vectorization potential of the static instruction being characterized.

3.2.5 Non-Unit Constant Stride Access

The contiguous access check performed in the previous stage is important for discovering efficient vectorization potential, but a variation of the check can be used to explore the potential benefit of data layout transformations on the original code.
Listing 3.3 Vectorization benefits of data layout transformations: stride-N column access and array-of-structures access.

```
for ( i = 0 ; i < N ; i++ )
    for ( j = 2 ; j < N ; j++ )

for ( i = 0 ; i < N ; i++ ) {
    C[i].x = B[i].x + B[i].y; // S2
    C[i].y = B[i].x - B[i].y; // S3
}
```

It is not uncommon to find computations where fine-grained concurrency exists but the data accessed has a non-unit but constant stride. In the first loop in Listing 3.3 there is a loop-carried dependence along the inner \( j \) loop but the \( i \) loop is parallel. If the loops were permuted, we would have fine-grained parallelism in the inner loop for the instances of S1, but the access stride would be \( N \). A data layout transformation to transpose the array (i.e., swap the first and second array dimensions) would enable unit-stride access and efficient vectorization. Listing 3.4 shows how the code could be transformed.

The second loop in Listing 3.3 illustrates a scenario with arrays of structures that results in fine-grained concurrent operations but non-unit access stride. The instances of S2 are all independent but they exhibit stride-2 access (e.g., 8 bytes if \( x \) and \( y \) are single-precision floating-point). The same is true for the instances of S3. In this case, changing the data structure from an array of structures to a structure of arrays would enable parallel, stride-1 access which could be automatically vectorized.

By relaxing the unit/zero-stride condition to instead check for any non-unit constant stride, we can detect cases such as the ones illustrated in Listing 3.3. Given the partitions produced by Algorithm 3.1, we apply the unit-stride analysis from
the previous subsection. At the end, any instruction instance that belongs to a sub-
partition of size one is identified. All such instances (of the same static instruction,
and with the same timestamp) are then sorted and scanned. When the currently
observed stride does not match the previously observed one, the instruction is put
on a waitlist for future processing, and the scanning based on the current stride
continues until the end of the list is reached; this results in one subpartition. Any
waitlisted instructions are then traversed again, in sorted order, so that the next
subpartition can be formed.

### 3.3 Experimental Evaluation

In this section we present a number of studies to illustrate the use of the dy-
namic analysis tool. Although the studies are restricted to analysis of sequen-
tial programs, the tool can also be used with parallel programs using Pthreads,
OpenMP, MPI, etc.—the instrumentation and trace generation would be applied to
one or more sequential processes or threads of the parallel program to assess the
potential for SIMD vector parallelism within a process/thread. Further, although
we only concentrate on characterizing floating-point operations (because they tend

**Listing 3.4** Loop transformations and data layout transformations applied to Listing 3.3

```c
// transposed declarations for A, B, and C
for ( j = 2 ; j < N ; j++ )
    for ( i = 0 ; i < N ; i++ )

for ( i = 0 ; i < N ; i++ ) {
    C.x[i] = B.x[i] + B.y[i]; // S2
    C.y[i] = B.x[i] - B.y[i]; // S3
}
```
to be the focus of most SIMD optimization efforts), such analysis can be carried out for any type of operations, e.g., integer arithmetic.

The experiments were performed on a machine with an Intel Xeon E5630 processor and 12GB memory, running Linux 2.6.32. To obtain performance measurements, the Intel icc compiler (12.1.3) was used to compile the program code, at the O3 optimization level. Profiling data was obtained with HPCToolkit [20] version 5.2.1, at sampling period 500 thousand cycles. The instrumentation infrastructure was implemented in LLVM 3.0.

### 3.3.1 Characterization of Applications

We illustrate the use of the tool for characterizing software collections by applying it to the SPEC CFP2006 floating-point benchmarks, and kernels from the UTDSP benchmark suite [65]. We also include two stand-alone compute kernels: a 2-D Gauss-Seidel stencil code and a kernel from a 2-D PDE grid-based solver; they are elaborated upon later as case studies illustrating potential use of the analysis for performance optimization and compiler enhancement.

The stand-alone kernels and UTDSP benchmarks were directly analyzed by the tool and the results are shown in Tables 3.2 and 3.3. For full application codes, such as the SPEC CFP2006 floating-point benchmarks, the output produced by the dynamic analysis will be very extensive. Therefore, we only analyze and report characteristics for time-consuming loops identified via profiling by HPCToolkit, analyzing only loops that account for at least 10% of total execution cycles during a run of the benchmark using the SPEC reference data set (the 10% threshold was selected to reduce the amount of data presented; we also collected data at a threshold...
of 5%). To perform the DDG analysis for a particular loop, we collected several sub-traces corresponding to separate instances of the loop (using the SPEC train data set, and for a few large loops the test data set). A subtrace was started upon loop entry and terminated upon loop exit and its DDG was constructed and analyzed. We randomly chose several instances of the loop, analyzed each corresponding sub-trace to obtain the various metrics described later, and chose one representative subtrace to be included in the measurements presented in the paper.

The results of the analysis for SPEC CFP2006 are shown in Table 3.1. For each benchmark, we only show loops that account for at least 10% of total execution cycles. We start with all innermost loops, and only include a parent loop if the total percentage of execution cycles spent in it is at least 10 percentage points greater than the sum of the percentages for its inner loops. The games benchmark could not be compiled with LLVM and was not used in the experiments.

The Percent Packed metric shows the percentage of floating-point run-time operations that were executed using packed (i.e., vector) SSE instructions, as reported by HPCToolkit. This column provides information on the effectiveness of current compilers (we used Intel icc since we have found it to be superior to other production compilers in vectorization capability) in vectorizing each of the identified hot loops. A high value indicates that a significant fraction of the floating-point operations in that loop were executed via packed SIMD instructions. A zero value indicates that the compiler was unable to achieve any vectorization at all for the loop.

The Average Concurrency metric was computed by determining the average partition size across the collection of partitions for all floating-point instructions in the
graph, where partitions were formed by considering only instruction independence (Sect. 3.2.3). For this metric, both singleton and non-singleton partitions were considered. From the non-singleton parallel partitions, subpartitions containing unit-stride operations were formed (as described in Sect. 3.2.4). The Percent Vec. Ops metric (i.e., potentially vectorizable run-time instructions) shows the number of operations that belong to non-singleton unit-stride subpartitions, as percent of the total number of operations in the graph. The Average Vec. Size metric represents the average size of these non-singleton vectorizable subpartitions. The general trend is that for most of the loops, many run-time instructions belong to partitions that exhibit both independence and contiguous memory access. Furthermore, the sizes of these partitions are large—in many cases, much larger than the vector sizes in existing and emerging architectures. Thus, the analysis indicates that the majority of the analyzed loops may have high vectorization potential.

The run-time instructions within a non-singleton parallel partition that did not belong in any unit-stride subpartition were further analyzed with the non-unit stride analysis described in Section 3.2.5. The analysis reported the number of such instructions that could be placed in subpartitions accessing data at some fixed non-unit stride. This number, as percent of the total number of all run-time instructions in the graph, is shown in column Percent Vec. Ops. The average size of such subpartitions is given in the last column. There are several examples where a significant number of independent instructions can be combined together using non-unit stride, and the sizes of the partitions are large as well. This indicates that data layout transformations may be beneficial in these cases.
There may be cases where the percentage of packed instructions observed via profiling exceeds the sum of the values in columns Percent Vec. Ops (e.g., utilitiesDV.c:1241 in 454.calculix and vector.c:521 in 482.sphinx3). This happens in the presence of a reduction (e.g., s+=expr): our analysis considers the chain of dependences and treats the computation as non-vectorizable. However, there exist approaches to vectorize reductions, and icc employs some of them. In future work, our approach could be extended to ignore dependences due to reductions, which would uncover these additional vectorization opportunities.

As is typical of other work on dynamic analysis of fine-grained dependences (e.g., [29, 72]), the instrumentation incurs an overhead of two to three orders of magnitude, relative to the execution time of the original unmodified code. The cost of DDG analysis depends on graph size and memory access patterns, and is typically of the order of tens to hundreds of microseconds per DDG node. Although we have not focused on tool optimization, the utility of the current unoptimized implementation of our prototype is not hampered for two reasons. First, the analysis is intended to be performed offline, e.g., during performance tuning. Many profiling analyses have been successfully used in this setting, and various existing techniques can be readily applied to reduce their cost (e.g., [72, 74]). Second, the instrumented code can be run with much smaller problem sizes than the “production” size: in our experience, although metrics such as average vector size can vary with problem size, the qualitative insights about potential vectorizability do not change.
3.3.2 Assisting Vectorization Experts

Many institutions possess large code bases that were largely developed before the recent emergence of SIMD parallelism in all CPUs/GPUs. When the original developers of the code are not available to adapt it for improved vectorization, an automated tool can be very valuable. For some loops, a quick scan of the code of a hot loop by a vectorization expert will immediately reveal the opportunities for enhancing vectorizability through code changes. But this is certainly not the norm, especially with C++ codes or C codes that make heavy use of pointers. An automated tool allows the vectorization expert to quickly eliminate loops with little to no vectorization potential, and concentrate on the loops with high potential. With these, some of the code structures involve multiple levels of function calls and the output from the tool is valuable input to the expert, indicating that the effort to unravel the code is likely worth it. As an example, the hot loops in 444.namd are generated using C preprocessor macros and it is very difficult to get an understanding of the code just by scanning it. If we examine the HPCToolkit profile data, we know the loops are hot, but not whether or not we have any hope of vectorizing them. However, our analysis shows that there is a high potential for vectorization in this part of code, so it may be worth the time investment of a vectorization expert to carefully analyze these loops.

Another use case is for identifying missed opportunities in compiler test suites. Vendor compilers are typically tested against large amounts of code to gauge the performance of the compiler’s vectorizer. It is easy to automate this testing to see how much of the code is vectorized, but for the remaining code, it is not clear
whether the code is just not vectorizable, or if the compiler is missing an opportunity. It would take considerable effort for a vectorization expert to manually analyze all of the non-vectorized code. The analysis tool can help to automate the process and focus the expert’s effort on identifying why code that has been identified as being potentially vectorizable is not actually being vectorized by the compiler.

### 3.3.3 Array-Based vs. Pointer-Based Code

Auto-vectorizing compilers are becoming increasingly good at vectorizing array-based code, but pointer-based code is often not vectorized due to the added complexities with pointer aliasing and the verification of contiguous access during the compiler’s static analysis. A primary benefit of the proposed dynamic analysis technique is the ability to analyze pointer based code just as easily as array based code. Both versions of the same computation will provide the same analysis results, since the dynamic analysis considers IR-level arithmetic operations, and does not make a distinction between data that is read from arrays or pointer dereferencing.

To test this facet of our analysis, we used the UTDSP [65] benchmark suite, which contains both array- and pointer-based versions of several computation kernels for digital signal processors (DSP). The suite was created to evaluate the quality of code generated by a high-level language compiler (e.g., a C compiler) targeting a programmable DSP. Thus, each kernel was written in different styles, including an array-based version and a pointer-based version. Both versions provide identical functionality, except for the use of arrays or pointers to traverse the data structures.

Table 3.3 shows the results of this experiment. The measurements include the percentage of vectorizable operations found in the program, the average vector size,
and the percentage of operations that are actually vectorized by the Intel icc compiler. We see that our analysis is invariant to the form of the code\textsuperscript{1}, but icc fails to vectorize some of the pointer-based code. Such knowledge would be very useful in optimizing certain applications, where a conversion from pointer-based code to array-based code may be worthwhile if the potential benefits are high. The dynamic analysis from our tool could be a valuable first step in the process.

### 3.3.4 Case Studies

Based on the results from the previous subsections, some benchmarks were manually transformed to enable vectorization by icc. The targeted benchmarks were the Gauss-Seidel stencil, the PDE grid solver, and the 410.bwaves, 433.milc, and 435.gromacs benchmarks from SPEC CFP2006. A comparison of the performance of the original and modified versions is shown in Table 3.4. In addition to the Intel Xeon E5630 machine used for the measurements presented earlier, two other machines were used in these experiments: an Intel Core i7 2600K and an AMD PhenomII 1100T, both with the same icc configuration. For each benchmark, we show the total execution time for both versions, as well as the achieved speedup. When the target of the optimization is a particular loop (e.g., bwaves and gromacs), the measurements in the table are based on the total time spent in the loop. The reference data sets were used when running the SPEC benchmarks.

**Gauss-Seidel** In this case study we analyze the vectorization potential of a 9-point Gauss-Seidel stencil code. This code has been identified by our analysis as being not

\textsuperscript{1}The discrepancy in LMSFIR is due to a difference in the way the two versions are written, resulting in slightly different distributions of operations.
auto-vectorized by the vendor compiler, but possessing non-trivial vectorization potential (see Table 3.2). Listing 3.5 shows the original kernel. It has a loop-carried dependence in the innermost \( j \) loop, since the fourth operand \( A[i][j-1] \) is produced in the previous iteration of this loop. Similarly, the outer \( i \) loop also has loop-carried dependences. Due to the dependences, icc was unable to vectorize the code. This was not unexpected. However, what surprised us was that the dynamic analysis revealed vectorization potential for this code.

The analysis classified two out of the eight addition operations \( (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]) \) as vectorizable. This is because the operands were all produced in the previous iteration of the \( i \) loop. The only true dependence in the loop is due to \( A[i][j-1] \). The operations involving elements from row \( i+1 \), and even the
addition of $A[i][j]$ and $A[i][j+1]$, could be performed in vectorized mode by splitting the loop into a sequence of two loops, as shown in Listing 3.5. The first $\text{\textit{j}}$ loop in the transformed code is now completely vectorized by icc, resulting in significant performance improvement (see the results in Table 3.4, obtained for $N = 1000$ and $T = 20$).

The extent of vectorizability of the Gauss-Seidel code was a surprise to us since our initial expectation was that the code would not exhibit vectorization potential due to loop-carried dependences. Closer examination of the dependences shows that all the information needed to transform the code is actually derivable from purely static analysis. However, to our knowledge, no research or production compiler can perform the transformation we performed manually. This example illustrates how the developed dynamic analysis can be valuable for compiler writers to identify scenarios where enhancements to static analysis and transformation capabilities can enable improved code transformations for vectorization.

**2-D PDE Solver** In this case study we analyze the core computation from a 2-D PDE grid-based solver. The code is from the examples included with PETSc [48] 3.1-p7 and solves the solid fuel ignition problem, modeled as a partial differential equation. The original source can be found under `/src/snes/examples/tutorials/ex5.c` in the PETSc distribution. We see that this code is not auto-vectorized by icc, but our analysis shows very high vectorization potential. In this kernel, the 2-D computation grid is distributed onto a 2-D grid of blocks, where the computation is performed by iterating over every cell within every block. For our purposes, we consider only sequential execution of the program.
The per-block kernel code is shown in Listing 3.6. The if condition in the innermost loop is a boundary condition check that forces grid points on the boundary to follow a different path of execution as compared to the other interior points. The loop bounds for this loop nest, along with two of the four conditions that can trigger the if statement, are data dependent. As a result, compilers are forced to be conservative and assume that for each iteration of the loop, it is unknown whether the then or else clause will be executed. Due to this constraint, the vectorizability of this particular loop nest, as written, is very low. Further, without more constraints
on the values within the info structure, static analysis cannot determine transformations that would enable vectorization.

However, the results of the dynamic analysis show a great potential for vectorizability within this code (see Table 3.2). Specifically, the else clause exhibits perfect vectorizability. To allow a compiler to vectorize this loop, we can rewrite the code to extract the if/then/else construct and then hoist an if to provide a vectorizable loop. The modified code is shown in Listing 3.6. The key to the vectorization-enabling transformation is the observation that cells which correspond to the boundary condition can only occur on boundary blocks. We observe that i and j cannot be zero except within blocks along the top or left edge of the grid, and cannot be equal to the maximum index value (mx-1 and my-1) except within blocks along the right or bottom edge of the grid. Therefore, the kernel code can be split into two separate versions; one for boundary blocks and one for interior blocks, as shown in Listing 3.6. While this code should provide no speed-up for boundary blocks, it enables vectorization for interior blocks and provides an advantage when the blocks are in at least a 3 × 3 grid.

Table 3.4 shows the total execution time for the original and modified code for a case where the block size is 512 × 512 and blocks are in a grid of size 16 × 16. As shown in the table, the performance improvement is substantial.

410.bwaves In the 410.bwaves benchmark, one of the loops we analyzed in our extended study (at 5% threshold for hot loops) is at jacobi_lam.f:30. This loop exhibits a very low percentage of packed instructions, while the dynamic analysis shows that there are significant numbers of vectorizable operations at unit and non-unit
stride. The result from the non-unit stride analysis suggests possible improvements through data layout transformation.

Listing 3.7 is representative of the computation within this loop. Arrays $je$ and $jv$ are of size $(5,5,nx,ny,nz)$, and array $q$ is of size $(5,nx,ny,nz)$ where $nx$, $ny$ and $nz$ are program constants. While there is no dependence between the iterations of the innermost loop $i$, there are two factors that hinder vectorization. First, there is no unit-stride data access pattern since $i$ is used to access the third dimension of the arrays. Second, the use of $mod$ operations to calculate the neighbor with wrap-around boundary conditions hampers the vectorization of accesses to array $q$.

To address these problems, a data layout transformation was performed on arrays $je$, $jv$, and $q$: the dimension which was originally accessed by $i$ (and $ip1$) was moved to become the fastest varying dimension of the arrays. This transformation is shown in Listing 3.7. The modified code has a stride-1 data access pattern. The $mod$ operations were removed by peeling the last iteration of the $i$ loop, and introducing $ip1=i+1$ within the loop. The value of $ip1$ for the peeled iteration was set to 1. Table 3.4 shows the performance of the original and modified versions.

433.milc In this case study we explore the benefits of data layout transformations on the 433.milc benchmark. Specifically, we focus on one of the loops from Table 3.1. The loop starting at quark_stuff.c:1452 shows no automatic vectorization by the compiler. There is also limited potential for vectorization at unit stride. However, the non-unit stride analysis shows significant potential for vectorization, implying that a data layout transformation may speed up the computation.
This loop iterates over every point in a lattice, applying a matrix-vector multiplication operation at each point. The matrices are of size $3 \times 3$ and the vectors are of size 3, both containing complex numbers. The lattice itself holds a matrix at each point, and external arrays of vectors are used for the vector inputs and outputs. The matrix-vector multiplication is not vectorized by the compiler due to non-unit stride access (distribution of real/imaginary components) and a small inner loop trip count (3).

To help us isolate the required changes, we created a version of the benchmark that only contains the computation we identified. The full benchmark was too large to optimize manually without in-depth understanding of the entire application, and the smaller, kernel-ized version allowed us to show proof-of-concept benefits of a data layout transformation. To optimize this operation, the transformation was applied to the lattice data structure, where the lattice of matrices was converted to a matrix of lattices. This modification exposes unit-stride operations within the inner loop. The original and transformed code are shown in Listing 3.8. Table 3.4 demonstrates a significant speedup for the modified kernel.

**435.gromacs** For this case study we focus on the loop at line 3960 in `inner.f` from 435.gromacs. While icc is not able to vectorize this loop in any significant manner, the dynamic analysis results indicate the existence of unit-stride operations. Lines 1–14 of Listing 3.9 represent the computation within the loop.

The value of $j3$ is data dependent (based on the run-time values in indirection array $jjnr$) and is used to index into arrays $pos$ and $faction$. The compiler has to assume that the loop is not parallel, in case two or more elements of $jjnr$ have the
same value and thus create a dependence through $\text{faction}(j3)$. Furthermore, the access patterns for $\text{pos}$ and $\text{faction}$ are not regular due to the arbitrary values of $j3$. Thus, the loop is not vectorized by icc.

When the dynamic analysis results were examined at the level of individual statements, it became clear that the loop is in fact parallel: the values in $\text{jjnr}$ ensure that distinct elements of $\text{faction}$ are accessed by each iteration. (The relatively low average concurrency in Table 3.1 is due to the small number of loop iterations and to a few chains of reductions.) Furthermore, although the accesses to $\text{pos}$ and $\text{faction}$ do not exhibit any patterns, the rest of the computation in the loop body is done through scalars and can be easily vectorized.

For better vectorization, the loop was strip-mined as shown in Lines 15–38 of Listing 3.9. (The cleanup loop is not shown.) Loop distribution of the inner $k_{\text{vect}}$ loop was then applied to move all reads from $\text{pos}$ and $\text{faction}$ above the computation, and to move all writes to $\text{faction}$ after the computation. Array expansion of temporary $j3$ was also performed to hold the necessary indices of $\text{faction}$. The middle $k_{\text{vect}}$ loop is now vectorized by icc. Table 3.4 shows the reduction in the total time spent in the loop due to the transformation.

Unlike in the earlier case studies, here the analysis results do not necessarily generalize to arbitrary input data. Specifically, the fact that all iterations of the loop were independent affects both the vectorization potential and the code modifications. As written, the transformed code in Listing 3.9 is not correct for all possible inputs. It becomes necessary to assert this correctness with the help of additional
information—e.g., an expert’s knowledge of certain properties of the problem domain, or some compiler analysis of the intra- and inter-procedural code context surrounding the loop.

**Limitations** Although these case studies demonstrate the usefulness of the analysis reports, the proposed technique has a number of limitations. For example, for the loop from 435.gromacs, the conclusions about vectorizability are dependent on properties of the input data. As another example, we investigated loop bbox.cpp:894 from 453.povray in greater depth. This benchmark is a ray-tracer, and the loop in question implements a worklist algorithm that intersects a ray with a tree of bounding boxes. The computation is driven by a priority queue. Each iteration of the loop removes a bounding box from the queue and, if necessary, adds other bounding boxes to the queue. Inside an iteration, what processing is performed on the current bounding box depends on whether it is an inner node or a leaf of the tree, and whether the ray intersects the boxes of its children. The overall structure of the computation is very irregular and heavily depends on the actual run-time data. As part of the intersection tests for boxes and the scene objects contained in them, some low-level operations (e.g., computing the angle between two vectors) occur repeatedly, with high concurrency and with some potential vectorizability for certain subcomputations. However, the highly-irregular structure of the control flow makes it extremely challenging to exploit the vectorization potential without significant changes to the code by a domain expert with a deep understanding of the algorithm.
An interesting direction for future work is to refine the dynamic analysis to distinguish computations with irregular data-dependent control flow from ones where the control flow is more structured and vectorization potential is more likely to be actually realizable through code transformations.

3.4 Conclusion

In this chapter, we have introduced a dynamic analysis technique for assessing the vectorizability of applications. The output of the analysis is a profile that shows the portions of code which would benefit from manual or semi-automatic code transformations for vectorization. Such an analysis is useful both for developers to identify where to focus their efforts, and to compiler developers for identifying code segments that would benefit from vectorization but are not currently being auto-vectorized. We also present an experimental evaluation of the usefulness of the analysis, including case studies that show significant speedup from manual optimization of analysis-identified code segments.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Loop</th>
<th>Percent Cycles</th>
<th>Percent Packed</th>
<th>Average Concur.</th>
<th>Unit Stride</th>
<th>Non-unit Stride</th>
</tr>
</thead>
<tbody>
<tr>
<td>410.bwaves</td>
<td>block_solver.f : 55</td>
<td>79.2%</td>
<td>50.0%</td>
<td>7.0%</td>
<td>8.6%</td>
<td>2.0%</td>
</tr>
<tr>
<td></td>
<td>block_solver.f : 176</td>
<td>65.8%</td>
<td>66.4%</td>
<td>10.0%</td>
<td>5.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>gauge_stuff.c : 258</td>
<td>22.0%</td>
<td>0.0%</td>
<td>10453.4</td>
<td>36.2%</td>
<td>10427.4</td>
</tr>
<tr>
<td></td>
<td>path_product.c : 49</td>
<td>17.9%</td>
<td>0.0%</td>
<td>73316.6</td>
<td>36.4%</td>
<td>69441.5</td>
</tr>
<tr>
<td></td>
<td>quarantine.c : 666</td>
<td>15.2%</td>
<td>0.0%</td>
<td>23687.7</td>
<td>88.3%</td>
<td>11.4%</td>
</tr>
<tr>
<td></td>
<td>quarantine.c : 968</td>
<td>44.9%</td>
<td>0.0%</td>
<td>11447.3</td>
<td>65.1%</td>
<td>15.5%</td>
</tr>
<tr>
<td></td>
<td>quarantine.c : 973</td>
<td>35.0%</td>
<td>0.0%</td>
<td>61566.7</td>
<td>57.4%</td>
<td>13.8%</td>
</tr>
<tr>
<td></td>
<td>quarantine.c : 1452</td>
<td>14.2%</td>
<td>0.0%</td>
<td>20736.0</td>
<td>36.4%</td>
<td>20736.0</td>
</tr>
<tr>
<td></td>
<td>quarantine.c : 1468</td>
<td>13.6%</td>
<td>0.0%</td>
<td>20736.0</td>
<td>36.4%</td>
<td>20736.0</td>
</tr>
<tr>
<td></td>
<td>quarantine.c : 1523</td>
<td>15.4%</td>
<td>0.0%</td>
<td>2921.1</td>
<td>55.0%</td>
<td>2000.0</td>
</tr>
<tr>
<td>433.milc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>434.zeusmp</td>
<td>advx3.f : 437</td>
<td>13.1%</td>
<td>35.0%</td>
<td>66613.9</td>
<td>74.3%</td>
<td>442.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16.6%</td>
</tr>
<tr>
<td>435.gromacs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>436.cactusADM</td>
<td>Staggered.eapFrog2.F : 342</td>
<td>18.4%</td>
<td>100.0%</td>
<td>80.0%</td>
<td>100.0%</td>
<td>80.0%</td>
</tr>
<tr>
<td></td>
<td>Staggered.eapFrog2.F : 366</td>
<td>81.1%</td>
<td>96.9%</td>
<td>78.0%</td>
<td>100.0%</td>
<td>78.0%</td>
</tr>
<tr>
<td>437.leslie3d</td>
<td>tnl.F : 522</td>
<td>15.6%</td>
<td>98.5%</td>
<td>8805.3</td>
<td>100.0%</td>
<td>136.3</td>
</tr>
<tr>
<td></td>
<td>tnl.F : 889</td>
<td>13.4%</td>
<td>99.2%</td>
<td>7434.2</td>
<td>99.9%</td>
<td>178.4</td>
</tr>
<tr>
<td></td>
<td>tnl.F : 1269</td>
<td>12.4%</td>
<td>99.2%</td>
<td>438.3</td>
<td>22.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td>tnl.F : 3569</td>
<td>21.6%</td>
<td>98.6%</td>
<td>8100.0</td>
<td>100.0%</td>
<td>90.0%</td>
</tr>
<tr>
<td>444.namd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>447.dealII</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>450.soplex</td>
<td>sxvector.cc : 983</td>
<td>10.6%</td>
<td>0.0%</td>
<td>373.0</td>
<td>32.2%</td>
<td>18.5%</td>
</tr>
<tr>
<td></td>
<td>sxvector.cc : 983</td>
<td>10.6%</td>
<td>0.0%</td>
<td>373.0</td>
<td>32.2%</td>
<td>18.5%</td>
</tr>
<tr>
<td></td>
<td>sxvector.cc : 983</td>
<td>10.6%</td>
<td>0.0%</td>
<td>373.0</td>
<td>32.2%</td>
<td>18.5%</td>
</tr>
<tr>
<td></td>
<td>sxvector.cc : 983</td>
<td>10.6%</td>
<td>0.0%</td>
<td>373.0</td>
<td>32.2%</td>
<td>18.5%</td>
</tr>
<tr>
<td>453.povray</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>454.calculix</td>
<td>Chv_update.c : 736</td>
<td>13.6%</td>
<td>91.5%</td>
<td>27.4%</td>
<td>48.4%</td>
<td>15.0%</td>
</tr>
<tr>
<td></td>
<td>e_cld.f : 675</td>
<td>69.7%</td>
<td>0.1%</td>
<td>35.6%</td>
<td>100.0%</td>
<td>12.3%</td>
</tr>
<tr>
<td></td>
<td>Frontem_update.c : 267</td>
<td>16.4%</td>
<td>91.5%</td>
<td>774.0</td>
<td>96.4%</td>
<td>28.6%</td>
</tr>
<tr>
<td></td>
<td>Frontem_update.c : 38</td>
<td>14.0%</td>
<td>91.2%</td>
<td>1166.3</td>
<td>96.7%</td>
<td>12.9%</td>
</tr>
<tr>
<td></td>
<td>nautilus.f : 144</td>
<td>74.7%</td>
<td>0.4%</td>
<td>6068.4</td>
<td>99.2%</td>
<td>136.9</td>
</tr>
<tr>
<td></td>
<td>utilities Mvc : 124</td>
<td>11.4%</td>
<td>96.6%</td>
<td>2.0</td>
<td>50.0%</td>
<td>49.0%</td>
</tr>
<tr>
<td>459.GemFDTD</td>
<td>HFT.F : 1068</td>
<td>17.4%</td>
<td>0.0%</td>
<td>24.2%</td>
<td>69.9%</td>
<td>9.9%</td>
</tr>
<tr>
<td></td>
<td>update.F : 188</td>
<td>17.3%</td>
<td>97.4%</td>
<td>201.0%</td>
<td>100.0%</td>
<td>201.0%</td>
</tr>
<tr>
<td></td>
<td>update.F : 242</td>
<td>17.1%</td>
<td>97.3%</td>
<td>200.0%</td>
<td>100.0%</td>
<td>200.0%</td>
</tr>
<tr>
<td>465.tomto</td>
<td>mol.F : 5561</td>
<td>15.7%</td>
<td>80.4%</td>
<td>50779.4</td>
<td>99.2%</td>
<td>130.7</td>
</tr>
<tr>
<td></td>
<td>mol.F : 6659</td>
<td>15.9%</td>
<td>39.5%</td>
<td>266.6</td>
<td>97.2%</td>
<td>31.6%</td>
</tr>
<tr>
<td>470.lbm</td>
<td>lbn.c : 186</td>
<td>99.6%</td>
<td>100.0%</td>
<td>137487.0</td>
<td>61.6%</td>
<td>137487.0</td>
</tr>
<tr>
<td>481.wrf</td>
<td>solve_em.F : 179</td>
<td>87.9%</td>
<td>79.1%</td>
<td>1198.6</td>
<td>97.4%</td>
<td>39.7%</td>
</tr>
<tr>
<td></td>
<td>solve_em.F : 884</td>
<td>14.4%</td>
<td>89.3%</td>
<td>54721.8</td>
<td>99.8%</td>
<td>117.0%</td>
</tr>
<tr>
<td></td>
<td>solve_em.F : 1258</td>
<td>14.8%</td>
<td>89.6%</td>
<td>98871.7</td>
<td>93.6%</td>
<td>89.1%</td>
</tr>
<tr>
<td></td>
<td>solve_em.F : 1538</td>
<td>12.8%</td>
<td>87.4%</td>
<td>95531.4</td>
<td>95.4%</td>
<td>27.6%</td>
</tr>
<tr>
<td>482.sphinx3</td>
<td>approx_cont.m00 : 279</td>
<td>39.8%</td>
<td>68.1%</td>
<td>8949.0</td>
<td>75.2%</td>
<td>6886.1</td>
</tr>
<tr>
<td></td>
<td>approx_cont.m0 : 652</td>
<td>17.0%</td>
<td>72.8%</td>
<td>3.7</td>
<td>75.0%</td>
<td>39.0%</td>
</tr>
<tr>
<td></td>
<td>approx_cont.m00 : 456</td>
<td>30.8%</td>
<td>75.0%</td>
<td>19154.8</td>
<td>75.5%</td>
<td>15360.0</td>
</tr>
</tbody>
</table>

Table 3.1: Analysis results for analyzed benchmark loops.
### Table 3.2: Analysis results for computation kernels.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D Gauss-Seidel Stencil</td>
<td>0.0%</td>
<td>226</td>
<td>22.2%</td>
<td>46.1</td>
<td>77.4%</td>
<td>9.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-D PDE Grid Solver</td>
<td>0.0%</td>
<td>231426</td>
<td>100.0%</td>
<td>820.8</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.3: Analysis results for UTDSP benchmark suite.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Array</td>
<td>49.9%</td>
<td>568.9</td>
<td>79.3%</td>
<td>24.1</td>
<td>12.2%</td>
<td>2.0</td>
<td></td>
<td></td>
<td>12.2%</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.0%</td>
<td></td>
<td>568.9</td>
<td>79.3%</td>
<td>24.1</td>
<td></td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.0%</td>
<td></td>
<td>99.9</td>
<td>100.0%</td>
<td>57.4</td>
<td>0.0%</td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.0%</td>
<td></td>
<td>99.9</td>
<td>100.0%</td>
<td>57.4</td>
<td>0.0%</td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.00%</td>
<td></td>
<td>43.6</td>
<td>64.8%</td>
<td>14.3</td>
<td>0.0%</td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.00%</td>
<td></td>
<td>43.6</td>
<td>64.8%</td>
<td>14.3</td>
<td>0.0%</td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.00%</td>
<td></td>
<td>7.4</td>
<td>74.6%</td>
<td>23.9</td>
<td>0.0%</td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.00%</td>
<td></td>
<td>7.4</td>
<td>74.6%</td>
<td>23.9</td>
<td>0.0%</td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.0%</td>
<td></td>
<td>2.7</td>
<td>48.3%</td>
<td>22.1</td>
<td>16.5%</td>
<td></td>
<td></td>
<td>16.2%</td>
<td>21.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.0%</td>
<td></td>
<td>2.8</td>
<td>49.4%</td>
<td>28.0</td>
<td>16.2%</td>
<td></td>
<td></td>
<td>16.2%</td>
<td>21.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td>Array</td>
<td>50.4%</td>
<td>181.9</td>
<td>100.0%</td>
<td>18.2</td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pointer</td>
<td>0.00%</td>
<td></td>
<td>181.9</td>
<td>100.0%</td>
<td>18.2</td>
<td>0.0%</td>
<td></td>
<td></td>
<td>0.0%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Performance measurements for the case studies.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>Trans.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D Seidel</td>
<td>Xeon E5630 0.228s</td>
<td>Core i7 2600K 0.170s</td>
<td>1.386×</td>
</tr>
<tr>
<td></td>
<td>Xeon E5630 0.355s</td>
<td>Core i7 2600K 0.198s</td>
<td>1.799×</td>
</tr>
<tr>
<td>2-D PDE</td>
<td>Xeon E5630 2.42e-1s</td>
<td>Core i7 2600K 1.72e-1s</td>
<td>1.382×</td>
</tr>
<tr>
<td></td>
<td>Xeon E5630 2.74e-1s</td>
<td>Core i7 2600K 2.50e-1s</td>
<td>1.100×</td>
</tr>
<tr>
<td>410.bwaves</td>
<td>Xeon E5630 9.10e-3s</td>
<td>Core i7 2600K 7.30e-3s</td>
<td>1.245×</td>
</tr>
<tr>
<td></td>
<td>Xeon E5630 3.05e-2s</td>
<td>Core I1100T 1.99e-2</td>
<td>1.528×</td>
</tr>
<tr>
<td>433.milc</td>
<td>Xeon E5630 238.5s</td>
<td>Core i7 2600K 163.6s</td>
<td>1.386×</td>
</tr>
<tr>
<td></td>
<td>Xeon E5630 3.05e-2s</td>
<td>Core I1100T 1.99e-2</td>
<td>1.528×</td>
</tr>
<tr>
<td>435.gromacs</td>
<td>Xeon E5630 210.3s</td>
<td>Core I1100T 151.8s</td>
<td>1.386×</td>
</tr>
</tbody>
</table>

54
Listing 3.7 Original and transformed bwaves code.

1  
2  \texttt{ish = 1 }  
3  \texttt{ksh = 1 }  
4  \texttt{jsh = 1 }  
5  \texttt{!! Original}  
6  \texttt{do } \texttt{k = 1,nz}  
7  \texttt{kp1 = mod(k,nz+1-ksh)+ksh}  
8  \texttt{do } \texttt{j=1,ny}  
9  \texttt{jp1=mod(j,ny+1-jsh)+jsh}  
10 \texttt{do } \texttt{i=1,nx}  
11 \texttt{ip1=mod(i,nx+1-ish)+ish}  
12 \texttt{!! Some computation}  
13 \texttt{je(1,1,i,j,k) = ... }  
14 \texttt{je(1,2,i,j,k) = ... }  
15 \texttt{... }  
16 \texttt{je(4,5,i,j,k) = ... }  
17 \texttt{je(5,5,i,j,k) = ... }  
18 \texttt{!! Some computation}  
19 \texttt{ros = q(1,ip1,jp1,kp1)}  
20 \texttt{!! Similar computation for jv as for je}  
21 \texttt{enddo}  
22 \texttt{enddo}  
23 \texttt{!! Transformed}  
24 \texttt{do } \texttt{k = 1,nz}  
25 \texttt{kp1 = mod(k,nz+1-ksh)+ksh}  
26 \texttt{do } \texttt{j=1,ny}  
27 \texttt{jp1=mod(j,ny+1-jsh)+jsh}  
28 \texttt{do } \texttt{i=1,nx-1}  
29 \texttt{ip1=i+1}  
30 \texttt{!! Some computation}  
31 \texttt{je(i,1,1,j,k) = ... }  
32 \texttt{je(i,1,2,j,k) = ... }  
33 \texttt{... }  
34 \texttt{!! Some computation}  
35 \texttt{ros = q(ip1,1,jp1,kp1)}  
36 \texttt{!! Similar computation for jv as for je}  
37 \texttt{enddo}  
38 \texttt{i = nx}  
39 \texttt{ip1 = 1}  
40 \texttt{!! Inner loop computation (lines 11-19)}  
41 \texttt{enddo}  
42 \texttt{enddo}
Listing 3.8 Original and transformed milc code.

```c
/* Original data layout */
typedef struct { double r, i; } complex;
typedef struct { complex c[3]; } su3_vector;
typedef struct { complex e[3][3]; } su3_matrix;
su3_matrix lattice[NUM_SITES];
su3_vector vec[NUM_SITES], out_vec[NUM_SITES];

/* Original computation */
for (s = 0; s < NUM_SITES; ++s) {
    for (i = 0; i < 3; ++i) {
        complex x = { 0.0, 0.0 };  
        for (j = 0; j < 3; ++j) {
            complex y;
            y.r = lattice[s].e[i][j].r * vec[s].c[j].r -
                lattice[s].e[i][j].i * vec[s].c[j].i;
            y.i = lattice[s].e[i][j].r * vec[s].c[j].i +
                lattice[s].e[i][j].i * vec[s].c[j].r;
            x.r += y.r; x.i += y.i;
        }
        out_vec[s].c[i] = x;
    }
}

/* Transformed data layout */
typedef struct {
    double r[3][3][NUM_SITES];  
    double i[3][3][NUM_SITES];
} lattice_dlt;
typedef struct {
    double r[3][NUM_SITES];
    double i[3][NUM_SITES];
} vec_dlt;
lattice_dlt lattice;
vec_dlt vec, out_vec;

/* Transformed computation */
/* Initialize the elements of out_vec to 0.0 */
for (i = 0; i < 3; ++i) {
    for (j = 0; j < 3; ++j) {
        for (s = 0; s < NUM_SITES; ++s) {
            double x_r, x_i;
            x_r = lattice.r[i][j][s] * vec.r[j][s] -
                lattice.i[i][j][s] * vec.i[j][s];
            x_i = lattice.r[i][j][s] * vec.i[j][s] +
                lattice.i[i][j][s] * vec.r[j][s];
            out_vec.r[i][s] += x_r;
            out_vec.i[i][s] += x_i;
        }
    }
}
```

56
Listing 3.9 Original and transformed gromacs code.

```plaintext
!! Original

```
CHAPTER 4

Stencil Grids

Before introducing our code generation algorithm for stencil computations, we must first introduce our definition of a stencil computation and its computational domain. A stencil computation is a nearest-neighbor computation that operates on each point in a grid and updates each grid point using data in neighboring points. These computations are very prevalent in scientific computing and represent an interesting optimization challenge for accelerator architectures such as GPUs. The definitions in this chapter will be used to drive our code generation framework in the following chapter.

4.1 Computation Grid

Every stencil computation that we consider is defined on a computation grid, which is a bounded, rectangular region in $\mathbb{Z}^N$ that is defined over $[0, B_i)$ for some $B_i \in \mathbb{Z}$ in each dimension. Formally, the grid is completely defined by an N-dimensional bound vector:

$$\vec{B} \in \mathbb{Z}^N = (B_0, ..., B_{N-1}) \quad (4.1)$$
where \( N \) is the dimensionality of the grid. Then the grid can be built by taking the cartesian product of all intervals between zero and the bound for each dimension:

\[
\mathcal{G}_{\vec{B}} = [0, B_0 - 1] \times [0, B_1 - 1] \times \ldots \times [0, B_n - 1]
\] (4.2)

As a concrete example, consider the grid in Figure 4.1. Here, we define the bounds of the grid to be \( \vec{B} = (6, 7) \). We then construct our grid \( \mathcal{G}_{\vec{B}} \) as shown, giving us the shaded region in \( \mathbb{Z}^2 \). Note that there is no restriction forcing us to assign known integer values to the bounds of the grid. We could just as easily leave them as parameters, making the size (and shape) of our computation grid parametric in \( \vec{B} \).
4.2 Data Grids

Once we have defined our computation grid, we can define the data grids for a particular computation. A data grid \( D_G \) on computation grid \( G \) is a data structure that maps a value to every point in the data grid. The type of these values can be simple scalar values or complex recursive types, defined by the following grammar:

\[
\begin{align*}
\text{ElemType} & \rightarrow \text{real} \mid \text{integer} \mid \text{vector of ElemType} \mid \text{StructType} \\
\text{StructType} & \rightarrow \{ \text{field}_1 : \text{ElemType}_1, \text{field}_2 : \text{ElemType}_2, \ldots \}
\end{align*}
\]

A stencil computation can contain any number of distinct data grids. Each data grid is defined for every point in the computation grid \( G \), so we can think of a data grid as superimposing meaningful data on an otherwise abstract grid.

4.3 Stencil Functions

Another component of a stencil computation is the sequence of stencil functions that are applied iteratively to the data grids defined on the computation grid. A stencil function defines a computation that is applied to every point of a rectangular region of the computation grid. This function uses neighboring data grid points in the same data grid or other data grids in its computation of the new value for the point.

Formally, if \( T_D \) is the value type associated with data grid \( D_G \), then a stencil function \( f \) is a relation \( f : T_A^N \times T_B^M \times \ldots \times T_C^P \rightarrow T_D \) where \( A, B, C \) are data grids and \( N, M, P \) are any integers greater than zero and depend on the number of grid points of the
particular data grid needed in the computation. The domain for this relation is an arbitrary rectangular region of the computational grid, denoted $\mathcal{D}_B$.

As an example, consider a simple 2-D stencil function that is defined on the computation grid introduced in Figure 4.1. We define the stencil as:


where the notation $A[i, j]$ shows an access to data grid $A$ at offset $(i, j)$ from the current grid point, and $[1..N - 2, 1..M - 2]$ specifies the range of the computation grid over which the stencil function should be applied\(^2\). This notation naturally extends to grids of any dimensionality. This stencil function effectively computes a new value for a grid point based on the immediate neighbors in both the horizontal and vertical directions.

Note that we have not specified the type for data grid $A$. In this example, we assume that the addition operator (+) is well-defined for the value type of data grid $A$. For scalar, vector, and matrix types, the operator is clearly well-defined. However, if our value type is of structure type, the addition operator may not be well-defined in general.

### 4.4 Program Specification

Using the previous definitions of computation grid, data grid, and stencil function, we can now define a complete stencil program. A stencil program defines the underlying computation grid, a set of one or more data grids that are superimposed on the computation grid, and a sequence of stencil functions that are applied to the

\(^2\)Here, it is assumed that $N$ and $M$ are the bounds of the computation grid.
data grids. For each data grid, we must define the value type for the points. We must also define the region over which each stencil function operates. Finally, we must define the number of iterations over which the stencil computation will occur. In each iteration, every stencil function is evaluated in the proper region.

In the presence of multiple stencil functions, the semantics of a stencil program asserts that they are executed in sequential order. That is, the first stencil function is applied to each point in its region and the results are written back to the data grid before any evaluation of the second stencil function occurs. There are no ordering constraints between grid points within a stencil function evaluation. That is, each evaluation of a stencil function within a single outer iteration occurs concurrently.

4.5 Example

As a concrete example, we will now consider the following stencil program, where $M$ is a parameter:

$$\vec{B} = (M)$$

$$\mathbb{G}_{\vec{B}} = [0, M - 1]$$

$$A_c : \text{real}$$

$$f_1(A)[1..M - 2] = 0.333 \times (A[-1] + A[0] + A[1])$$

In this program, we define the computation grid as the region $[0, M - 1]$ in $\mathbb{Z}$. We also define one data grid $A_c$ which superimposes a grid of real numbers onto the computation grid. Finally, we define a single stencil function $f_1$ that computes
an average of grid point values, defined in the range \([1, M - 2]\) on the computation grid. Note that the edge points are not updated in the computation.

To help understand the semantics of this program, we can implement it in C-like pseudocode:

1. \texttt{real A[M];}
2. \texttt{real A\_tmp[M];}
3. \texttt{copy(A\_tmp /* dest */, A /* src */);}
4. \texttt{for (n = 0; n < T; ++n) {}
5. \hspace{2em} \texttt{for (i = 1; i < M-1; ++i) {
6. \hspace{4em} \texttt{A[i] = 0.333 \times (A\_tmp[i-1] + A\_tmp[i] + A\_tmp[i+1]);
7. \hspace{2em}}}
8. \texttt{copy(A\_tmp, A);}
9. }

Note the use of the \texttt{A\_tmp} array to help implement the semantics of the stencil program. In each iteration (\(n\)-loop), the new values for \(A\) are computed using the \textit{old} values of \(A\) from the previous iteration. Therefore, the \texttt{A\_tmp} array is used to cache the old values of \(A\). Also, the variable \(T\) is used as a parameter for the number of iterations over which to perform the stencil computation.

### 4.6 OverTile: Stencil-DSL Compiler for GPGPUs

In this chapter, OverTile is introduced as a stencil-DSL compiler library for GPGPU architectures. It incorporates the overlapped tiling code generation algorithms presented in Chapter 5 and the performance model presented in Chapter 6.

#### 4.6.1 Overview

OverTile is a Stencil-DSL compiler library for GPGPUs. It can be used as either a library or a stand-alone compiler, depending on the input format and level of integration with a hosting compiler. The core IR for OverTile is an AST structure
called ExprAPI. ExprAPI is a C++ class hierarchy that can be used to describe a stencil program. An ExprAPI structure is then passed to the OverTile code generator, which produces either CUDA or OpenCL device code and supporting host code. In addition to direct use of ExprAPI, a text specification language is provided. A stand-alone compiler exists as part of OverTile that generates CUDA or OpenCL code directly from a Stencil Specification (SSP) text file. This process is depicted in Figure 4.2.

The stand-alone OverTile compiler also provides facilities to extract an embedded stencil specification from C/C++ code, generate CUDA or OpenCL code for the stencil, and place it in the C/C++ source file. This allows users to embed a stencil specification directly into C/C++ code and the compiler takes care of the host/device glue code. Other DSL implementations can generate ExprAPI and make use of the resulting CUDA or OpenCL code to implement high-performance GPGPU code generation into their compilers.
4.6.2 Inputs

The OverTile compiler works on an intermediate representation called ExpressionAPI (ExprAPI) that defines the stencil program to be generated. This representation is described below. Additionally, a text version of the intermediate representation is available that can be used as a description language.

ExpressionAPI

The ExpressionAPI representation is an abstract syntax-tree structure that defines the fields and functions for a stencil program. It is defined as a C++ class hierarchy.

Grid  At the head of the AST structure is a Grid class which contains the definitions for all fields and functions.

```cpp
/**
 * Abstraction of a space in Z^n
 */

class Grid {
public:
    /// Grid - default constructor
    Grid();
    /// Grid - Constructs a Grid instance of the specified dimensionality.
    Grid(unsigned Dim);
    /// Grid - default destructor
    ~Grid();

    /// attachField - Attaches a Field object to the grid.
    void attachField(Field *F);

    /// appendFunction - Appends the Function \p F to the list of stencil point
    /// functions that act on this grid.
    void appendFunction(Function *F);

    /// getNumDimensions - Returns the dimensionality of the grid.
    unsigned getNumDimensions() const;
    /// setNumDimensions - Sets the dimensionality of the grid.
    void setNumDimensions(unsigned Dim);
```
Field class defines a single field in the stencil program, and contains the data type and copy-semantics for the field. The copy semantics defines if the field is copy-in, copy-out, or copy-inout.
Function  The `Function` class defines an expression to be applied to one or more input fields to produce an output field. A function also contains a bound for each dimension to control the region for which the function is applied. Many stencil functions are applied to only a subset of the entire grid, with different functions being used for boundary and non-boundary regions.

```cpp
/**
 * Representation of a stencil point function.
 */
class Function {
  public:
    /// Function - Constructs a new Function instance.
    Function(Field *Out, Expression *E);
    /// Function - Default destructor
    ~Function();

    // setLowerBound - Sets a bound on the function so it is only applicable
    // starting with element \p Offset in the \p Dim dimension.
    void setLowerBound(unsigned Dim, unsigned Offset);

    // setUpperBound - Sets a bound on the function so it is only applicable
    // below element N - \p Offset in the \p Dim dimension.
    void setUpperBound(unsigned Dim, unsigned Offset);

    // getInputFields - Returns a set of Field objects that are read when
    // evaluating this function.
    std::set<Field*> getInputFields() const;

    // adjustRegion - Adjusts the Region \p FRegion for Field \p F to include
    // and points that are needed to evaluate the function in Region \p InRegion.
    void adjustRegion(Field *F, Region &FRegion, const Region &InRegion) const;

    // getMaxOffsets - Returns in \p LeftMax and \p RightMax the left-most and
    // right-most offsets for field \p F that are touched by this function for
    // dimension \p Dim.
    void getMaxOffsets(const Field *F, unsigned Dim, unsigned &LeftMax,
                       unsigned &RightMax) const;

    // countFlops - Returns the number of floating-point ops computed by an
    // invocation of this function.
    double countFlops() const;

    //==-- Accessors --======================================================== //

    /// getOutput - Returns the field written by this function.
    Field *getOutput() { return OutField; }
    const Field *getOutput() const { return OutField; }
};
```
A Jacobi 2D stencil program written in SSP.

Listing 4.1

```c
1 program j2d is
2 grid 2
3 field A float inout

4 A =
5 @[1:$-1][1:$-1] = 0.2*(A[-1][0]+A[0][0]+A[1][0]+A[0][1]+A[0][-1])
```
A single field is then defined by the `field A float inout` statement with a name of `A` and type `float`. The `inout` attribute is a hint to the compiler that the field is both input and output to the program, and should be copied between the host and accelerator appropriately. The compute statement defines a new value for each point in field `A` based on the old values of `A` at the same point and neighboring points. The syntax `A[0][1]` specifies the point to the right of the current point. Note that the semantics of the program state that all new values of `A` are computed before any results are written back to `A` (Jacobi-style instead of Seidel-style). The compute statement also specifies that the function is applied in the range `[1, N - 2] × [1, M - 2]` where `N × M` is the problem size.

As a more complex example, Figure 4.2 shows a Finite- Difference stencil program. In this example, three fields are defined, all of which are both inputs and outputs to the stencil program. There are also three compute statements that update the fields in the order that they are defined in the program. Each statement is defined over a different region of the grid.
4.6.4 Code Generation

Code generation within OverTile implements the algorithms presented in Chapter 5. The ExprAPI representation is a one-to-one mapping with the algorithm inputs defined in that chapter, and the resulting code is translated into either CUDA or OpenCL depending on the user-specified target. The performance model presented in Chapter 6 is used to automatically select block and tile sizes if they are not specified by the user.

4.7 Conclusion

In this chapter, we have introduced the notation and formalisms we will be using throughout the remainder of this work. We have given our definition of a stencil computation and its mathematical components. We have also provided a concrete example to help understand how we specify stencil programs and hence how we define the input to our code generation algorithm. We have also introduced OverTile as a Stencil-DSL compiler library that implements the code generation and performance modeling work presented in this dissertation. It is useful as both a compiler library for adding GPGPU code generation to other DSL front-ends, as well as a stand-alone or embedded DSL.
\[ \text{⟨program} \rangle ::= \text{‘program’ ident ‘is’ (grid-def) (field-list) (func-list)} \]
\[ \text{⟨grid-def} \rangle ::= \text{‘grid’ (int-constant)} \]
\[ \text{⟨field-list} \rangle ::= \text{(field-def) (field-list)} | \epsilon \]
\[ \text{⟨field-def} \rangle ::= \text{‘field’ ident (type) (copy-semantic)} \]
\[ \text{⟨func-list} \rangle ::= \text{(func-def) (func-list)} | \epsilon \]
\[ \text{⟨func-def} \rangle ::= \text{ident (bounded-expr-list)} \]
\[ \text{⟨bounded-expr-list} \rangle ::= \text{(bounded-expr) (bounded-expr-list)} | \text{(bounded-expr)} \]
\[ \text{⟨bounded-expr} \rangle ::= \text{‘@’ (bounds) ‘=’ (expr)} \]
\[ \text{⟨bounds} \rangle ::= \text{(bound-def) (bounds)} | \epsilon \]
\[ \text{⟨bound-def} \rangle ::= \text{‘[’ (bound-expr) ‘:’ (bound-expr) ‘]’} \]
\[ \text{⟨bound-expr} \rangle ::= \text{‘$’ ‘-’ (int-constant)} \]
\[ \text{⟨expr} \rangle ::= \text{(expr) ‘+’ (expr)} | \text{(expr) ‘-’ (expr)} | \text{(expr) ‘*’ (expr)} | \text{(expr) ‘/’ (expr)} | \text{‘(’ (expr) ‘)’} | \text{‘let’ ident ‘=’ (expr) ‘in’ (expr)} | \text{⟨func-call⟩} | \text{⟨field-ref⟩} | \text{⟨int-constant⟩} | \text{⟨float-constant⟩} \]
\[ \text{⟨func-call} \rangle ::= \text{ident ‘(’ (expr-list) ‘)’} \]
\[ \text{⟨expr-list} \rangle ::= \text{(expr) ‘,’ (expr-list)} | \text{(expr)} \]
\[ \text{⟨field-ref} \rangle ::= \text{ident (offset-list)} \]
\[ \text{⟨offset-list} \rangle ::= \text{(offset) (offset-list)} | \text{(offset)} \]
\[ \text{⟨offset} \rangle ::= \text{‘[’ (int-constant) ‘]’} \]

**Figure 4.3:** Grammar for OverTile Stencil Specification

71
CHAPTER 5

GPU Code Generation for Overlapped Tiling

5.1 Overview

Tiling for stencil computations is complicated by data sharing between neighboring tiles. Cells along the boundary of a tile are often needed by computations in surrounding tiles, requiring communication between tiles when neighboring tiles are computed by different processors. To compute a stencil on a cell of a grid, data from neighboring cells is required. These cells are often referred to as the halo region. In general, to compute an $N \times M$ block of cells on a grid, we need an $(N+n)\times(M+m)$ block of data to account for the cells we are computing as well as the surrounding halo region, where $n$ and $m$ are constants derived from the shape of the stencil. For GPU devices, the halo region needs to be re-read from global memory for every time step as surrounding tiles may update the values in these cells. This limits the amount of re-use we can achieve in scratchpad memory before having to go back to global memory for new data. The new cell values produced in each time step must also be re-written to global memory as other tiles may require the data in their halo regions. In addition to the cost of going to global memory for each time step, global synchronization is also required to ensure all surrounding
tiles have completed their computation and written their results to global memory before new halo data is read for each tile.

To get around these issues, overlapped tiling has been proposed [24] as a technique to reduce the data sharing requirements for stencil computations by introducing redundant computations. Instead of forcing tile synchronization after each time step to update the halo region for each tile, each tile instead redundantly computes the needed values for the halo region. This allows us to efficiently perform time tiling and achieve high performance on GPU targets.

Consider a simple 2-D Jacobi 5-point stencil like the one shown in Figure 5.1. In order to compute one time step of an $n \times n$ tile, we need to read $(n + 2) \times (n + 2)$ cells into scratchpad memory, compute the stencil operation on each of the $n \times n$ points, then write back $n \times n$ points to global memory. Now, let us consider the computation of two time steps of the stencils on a single block, without having to go back to global memory between the two time steps. If we read $(n + 4) \times (n + 4)$ cells into memory, we can compute a $(n+2) \times (n+2)$ tile of cells in the first time step, which includes the results for our original $n \times n$ tile as well as the halo region needed for the next time step. If we again apply the stencil operation to the $(n + 2) \times (n + 2)$ tile, we correctly compute the inner $n \times n$ tile for the second time step but the results computed in the halo region for the second time step are not correct. However, this is not a problem since we only care about the inner $n \times n$ region. An illustration of this computation is presented in Figure 5.2.
for(t = 0; t < T; ++t) {
for(i = 1; i < N-1; ++i) {
for(j = 1; j < N-1; ++j) {
}
}
for(i = 1; i < N-1; ++i) {
for(j = 1; j < N-1; ++j) {
    B[i][j] = A[i][j];
}
}
}

Figure 5.1: Jacobi 5-Point Stencil. In (a), C code is shown for the stencil, and in (b), the accessed data space is shown for one grid point. The orange cell is the (i,j) point, and each blue cell is read during the computation of the (i,j) cell.

Figure 5.2: Overlapped Jacobi 5-Point Stencil for T = 2.

5.2 Code Generation

To generate efficient GPU code, we need to tile the data space of the stencil grids into units that can be computed by a single thread block on a GPU device without
### Algorithm 5.1: Overlapped-tiling code generation algorithm

<table>
<thead>
<tr>
<th>Input</th>
<th>P : Program, B : Block Size, T : Time Tile Size, E : Elements Per Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>P&lt;sub&gt;gpu&lt;/sub&gt; : GPU Kernel, P&lt;sub&gt;host&lt;/sub&gt; : CPU function</td>
</tr>
</tbody>
</table>

1. \( R \leftarrow \text{DetermineTileSize}(P, B, T) \)
   //Generate GPU Kernel
2. \( P<sub>gpu</sub> \leftarrow \text{GenerateGPUKernel}(P, B, R, T) \) (§ 5.3)
   //Generate CPU Function
3. \( P<sub>host</sub> \leftarrow \text{GenerateHostCode}(P, B, R, T) \) (§ 5.4)

any needed block synchronization. For each stencil operation, we need to determine the region of grid points that will be computed by each thread block, including any needed redundant computation for intermediate time steps. The input to our code generation algorithm is a sequence of stencil operations and the output is overlapped-tiled GPU code and a host driver function.

Code generation for overlapped tiling on GPU architectures requires us to first identify the footprints of the stencil(s), determine proper block sizes for the target architecture, and then generate the GPU kernel code and host wrapper code. The overall algorithm is presented in Algorithm 5.1. The following sections describe all of the steps for the code generation algorithm.

**Determining Abstract Tiles:** To implement overlapped tiling, we need to create tiles such that a single thread block can compute all grid points for all stencil operations for \( T \) time steps, where \( T \) is the time tile size, without requiring any inter-block synchronization. For single-stencil computations, we only need to worry about the halo regions that are needed in intermediate time steps. However, for multi-stencil computations, we need to account for grid cells that may be needed for later stencil computations.
To determine the required computation for each sub-grid, we work backwards across all stencils and all time steps within our time tile. We start by assuming the last stencil computation in the last time step operates on a rectangular tile that is defined by an origin $\vec{x}_0$ and length $\vec{l}_0$. For each sub-grid used to compute this stencil, we can build a rectangular region that forms a tightest-possible bound on the sub-grid that includes all grid points needed for that stencil computation.

Once rectangular regions have been built for all involved sub-grids, we move onto the previous stencil in execution order and perform the same computation. Once we finish with the first stencil in execution order, we wrap around to the last stencil and continue the process again until we have processed all stencils for all time steps in our time tile. When we generate a new rectangular tile for a sub-grid for which we have already computed a tile, we form the new tile by forming the union of the old and new result and then taking the tightest-possible rectangular bound.

This tile size selection procedure is formalized in Algorithm 5.2. The $\text{GetUpdatedGrid}(s)$ function returns the sub-grid that is updated for the given stencil, and the function $\text{GetSourceGrids}(s)$ returns all sub-grids that are used on the right-hand side of the given stencil. The function $\text{GetRequiredBounds}(g, s, \vec{x}, \vec{l})$ returns a rectangular region that represents the grid cells that are needed in grid $g$ to compute stencil $s$ in the region defined by $\vec{x}$ and $\vec{l}$. For a given stencil program $P$, the $\text{ExtractSubGrids}(P)$ function returns all sub-grids that are touched by any point function. Finally, the $\text{RectangularHull}(\bigcup_{i}(\vec{x}_i, \vec{l}_i))$ function returns the region that forms the tightest-possible rectangular bound on the given (possibly non-rectangular) region.
**Algorithm 5.2:** Finding the tile size.

Name: DetermineTileSize  
Input: P : Stencil Program, T : Time Tile Size  
Output: \((\vec{x}_g', \vec{l}_g')\) \(\forall\) Sub-Grids

1. \(S \leftarrow \text{ExtractStencilFunctions}(P)\)
2. \(DG \leftarrow \text{ExtractSubGrids}(P)\)
3. \(\text{foreach} \text{ SubGrid } g \in DG \text{ do}\)
   4. \((\vec{x}_g', \vec{l}_g') \leftarrow (\vec{x}_0, \vec{l}_0)\)
4. \(\text{end}\)
5. \(\text{foreach} \text{ TimeStep } t \in (T, ..., 1) \text{ do}\)
6. \(\text{foreach} \text{ StencilFunction } s \in \text{Reverse}(S) \text{ do}\)
7. \(g \leftarrow \text{GetUpdatedGrid}(s)\)
8. \(\text{src} \leftarrow \text{GetSourceGrids}(s)\)
9. \(\text{foreach} \text{ SubGrid } dg \in \text{src} \text{ do}\)
   10. \((\vec{x}'_{dg}, \vec{l}'_{dg}) \leftarrow \text{GetRequiredBounds}(dg, s, \vec{x}_g', \vec{l}_g')\)
   11. \((\vec{x}_{dg'}, \vec{l}_{dg'}) \leftarrow \text{RectangularHull}((\vec{x}_g', \vec{l}_g') \cup (\vec{x}'_{dg}, \vec{l}'_{dg}))\)
12. \(\text{end}\)
13. \(\text{end}\)
14. \(\text{end}\)

The results of this algorithm are a set of rectangular regions defined by \(\vec{x}_g'\) and \(\vec{l}_g'\), one for each sub-grid \(g\). These regions define the grid cells that must be processed in each time step for each sub-grid in order to properly compute the final stencil in the final time step of the time tile. These regions will later be used to derive the per-block computation code. For the case of multiple grid points processed per thread, the \(\vec{l}_g'\) quantities are just multiplied by the number of elements per thread, element-wise in each dimension.

Note that to compute the final result for our \((\vec{x}_0, \vec{l}_0)\) region for each sub-grid, we do not need to compute values for the entire \((\vec{x}_g', \vec{l}_g')\) grid in each time step. However, on GPU architectures, it is more efficient to perform this redundant computation and let each thread perform the same amount of work in each time step. Otherwise, threads would contain control-flow instructions that would cause branch divergence and lower overall performance.
**1-D Single-Stencil Example:** To help illustrate how we create the abstract tiles for a stencil computation, a few examples are presented below that apply the given algorithm to a concrete stencil computation.

First, let us consider the base case of a single-stencil computation over three time steps. Let the stencil computation be defined as:


In this stencil, each grid point is updated using data from the previous iteration at the same grid point as well as immediately surrounding grid points. We start by abstractly defining our initial tile as $(\vec{x}_A, \vec{l}_A) = (x_0, l_0)$. Now we compute the needed tile regions for all input grids, which in this case is just the same grid. Based on the index expressions, if $(x_0, l_0)$ is the region we wish to compute, then we need a $(x_0 - 1, l_0 + 1)$ region of grid points.

**Figure 5.3:** Tile region computation for 1-D stencil.
We then proceed to the previous time step and apply the same algorithm. In this case, we know we need to compute a region \((x_0 - 1, l_0 + 2)\) of grid points, so we need a \((x_0 - 2, l_0 + 4)\) region of grid points from the previous time step. Extending this to the first time step will give us a final region of \((x_0 - 3, l_0 + 6)\). A visual depiction of this process is provided in Figure 5.3.

**1-D Multi-Stencil Example:** Let us now consider a multi-stencil computation. In the following example, we use two stencil computations operating over two data grids, defined as:

\[
\begin{align*}
\mathbf{f_A} & (B_0)[1..N-2] = B[-1] + B[0] \\
\mathbf{f_B} & (A_0)[1..N-2] = A[0] + A[1]
\end{align*}
\]

Again, we use a time tile size of three. We start by assigning an initial tile of \((\vec{x}_B, \vec{l}_B) = (x_0, l_0)\) to grid B. We see that the computation of B only depends on the grid A, so we use the index expressions to compute the needed region of A, which is \((\vec{x}_A, \vec{l}_A) = (x_0, l_0 + 1)\). We then backtrack to the computation for grid A, and determine that the needed region in grid B to compute A is \((\vec{x}_B, \vec{l}_B) = (x_0 - 1, l_0 + 2)\). This completes time step three, so we reverse to time step two and perform the same computation. Backtracking all of the way to the beginning of time step one, we have:

\[
\begin{align*}
(\vec{x}_A, \vec{l}_A) & = (x_0 - 2, l_0 + 5) \\
(\vec{x}_B, \vec{l}_B) & = (x_0 - 2, l_0 + 4)
\end{align*}
\]
5.3 Generation of GPU Kernel Code

Now that we know the amount of redundant computation that is needed for each sub-grid, we can generate the GPU kernel code for the stencil time tile. For this, we need to define a set of parameters that are used as input to the code generation algorithm:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>The number of cells to process per thread</td>
</tr>
<tr>
<td>$B$</td>
<td>The desired GPU block size ($x$, $y$, $z$)</td>
</tr>
<tr>
<td>$T$</td>
<td>The time tile size</td>
</tr>
</tbody>
</table>

Using the tile regions determined by Algorithm 5.2, we know that for each sub-grid $g$, the region of computation for a given thread block is given by $(\vec{x}_g, \vec{l}_g)$. Let us define the region with the maximum size as $(\vec{x}_{\text{max}}, \vec{l}_{\text{max}})$, where $\vec{x}_{\text{max}}$ and $\vec{l}_{\text{max}}$ are...
affine expressions in \( \vec{x}_0 \) and \( \vec{l}_0 \), our abstract tile size. Here, our definition of maximum size is the largest number of grid points contained in the region defined by \( (\vec{x}_g, \vec{l}_g) \). Then, given that \( \vec{B} \) is our desired thread block size, we can solve for \( \vec{l}_0 \) with:

\[
\vec{l}_0 = \vec{B}
\]

The value \( \vec{x}_0 \) is still a symbolic entity, but we now have a concrete value for \( \vec{l}_0 \) which we can use to generate GPU kernel code. An important observation at this point is that the region \( (\vec{x}_g, \vec{l}_g) \) defines the region of real computation for the stencil operation writing sub-grid \( g \) and the halo region (e.g. the region that is computed redundantly) is defined as:

\[
\text{Halo}(g) = (\vec{x}_\text{max}, \vec{l}_\text{max}) - (\vec{x}_g, \vec{l}_g)
\]

Note that the halo region is, in general, not a rectangular region. It is instead a rectangular bound around the non-halo region.

**Shared/Local Memory Size:** To take advantage of the GPU memory hierarchy, it is necessary to use shared/local memory to cache results whenever possible. To generate high-performance GPU code using overlapped tiling, we need to determine how much shared memory is needed to cache the redundant computations that are performed. This computation is straightforward, since we only need to account for the results produced by each thread. Therefore, the amount of shared memory we need for the computation (in number of data elements) is simply:

\[
\text{SharedSize} = \sum_g \text{(Area}(\vec{B}))
\]
Algorithm 5.3: Generating shared-memory definitions

<table>
<thead>
<tr>
<th>Name: GenerateSharedMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> P : GPU Program, ( \vec{B} ) : Block Size, ( \vec{E} ) : Elements Per Thread, D : Sub-Grids</td>
</tr>
<tr>
<td><strong>Output:</strong> P : GPU Program</td>
</tr>
</tbody>
</table>

1. **foreach** SubGrid \( g \) in D do
   2. \( P \leftarrow \text{DeclareSharedRegion}(P, \text{NumberOfPoints}(\vec{B}, \vec{E}), g) \)
3. **end**

This allows each thread to cache the result it produces for each stencil computation. The \( \text{Area()} \) function simply returns the number of grid points within the region defined by \( (\vec{x}_g, \vec{l}_g) \).

This process is formalized in Algorithm 5.3. Given a block size and the number of grid points to process per device thread, a shared memory region is declared for each sub-grid. This shared memory region is declared for the GPU program \( P \). The \( \text{NumberOfPoints} \) function simply returns the number of integer points contained within the block defined by taking the component-wise multiplication of \( \vec{B} \) and \( \vec{E} \).

**Thread Synchronization:** Thread barriers are used between stencil operations to ensure that all threads have finished the computation for a particular stencil operation before any thread starts computing a value for the next stencil operation. The thread blocks compute completely independently, but the threads within a thread block must be synchronized as one thread may produce a result that is needed by another thread in the next stencil operation.

**Block Synchronization:** Our computation model dictates that for each stencil operation, a snapshot of the data is used as input for every evaluation of every grid...
point, and writes back to the grid are done after all evaluations have finished. In other words, all reads from global memory must see the same data. Unfortunately, this is hard to guarantee on GPU architectures which lack block synchronization and GPU-wide memory fences. To get around this issue, a buffering approach is used. For each sub-grid used as an output for a stencil computation, two grids are actually maintained in GPU memory. For each time tile, one version is used as input and another is used as output. Between time tiles, the host will swap the buffer pointers before the next kernel invocation. This ensures that all reads within a kernel invocation see the same data and no issues can arise from one thread block completely finishing a computation before another block starts.

**Thread Code:** We can now generate the per-thread code that will implement the stencil computation. This code implements the computation of all sub-stencils for $|\vec{E}|$ data elements over $T$ time steps. In the first time step, each thread reads its needed data from global memory, performs the computation of the first stencil operation, and stores the result to shared/local memory. This process is repeated for each stencil operation in the stencil computation. For each subsequent time step, each thread reads its needed data from shared/local memory, performs the computation of each sub-stencil, in order, and stores the result to shared/local memory. In the last time step, *if the thread is not part of the halo*, the final result is written back to global memory.

This process is formalized in Algorithm 5.4. Here, code is generated for each stencil function evaluation for every time step of our time tile. A new GPU program object is created that represents our generated kernel code, and we generate code
Algorithm 5.4: Thread code generation

Name: GenerateGPUKernel
Input: P : Stencil Program, ⃗B : Block Size, ⃗E : Elements Per Thread, T : Time Tile Size
Output: P_gpu : GPU Program

1 S ← ExtractStencilFunctions (P)
2 D ← ExtractSubGrids (P)
3 P_gpu ← NewGPUProgram ()
4 P_gpu ← GenerateSharedMemory (P_gpu, ⃗B, ⃗E, D) (Algorithm 5.3)
5 InShared ← Ø
6 foreach TimeStep t in T do
7     foreach Function f in S do
8         foreach Element e in Iterate (⃗E) do
9             foreach SubGrid g in GetSources (f) do
10                if g ∈ InShared then
11                   P_gpu ← GenerateSharedMemoryReads (P_gpu, g, e, f)
12                else
13                   P_gpu ← GenerateGlobalMemoryReads (P_gpu, g, e, f)
14            end
15            P_gpu ← GenerateFunctionEvaluation (P_gpu, f, e)
16        end
17     end
18     P_gpu ← GenerateThreadSync (P_gpu)
19     P_gpu ← GenerateSharedMemoryWrite (P_gpu, g, e)
20     InShared ← InShared ∪ GetDestination (f)
21     P_gpu ← GenerateThreadSync (P_gpu)
22 end
23 foreach SubGrid g in D do
24     foreach Element e in Iterate (⃗E) do
25         if ¬ Halo (g, e) then
26             P_gpu ← GenerateGlobalMemoryWrite (P_gpu, g, e)
27         end
28     end
29 end
30
to evaluate multiple grid points per thread (if needed), and use shared memory to cache results whenever possible. Different implementations of the generate functions can be used to target different languages, including CUDA, OpenCL, LLVM IR, etc.
**Example:** As an example, let us consider a Jacobi 5-point stencil over 2 time steps. The stencil function can be defined as:

\[
f(A_t)(1..N - 1, 1..M - 1) = 0.2 \times (A[-1, 0] + A[0, 0]+ A[1, 0] + A[0, -1] + A[0, 1])
\]

The generated code for an OpenCL target will apply the stencil function twice, once at time \( t \) and again at time \( t + 1 \). Global memory will only be read at the beginning of time step \( t \), and written at the end of time step \( t + 1 \). The results computed in time step \( t \) will be cached in shared memory and read in time step \( t + 1 \). A barrier will be placed between the time steps to ensure all shared memory writes complete before any shared memory reads occur in time step \( t + 1 \).

### 5.4 Generation of Host Code

To be a complete code generation framework, host code is also needed to properly configure and execute the generated GPU kernels. For our purposes, this involves creating a C/C++ function that is responsible for copying input data to the GPU device before kernel invocation, setting up the proper grid and block sizes, invoking the kernel, and copying output data back to the host after kernel invocation. The general procedure is outlined in Algorithm 5.5, and details are provided in the following subsections.

**Copy In/Out:** Copy-in/copy-out code is generated by determining the amount of global halo that is needed, allocating buffers of the appropriate size, and copying data to/from the device at the appropriate time. A global halo is used to eliminate
Algorithm 5.5: Host code generation

Name: GenerateHostCode
Input: \( P : \) Stencil Program, \( P_{\text{gpu}} : \) GPU Program, \( \vec{B} : \) Block Size, \( \vec{E} : \) Elements Per Thread, \( T : \) Time Tile Size, \( \vec{N} : \) Problem Size, \( T_{\text{total}} : \) Total Number of Time Steps, \((\vec{x}_0, \vec{l}_0)\)
Output: \( P_{\text{host}} : \) Host Program

1. \( P_{\text{host}} \leftarrow \text{NewHostProgram}() \)
2. \( \text{DG} \leftarrow \text{ExtractSubGrids}(P) \)
3. \( \text{NumBlocks} \leftarrow (N_x/l_{0,x}, N_y/l_{0,y}, ...) \)
4. \textbf{foreach} SubGrid \( g \) \textbf{in} \( \text{DG} \) \textbf{do}
5. \( g_{\text{gpu},0} \leftarrow \text{AllocateGPUBuffer}(g) \)
6. \( g_{\text{gpu},1} \leftarrow \text{AllocateGPUBuffer}(g) \)
7. \( P_{\text{host}} \leftarrow \text{CopyHostToDevice}(P_{\text{host}}, g_{\text{gpu},0}, g) \)
8. \( P_{\text{host}} \leftarrow \text{CopyHostToDevice}(P_{\text{host}}, g_{\text{gpu},1}, g) \)
9. \textbf{end}
10. Body \( \leftarrow \{ \text{InvokeKernel}(P_{\text{gpu}}, \text{NumBlocks}); \text{Swap}(g_{\text{gpu},0}, g_{\text{gpu},1}) \ \forall \ \text{SubGrid } g \ \text{in } \text{DG} \} \)
11. \( P_{\text{host}} \leftarrow \text{GenerateTimeLoop}(P_{\text{host}}, T_{\text{total}} / T, \text{Body}) \)
12. \textbf{foreach} SubGrid \( g \) \textbf{in} \( \text{DG} \) \textbf{do}
13. \( P_{\text{host}} \leftarrow \text{CopyDeviceToHost}(P_{\text{host}}, g_{\text{gpu},1}, g) \)
14. \textbf{end}

Conditional behavior around the boundary in GPU kernel code. When time tiling is used, a thread that would ordinarily access the sub-grid boundary may now read several points beyond the edge of the grid. To make sure these memory accesses stay within bounds and that the intermediate results are correct, the boundary grid cells are replicated into a halo region of width equal to the maximum radius of all stencils that read from the sub-grid.

Thread Blocks: The GPU block size is pre-determined as an input to the code generation algorithm, but we also need to determine the number of thread blocks that will be executed. Remember that each thread block computes a region of \((\vec{x}_{\text{max}}, \vec{l}_{\text{max}})\) for grid \( g \), where only \((\vec{x}_0, \vec{l}_0)\) is useful work. If \( \vec{N} \) is our total problem size, then we need \(|\vec{N}/\vec{l}_0|\) total blocks, where the division is performed element-wise and the length is a measure of the total number of blocks in the volume.
Table 5.1: Characteristics of stencil programs used for performance evaluation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Elements Per Pt</th>
<th>Number of Arrays</th>
<th>Ops Per Pt</th>
<th>Dim.</th>
<th>Uses SFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denoise 2D</td>
<td>5/5/1</td>
<td>3</td>
<td>187</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>FDTD 2D</td>
<td>2/2/3</td>
<td>3</td>
<td>11</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>Gradient 2D</td>
<td>5</td>
<td>1</td>
<td>18</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Heat 2D</td>
<td>5</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>Jacobi 2D</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>Poisson 2D</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>TV Update 2D</td>
<td>7</td>
<td>3</td>
<td>87</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Denoise 3D</td>
<td>7/7/1</td>
<td>3</td>
<td>357</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>Gradient 3D</td>
<td>7</td>
<td>1</td>
<td>26</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>Heat 3D</td>
<td>7</td>
<td>1</td>
<td>15</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>Jacobi 3D</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>Poisson 3D</td>
<td>9</td>
<td>1</td>
<td>27</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>TV Update 3D</td>
<td>12</td>
<td>2</td>
<td>113</td>
<td>3</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Time Loop:** The host code is also responsible for implementing the outer time loop. Each invocation of the GPU kernel will compute $T$ time steps of the stencil. If $S$ is the total number of time steps, then the host code will invoke the kernel $\frac{S}{T}$ times. After each time tile, the input and output buffers are swapped to make the output from the previous time tile the input to the next time tile.

### 5.5 Experimental Evaluation

In this section, experimental results are presented that show the effectiveness of the GPU code generation algorithms presented in this chapter. We analyze the performance of the generated GPU kernels and explore the effects of block and tile sizes on the achieved performance.
Figure 5.5: Performance evaluation of OverTile-generated GPU kernels on Tesla C2050 and Tesla K10 (single-precision). The numbers above the bars show the achieved GFlop/s.

5.5.1 OverTile Performance Results

We start by looking at the achieved performance of OverTile-generated GPU kernels for a variety of stencil programs. Table 5.1 gives an overview of the stencil programs used in this experiment, including stencil program characteristics that have an impact on the performance of the generated code. For each stencil program, we test each possible block and tile size to determine the best-performing point.
Figure 5.6: Performance evaluation of OverTile-generated GPU kernels on Tesla C2050 and Tesla K10 (double-precision). The numbers above the bars show the achieved GFlop/s.

The results are shown in Figures 5.5 and 5.6 for Tesla C2050, a Fermi-generation card, and Tesla K10, a Kepler-generation card. For each stencil program, we show the achieved GStencils/sec both with and without the memory transfer times. The Compute Only results show the performance of the generated code on the device, while the Compute+Transfer results show the expected performance if the data is not already resident on the GPU device. For all 2D programs, a problem size of
6000 × 6000 was used; for all 3D programs, a problem size of 300 × 300 × 300 was used. In all cases, the computation was performed over 100 iterations.

5.5.2 Comparison with Automatically Optimized CPU Code

In this experiment, we compare the performance of OverTile-generated GPU code with Pochoir [60], a state-of-the-art code optimizer for stencils on multi-core processors. The set of stencil programs used for this experiment is the set of all previous stencil programs that are recognized by the Pochoir compiler. The remaining stencil programs can still be passed through the Pochoir compiler, but are not optimized and are hence not considered here.

For each benchmark, we produce a C++ version written in Pochoir’s embedded DSL that is optimized by the Pochoir compiler and compiled with Intel Composer XE 2013 (13.0.0) with Cilk Plus [13] extensions. The Pochoir optimizer uses Cilk Plus to expose parallelism. The compiled benchmarks were then executed on a dual-socket Intel Xeon E5630 using 8 threads (one for each physical core).

An equivalent stencil program was then written in the OverTile Stencil Specification language, and compiled with the OverTile compiler driver with the CUDA back-end. The NVIDIA CUDA Compiler (nvcc) version 4.2 was used to compile the resulting code. Auto-tuning was performed to select an optimal block and tile size to use for each benchmark on each GPU card. The results of this experiment are shown in Figures 5.7 and 5.8. For each benchmark, the achieved GStencils per second is presented. As before, the Compute Only results show the performance of the generated code on the device, while the Compute+Transfer results show the expected performance if the data is not already resident on the GPU device. For all
Figure 5.7: Performance evaluation of OverTile-generated GPU kernels vs. Pochoir, a state-of-the-art stencil compiler for multi-core CPUs. The CPU benchmarks were executed on a dual-socket Xeon E5630 with 8 threads. (single-precision)

2D programs, a problem size of 6000 × 6000 was used; for all 3D programs, a problem size of 300 × 300 × 300 was used. In all cases, the computation was performed over 100 iterations.

5.5.3 Parameter Sensitivity

The choice of block and tile size plays an important role in the overall performance of the generated GPU stencil programs. Block and spatial tile sizes affect the
Figure 5.8: Performance evaluation of OverTile-generated GPU kernels vs. Pochoir, a state-of-the-art stencil compiler for multi-core CPUs. The CPU benchmarks were executed on a dual-socket Xeon E5630 with 8 threads. (double-precision)

percentage of halo vs non-halo threads for a fixed time tile size, and larger time tile sizes reduce the required global memory traffic but require more redundant computation. Therefore, the global memory savings from larger time tile sizes may be offset by the increase in computation. These competing factors make the choice of block and tile size for stencil programs a non-trivial decision.

Figure 5.9 shows the performance of the Jacobi 2D and Heat 2D stencil programs on a GTX 280, a Tesla C2050, and a Tesla K10 for various time tile sizes. The block
Figure 5.9: Performance variation with respect to changes in time tile size for Jacobi 2D and Heat 2D stencil programs. Problem size: $4000^2$

and spatial tile sizes are held constant. For each stencil program, four combinations of block/spatial tile size are used: $[32 \times 8, 64 \times 8] \times [2, 6]$, where 2 and 6 represent the spatial tile size in the Y-dimension.
Figure 5.10: Performance variation with respect to changes in spatial tile size (Y) for Jacobi 2D and Heat 2D stencil programs. Problem size: $4000^2$

To evaluate spatial tiling, Figure 5.10 shows the performance of the Jacobi 2D and Heat 2D stencil programs on a GTX 280, a Tesla C2050, and a Tesla K10 for various spatial tile sizes in the Y-dimension. The block and time tile sizes are held constant. For each stencil program, four combinations of block/spatial tile size
Figure 5.11: Performance counter data for Jacobi 2D stencil on Tesla C2050 for a range of time tile sizes.

are used: \([32 \times 8, 64 \times 8] \times [2, 6]\), where 2 and 6 represent the time tile size in the Y-dimension.

5.5.4 CDSC Medical Imaging Benchmarks

An important motivation for this work is optimizing the medical imaging pipeline from the Center for Domain-Specific Computing. This pipeline includes several image processing stages that are implemented as stencil operations. These stages have been converted to the OverTile Stencil Specification language and compiled into GPU kernels. The performance of the generated kernels was then compared with the reference GPU implementations of these stencils, compiled with the Intel Composer XE 2013 compiler (version 13.0.0). The results of this experiment are shown in Figure 5.12.
The 2D stencils show significant speedup over the reference C implementations. Time tiling is particularly effective for reducing the memory bandwidth requirement. However, the 3D stencil does not benefit as much due to the increased redundancy in the computation. As previously shown, the computation redundancy in applying overlapped tiling to 3D stencils offsets the benefits of time tiling.

Figure 5.12: Performance results for CDSC medical imaging benchmarks.
Case Study: Rician Denoise 3D

The original implementation of the Rician Denoise 3D imaging pipeline stage offered an interesting optimization challenge. The stencil program, as translated from the original C++ source and shown in the OverTile specification language in Listing 5.1, involves the use of three fields: $u$, the output image; $f$, the input image; and $g$, a temporary field. In each iteration of the stencil, both $g$ and $u$ are updated with new values. Both of these stencil functions are order-1 stencils, and the generated code will need to use shared memory to store local values for both.

This has two implications on the performance of the generated code. First, storing local values for two fields doubles the amount of shared memory needed for a thread block and therefore decreases the maximum size of the thread block. This limits occupancy and the amount of parallelism that is available within an SM to hide latencies. Second, the first stencil function (the update to $g$) must finish before the second stencil function starts for a particular thread block, which forces a thread synchronization point to be emitted by the code generator. This also limits the available parallelism by forcing threads to wait at this barrier.

To help alleviate these issues, we created a new version of the stencil program that eliminates the $g$ array. In each stencil iteration, the values of $g$ are created and then immediately consumed by the update to field $u$. Therefore, we do not need to keep $g$ around after the stencil iteration and we do not need to allocate store for it. Instead, we compute each needed value of $g$ as needed in the update of $u$. While this reduces the shared memory requirement for the generated code as well as eliminating a thread synchronization point, it does significantly increase the computation workload of the kernel. Instead of neighboring threads cooperating
Table 5.2: Rician Denoise 3D performance results before and after the elimination of field g.

to produce the needed region of g, each thread must compute its own region. This increases the amount of redundant computation between threads.

The transformed code is shown in Listing 5.2, and the relative performance of the two versions, shown in GStencils/sec, are given in Table 5.2. We see that the increase in computational workload is easily offset of the increase in parallelism exposed by a lower shared memory requirement and fewer thread synchronization points. On the Tesla C2050, we see a 1.81x speed-up in single-precision and a 2.37x speed-up in double-precision.

5.6 Conclusion

In this chapter, we have introduced an automatic code generation scheme for stencil computations on GPU architectures. This scheme uses overlapped tiling to provide efficient time tiling on GPU architectures, which are massively threaded but are susceptible to performance degradation due to branch divergence and a lack of memory coalescing. We have shown that our scheme produces high performance code on a variety of GPU devices for many stencil programs. We further performed an analysis of the resulting code for various time tile sizes to identify the limiting factors, showing that global memory access is the limiting factor for
smaller time tile sizes, and computational overhead is the limiting factor for larger time tile sizes.
Listing 5.1 Original Rician Denoise 3D stencil program.

```plaintext
program rician3d is
  grid 3
  field G double inout
  field U double inout
  field F double in
  F[0:0][0:0][0:0] = F[0][0][0]
  G[1:1][1:1][1:1] =
    let left  = ((U[0][0][0] - U[0][0][-1])*(U[0][0][0] - U[0][0][-1])) in
    let right = ((U[0][0][0] - U[0][0][1])*(U[0][0][0] - U[0][0][1])) in
    let top   = ((U[0][0][0] - U[0][-1][0])*(U[0][0][0] - U[0][-1][0])) in
    let bottom= ((U[0][0][0] - U[0][1][0])*(U[0][0][0] - U[0][1][0])) in
    let back  = ((U[0][0][0] - U[-1][0][0])*(U[0][0][0] - U[-1][0][0])) in
    let front = ((U[0][0][0] - U[1][0][0])*(U[0][0][0] - U[1][0][0])) in
    let epsilon = 1.0e-20 in
    rsqrt(epsilon + right + left + top + bottom + back + front)
  U[1:1][1:1][1:1] =
    let DT   = 5.0 in
    let sigma = 1.00001 in
    let sigma2 = sigma*sigma in
    let lambda = 1.00001 in
    let gamma  = lambda/sigma2 in
    let r_inner = U[0][0][0]*F[0][0][0]/sigma2 in
    let r       = (r_inner* (2.38944 + r_inner*(0.950037 + r_inner))) /
                   (4.65314 + r_inner* (2.57541 + r_inner*(1.48937 + r_inner))) in
    let left  = U[0][-1][0]*G[0][-1][0] in
    let right = U[0][1][0]*G[0][1][0] in
    let top   = U[0][0][-1]*G[0][0][-1] in
    let bottom= U[-1][0][0]*G[-1][0][0] in
    let back  = U[1][0][0]*G[1][0][0] in
    let front = U[0][0][0]*G[0][0][0] in
    (U[0][0][0] + DT*(right + left + top + bottom + back + front + gamma*F[0][0][0]*r)) /
     (1.0 + DT*(G[0][0][1] + G[0][0][-1] + G[-1][0][0] + G[0][1][0] +
                   G[-1][0][0] + gamma))
```

100
Listing 5.2 Transformed Rician Denoise 3D stencil program.

```plaintext
program rician3d is

grid 3
field U double inout
field F double in

U[1:1][1:1][1:1] =

let DT = 5.0 in
let sigma = 1.00001 in
let sigma2 = sigma*sigma in
let lambda = 1.00001 in
let r_inner = U[0][0][0]*F[0][0][0]/sigma2 in
let r = (r_inner*(2.38944 + r_inner*(0.950037+r_inner))) / (4.65314 + r_inner*(2.57541 + r_inner*(1.48937 + r_inner))) in
let epsilon = 1.0e-20 in

let left_1_0_0 = ((U[ 1][0][0] - U[1][0][-1])*(U[1][0][0] - U[1][0][-1])) in
let right_1_0_0 = ((U[ 1][0][0] - U[1][0][1])*(U[1][0][0] - U[1][0][1])) in
let top_1_0_0 = ((U[ 1][0][0] - U[1][-1][0])*(U[1][0][0] - U[1][-1][0])) in
let bottom_1_0_0 = ((U[1][0][0] - U[1][1][0])*(U[1][0][0] - U[1][1][0])) in
let back_1_0_0 = ((U[ 1][0][0] - U[0][0][0])*(U[1][0][0] - U[0][0][0])) in
let front_1_0_0 = ((U[ 1][0][0] - U[2][0][0])*(U[1][0][0] - U[2][0][0])) in
let g_1_0_0 = rsqrt(epsilon + right_1_0_0 + left_1_0_0 + top_1_0_0 +
    bottom_1_0_0 + back_1_0_0 + front_1_0_0) in

(* Compute for g[-1][0][0] ... *)
(* Compute for g[0][1][0] ... *)
(* Compute for g[0][-1][0] ... *)
(* Compute for g[0][0][1] ... *)
(* Compute for g[0][0][-1] ... *)

let left = U[-1][0][0]*g_0_m1_0 in
let right = U[ 0][1][0]*g_0_1_0 in
let top = U[ 0][0][-1]*g_0_0_m1 in
let bottom = U[0][0][1]*g_m1_0 in
let back = U[-1][0][0]*g_m1_0_0 in
let front = U[1][0][0]*g_1_0_0 in

(U[0][0][0] + DT*(right + left + top + bottom + back + front + gamma*F[0][0][0]*r)) /
    (1.0 + DT*(g_0_0_1 + g_0_0_m1 + g_0_m1_0 + g_0_1_0 + g_m1_0_0 + g_1_0_0 + gamma))
```

101
CHAPTER 6

Performance Modeling for GPU Stencil Code

In this chapter, a modeling technique is presented with the goal of determining the relative performance of generated stencil code for various block and tile sizes.

6.1 Motivation

The performance of tiled code on GPU architectures is very dependent on the choice of block and tile sizes. The code generation algorithms presented in the previous chapter are parameterized on a choice for block and tile sizes, but the set of all possible configurations is quite large (thousands of potentially viable sizes). A compiler that implements these algorithms could use auto-tuning to select an optimal block and tile size, but such a process can be time-consuming. Instead, we aim to build an analytic performance model that a compiler can use to automatically select a near-optimal block and tile size at compile-time.

One metric for determining device efficiency is occupancy, which is the ratio of active threads per multi-processor to the total number of threads that can be active on the multi-processor. Generally, kernels with higher occupancy are able to better hide the latency of memory and arithmetic instructions by giving the thread scheduler more parallel work from which to pick ready threads. However, higher
occupancy does not necessarily translate to higher performance. Once the latency of individual instructions can be hidden with sufficient parallelism, then the benefits of increasing parallelism vanish. Further, having a larger number of threads per multi-processor can increase the memory working set size for the multi-processor, leading to more cache misses. Figure 6.1 shows the elapsed time and occupancy for several configurations of a Jacobi 2D kernel. The relationship between elapsed time and occupancy is clearly not very strong, and the correlation for the data is only 29.9%.

Hardware limitations can constrain the number of threads that can be active on a Streaming Multiprocessor (SM) and limit the achievable occupancy, as shown in Table 6.1. For each constraint, the maximum number of threads that can be active on an SM is dictated by the per-thread or per-block resource usage of the kernel.
Because of these limitations, choices for block and tile sizes which may improve per-thread efficiency may also ultimately decrease performance due to a reduction in the total number of active threads on the device.

### 6.2 Kernel Execution Estimation

To estimate the execution time for a GPU kernel, we create an analytic model that attempts to determine total execution time based on the memory and computation patterns found in the kernel. Since the model is to be used in a compiler to select a near-optimal block and tile size for tiled stencil code, we are able to focus our model on the particular kernel code generated from the algorithms in the previous chapter instead of dealing with the general problem of arbitrary kernel code. Since the generated code has constant loop bounds and known memory access patterns, we can develop our model without the need for code analysis or assumptions about run-time behavior.

Figure 6.2 shows the form of generated stencil kernels. Phase 1 performs kernel initialization like base pointer computations and phase 2 performs the first time step of the time tile. In this phase, data is read from global memory, but some stencils that involve multiple functions may cache intermediate results in shared memory.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>GTX 280</th>
<th>Tesla C2050</th>
<th>Tesla K10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>1.3</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Registers Per Block</td>
<td>16384</td>
<td>32768</td>
<td>65536</td>
</tr>
<tr>
<td>Warps Per SM</td>
<td>32</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>Threads Per SM</td>
<td>1024</td>
<td>1536</td>
<td>2048</td>
</tr>
<tr>
<td>Shared Memory Per SM</td>
<td>16384</td>
<td>49152</td>
<td>49152</td>
</tr>
</tbody>
</table>

**Table 6.1:** Hardware constraints for NVIDIA GPUs
memory. After this phase, the result of the computation in the first time step is stored in shared memory for all fields and a barrier is issued to ensure threads do not progress to a later stage and read shared memory before all threads have written their values to shared memory. At this point, and kernel may branch directly to phase 3 if no time tiling is desired.

Phase 3 represents the time-tiled computation. This phase is executed \((T - 1)\) times, where \(T\) is the time tile size. All data is read from shared memory and all results are written back to shared memory, after which a branch back to the beginning of phase 3 occurs. After all time steps have been executed, the kernel branches...
to phase 4. In phase 4, all non-halo threads write back their results to global memory. This step usually introduces divergence in the kernel since non-halo threads may be in the same warp as halo threads. However, the impact of divergence is low.

The barriers between stages enforce thread synchronization that also makes it easier for us to model the performance of the kernel as a whole. We know that the warp scheduler will not interleave instructions from before and after the barrier in the same block, so we can model each phase independently and simply sum the estimated execution times for each phase.

6.2.1 Phase 2 Modeling

Modeling the performance of phase 2 requires us to model global and shared memory access, as well as the latency of arithmetic instructions. Especially in the presence of spatial timing where a single thread processes multiple field elements, the GPU compiler will try to schedule all loads first followed by the compute instructions. This gives the warp schedulers the best chance to overlap loads and compute, and hide the latency of load instructions. Additionally, the run-time behavior of a block is dependent not only on its threads but also on the threads in blocks that are concurrently executing on the same SM. We must therefore consider all threads for all active blocks on an SM when we model phase 2.

The model we use assumes that all load instructions will be issued first by the warp scheduler, followed by all compute instructions. As a lower bound on execution time, we can use the time required just to issue all instructions:
where $c_{ld}/c_{op}/c_{sfu}$ is the number of cycles required to issue a load, floating-point instruction, and special-function unit instruction, respectively, for a warp, $k_{ld}/k_{op}/k_{sfu}$ is the number of loads, floating-point instructions, and special-function unit instructions, respectively, for a warp, and $W$ is the total number of active warps on an SM. If sufficient global memory bandwidth was available, this would be an accurate model for execution cycles since the needed data would be available when a compute instruction was issued. However, global memory bandwidth on GPU devices is often not sufficient to feed computations, and we therefore need to model execution cycles in the presence of bandwidth constraints.

To model the time required to feed memory requests, we adopt a simple queuing model where a request to global memory is serviced after $L_{gmem}$ cycles. Assuming the warp scheduler continues to issue load instructions while the first load is still waiting for completion, the memory system will not require an additional $L_{gmem}$ cycles for subsequent loads. Instead, after the first $L_{gmem}$ cycles, then each queued load will be serviced after $B_{gmem}$ cycles, which is derived from the available bandwidth. We model shared memory similarly with $L_{smem}$ and $B_{smem}$ constants.

Since phase 2 may involve both global and shared memory loads, we must consider the number of cycles required to service both types of requests. However, global and shared memory loads may be serviced at the same time allowing us to consider only the one that contributes more cycles, leading us to the following equation for bandwidth-constraint execution cycles: 

$$L_{p2} = c_{ld} \times k_{ld} \times W + c_{op} \times k_{op} \times W + c_{sfu} \times k_{sfu} \times W$$ (6.1)
\[ B_{p2} = L_{\text{gmem}} + c_{\text{id}} + \max(B_{\text{gmem}} * k_{\text{id, gmem}} * W, B_{\text{smem}} * k_{\text{id, smem}} * W) \quad (6.2) \]

Now that we have estimates for both latency-bound and boundwidth-bound cases, we can derive an execution estimate for phase 2 as:

\[ t_{p2} = \max(L_{p2}, B_{p2}) \quad (6.3) \]

### 6.2.2 Phase 3 Modeling

Modeling phase 3 is equivalent to modeling phase 2 without any global memory accesses. Therefore, we can just remove the global memory terms from the phase 2 equations:

\[ L_{p3} = c_{\text{id}} * k_{\text{id}} * W + c_{\text{op}} * k_{\text{op}} * W + c_{\text{sfu}} * k_{\text{sfu}} * W \quad (6.4) \]

\[ B_{p3} = L_{\text{smem}} + c_{\text{id}} + B_{\text{smem}} * k_{\text{id, smem}} * W \quad (6.5) \]

\[ t_{p3} = \max(L_{p3}, B_{p3}) \quad (6.6) \]

where \( c_{\text{id}} / c_{\text{op}} / c_{\text{sfu}} \) is the number of cycles required to issue a load, floating-point instruction, and special-function unit instruction, respectively, for a warp, \( k_{\text{id}} / k_{\text{op}} / k_{\text{sfu}} \) is the number of loads, floating-point instructions, and special-function unit instructions, respectively, for a warp, and \( W \) is the total number of active warps on an SM.
6.2.3 Phase 4 Modeling

Phase 4 contains only the global memory writes for non-halo threads. As an estimate for total execution cycles, we assume each thread performs the write and consider the time required by the memory system to service the writes. This amounts to an equivalent of the bandwidth equation for phases 2 and 3:

\[ t_{p4} = L_{gmem} + c_{st} + B_{gmem} \ast k_{st} \ast W \] (6.7)

6.2.4 All-Phase Modeling

We can now assemble the equation to estimate the total execution cycles for a single set of active blocks on an SM. For convenience, we use the term \textit{stage} to denote a set of active blocks on an SM.

\[ t_{stage} = t_{p2} + (T - 1)t_{p3} + t_{p4} \] (6.8)

Note that we omit an estimation for phase 1. In our experience, the total execution cycles for kernel initialization does not vary much between configurations. Since we are most concerned with ordering configurations in relative performance, an estimation for phase 1 is not needed.

To model the entire kernel execution, we must also consider how many stages are actually needed. The choice of block and tile sizes determines how many field points are updated in each block, which can be used to determine the total number of blocks needed to cover the entire field. For each dimension \( i \), we compute the number blocks needed in that dimension:
\( K_{\text{blocks},i} = \left\lceil \frac{N_i}{\text{RealPerBlock}(i)} \right\rceil \)  

(6.9)

where \( \text{RealPerBlock}(i) \) returns the number of field elements which are updated by a block in dimension \( i \). This function is dependent on the stencil specification, and the choice of block and tile sizes. The ceiling function is used to ensure we have enough blocks even if the problem size is not evenly divisible by \( \text{RealPerBlock}(i) \).

We can now determine the total number of blocks as:

\[
K_{\text{blocks}} = \prod_i K_{\text{blocks},i} 
\]

(6.10)

Assuming a consistent mapping of blocks to SMs, we can then determine the total number of blocks processed by an SM as:

\[
K_{\text{SM}} = \left\lceil \frac{K_{\text{blocks}}}{N_{\text{SM}}} \right\rceil 
\]

(6.11)

where \( N_{\text{SM}} \) is the total number of SMs in the device. To determine the number of stages needed, we must first determine how many blocks can be active on an SM at once. The number of blocks active on an SM is influenced by several architectural limitations:

- Register usage per block
- Shared memory usage per block
- Maximum warps per SM

We therefore model the total number of active blocks as a minimum across all constraints:
where $W_{\text{max}}$ is the maximum number of active warps per SM, $W_{\text{block}}$ is the number of warps per block, $N_{\text{shared}}$ is the total amount of shared memory per SM (in bytes), $N_{\text{shared,block}}$ is the shared memory requirement for a block, $N_{\text{regs}}$ is the total number of registers available on an SM, and $N_{\text{regs,block}}$ is the register usage for a block (register usage per thread times threads per block). If we want to select a block and tile size without first compiling all possible combinations, then register usage per thread is not available to us. In this case, we omit the register constraint which can decrease the accuracy of the model. This is a trade-off between compile time and accuracy.

We can now determine the total number of stages as:

$$K_{\text{stages}} = \left\lfloor \frac{K_{\text{SM}}}{K_{\text{active}}} \right\rfloor$$  \hfill (6.16)

The total execution cycles for a kernel invocation is then:

$$t_{\text{kernel}} = K_{\text{stages}} * t_{\text{stage}}$$  \hfill (6.17)
6.3 Model Calibration

The accuracy of the presented model is dependent on reasonable choices for the architecture-specific parameters. This includes $L_{gmem}$, $B_{gmem}$, $L_{smem}$, $B_{smem}$, $c_{op}$, $c_{ld}$, and $c_{sfu}$. The global and shared memory latency and bandwidth terms are derived by creating micro-benchmarks that mimic simple memory accesses. However, we must be careful to ensure similar access patterns to those seen in generated stencil code. Specifically, most global memory accesses are not perfectly aligned to cache line sizes, so two transactions are needed to service a global memory load for a warp even though the accesses are guaranteed to be consecutive (e.g. coalescable). Similarly, our generated shared memory access patterns are guaranteed to avoid bank conflicts so we can ignore the case of bank conflicts in our micro-benchmarks.

In our micro-benchmarks, the CUDA `clock()` function is used to time the number of cycles needed to service a load. Since this function returns the SM’s clock counter, we take the minimum of all starting clock values and the maximum of all final clock values to derive an approximation to the total number of cycles required to service loads across all warps on an SM.

The cycles-per-instruction parameters $c_{op}$, $c_{ld}$ and $c_{sfu}$ are determined directly from block diagrams of the architecture. For example, on Fermi, there are 32 scalar SPs, 16 load/store units, and 4 special-function units. Therefore, we can issue one warps worth of compute instructions per clock ($c_{op} = 1$), a half-warps worth of loads/stores per clock ($c_{op} = 2$), or one quarter-warps worth of special function unit instructions per clock ($c_{sfu} = 4$). In reality, Fermi has two warp schedulers that can issue a half-warp per clock (ignoring dual issue on GF110), but for simplicity we assume that instead we issue one warp per clock.
6.4 Experimental Evaluation

To evaluate the accuracy of the GPU performance model, we used the model to estimate the execution time for many runs of our suite of stencil benchmarks. We also timed the execution of different kernel phases independently to see how well the model predicted actual execution time for phase 2 (global memory loads, shared memory stores) and phase 3 (shared memory loads and stores). The results for a subset of the runs are shown in Figures 6.3, 6.4, and 6.5 for a Jacobi 2D, a Poisson 2D, and a Total Variation 2D stencil, respectively, on a Tesla C2050. In each
figure 6.4: Model fit for a Poisson 2D stencil on Tesla C2050.

Generally, we see that the model is very accurate for phase 3 execution time (shared memory loads and stores). Shared memory on GPU devices is implemented as a scratchpad memory without caches, making the modeling easier. In addition, we have known, bank conflict-free accesses so we can accurately determine access latencies for shared memory. In contrast, the model is not as accurate for phase 2 kernel execution, which involves global memory access. The Tesla
C2050 uses per-SM caches for reads from global memory, which makes the modeling harder.

Since multiple thread blocks are simultaneously executing on each SM, the cache hit/miss rates are a function of not only the global memory access patterns of the threads, but also of the mapping of blocks to SMs. If two blocks are mapped to the same SM and access the same data around their boundaries, then cache affects could improve the memory access time for the second block. However, the larger memory footprint could also increase the amount of capacity and/or conflict misses between the threads in the two blocks. Since there is no software control

**Figure 6.5:** Model fit for a Total Variation stencil on Tesla C2050.
over the block-to-SM mapping strategy, nor is the mapping strategy known, we are unable to accurately model cache effects.

### 6.4.1 Parameter Sensitivity

To better understand how well the GPU performance model captures the effects of changes to block and tile sizes, we independently looked at the prediction accuracy as a function of spatial tile size, time tile size, and block size. For each run, the other factors were held constant. The results of this experiment are shown in Figures 6.6, 6.7, and 6.8. Note that even though the absolute error may be high in some cases, the trends as parameters are varied are captured. Since the goal of this work is to predict the optimal block and tile sizes, error is acceptable as long as the trends match.
6.4.2 Block/Tile Size Selection Accuracy

The primary goal of the GPU performance model is to enable fast, compile-time selection of optimal block/tile sizes for generated stencil kernels. To test the effectiveness of the model, we used it to estimate the execution time of multiple stencil kernels, and selected the block/tile size with the lowest execution time. We then compared the actual performance of this selection with the space-optimal selection as determined by exhaustive auto-tuning. The results are shown in Table 6.2.

In this table, the average error in elapsed time prediction is shown along with the performance degradation from selecting the model-chosen point, and the performance degradation from auto-tuning from the model-chosen 1% best points. For some benchmarks, the model-selected block/tile size is clearly non-optimal. However, if we consider the top 1% of the model-selected sizes and auto-tune on that
set, the performance degradation is significantly lower. The model can thus complement auto-tuning by providing a guide for which block/tile sizes to examine. If we only need to consider 1% of the total points in the configuration space, then we can auto-tune in only 1% of the time.
6.5 Conclusion

In this chapter, we have introduced a compile-time model-driven tile size selection scheme for a GPU stencil compiler and we’ve shown that our scheme produces high performance code on a variety of GPU devices for many stencil programs, and comes very close to the empirically-determined best tile sizes. Our model-driven approach to performance prediction leads us to automatically select optimal block and tile sizes for generated GPU kernels for OverTile, our GPU stencil compiler, without needing full auto-tuning.
CHAPTER 7

Related Work

There exists a significant body of prior and related work regarding dynamic analysis and high-performance code generation and optimization. We break down this related work into the categories of dynamic analysis for parallelism, GPU code generation and optimization, high-level languages for GPU programming, domain-specific languages for stencil computations, and GPU performance modeling.

7.1 Dynamic Analysis for Parallelism

A number of approaches have been proposed to measure the available parallelism at the statement (or instruction) level (e.g., [2, 26, 27, 36, 42, 49, 50, 56, 61, 67]). The work in [67] considers the limits of instruction-level parallelism (ILP) and how it is affected by several hardware features and compiler optimizations. The approach from [27] focuses on the ILP effects of control dependences and various hardware and compiler means of handling them. In [2], a dynamic dependence graph is first constructed and then ILP is measured on it for several configurations. Typically, in these and similar studies, a program execution trace is first created; next, possible parallel schedules are defined by taking into account the dependences between binary instructions in the trace, under various assumptions.
(e.g., anti and output dependences may be ignored). A notable exception is the work from [26], which does not create a trace and instead instruments the program to compute the parallel schedule online.

Researchers have also considered dynamic analysis of loop-level parallelism, where all iterations of a loop may run concurrently with each other. The approach in [29] generates an execution trace and analyzes it to model the effects of loop-level parallelism. A related technique is applied in the context of speculative parallelization of loops, where shadow locations are used to track dynamic dependences across loop iterations [51]. A few recent approaches of similar nature include [9, 46, 62, 63, 70, 76].

The efficient collection of dynamic data-dependence information has also been explored by previous work. Tallam et al. use run-time control-dependence information to re-construct significant portions of the dynamic data-dependence graph of the program run [58], and have proposed a technique for producing light-weight tracing of multi-threaded code [59]. Zhang et al. use a novel approach to collect compressed profiles from program executions, including data dependence information [73]; they also propose a technique to greatly decrease the cost of maintaining a dynamic dependence graph in the context of dynamic slicing [72, 74].

Automatic parallelization is also closely related to the characterization of vectorization in programs. Techniques to automatically exploit fine-grained parallelism must first determine the dependences between instructions. There is a large body of work on the characterization of dynamic data dependencies (e.g., [62, 77]). Tournavitis et al. [63] use a combination of static and dynamic analysis to exploit parallelism that static analyses alone fail to identify. Rus et al. [52] optimize codes with
data-dependent parallelism by developing minimal checks performed at run-time to select between parallel and sequential versions of the same code.

Automatic SIMD vectorization has become a topic of great interest in recent years and has spawned a large body of work [43, 44, 68]. These approaches use static analysis to analyze the dependences between instructions in a program and convert scalar instructions to vector instructions where the conditions for efficient vectorization are met. To the best of our knowledge, no prior work has addressed the topic of this paper—the development of an approach for dynamic analysis of vectorizability of computations.

7.2 GPU Code Generation and Optimization

There is significant prior work on code generation and optimization for GPU and other heterogeneous architectures [6–8, 25, 34, 39].

In [39], Micikevicius et al. hand-tune a 3-D finite difference computation stencil and achieve an order of magnitude performance increase over existing CPU implementations on GT200-based Tesla GPUs. Datta et al. [15] propose an optimization and auto-tuning framework for stencil computations, targeting multi-core systems. However, they also consider accelerator platforms such as NVidia GPUs and Cell SPUs. They propose that autotuning is necessary to achieve performance levels where the benefits outweigh the cost of sending data across the PCIe bus. Xu et al. [71] look at the effects of tiling on different generations of NVIDIA GPUs and conclude that the tiling needing to obtain good performance on stencil-like computations varies between generations.


7.2.1 Polyhedral Code Optimization

In [6, 7], Baskaran et al. introduce an automatic code generation framework that
uses the polyhedral model to generate CUDA code from affine sequential C code.
Multi-level tiling is used to partition the program iteration space into thread blocks,
where each thread processes a tile in the iteration space. For stencil computations, a
custom, device-specific block-synchronization primitive is used to handle inter-tile
dependencies. This limits the number of thread blocks to the physical number of
streaming multiprocessors in the device. Instead of scheduling many more thread
blocks than multiprocessors, which is common practice in CUDA programming for
hiding memory and instruction latencies, only a limited number of thread blocks
are actually created and a per-block work list is used to iterate over the whole space
of thread blocks.

7.2.2 Synchronization

Thread synchronization is a known bottleneck of stencil computations, espe-
cially on GPU architectures, and there exists significant prior work on reducing the
amount of synchronization that is needed in stencil computations [25, 66]. In [25],
Venkatasubramanian et al. study the Jacobi iterative solver and propose a “wildly
asynchronous” implementation based on chaotic relaxation [10]. In this scheme,
synchronization is not performed after every iteration. Instead, asynchronous ex-
ecution is used to trade higher performance for an increase in the number of iter-
ations needed to reach the fixed point. This leads to efficient (98% of peak) use of
GPU bandwidth.
A similar approach is used by Liu et al. [35] to increase the amount of exploitable parallelism in the presence of tiling. This allows a program to be optimized for both locality and loop-level parallelism.

Overlapped tiling, a technique used in our automatic code generation framework, is introduced in [25]. In this technique, time tiling for stencils is enabled by introducing redundant computations in the halo regions of tiles instead of performing a global synchronization after each iteration. This leads to better utilization of the compute resources available on GPU architectures.

7.2.3 Data-Layout Transformations

In [57], Sung et al. propose an automatic data-layout transformation framework that optimizes the layout of grid data using a model for the target architecture. Their target is structured grid computations, like those found in many stencil computations. However, they require the code to be written using variable-length C99 arrays. They are able to achieve over a 500% speedup on some computations by automatically re-arranging data to maximize the available memory bandwidth.

7.2.4 Multi-GPU Systems

In [54], Spafford et al. evaluate the bottlenecks in multi-GPU NUMA systems. They conducted performance tests on single systems that contained multiple GPU units, as well as on clusters containing GPU nodes. Contention on the PCIe bus, as well as sharing of GPU resources on MPI-based applications were investigated. Babich [3] et al. explore the problems encountered when utilizing MPI on multi-GPU clusters for the QUDA library for Lattice QCD. Their focus is compute nodes
with multiple GPU devices, where the computation at a single compute node must be split up among multiple GPU devices.

### 7.2.5 Memory Hierarchy Optimizations

In [41], Nguyen et al. propose a data blocking scheme that optimizes both the memory bandwidth and computation resources on GPU devices. Instead of performing full 3D blocking for 3D stencils, they perform “2.5D blocking” by using normal spatial tiling in two dimensions and streaming the third dimension as needed. When incorporating time tiling, their approach is very similar to that of Krishnamoorthy [25].

### 7.3 High-Level Languages for GPUs

The Pervasive Parallelism Laboratory at Stanford has been developing DSLs to enable high-performance code execution from high-level specifications [55]. Their work is based on the Scala language [31], which is based on the Java VM. Their DSLs enable high-performance program execution on both multi-core CPUs and GPU architectures.

In [33], Leung et al. propose an extension to a Java VM to automatically offload parallel computations to a GPU device. In their system, a cost model is used to determine if offloading a given piece of code to a GPU device would be profitable. Since the code generation occurs within the Java VM, no program modification is needed to take advantage of their system.

The OpenMP programming model is very popular for exploiting multi-core CPU architectures, and Lee et al. [32] propose an extension to OpenMP, called
OpenMPC, that enables automatic offloading of computations to GPU devices using OpenMP-like directives. By using their OpenMPC directives, they are able to achieve 88% of the performance of hand-tuned CUDA implementations of the same code.

Embedded DSLs have gained popularity recently for their ability to provide a high-level abstraction for programmers while helping the compiler writers to better optimize the code for diverse architectures. In [28], Larsen proposes a declarative language embedded in Haskell that is automatically compiled to CUDA code. The declarative nature of the DSL makes several forms of code optimization easier due to a relaxation of program semantics. In [47], Orchard et al. propose another embedded DSL within Haskell for structured grid computations. They also use a declarative form for specifying the computation, which gives the compiler more information with which to perform GPU-based optimizations.

In [64], Unat et al. propose an automatic C-to-CUDA translator that uses a small set of user-supplied \#pragma annotations to generate CUDA code for stencil computations. Their optimization framework achieves 80% the performance of equivalent hand-coded CUDA code.

### 7.4 Stencil DSL Compilers

Closely related to our work is the Pochoir [60] stencil compiler by Tang et al. that optimizes stencil programs for multi-core CPUs and uses a DSL embedded in C++ to define stencil kernels. The embedded DSL is syntactically-valid C++ and can be compiled by any standards-compliant C++ compiler using the Pochoir template.
library instead of the Pochoir compiler. The Pochoir compiler parses the embedded

PATUS [11] is a stencil compiler proposed by Christen et al. that uses both a
stencil description and a machine mapping description to generate efficient CPU
and GPU code for stencil programs.

7.5 GPU Performance Modeling

In recent years, a large body of work has been produced for performance mod-
eling on GPU architectures. Meng et al. [37, 38] propose a performance model for
the evaluation of ghost zones for stencil computations on GPU architectures. They
consider the effects of tile size selection on the performance of the final code, and
propose an approach based on user provided annotations for GPU code generation,
but do not consider fully automated code generation. Kothapalli et al. [23] use a
combination of existing models to create a complete model for the GT200-generation
of NVIDIA GPUs. However, this model does not take into account caches that are
present on later-generation cards.

The problem of formulating practical performance models for GPGPUs has at-
ttracted considerable research attention over the last few years. This body of work
could be roughly distinguished in two main categories based on the modelling ap-
proaches used.

The studies from Hong et al., Sim et al. and Baghsorkhi et al. [4, 18, 53] are
based on theoretical GPU abstractions which are then validated through micro-
benchmarks. In particular, Hong et al. [18, 53] introduces the concepts of Memory
Warp Parallelism (MWP) and Compute Warp Parallelism (CWP) as a way of expressing
global memory bandwidth or instruction throughput bottlenecks while Baghsorkhi et al. [4] defines the notion of work flow graphs that are used to model the interactive effects of different performance factors and facilitate auto-tuning. Recently, Hong et al. [19] have extended their model to incorporate power modeling.

On the other hand our work is closely related to Zhang and Owens approach [75] that try to quantitatively analyse hardware components and corresponding bottlenecks through micro-benchmarks and subsequently formulate suitable models - instead of the other way around. Other hardware-oriented studies concentrate on revealing low level hardware mechanisms through simulation [5] or micro benchmarks [69]. None of those studies though provide a way of statically estimating the runtime of a GPU computation kernel. In particular, Zhang and Owens [75] use runtime profiling of synthetic benchmarks in order to estimate the actual global memory performance of a kernel while Wong et al. [69] do not provide a runtime estimation and the simulation work [5] implies a much higher modeling cost.
CHAPTER 8

Future Work

In this chapter, we present ideas for improvement of the previously-presented research that make up the future research goals for this work.

8.1 Dynamic Analysis for Vectorization Potential

The primary issue with the proposed dynamic analysis technique is scalability. Performing a graph-based analysis of the access patterns of a program does not currently scale well to large programs (e.g. SPEC benchmarks with reference input). To address this issue, a few possible topics of research are: trace compression and course-grain instrumentation events.

In the current implementation, there is a trade-off between analysis speed and memory usage. To overcome the memory usage limitations, we can always use hard disk storage such as on-disk databases to store trace and analysis data. However, such an approach would greatly increase the analysis time. On the other hand, we can store all trace and analysis data in memory, which greatly improves the analysis time, but we quickly exhaust the amount of available memory even for smaller applications.
The proposal for this work is to improve the performance of the analysis, both in terms of memory usage and analysis time, and allow it to be used to analyze sections of large programs on reasonable data set sizes. This involves devising a way to store the trace more efficiently while not impacting the analysis time. Techniques for shortening the analysis time will also be explored.

### 8.2 GPU Code Generation from High-Level Specifications

In this work, the problem of high-performance GPU code generation is approached by proposing an automatic code generation technique for generating GPU kernels for arbitrary stencil code. This technique can be used to generate GPU kernel code from high-level specifications such as those found in domain-specific languages.

Automatic code generation for overlapped tiling has produced promising results for GPU stencil code, but a focus of our future work will be to improve the quality of the generated code by (1) taking advantage of low-level optimizations that can improve the performance of the GPU code such as memory coalescing and SM utilization, and (2) improving the GPU performance model to more accurately select optimal block/tiles sizes. Three-dimensional stencils often do not benefit from overlapped tiling due to the computational overhead associated with this form of time tiling, so other optimizations are needed to improve the performance of the generated GPU kernels.

The selection of time tile size, GPU block size, number of elements to process per thread, and the use of shared/local memory as a cache can have a large impact on
the performance of the code. A more accurate model is would improve the quality of the generated code in the absence of auto-tuning.

8.3 Hybridization of Dynamic Analysis and GPU Code Optimization

Another facet of this work is to merge the dynamic analysis technique with our GPU code generation framework. The vectorization analysis can be extended to identify sections of code that would optimize well on GPU architectures, as well as providing feedback on the access patterns of the code which can be used to help the distribution of the code across thread blocks in a GPU implementation.
CHAPTER 9

Conclusion

The work presented in this document has explored both the use of dynamic analysis to characterize the vectorization potential of sequential programs and the optimization of GPU kernel code for scientific applications utilizing stencil iteration methods. The vectorization analysis work provides meaningful insights into program behavior that can be used by both applications developers to optimize their programs and by compiler writers to identify vectorization opportunities that are missed by compilers. The output of the analysis exposes portions of code that would benefit from vectorization, especially portions of code that are not auto-vectorized by vendor compilers.

The GPU optimization work develops a code generation framework that implements efficient time-tiling for GPU architectures. This work is implemented in a GPU compiler that generates high-performance GPU code from a stencil specification language. A GPU performance model is also developed to help select optimal block/tile sizes. The model can either be used to choose a block/tile size, or help guide auto-tuning. Directions for future work have also been presented.
APPENDIX A

Generated Code

This appendix contains automatically generated code that was produced by the OverTile code generation framework.

A.1 Jacobi 2D 5-Point

The following code is the input and output to the OverTile stencil compiler for a Jacobi 2D stencil program.

Stencil Specification:

```plaintext
program j2d is
  grid 2
  field A float inout
  A = @[1:$-1][1:$-1] = 0.2*(A[-1][0]+A[0][0]+A[1][0]+A[0][1]+A[0][-1])
```

OverTile Output:

```plaintext
//
// Generated by OverTile
//
// Description:
// CUDA device code

__global__ static void ot_kernel(float *In_A, float *Out_A, int Dim_0, int Dim_1) {
  __shared__ float Shared_A[2244];
  const int Halo_Left_0 = 3;
```
const int Halo_Right_0 = 3;
const int Halo_Left_1 = 3;
const int Halo_Right_1 = 3;
int real_per_block_0 = 1*blockDim.x - Halo_Left_0 - Halo_Right_0;
int real_per_block_1 = 4*blockDim.y - Halo_Left_1 - Halo_Right_1;
const int array_size = Dim_0 * Dim_1;
const int ts_0 = 1;
const int ts_1 = 4;
float Buffer_A[4][1];
int max_left_offset_0 = 1;
int max_right_offset_0 = 1;
innt shared_size_0 = ts_0*blockDim.x + 2;
int max_left_offset_1 = 1;
int max_right_offset_1 = 1;
innt shared_size_1 = ts_1*blockDim.y + 2;
int AddrOffset;

// Kernel init
int local_0 = threadIdx.x;
int group_0 = blockIdx.x;
innt tid_0 = group_0 * real_per_block_0 + local_0 - Halo_Left_0;
int local_1 = threadIdx.y;
int group_1 = blockIdx.y;
innt tid_1 = group_1 * real_per_block_1 + local_1*4 - Halo_Left_1;

// First time step
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
  int thisid_0 = tid_0 + elem_0*blockDim.x;
  int thislocal_0 = local_0 + elem_0*blockDim.x;
  for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    if ((thisid_0 >= 1 && thisid_0 < Dim_0 - 1) && (thisid_1 >= 1 && thisid_1 < Dim_1 - 1)) {
      AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0;
      float A_0_m1 = *(In_A + AddrOffset);
      AddrOffset = (thisid_0+0) + (thisid_1+1)*Dim_0;
      float A_0_0 = *(In_A + AddrOffset);
      AddrOffset = (thisid_0+0) + (thisid_1+1)*Dim_0;
      float A_0_p1 = *(In_A + AddrOffset);
      AddrOffset = (thisid_0+1) + (thisid_1+0)*Dim_0;
      float A_p1_0 = *(In_A + AddrOffset);
      AddrOffset = (thisid_0+1) + (thisid_1+0)*Dim_0;
      float A_m1_0 = *(In_A + AddrOffset);
      float Res = (0.2f*(((A_0_m1+A_0_0)+A_0_p1)+A_p1_0)+A_m1_0));
      Buffer_A[elem_1][elem_0] = Res;
    } else if ((thisid_0 >= 0 && thisid_0 < Dim_0) && (thisid_1 >= 0 && thisid_1 < Dim_1) {
      AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0;
      float temp = *(In_A + AddrOffset);
      Buffer_A[elem_1][elem_0] = temp;
    } else {
Buffer_A[elem_1][elem_0] = 0;
}
}
}
_syncthreads();

for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
        AddrOffset = (thislocal_0+max_left_offset_0) +
        (thislocal_1+max_left_offset_1)*shared_size_0;
        *(Shared_A + AddrOffset) = Buffer_A[elem_1][elem_0];
    }
}
_syncthreads();

// Remaining time steps
for (int t = 1; t < 4; ++t) {
    for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
        int thisid_0 = tid_0 + elem_0*blockDim.x;
        int thislocal_0 = local_0 + elem_0*blockDim.x;
        for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
            int thisid_1 = tid_1 + elem_1;
            int thislocal_1 = threadIdx.y*ts_1 + elem_1;
            AddrOffset = ((thislocal_0+0)+max_left_offset_0) +
            ((thislocal_1+0)+max_left_offset_1)*shared_size_0;
            float A_0_m1 = *(Shared_A + AddrOffset);
            AddrOffset = ((thislocal_0+0)+max_left_offset_0) +
            ((thislocal_1+0)+max_left_offset_1)*shared_size_0;
            float A_0_0 = *(Shared_A + AddrOffset);
            AddrOffset = ((thislocal_0+1)+max_left_offset_0) +
            ((thislocal_1+0)+max_left_offset_1)*shared_size_0;
            float A_0_p1 = *(Shared_A + AddrOffset);
            AddrOffset = ((thislocal_0+0)+max_left_offset_0) +
            ((thislocal_1+1)+max_left_offset_1)*shared_size_0;
            float A_p1_0 = *(Shared_A + AddrOffset);
            AddrOffset = ((thislocal_0+1)+max_left_offset_0) +
            ((thislocal_1+0)+max_left_offset_1)*shared_size_0;
            float A_m1_0 = *(Shared_A + AddrOffset);
            float Res = (0.2f*(((A_0_m1+A_0_0)+A_0_p1)+A_p1_0)+A_m1_0));
            Buffer_A[elem_1][elem_0] = Res;
        }
    }
    __syncthreads();
}

for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
int thisid_0 = tid_0 + elem_0*blockDim.x;
int thislocal_0 = local_0 + elem_0*blockDim.x;
for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    if ((thisid_0 >= 1 && thisid_0 < Dim_0 - 1) && (thisid_1 >= 1 && thisid_1 < Dim_1 - 1)) {
        AddrOffset = (thislocal_0+max_left_offset_0) +
          (thislocal_1+max_left_offset_1)*shared_size_0;
        *(Shared_A + AddrOffset) = Buffer_A[elem_1][elem_0];
    }
}
__syncthreads();
}
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
        if ((thislocal_0 >= Halo_Left_0 && thislocal_0 < blockDim.x*ts_0 - Halo_Right_0 &&
            thisid_0 >= 1 && thisid_0 < Dim_0 - 1) && (thislocal_1 >= Halo_Left_1 &&
            thislocal_1 < blockDim.y*ts_1 - Halo_Right_1 && thisid_1 >= 1 && thisid_1 < Dim_1 - 1)) {
            AddrOffset = thisid_0 + thisid_1*Dim_0;
            *(Out_A + AddrOffset) = Buffer_A[elem_1][elem_0];
        }
    }
} // End of kernel

void ot_program(int timesteps, float *Host_A, int Dim_0, int Dim_1) {
    cudaError_t Result;
    int ArraySize = Dim_0*Dim_1;
    float *deviceA_In;
    float *deviceA_Out;
    Result = cudaMalloc(&deviceA_In, sizeof(float)*ArraySize);
    assert(Result == cudaSuccess);
Result = cudaMalloc(&deviceA_Out, sizeof(float)*ArraySize);
assert(Result == cudaSuccess);
float *deviceA_InPtr = deviceA_In;
float *deviceA_OutPtr = deviceA_Out;
Result = cudaMemcpy(deviceA_In, Host_A, sizeof(float)*ArraySize,
cudaMemcpyHostToDevice);
assert(Result == cudaSuccess);
Result = cudaMemcpy(deviceA_Out, deviceA_In, sizeof(float)*ArraySize,
cudaMemcpyDeviceToDevice);
assert(Result == cudaSuccess);
const int Halo_Left_0 = 3;
const int Halo_Right_0 = 3;
const int real_per_block_0 = 64 - Halo_Left_0 - Halo_Right_0;
const int Halo_Left_1 = 3;
const int Halo_Right_1 = 3;
const int real_per_block_1 = 32 - Halo_Left_1 - Halo_Right_1;
dim3 block_size(64, 8);
int num_blocks_0 = Dim_0 / real_per_block_0;
int extra_0 = Dim_0 % real_per_block_0;
num_blocks_0 = num_blocks_0 + (extra_0 > 0 ? 1 : 0);
int num_blocks_1 = Dim_1 / real_per_block_1;
int extra_1 = Dim_1 % real_per_block_1;
num_blocks_1 = num_blocks_1 + (extra_1 > 0 ? 1 : 0);
dim3 grid_size(num_blocks_0, num_blocks_1);
cudaThreadSynchronize();
cudaEvent_t StartEvent, StopEvent;
cudaEventCreate(&StartEvent);
cudaEventCreate(&StopEvent);
cudaEventRecord(StartEvent, 0);
for (int t = 0; t < timesteps; t += 4) {
  ot_kernel«<grid_size, block_size»>(deviceA_InPtr, deviceA_OutPtr, Dim_0, Dim_1);
  std::swap(deviceA_InPtr, deviceA_OutPtr);
}
assert(cudaEventRecord(StopEvent, 0) == cudaSuccess);
assert(cudaEventSynchronize(StopEvent) == cudaSuccess);
Result = cudaMemcpy(Host_A, deviceA_InPtr, sizeof(float)*ArraySize,
cudaMemcpyDeviceToHost);
assert(Result == cudaSuccess);
double Flops = 0.0;
double Points;
Points = (Dim_0-2) * (Dim_1-2);
Flops = Flops + Points * 5.000000e+00;
Flops = Flops * timesteps;
float ElapsedMS;
cudaEventElapsedTime(&ElapsedMS, StartEvent, StopEvent);
double Elapsed = ElapsedMS / 1000.0;
double GFlops = Flops / Elapsed / 1e9;
std::cerr << "GFlops: " << GFlops << "\n";
cudaEventDestroy(StartEvent);
cudaEventDestroy(StopEvent);
cudaFree(deviceA_In);
A.2 Rician 3D

The following code is the input and output to the OverTile stencil compiler for the Rician 3D benchmark from the CDSC medical imaging pipeline.

Stencil Specification:

```
program rician3d is
  grid 3
  field G float inout
  field U float inout
  field F float in
  F[0:$][0:$][0:$] = F[0][0][0]
  G[1:$-1][1:$-1][1:$-1] =
  let left = ((U[0][0][0] - U[0][0][-1])*(U[0][0][0] - U[0][0][-1])) in
  let right = ((U[0][0][0] - U[0][0][1])*(U[0][0][0] - U[0][0][1])) in
  let top = ((U[0][0][0] - U[0][-1][0])*(U[0][0][0] - U[0][-1][0])) in
  let bottom = ((U[0][0][0] - U[0][1][0])*(U[0][0][0] - U[0][1][0])) in
  let back = ((U[0][0][0] - U[-1][0][0])*(U[0][0][0] - U[-1][0][0])) in
  let front = ((U[0][0][0] - U[1][0][0])*(U[0][0][0] - U[1][0][0])) in
  let epsilon = 1.0e-20 in
  rsqrt(epsilon + right + left + top + bottom + back + front)
  U[1:$-1][1:$-1][1:$-1] =
  let DT = 5.0 in
  let sigma = 1.00001 in
  let sigma2 = sigma*sigma in
  let lambda = 1.00001 in
  let gamma = lambda/sigma2 in
  let r_inner = U[0][0][0]*F[0][0][0]/sigma2 in
  let r = (r_inner*(2.38944 + r_inner*(0.950037+r_inner))) / (4.65314 +
                 r_inner*(2.57541 + r_inner*(1.48937 + r_inner))) in
  let left = U[0][-1][0]*G[0][-1][0] in
  let right = U[0][1][0]*G[0][1][0] in
  let top = U[0][0][-1]*G[0][0][-1] in
  let bottom = U[0][0][1]*G[0][0][1] in
  let back = U[-1][0][0]*G[-1][0][0] in
  let front = U[1][0][0]*G[1][0][0] in
```
\[
(U[0][0][0] + DT*(right + left + top + bottom + back + front + gamma*F[0][0][0]*r)) / \\
(1.0 + DT*(G[0][0][1] + G[0][0][-1] + G[-1][0][0] + G[0][1][0] + G[-1][0][0] + \\
G[1][0][0] + gamma))
\]

**OverTile Output:**

```c
// BEGIN OVERTILE GENERATED CODE

// Generated by OverTile

// Description:
// CUDA device code

__global__
static void ot_kernel_rician3d(float *In_G, float *Out_G, float *In_U, float *Out_U, 
    float *In_F, float *Out_F, int Dim_0, int Dim_1, int Dim_2) {

    __shared__ float Shared_G[16+1+1][8+1+1][16+1+1];
    __shared__ float Shared_U[16+1+1][8+1+1][16+1+1];
    __shared__ float Shared_F[16+1+1][8+1+1][16+1+1];

    const int Halo_Left_0 = 1;
    const int Halo_Right_0 = 1;
    const int Halo_Left_1 = 1;
    const int Halo_Right_1 = 1;
    const int Halo_Left_2 = 1;
    const int Halo_Right_2 = 1;

    int real_per_block_0 = 1*blockDim.x - Halo_Left_0 - Halo_Right_0;
    int real_per_block_1 = 1*blockDim.y - Halo_Left_1 - Halo_Right_1;
    int real_per_block_2 = 2*blockDim.z - Halo_Left_2 - Halo_Right_2;

    int array_size = Dim_0 * Dim_1 * Dim_2;
    const int ts_0 = 1;
    const int ts_1 = 1;
    const int ts_2 = 2;

    float Buffer_G[2][1][1];
    float Buffer_U[2][1][1];
    float Buffer_F[2][1][1];

    int max_left_offset_0 = 1;
    int max_right_offset_0 = 1;
    int max_left_offset_1 = 1;
    int max_right_offset_1 = 1;
    int max_left_offset_2 = 1;
    int max_right_offset_2 = 1;

    int AddrOffset;

    __global__
    static void ot_kernel_rician3d(float *In_G, float *Out_G, float *In_U, float *Out_U, 
        float *In_F, float *Out_F, int Dim_0, int Dim_1, int Dim_2) {

        __shared__ float Shared_G[16+1+1][8+1+1][16+1+1];
        __shared__ float Shared_U[16+1+1][8+1+1][16+1+1];
        __shared__ float Shared_F[16+1+1][8+1+1][16+1+1];

        const int Halo_Left_0 = 1;
        const int Halo_Right_0 = 1;
        const int Halo_Left_1 = 1;
        const int Halo_Right_1 = 1;
        const int Halo_Left_2 = 1;
        const int Halo_Right_2 = 1;

        int real_per_block_0 = 1*blockDim.x - Halo_Left_0 - Halo_Right_0;
        int real_per_block_1 = 1*blockDim.y - Halo_Left_1 - Halo_Right_1;
        int real_per_block_2 = 2*blockDim.z - Halo_Left_2 - Halo_Right_2;

        int array_size = Dim_0 * Dim_1 * Dim_2;
        const int ts_0 = 1;
        const int ts_1 = 1;
        const int ts_2 = 2;

        float Buffer_G[2][1][1];
        float Buffer_U[2][1][1];
        float Buffer_F[2][1][1];

        int max_left_offset_0 = 1;
        int max_right_offset_0 = 1;
        int max_left_offset_1 = 1;
        int max_right_offset_1 = 1;
        int max_left_offset_2 = 1;
        int max_right_offset_2 = 1;

        int AddrOffset;

        // Kernel init
        int local_0 = threadIdx.x;
        int group_0 = blockIdx.x;
        int tid_0 = group_0 * real_per_block_0 + local_0 - Halo_Left_0;
        int local_1 = threadIdx.y;
        int group_1 = blockIdx.y;
        int tid_1 = group_1 * real_per_block_1 + local_1*1 - Halo_Left_1;
        int local_2 = threadIdx.z;
        int group_2 = blockIdx.z;
```
int tid_2 = group_2 * real_per_block_2 + local_2*2 - Halo_Left_2;

// First time step
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
        for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
            int thisid_2 = tid_2 + elem_2;
            int thislocal_2 = threadIdx.z*ts_2 + elem_2;
            if ((thisid_0 >= 0 && thisid_0 < Dim_0 - 0) && (thisid_1 >= 0 && thisid_1 < Dim_1 - 0) &&
                (thisid_2 >= 0 && thisid_2 < Dim_2 - 0)) {
                AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 + (thisid_2+0)*Dim_0*Dim_1;
                float F_0_0_0 = *(In_F + AddrOffset);
                float Res = F_0_0_0;
                Buffer_F[elem_2][elem_1][elem_0] = Res;
            } else if (((thisid_0 >= 0 && thisid_0 < Dim_0) && (thisid_1 >= 0 && thisid_1 < Dim_1) &&
                (thisid_2 >= 0 && thisid_2 < Dim_2)) {
                AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 + (thisid_2+0)*Dim_0*Dim_1;
                float temp = *(In_F + AddrOffset);
                Buffer_F[elem_2][elem_1][elem_0] = temp;
            } else {
                Buffer_F[elem_2][elem_1][elem_0] = 0;
            }
        }
    }
}
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
        for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
            int thisid_2 = tid_2 + elem_2;
            int thislocal_2 = threadIdx.z*ts_2 + elem_2;
            Shared_F[thislocal_2+1][thislocal_1+1][thislocal_0+1] = Buffer_F[elem_2][elem_1][elem_0];
        }
    }
}
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;

for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
    int thisid_2 = tid_2 + elem_2;
    int thislocal_2 = threadIdx.z*ts_2 + elem_2;
    if ((thisid_0 >= 1 && thisid_0 < Dim_0 - 1) && (thisid_1 >= 1 && thisid_1 < Dim_1 - 1) &&
        (thisid_2 >= 1 && thisid_2 < Dim_2 - 1)) {
        AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 + (thisid_2+0)*Dim_0*Dim_1;
        float U_0_0_0 = *(In_U + AddrOffset);
        AddrOffset = (thisid_0+1) + (thisid_1+0)*Dim_0 + (thisid_2+0)*Dim_0*Dim_1;
        float U_p1_0_0 = *(In_U + AddrOffset);
        AddrOffset = (thisid_0+0) + (thisid_1+1)*Dim_0 + (thisid_2+0)*Dim_0*Dim_1;
        float U_0_m1_0 = *(In_U + AddrOffset);
        AddrOffset = (thisid_0+0) + (thisid_1-1)*Dim_0 + (thisid_2+0)*Dim_0*Dim_1;
        float U_0_0_m1 = *(In_U + AddrOffset);
        AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 + (thisid_2+1)*Dim_0*Dim_1;
        float U_0_0_p1 = *(In_U + AddrOffset);
        float Res = rsqrtf(((((9.9999996E-21f+((U_0_0_0-U_p1_0_0)*(U_0_0_0-U_p1_0_0)))+
            ((U_0_0_0-U_m1_0_0)*(U_0_0_0-U_m1_0_0)))+((U_0_0_0-U_0_m1_0_0))*(U_0_0_0-U_0_m1_0_0))+
            ((U_0_0_0-U_0_0_m1)*(U_0_0_0-U_0_0_m1)))+((U_0_0_0-U_0_0_p1)*(U_0_0_0-U_0_0_p1)));
        Buffer_G[elem_2][elem_1][elem_0] = Res;
    } else if ((thisid_0 >= 0 && thisid_0 < Dim_0) && (thisid_1 >= 0 && thisid_1 < Dim_1) &&
        (thisid_2 >= 0 && thisid_2 < Dim_2)) {
        AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 + (thisid_2+0)*Dim_0*Dim_1;
        float temp = *(In_G + AddrOffset);
        Buffer_G[elem_2][elem_1][elem_0] = temp;
    } else {
        Buffer_G[elem_2][elem_1][elem_0] = 0;
    }
}
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
        for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
            int thisid_2 = tid_2 + elem_2;
            int thislocal_2 = threadIdx.z*ts_2 + elem_2;
            Shared_G[thislocal_2+1][thislocal_1+1][thislocal_0+1] = Buffer_G[elem_2][elem_1][elem_0];
        }
    }
}
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
        for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
            int thisid_2 = tid_2 + elem_2;
            int thislocal_2 = threadIdx.z*ts_2 + elem_2;
            if ((thisid_0 >= 1 && thisid_0 < Dim_0 - 1) &&
                (thisid_1 >= 1 && thisid_1 < Dim_1 - 1) &&
                (thisid_2 >= 1 && thisid_2 < Dim_2 - 1)) {
                AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 +
                    (thisid_2+0)*Dim_0*Dim_1;
                float U_0_0_0 = *(In_U + AddrOffset);
                AddrOffset = (thisid_0+0) + (thisid_1+1)*Dim_0 +
                    (thisid_2+0)*Dim_0*Dim_1;
                float U_0_0_0_0 = *(In_U + AddrOffset);
                float G_0_0_0_0 = Shared_G[thislocal_2+1+0][thislocal_1+1+1][thislocal_0+1+0];
                AddrOffset = (thisid_0+0) + (thisid_1+1)*Dim_0 +
                    (thisid_2+0)*Dim_0*Dim_1;
                float U_0_0_0_0_0 = *(In_U + AddrOffset);
                float G_0_0_0_0_0 = Shared_G[thislocal_2+1+0][thislocal_1+1+1][thislocal_0+1+0];
                AddrOffset = (thisid_0+1) + (thisid_1+0)*Dim_0 +
                    (thisid_2+0)*Dim_0*Dim_1;
                float U_0_0_0_0_0_0 = *(In_U + AddrOffset);
                float G_0_0_0_0_0_0 = Shared_G[thislocal_2+1+0][thislocal_1+1+1][thislocal_0+1+0];
                AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 +
                    (thisid_2+0)*Dim_0*Dim_1;
                float U_0_0_0_0_0_0_0 = *(In_U + AddrOffset);
                float G_0_0_0_0_0_0_0 = Shared_G[thislocal_2+1+0][thislocal_1+1+1][thislocal_0+1+0];
            }
    }
    else if (((thisid_0 >= 0 && thisid_0 < Dim_0 - 1) &&
        (thisid_1 >= 0 && thisid_1 < Dim_1 - 1) &&
        (thisid_2 >= 0 && thisid_2 < Dim_2 - 1)) {
        AddrOffset = (thisid_0+0) + (thisid_1+0)*Dim_0 +
            (thisid_2+0)*Dim_0*Dim_1;
        float temp = *(In_U + AddrOffset);
        Buffer_U[elem_2][elem_1][elem_0] = temp;
    }
}
} else {
    Buffer_U[elem_2][elem_1][elem_0] = 0;
}
}
}
}
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    }
    for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
        int thisid_2 = tid_2 + elem_2;
        int thislocal_2 = threadIdx.z*ts_2 + elem_2;
    }
    Shared_U[thislocal_2+1][thislocal_1+1][thislocal_0+1] = Buffer_U[elem_2][elem_1][elem_0];
}
__syncthreads();

// Remaining time steps
for (int t = 1; t < 1; ++t) {
    for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
        int thisid_0 = tid_0 + elem_0*blockDim.x;
        int thislocal_0 = local_0 + elem_0*blockDim.x;
        for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
            int thisid_1 = tid_1 + elem_1;
            int thislocal_1 = threadIdx.y*ts_1 + elem_1;
        }
        for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
            int thisid_2 = tid_2 + elem_2;
            int thislocal_2 = threadIdx.z*ts_2 + elem_2;
            float F_0_0_0 = Shared_F[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+0];
            float Res = F_0_0_0;
            Buffer_F[elem_2][elem_1][elem_0] = Res;
        }
    }
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
    for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
        int thisid_1 = tid_1 + elem_1;
        int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    }
    for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
        int thisid_2 = tid_2 + elem_2;
        int thislocal_2 = threadIdx.z*ts_2 + elem_2;
        if ((thisid_0 >= 0 && thisid_0 < Dim_0 - 0) && (thisid_1 >= 0 && thisid_1 < Dim_1 - 0) &&
(thisid_2 >= 0 && thisid_2 < Dim_2 - 0) {
  Shared_F[thislocal_2+1][thislocal_1+1][thislocal_0+1] = Buffer_F[elem_2][elem_1][elem_0];
}

__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
  int thisid_0 = tid_0 + elem_0*blockDim.x;
  int thislocal_0 = local_0 + elem_0*blockDim.x;
  for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
      int thisid_2 = tid_2 + elem_2;
      int thislocal_2 = threadIdx.z*ts_2 + elem_2;
      { float U_0_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+0];
        float U_p1_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+1];
        float U_m1_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+-1];
        float U_0_m1_0 = Shared_U[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_m1 = Shared_U[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_p1 = Shared_U[thislocal_2+1+1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+0];
        float U_p1_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+1];
        float U_m1_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+-1];
        float U_0_m1_0 = Shared_U[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_m1 = Shared_U[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_p1 = Shared_U[thislocal_2+1+1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+0];
        float U_p1_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+1];
        float U_m1_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+-1];
        float U_0_m1_0 = Shared_U[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_m1 = Shared_U[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
        float U_0_0_p1 = Shared_U[thislocal_2+1+1][thislocal_1+1+0][thislocal_0+1+0];
        float Res = rsqrtf((((((9.9999996E-21f+((U_0_0_0-U_p1_0_0)*(U_0_0_0-U_p1_0_0)))+((U_0_0_0-U_m1_0_0)*(U_0_0_0-U_m1_0_0)))+((U_0_0_0-U_0_m1_0)*(U_0_0_0-U_0_m1_0)))+((U_0_0_0-U_0_p1_0)*(U_0_0_0-U_0_p1_0)))+((U_0_0_0-U_0_0_m1)*(U_0_0_0-U_0_0_m1)))+((U_0_0_0-U_0_0_p1)*(U_0_0_0-U_0_0_p1)));
        Buffer_G[elem_2][elem_1][elem_0] = Res;
      }
    }
  }

__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
  int thisid_0 = tid_0 + elem_0*blockDim.x;
  int thislocal_0 = local_0 + elem_0*blockDim.x;
  for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
      int thisid_2 = tid_2 + elem_2;
      int thislocal_2 = threadIdx.z*ts_2 + elem_2;
      if ((thisid_0 >= 1 && thisid_0 < Dim_0 - 1) && (thisid_1 >= 1 && thisid_1 < Dim_1 - 1) &&
          (thisid_2 >= 1 && thisid_2 < Dim_2 - 1)) {
        Shared_G[thislocal_2+1][thislocal_1+1][thislocal_0+1] = Buffer_G[elem_2][elem_1][elem_0];
      }
    }
  }
}

144
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
}
for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
}
for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
    int thisid_2 = tid_2 + elem_2;
    int thislocal_2 = threadIdx.z*ts_2 + elem_2;
}
{
    float U_0_0_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+0];
    float U_0_p1_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+1][thislocal_0+1+0];
    float G_0_p1_0 = Shared_G[thislocal_2+1+0][thislocal_1+1+1][thislocal_0+1+0];
    float U_0_m1_0 = Shared_U[thislocal_2+1+0][thislocal_1+1+-1][thislocal_0+1+0];
    float G_0_m1_0 = Shared_G[thislocal_2+1+0][thislocal_1+1+-1][thislocal_0+1+0];
    float U_m1_0_0 = Shared_U[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
    float G_0_0_m1 = Shared_G[thislocal_2+1+-1][thislocal_1+1+0][thislocal_0+1+0];
    float U_0_0_p1 = Shared_U[thislocal_2+1+1][thislocal_1+1+0][thislocal_0+1+0];
    float G_0_0_p1 = Shared_G[thislocal_2+1+1][thislocal_1+1+0][thislocal_0+1+0];
    float F_0_0_0 = Shared_F[thislocal_2+1+0][thislocal_1+1+0][thislocal_0+1+0];
    float Res = ((U_0_0_0+(5.0f*(((U_0_p1_0*G_0_p1_0)+(U_0_m1_0*G_0_m1_0))+(U_m1_0_0*G_m1_0_0))+(U_p1_0_0*G_p1_0_0))+(U_0_0_m1*G_0_0_m1))+(1.00001f*(1.00001f*1.00001f)*F_0_0_0)*
        (((U_0_0_0+F_0_0_0)/(1.00001f*1.00001f))*(2.3894401f+(((U_0_0_0*F_0_0_0)/(1.00001f*1.00001f))*(0.950037f+((U_0_0_0*F_0_0_0)/(1.00001f*1.00001f)))))/(4.6531401f+(((U_0_0_0+F_0_0_0)/(1.00001f*1.00001f))*(2.5754099f+((U_0_0_0*F_0_0_0)/(1.00001f*1.00001f))*(1.48937f+((U_0_0_0*F_0_0_0)/(1.00001f*1.00001f)))))))/(1.0f+(5.0f*(((G_0_0_0+G_0_0_0)+G_0_0_0)+G_0_0_0)*
        G_0_0_0)+G_0_0_0)*F_0_0_0)+(1.00001f/(1.00001f*1.00001f))); Buffer_U[elem_2][elem_1][elem_0] = Res;
}
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
    int thisid_0 = tid_0 + elem_0*blockDim.x;
    int thislocal_0 = local_0 + elem_0*blockDim.x;
}
for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
}
for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
    int thisid_2 = tid_2 + elem_2;
    int thislocal_2 = threadIdx.z*ts_2 + elem_2;
    if (((thisid_0 >= 1 && thisid_0 < Dim_0 - 1) && (thisid_1 >= 1 && thisid_1 < Dim_1 - 1) && (thisid_2 >= 1 && thisid_2 < Dim_2 - 1)) {
Shared_U[thislocal_2+1][thislocal_1+1][thislocal_0+1] = Buffer_U[elem_2][elem_1][elem_0];
}
}
}
__syncthreads();
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
  int thisid_0 = tid_0 + elem_0*blockDim.x;
  int thislocal_0 = local_0 + elem_0*blockDim.x;
  for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
      int thisid_2 = tid_2 + elem_2;
      int thislocal_2 = threadIdx.z*ts_2 + elem_2;
      if ((thislocal_0 >= Halo_Left_0 && thislocal_0 < blockDim.x*ts_0 - Halo_Right_0 &&
        thisid_0 >= 0 && thisid_0 < Dim_0 - 0) && (thislocal_1 >= Halo_Left_1 &&
        thislocal_1 < blockDim.y*ts_1 - Halo_Right_1 && thisid_1 >= 0 &&
        thisid_1 < Dim_1 - 0) && (thislocal_2 >= Halo_Left_2 &&
        thislocal_2 < blockDim.z*ts_2 - Halo_Right_2 && thisid_2 >= 0 &&
        thisid_2 < Dim_2 - 0)) {
        AddrOffset = thisid_0 + thisid_1*Dim_0 + thisid_2*Dim_0*Dim_1;
        *(Out_F + AddrOffset) = Buffer_F[elem_2][elem_1][elem_0];
      }
    }
  }
}
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
  int thisid_0 = tid_0 + elem_0*blockDim.x;
  int thislocal_0 = local_0 + elem_0*blockDim.x;
  for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
    int thisid_1 = tid_1 + elem_1;
    int thislocal_1 = threadIdx.y*ts_1 + elem_1;
    for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
      int thisid_2 = tid_2 + elem_2;
      int thislocal_2 = threadIdx.z*ts_2 + elem_2;
      if ((thislocal_0 >= Halo_Left_0 && thislocal_0 < blockDim.x*ts_0 - Halo_Right_0 &&
        thisid_0 >= 1 && thisid_0 < Dim_0 - 1) && (thislocal_1 >= Halo_Left_1 &&
        thislocal_1 < blockDim.y*ts_1 - Halo_Right_1 && thisid_1 >= 1 &&
        thisid_1 < Dim_1 - 1) && (thislocal_2 >= Halo_Left_2 &&
        thislocal_2 < blockDim.z*ts_2 - Halo_Right_2 && thisid_2 >= 1 &&
        thisid_2 < Dim_2 - 1)) {
        AddrOffset = thisid_0 + thisid_1*Dim_0 + thisid_2*Dim_0*Dim_1;
        *(Out_G + AddrOffset) = Buffer_G[elem_2][elem_1][elem_0];
      }
    }
  }
}
for (unsigned elem_0 = 0; elem_0 < ts_0; ++elem_0) {
  int thisid_0 = tid_0 + elem_0*blockDim.x;
int thislocal_0 = local_0 + elem_0*blockDim.x;
for (unsigned elem_1 = 0; elem_1 < ts_1; ++elem_1) {
  int thisid_1 = tid_1 + elem_1;
  int thislocal_1 = threadIdx.y*ts_1 + elem_1;
  for (unsigned elem_2 = 0; elem_2 < ts_2; ++elem_2) {
    int thisid_2 = tid_2 + elem_2;
    int thislocal_2 = threadIdx.z*ts_2 + elem_2;
    if ((thislocal_0 >= Halo_Left_0 && thislocal_0 < blockDim.x*ts_0 - Halo_Right_0 &&
         thisid_0 >= 1 && thisid_0 < Dim_0 - 1) &&
        (thislocal_1 >= Halo_Left_1 &&
         thislocal_1 < blockDim.y*ts_1 - Halo_Right_1 &&
         thisid_1 >= 1 &&
         thisid_1 < Dim_1 - 1) &&
        (thislocal_2 >= Halo_Left_2 &&
         thislocal_2 < blockDim.z*ts_2 - Halo_Right_2 &&
         thisid_2 >= 1 &&
         thisid_2 < Dim_2 - 1)) {
      AddrOffset = thisid_0 + thisid_1*Dim_0 + thisid_2*Dim_0*Dim_1;
      *(Out_U + AddrOffset) = Buffer_U[elem_2][elem_1][elem_0];
    }
  }
}
} // End of kernel

// Generated by OverTile
//
// Description:
// CUDA host code
//
#include <iostream>
#include <algorithm>
#include <cassert>

void ot_program_rician3d(int timesteps, float *Host_G, float *Host_U, float *Host_F, int Dim_0, int Dim_1, int Dim_2) {
  cudaError_t Result;
  int ArraySize = Dim_0*Dim_1*Dim_2;
  cudaEvent_t TotalStartEvent, TotalStopEvent;
  cudaEventCreate(&TotalStartEvent);
  cudaEventCreate(&TotalStopEvent);
  cudaEventCreate(&TotalStartEvent);
  cudaEventCreate(&TotalStopEvent);
  cudaEventRecord(TotalStartEvent, 0);
  float *deviceG_In;
  float *deviceG_Out;
  Result = cudaMalloc(&deviceG_In, sizeof(float)*ArraySize);
  assert(Result == cudaSuccess);
  Result = cudaMalloc(&deviceG_Out, sizeof(float)*ArraySize);
  assert(Result == cudaSuccess);
  float *deviceG_InPtr = deviceG_In;
  float *deviceG_OutPtr = deviceG_Out;
  Result = cudaMemcpy(deviceG_In, Host_G, sizeof(float)*ArraySize, cudaMemcpyHostToDevice);
  assert(Result == cudaSuccess);
Result = cudaMemcpy(deviceG_Out, deviceG_In, sizeof(float)*ArraySize, cudaMemcpyDeviceToDevice);
assert(Result == cudaSuccess);
float *deviceU_In;
float *deviceU_Out;
Result = cudaMalloc(&deviceU_In, sizeof(float)*ArraySize);
assert(Result == cudaSuccess);
Result = cudaMalloc(&deviceU_Out, sizeof(float)*ArraySize);
assert(Result == cudaSuccess);
float *deviceU_InPtr = deviceU_In;
float *deviceU_OutPtr = deviceU_Out;
Result = cudaMemcpy(deviceU_In, Host_U, sizeof(float)*ArraySize, cudaMemcpyHostToDevice);
assert(Result == cudaSuccess);
Result = cudaMemcpy(deviceU_Out, deviceU_In, sizeof(float)*ArraySize, cudaMemcpyDeviceToDevice);
assert(Result == cudaSuccess);
float *deviceF_In;
float *deviceF_Out;
Result = cudaMalloc(&deviceF_In, sizeof(float)*ArraySize);
assert(Result == cudaSuccess);
Result = cudaMalloc(&deviceF_Out, sizeof(float)*ArraySize);
assert(Result == cudaSuccess);
float *deviceF_InPtr = deviceF_In;
float *deviceF_OutPtr = deviceF_Out;
Result = cudaMemcpy(deviceF_In, Host_F, sizeof(float)*ArraySize, cudaMemcpyHostToDevice);
assert(Result == cudaSuccess);
Result = cudaMemcpy(deviceF_Out, deviceF_In, sizeof(float)*ArraySize, cudaMemcpyDeviceToDevice);
assert(Result == cudaSuccess);
const int Halo_Left_0 = 1;
const int Halo_Right_0 = 1;
const int real_per_block_0 = 16 - Halo_Left_0 - Halo_Right_0;
const int Halo_Left_1 = 1;
const int Halo_Right_1 = 1;
const int real_per_block_1 = 8 - Halo_Left_1 - Halo_Right_1;
const int Halo_Left_2 = 1;
const int Halo_Right_2 = 1;
const int real_per_block_2 = 16 - Halo_Left_2 - Halo_Right_2;
dim3 block_size(16, 8, 8);
int num_blocks_0 = Dim_0 / real_per_block_0;
int extra_0 = Dim_0 % real_per_block_0;
num_blocks_0 = num_blocks_0 + (extra_0 > 0 ? 1 : 0);
int num_blocks_1 = Dim_1 / real_per_block_1;
int extra_1 = Dim_1 % real_per_block_1;
num_blocks_1 = num_blocks_1 + (extra_1 > 0 ? 1 : 0);
int num_blocks_2 = Dim_2 / real_per_block_2;
int extra_2 = Dim_2 % real_per_block_2;
num_blocks_2 = num_blocks_2 + (extra_2 > 0 ? 1 : 0);
dim3 grid_size(num_blocks_0, num_blocks_1, num_blocks_2);
cudaThreadSynchronize();
cudaEvent_t StartEvent, StopEvent;
cudaEventCreate(&StartEvent);
cudaEventCreate(&StopEvent);
cudaEventRecord(StartEvent, 0);
for (int t = 0; t < timesteps; t += 1) {
    ot_kernel_rician3d«<grid_size, block_size>»(deviceG_InPtr, deviceG_OutPtr, deviceU_InPtr,
    deviceU_OutPtr, deviceF_InPtr, deviceF_OutPtr,
    Dim_0, Dim_1, Dim_2);
    std::swap(deviceG_InPtr, deviceG_OutPtr);
    std::swap(deviceU_InPtr, deviceU_OutPtr);
    std::swap(deviceF_InPtr, deviceF_OutPtr);
}

assert(cudaEventRecord(StopEvent, 0) == cudaSuccess);
assert(cudaEventSynchronize(StopEvent) == cudaSuccess);
Result = cudaMemcpy(Host_G, deviceG_InPtr,
sizeof(float)*ArraySize, cudaMemcpyDeviceToHost);
assert(Result == cudaSuccess);
Result = cudaMemcpy(Host_U, deviceU_InPtr, sizeof(float)*ArraySize, cudaMemcpyDeviceToHost);
assert(Result == cudaSuccess);
Result = cudaMemcpy(Host_F, deviceF_InPtr, sizeof(float)*ArraySize, cudaMemcpyDeviceToHost);
assert(Result == cudaSuccess);
cudaEventRecord(TotalStopEvent, 0);
assert(cudaEventSynchronize(TotalStopEvent) == cudaSuccess);

float Flops = 0.0;

double Points;
Points = (Dim_0-0) * (Dim_1-0) * (Dim_2-0);
Flops = Flops + Points * 0.0000000e+00;
Points = (Dim_0-2) * (Dim_1-2) * (Dim_2-2);
Flops = Flops + Points * 2.5000000e+01;
Points = (Dim_0-2) * (Dim_1-2) * (Dim_2-2);
Flops = Flops + Points * 5.7000000e+01;
Flops = Flops * timesteps;
float ElapsedMS;
cudaEventElapsedTime(&ElapsedMS, StartEvent, StopEvent);
double Elapsed = ElapsedMS / 1000.0;

double GFlops = Flops / Elapsed / 1e9;
std::cerr << "GFlops: " << GFlops << "\n";
std::cerr << "Elapsed: " << Elapsed << "\n";
float TotalElapsedMS;
cudaEventElapsedTime(&TotalElapsedMS, TotalStartEvent, TotalStopEvent);
double TotalElapsed = TotalElapsedMS / 1000.0;

double TotalGFlops = Flops / TotalElapsed / 1e9;
std::cerr << "Total GFlops: " << TotalGFlops << "\n";
std::cerr << "Total Elapsed: " << TotalElapsed << "\n";
cudaEventDestroy(StartEvent);
cudaEventDestroy(StopEvent);
cudaEventDestroy(TotalStartEvent);
cudaEventDestroy(TotalStopEvent);
cudaEventDestroy(TotalStopEvent);
cudaFree(deviceG_In);
cudaFree(deviceG_Out);
cudaFree(deviceU_In);
cudaFree(deviceU_Out);
cudaFree(deviceF_In);
cudaFree(deviceF_Out);


[39] Micikevicius, P. 3d finite difference computation on GPUs using CUDA. GPGPU-2, pp. 79–84.

154


[73] Zhang, X., and Gupta, R. Whole execution traces and their applications. ACM TACO 2, 3 (2005), 301–334.

