COMPILER OPTIMIZATIONS AND AUTOTUNING FOR STENCILS AND GEOMETRIC MULTIGRID

by

Protonu Basu

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2016

Copyright © Protonu Basu 2016

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Protonu Basu**

has been approved by the following supervisory committee members:

Name	Role	Date Approved
Mary Hall	Chair	**7-Dec-2015**
Samuel Williams	Member	**17-Nov-2015**
Rajeev Balasubramonian	Member	**10-Nov-2015**
Martin Berzins	Member	**10-Nov-2015**
Ganesh Gopalakrishnan	Member	**10-Nov-2015**

and by **Ross Whitaker**, Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Stencil computations are operations on structured grids. They are frequently found in partial differential equation solvers, making their performance critical to a range of scientific applications. On modern architectures where data movement costs dominate computation, optimizing stencil computations is a challenging task. Typically, domain scientists must reduce and orchestrate data movement to tackle the memory bandwidth and latency bottlenecks. Furthermore, optimized code must map efficiently to ever increasing parallelism on a chip.

This dissertation studies several stencils with varying arithmetic intensities, thus requiring contrasting optimization strategies. Stencils traditionally have low arithmetic intensity, making their performance limited by memory bandwidth. Contemporary *higher-order* stencils are designed to require smaller grids, hence less memory, but are bound by increased floating-point operations. This dissertation develops communication-avoiding optimizations to reduce data movement in memory-bound stencils. For higher-order stencils, a novel transformation, *partial sums*, is designed to reduce the number of floating-point operations and improve register reuse. These optimizations are implemented in a compiler framework, which is further extended to generate parallel code targeting multicores and graphics processor units (GPUs).

The augmented compiler framework is then combined with autotuning to productively address stencil optimization challenges. Autotuning explores a search space of possible implementations of a computation to find the optimal code for an execution context. In this dissertation, autotuning is used to compose sequences of optimizations to drive the augmented compiler framework. This compiler-directed autotuning approach is used to optimize stencils in the context of a linear solver, Geometric Multigrid (GMG). GMG uses sequences of stencil computations, and presents greater optimization challenges than isolated stencils, as interactions *between* stencils must also be considered.

The efficacy of our approach is demonstrated by comparing the performance of generated code against manually tuned code, over commercial compiler-generated code, and against analytic performance bounds. Generated code outperforms manually optimized codes on multicores and GPUs. Against Intel's compiler on multicores, generated code achieves up to 4x speedup for stencils, and 3x for the solver. On GPUs, generated code achieves 80% of an analytically computed performance bound.

To my parents and sister

CONTENTS

ABSTRACT . . . . **iii**

LIST OF FIGURES . . . . **x**

LIST OF TABLES . . . . **xiv**

ACKNOWLEDGEMENTS . . . . **xv**

CHAPTERS

1. INTRODUCTION **1**

1.1 Geometric Multigrid and Stencil Computations 2
1.2 Classification of Stencil Computations 3
1.2.1 Stencil Coefficient Types 5
1.2.2 Radius of Stencils 7
1.2.3 Common Stencil Iterations 7
1.2.3.1 Jacobi 7
1.2.3.2 Gauss-Seidel and Gauss-Seidel Red-Black 7
1.2.4 Arithmetic Intensity of Stencils 8
1.2.5 Summary of Stencils Optimized 9
1.2.5.1 Higher-Order Methods 9
1.2.5.2 Grid Boundary Conditions 10
1.3 Approaches to Reducing Data Movement 10
1.4 Optimization Challenges 12
1.5 Domain-Specific Optimization Techniques 14
1.6 Domain-Specific Extensions and Autotuning 14
1.7 Contributions 16
1.8 Thesis Structure 17

2. CHILL **19**

2.1 Organization of CHiLL 19
2.2 Integer Sets and Relations 20
2.3 Iteration Spaces 21
2.4 Relations for Loop Transformations 22
2.5 Dependence Graph and Legality of Transformations 24
2.6 Loop Transformations in CHiLL 24
2.6.1 Loop Permutation 25
2.6.2 Loop Skewing 25
2.6.3 Loop Tiling 26
2.6.4 Loop Fusion 26
2.7 Computing Footprint of Array References 30

2.8 Extending Polyhedral Technology in CHiLL 30
2.9 CUDA-CHiLL 31
2.9.1 Parallel Decomposition 31
2.10 Summary 32

3. THE miniGMG BENCHMARK 35

3.1 V-cycle 35
3.2 Domain-Decomposition and Parallelism 36
3.3 V-cycle and Operator Code Skeletons 41
3.3.1 Smooth 43
3.3.2 Residual 46
3.3.3 Restriction and Interpolation 46
3.4 Optimization Challenges 46

4. COMMUNICATION-AVOIDING OPTIMIZATIONS 49

4.1 Types of Communication 49
4.2 Communication-Avoiding Optimizations 50
4.2.1 Fusing Components of Smooth 50
4.2.2 Deep Ghost Zones 51
4.2.3 Wavefront Computation 53
4.2.4 Parallel Code Generation 61
4.2.5 Residual-Restriction Fusion 65
4.2.6 Smooth-Residual-Restriction Wavefront 66
4.3 Autotuning Opportunities 70
4.4 Compiler Implementation 71
4.4.1 Fusing Components of Smooth 72
4.4.2 Overlapping Ghost Zones 73
4.4.3 Stencils as Accumulation 74
4.4.4 Rebuilding the Dependence Graph 77
4.4.5 Wavefronts 78
4.4.6 Smooth Residual Restriction Fusion 80
4.4.7 Parallel Code Generation 80
4.5 Putting It Together 81
4.6 Results 82
4.6.1 Problem Specification 85
4.6.2 miniGMG Configuration 85
4.6.3 Manually Optimized and Baseline miniGMG 85
4.6.4 Evaluated Platforms 86
4.6.5 Analysis of GSRB Smooth Performance 86
4.6.5.1 Computing Roofline Memory Bounds 87
4.6.5.2 Optimizations and Smooth Performance 90
4.6.6 Smooth, Residual and Restriction Fusion 94
4.7 Summary and Conclusion 95

5. OPTIMIZATIONS FOR COMPUTE-INTENSIVE STENCILS 96

5.1 Stencil Definitions and Accuracy 97
5.2 Stencil Reordering: Partial Sums 98
5.2.1 Composition of Optimizations 105

5.3 Compiler Implementation 105
5.3.1 Background Review 106
5.3.2 Abstractions for Partial Sums 106
5.3.3 Deriving StencilPoints and BoundingBox (BB) 107
5.3.4 Deriving Coefficients (Coeff) 107
5.3.5 Deriving Partial Sums and Buffers 108
5.3.6 Exploiting Reuse in Partial Sums 109
5.3.7 Exploiting Symmetry to Reduce Floating-Point Operations 109
5.3.8 Code Generation 110
5.4 Experimental Results 111
5.4.1 Review of Evaluated Platforms 112
5.4.2 Problem Solved on miniGMG 112
5.4.3 Experimental Methodology 113
5.4.4 Computing Roofline Memory Bounds 116
5.4.5 Stencil Performance 117
5.4.6 Smooth Performance on the Fine Grid 119
5.4.7 Smooth Performance Throughout the V-Cycle 121
5.4.8 miniGMG Solver Performance and Error 122
5.4.9 Distributed Memory Results 127
5.5 Conclusions 127

6. GEOMETRIC MULTIGRID ON GPUs 130

6.1 Gauss-Seidel Red-Black (GSRB) Smooth on GPUs 131
6.1.1 Strategy for Parallel Decomposition of Smooth 131
6.2 Code Generation with CUDA-CHiLL 132
6.2.1 Mapping to Blocks and Threads 135
6.2.2 Extensions to Code Generation in CUDA-CHiLL 135
6.2.3 Space of Generated GSRB Variants 137
6.3 Experimental Results 137
6.3.1 miniGMG on GPUs 139
6.3.2 Experimental Methodology 139
6.3.3 Optimizations in Handtuned Code 141
6.3.4 Performance of Smooth 141
6.4 Conclusions 142

7. RELATED WORK 143

7.1 Optimizations to Reduce Data Movement 143
7.2 Stencil Reordering Optimizations 144
7.2.1 Comparison with Array Common Subexpression Elimination 144
7.2.2 Other Stencil Reordering Techniques 145
7.3 Optimizations for Solvers 146
7.4 Stencil Computations on GPUs 147
7.4.1 Manual Optimization Efforts 147
7.4.2 Domain-Specific Automated Optimization Efforts 148
7.5 Summary and Conclusions 149

8. FUTURE RESEARCH AND CONCLUSIONS 150

8.1 Contributions 150
8.1.1 Optimizing Memory Bandwidth Limited Stencils 151
8.1.2 Optimizing Compute-Intensive Higher-Order Stencils 151
8.1.3 Parallel Code Generation 152
8.2 Future Work 153
8.2.1 Nonperiodic Boundary Conditions 153
8.2.2 Target Emerging Many-Core Architectures 154
8.2.3 Code Generation for Emerging Runtimes 154
8.3 Conclusion 155

APPENDIX: STENCILS 156

REFERENCES 159

LIST OF FIGURES

1.1 The Geometric Multigrid V-cycle. The hierarchical linear solver starts with large, fine-resolution grids and comes down the V-cycle by successive application of smooth, residual, and restriction. The GMG goes back up the V-cycle by applications of smooths and interpolation. . . . 3

1.2 Visualization of stencils optimized: Figures(a)(b)(c)(d) are constant-coefficient 3D 7-, 13-, 27- and 125-point stencils, respectively. Figures(e) and (f) are 2D cross-sections of 3D stencils, (e) is a 7-point variable coefficient stencil, and (f) illustrates restriction and interpolation stencils. 4

1.3 Illustration of (a) Jacobi iterations and (b) Gauss-Seidel Red-Black (GSRB) iterations with a 2D 5-point stencil on grid. Jacobi is an out-of-place sweep. To compute a stencil it reads in 5 points from an input grid and writes out a computed values to an output grid. GSRB partitions the grid points into either red or black points such that each red point is surrounded by black points and vice versa. Each GSRB sweep updates either the red or black points. It is an in-place operation, as the grid being updated is also read. The figure illustrates the update of a red point which requires its own value and values from the black neighbors. 8

1.4 Visualization of periodic boundary conditions for structured grids. Figure (a) shows a 2D 5-point stencil and (b) illustrates application of the stencil on the upper-left corner of a 2D grid. Periodic boundary conditions mean points outside the boundary wrap around the grid. For the 2D 5-point stencil which reads point 0-4, the points 3 and 4 wrap around the grid as illustrated by the dashed lines. 11

1.5 Autotuning based on the CHiLL compiler framework. CHiLL has a scripting language interface. Novel transformations were built into CHiLL. The autotuners implemented in research presented here generated CHiLL scripts to direct optimization. CHiLL ran the scripts to generate code variants. 15

3.1 The multigrid V-cycle for solving $Lu^h = f^h$. Note, superscripts denote grid spacing. 37

3.2 Visualization of the domain/process/subdomain hierarchy in miniGMG. 38

3.3 Execution of the miniGMG V-cycle. A node is assigned a 256^3 domain which is decomposed into a list of 64^3 subdomains (boxes). The list of subdomains is partitioned into MPI processes. The subdomains owned by each MPI process are then computed in parallel by a single OpenMP thread. 39

3.4 Application of a 2D 5-point stencil of a 4x4 2D grid. (a) Shape of the stencil. (b) Application of this stencil of an interior and boundary point of the grid. Ghost zone is shaded gray. 40

3.5 Visualization of ghost zones and data exchange between subdomains. A 12×12 domain (grid) is geometrically decomposed to 9, 4×4 subdomains (tiles). Each subdomain needs a ghost zone which must be exchanged between neighboring tiles after a stencil computation is applied to the entire domain. 40

4.1 Two applications of a 3D 5-point stencil on a 4x4 grid with a two-deep ghost zone. (a) The stencil is first applied on a 8x8 grid to compute the values for the 6x6 grid (blue points). The outer layer of grid points shaded grey are read and used as the ghost zone. (b) The second stencil sweep uses the 6x6 grid to compute the output of the 4x4 grid (red points). For the second stencil sweep, the blue points are used as the ghost zone. 54

4.2 Deeper ghost zone changes the communication pattern and volume between a subdomain and its neighbors. A one-deep ghost zone requires exchange with left, right, top, and bottom neighbors. An additional layer of ghost zone now requires exchange with neighbor on the corners as well. In addition, a larger volume of data must be exchanged. 54

4.3 Progress of the GSRB wavefront. 57

4.4 Jacobi wavefront. 58

4.5 Multiple threads working collaboratively to process a subdomain/box . 62

4.6 Example parallel decompositions on Hopper, which has 6-cores per socket. All the boxes in a subdomain may work in parallel, or all the threads may work on one box collaboratively, or nested parallelism may be used. 65

4.7 Wavefront applying smooths, residual and restriction. 68

4.8 Code generation steps for accumulation transformation. 76

4.9 Loop skewing and loop permutation to create a wavefront. 79

4.10 Speedups of CHiLL-generated and manually tuned GSRB smooth relative to the baseline code on Hopper. The speedups are shown for all levels of the V-cycle. Generated code outperforms expert-written code on all sizes except the 32^3 box, and always outperforms baseline code. . 88

4.11 Speedups of CHiLL-generated and manually tuned GSRB smooth relative to the baseline code on Edison. The speedups are shown for all levels of the V-cycle. Generated code outperforms expert written code on all sizes except the 32^3 box, and always outperforms baseline code. 89

4.12 Speedup for the MG Solver attained from incrementally enabled optimizations in the CHiLL compiler. The performance results are categorized by type of smooth and architecture. "VC" is variable-coefficient. 93

5.1 (top) Visualizations of the discretized 3D Laplacian operators (stencils) used in this chapter. (bottom) 2D cross sections through the centers of 3D stencils. Color is used to denote the coefficient associated with that point. The 27- and 125-point stencils have complex symmetries that we exploit. 98

5.2 Visualization of 2D 9-point stencil application on a 2D grid. Figure shows stencil operator being applied on three consecutive iterations of the inner loop (j,i), (j,i+1) and (j, i+2). The edge of points {(j+1, i+1), (j, i+1), (j-1, i+1)}, bound by the bold rectangle, get reused by the three iterations. The coefficients of the stencil are color coded, blue = w1, green = w2, and red = w3. 101

5.3 Illustration of deriving partial sums. The right edge from the input array is loaded and multiplied by weights stored in the array of coefficients. The sum of products of the loaded points and the right, center, and left edges of the array of coefficients are BO, B1, and B2, respectively. 102

5.4 The reuse of the leading edge of points loaded at iteration <j,i> gets captured in three buffers R, C, and L (top). Buffer entires R[i] (1), C[i+1] (2), L[i+2] (3) correspond to B0, B1, and B2 from Figure 5.3. The loaded edge is factored into `r1` and `r2` based on symmetry of the color-coded coefficients. The factors are used to compute B0, B1, and B2. The final output out[j][i] is the sum of buffer entries. 102

5.5 Increasing symmetries in coefficients allows us to increasingly reduce floating-point computation. Symmetry about the $j-$axis (in b) permits discarding half the coefficients, and symmetry about the $i-$axis (c) and the diagonal (d) lets the compiler consider even fewer coefficients. 103

5.6 Visualization of a 2D plane from the 3D array of coefficients for the 125-point stencils (left). As shown in Figure 5.5(d), there are 6 unique coefficients 0-5. When applying the 125-point stencil at iteration <j,i> using partial sums, the leading 2D plane of 25 points is loaded and factored according to the unique coefficients (right). The six factors `r0` - `r5` are multiplied by appropriate coefficients to compute partial sums which are then buffered. The factors are created by summing loaded points which are multiplied by the same coefficient, and coeffient `0` corresponds to loaded point in[k][j][i+2]. 103

5.7 Code generation steps for partial sum transformation. 111

5.8 ApplyOp (y=Ax) stencil performance attained with the CHiLL compiler by optimization, operator, and platform. The Roofline memory bound is for the noncommunication-avoiding implementation. Partial sums can move compute-limited operations towards a memory-limited state. 118

5.9 Jacobi smooth performance attained with the CHiLL compiler by optimization, operator, and platform. The Roofline memory bound is for the noncommunication-avoiding (no wavefront) implementation and is lower than ApplyOp due to additional data movement like the RHS. The wavefront transformation allows CHiLL to exceed this limit. 120

5.10 Jacobi smooth performance on Edison attained with the CHiLL compiler as a function of level in the V-Cycle (256^3 fine grids down to 16^3 coarse grids) for the 27- and 125-point operators. Observe that the reference implementation of the memory-limited 27-point operator receives a cache boost on the coarser levels, while the compute-limited 125-point does not. 123

5.11 miniGMG performance (millions of degrees of freedom solved per second) using either the Intel compiler (baseline) or the CHiLL compiler. The labels indicate the overall solver speedup attained via CHiLL. The performance of the tenth-order 125-point solver is within a factor of 2 of the second-order 7-point solver, but provides nearly a million times lower error. 125

5.12 Error attained in miniGMG as a function of operator and grid size. A 1GB vector represents a single 512^3 grid. Multigrid will require several of these grids. Observe that the tenth-order method delivers a three-digit increase in accuracy for every 8× increase in memory. 126

5.13 The performance benefit of CHiLL is not lost as one weak scales miniGMG with a 27-point stencil to 31,944 cores (1331 nodes). The above figure illustrates weak-scaled miniGMG time to solution using either the baseline code or the CHiLL compiler. Four weighted Jacobi relaxations are used at each level, with BiCGStab as the bottom solver, and a 256^3 domain per node. 128

6.1 Organization of CUDA threads into a 3D (2,4,64) grid with 2D (32,16) thread blocks. Each 2D (X,Y) plane in the 3D grid has 8 2D thread blocks of dimension (32,16). Each 2D plane in the 3D grid works on a single subdomain (box) in miniGMG, and there are 64 such planes to process 64 subdomains. 133

6.2 Illustration of work done by each thread block. Figure (a) shows 8, (32,16) thread blocks working on a 64^3 subdomain (box), with each thread computing a column of 64 output points. The blue column in figure (b) represents the 32*16*64 = 32,768 output grid points computed by each thread block. 133

6.3 Execution times for 480 iterations of GSRB (240 Red, 240 Black) smooth on 64^3 boxes. Lower bar means better performance. The horizontal lines corresponds to performance of manually tuned codes. The x-axis are the 2D thread block sizes (TX,TY). The grid corresponding to each thread block is (64/TX, 64/TY, 64). Best performance was achieved with thread block (64, 16) and grid (1,4,64). 140

6.4 The fraction of the effective DRAM bandwidth achieved by GSRB smooth. Higher values correspond to better performance. 140

LIST OF TABLES

1.1 Description of stencils optimized. VC and CC stand for variable and constant coefficient, respectively. #Flops and #Bytes are per update of a grid point, and we only account for compulsory cache misses. We also take into account write allocation, that is, we do not consider cache bypass. 5

2.1 A subset of transformations available in CHiLL. 24

4.1 The table illustrates the classification of optimizations described in this chapter as reducing vertical communication, horizontal communication, or both. Parallel code generation has been been left out as it is not a communication-avoiding optimization but helps improve wavefront, which reduces vertical communication. 51

4.2 Overview of evaluated platforms. 87

4.3 Configurations of best-performing code variants for GSRB smooth on Hopper. 88

4.4 Configurations of best-performing code variants for GSRB smooth on Edison. 89

5.1 The table shows the Roofline memory (DRAM) bounds for ApplyOp computation with four different stencils. ApplyOp reads in a grid, applies the stencil operator to it and writes out the computed values to a different output grid. The bounds are expressed in terms of Million Stencils per Second that can be computed when DRAM bandwidth is the bottleneck. The stencil was applied to a 256^3 domain which was split into a list of 64^3 sudomains. To account for ghost zones a 64^3 sudomain translates to 66^3 and 68^3 grids for stencils with radius 1 and 2 respectively. Stencils with larger radius require larger data volumes and have lower Roofline memory bounds. 117

5.2 The table shows the Roofline memory (DRAM) bounds for the Jacobi smooths using four different stencils. The bounds are expressed in terms of Million Stencils per Second that can be computed when DRAM bandwidth is the bottleneck. In addition to input and output grids, smooths require reading in two more grids (arrays) rhs and lambda. This extra data movement means smooths have lower Roofline bounds than ApplyOp. 117

5.3 CHiLL was able to select optimizations uniquely for each multigrid level and platform. <#,#>denotes the number of inter- and intra-box threads with nested OpenMP. 124

6.1 Overview of evaluated NVIDIA GPU. 139

ACKNOWLEDGEMENTS

First, I'd like to thank my dissertation advisor, Mary Hall, for her encouragement and guidance. Her dedication to research and emphasis on tackling important, challenging, and real problems has been inspiring. Over the years, she has provided an incredible amount of help in writing and presenting my research. This dissertation has greatly benefited from her advice and editing.

Most of the research in this dissertation has been done in collaboration with researchers from Berkeley Lab as part of the x-tune project. I would like to thank Samuel Williams, Leonid Oliker, Brian Van Straalen, and Phillip Colella. They have helped give my research a new direction.

Samuel Williams, who is also on my committee, has been pivotal to my PhD. Sam's technical knowledge, attention to detail, and most importantly, enthusiasm, has benefited me immensely. I am thankful to Sam for all the discussions in person and over email, and for being on my PhD committee.

I am thankful to my dissertation committee members Martin Berzins, Ganesh Gopalakrishnan, and Rajeev Balasubramonian for their feedback and encouragement. I hope some of our discussions can lead to future collaborations.

My labmates in the CTOP group have been good friends, collaborators, and proofreaders. I would like to thank Anand Venkat, who has helped me debug code at all hours of the day. I also want to thank Manu Shantharam, Saurav Muralidharan, and Amit Roy for giving me feedback on drafts of my papers and dissertation. I am thankful to Chun Chen, who had implemented CHiLL. I wish he was around longer.

My friends in Salt Lake City and elsewhere have been crucial to my life as a graduate student. I'd like to especially thank Dan, Keita, Mike, Didem, Nil, Anusua, Anand, Shusen, Megan, Grant, Kristina, Grace, Nic, and Avishek for making the last six years enjoyable, and helping me handle the ups and downs of the PhD process.

I would like to thank my parents, Partha Sarathi and Subhra Basu, and my sister Rajashree Basu Kundu for their love, support and patience. I am forever grateful to them for the values they have instilled in me, and for believing in me even when most others didn't. I am humbled by their sacrifices that made it possible for me travel to the other side of the world and pursue my graduate studies.

I would like to thank the National Energy Research Scientific Computing Center (NERSC) for access to their machines Hopper and Edison. NERSC is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. I would also like to express my gratitude to Berkeley Lab for hosting me for the 2013 and 2014 summers. This research was funded by the Department of Energy Office of Science award #DE-SC0008682, the National Science Foundation award # CCF-1018881, and the the Department of Defense, through a contract to the University of Maryland.

CHAPTER 1

INTRODUCTION

Stencil computations are operations on structured grids. A structured grid [1] represents a uniformly discretized continuous domain, and a stencil application on the grid is simply an application of a differential operator. Stencil computations are a ubiquitous pattern in parallel computing, and they are frequently found at the heart of partial differential equation (PDE) solvers. As PDE solvers are used in a large fraction of scientific applications, ranging from fluid dynamics to electromagnetics, the performance of stencil computations is critical to scientific computing.

Stencil computations traditionally have very low arithmetic intensities, where arithmetic intensity is the number of floating-point operations performed per byte read from memory. Stencil computations execute as low as 0.2 floating-point operations (flops) per byte. This low arithmetic intensity, combined with the fact that memory bandwidth of modern machines lags far behind floating-point power, make stencil computations notoriously memory bandwidth-limited. Thus, improving performance of stencil computations requires reducing and managing data movement.

The importance of stencil computations in scientific computing, and the widening performance gap between computation and data movement has motivated efforts from diverse research communities to develop optimizations for stencils. Code optimization experts have designed manual and automatic optimizations to improve memory bandwidth use, and applied mathematicians have developed compute-intensive *higher-order* stencils that need far less memory.

In this dissertation we optimize both traditional memory bandwidth limited stencils and the more compute-intensive higher-order stencils using a compiler-based approach. Novel optimizations for memory- and compute-bound stencils are implemented in a compiler framework. A compiler-based approach not only helps improve application programmer productivity, but also allows the code optimization expert

to leverage decades of compiler research. This compiler-based approach is used to optimize stencil computations in isolation, and in the context of a linear solver, Geometric Multigrid.

1.1 Geometric Multigrid and Stencil Computations

Multigrid (MG) [2] methods are extensively used in a variety of numerical simulations to solve linear systems ($\mathbf{Ax}$=$\mathbf{b}$). Multigrid methods use a hierarchy of grids with different resolutions to accelerate the convergence of iterative linear solvers. Traditional iterative solvers operate on grids at a single resolution and require a higher number of iterations. Multigrid uses corrections of the solution from iterations on the coarser levels to improve the convergence rate of the solution at the finest level. Geometric Multigrid (GMG) is a special case of MG where the linear operator $\mathbf{A}$ is simply a stencil computation applied to the grid $\mathbf{x}$.

Geometric Multigrid (GMG) is a hierarchical approach to solving a linear system. GMG has four key operations: smooth, residual, restriction, and interpolation. These operations all involve computation of stencils and are applied in a sequence known as the **V-cycle** and illustrated in Figure 1.1. GMG starts with a large, fine resolution grid. It applies several iterations of smooth on it, calculates the residual and then restricts into a smaller, coarser grid. This sequence is applied multiple times until a bottom grid size is reached. Further smooths are applied to the coarsest grid at the bottom level. Once the coarsest grid is solved the algorithm applies the solution to the finer grids. This is done by consecutive applications of smooths followed by interpolation. Interpolation is the inverse of the restriction operation and interpolates values from a small coarse grid to a larger, finer one.

Smooth involves a stencil computation followed by pointwise grid updates. Multiple iterations of smooth are applied at each grid level in GMG; the time spent in smooth dominates runtime. Residual is also a stencil computation, but unlike smooth, only a single iteration is required at each level. Restriction and interpolation are stencil computations which are inverses of each other. Restriction is a reduction operation which averages multiple points on a fine grid and writes out the value to a smaller coarser grid. Interpolation is a scatter operation which maps a single point in a coarse grid to multiple points in a finer grid.

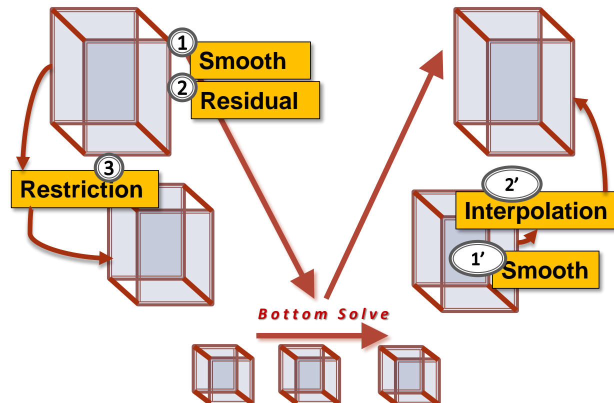


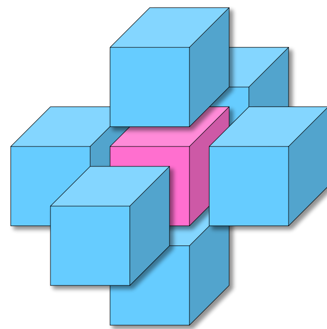
Figure 1.1: The Geometric Multigrid V-cycle. The hierarchical linear solver starts with large, fine-resolution grids and comes down the V-cycle by successive application of smooth, residual, and restriction. The GMG goes back up the V-cycle by applications of smooths and interpolation.

1.2 Classification of Stencil Computations

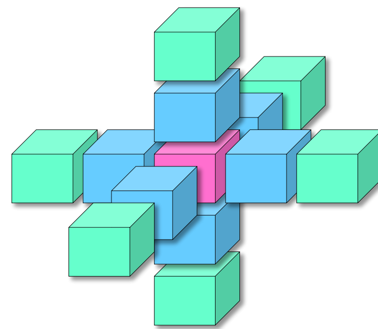
Figure 1.2 illustrates the various stencils optimized in this dissertation. The following four features of stencil computations listed below have been used to characterize the various stencils. These stencil features help understand performance characteristics and identify optimization challenges and opportunities.

1. Stencil coefficient types
2. Radius of stencils
3. Stencil iteration types
4. Arithmetic intensity of stencils

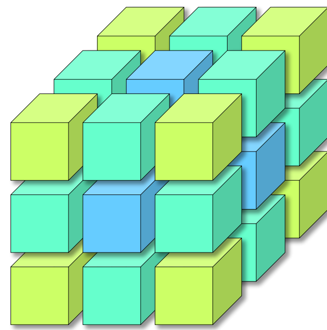
Table 1.1 lists the stencils from Figure 1.2 with the corresponding characteristics mentioned above. The first five stencils in Figure 1.2 ((a)-(e)) are used in smooth and residual. In GMG smooth and residual operations use the same stencil. For example, the GMG where smooth uses a 7-point stencil will also have a residual operation using the same 7-point stencil. The last stencil, Figure 1.2(f), represents both the restriction and interpolation operations. Restriction reads 8-points, averages



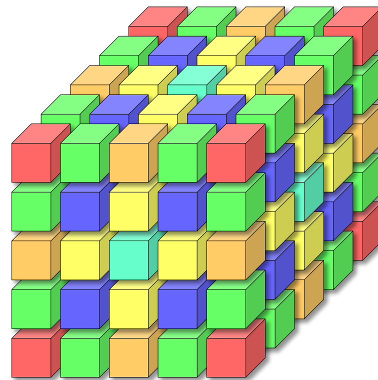
(a)



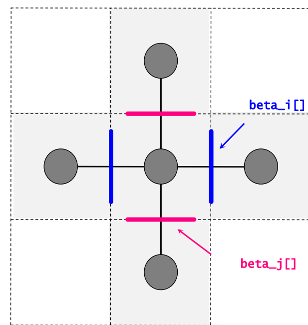
(b)



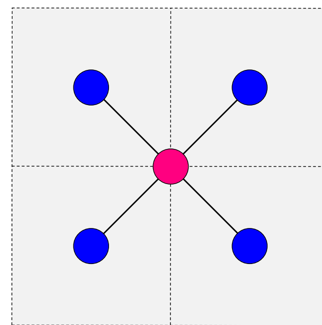
(c)



(d)



(e)



(f)

Figure 1.2: Visualization of stencils optimized: Figures(a)(b)(c)(d) are constant-coefficient 3D 7-, 13-, 27- and 125-point stencils, respectively. Figures(e) and (f) are 2D cross-sections of 3D stencils, (e) is a 7-point variable coefficient stencil, and (f) illustrates restriction and interpolation stencils.

Table 1.1: Description of stencils optimized. VC and CC stand for variable and constant coefficient, respectively. #Flops and #Bytes are per update of a grid point, and we only account for compulsory cache misses. We also take into account write allocation, that is, we do not consider cache bypass.

Stencil	Radius	Iterations	#Flops	#Bytes	Arithmetic Intensity
7-point VC	1	GSRB	17	80	0.21
7-point VC	1	Jacobi	17	48	0.35
7-point CC	1	Jacobi	8	24	0.33
13-point CC	2	Jacobi	15	24	0.63
27-point CC	1	Jacobi	32	24	1.33
125-point CC	2	Jacobi	134	24	5.58
8-pt Restriction	1	out-of-place	1	10	0.10

their value, and writes it to a coarser grid. The coarser grid is thus half the size of the fine grid in each dimension. Interpolation is the inverse operation, where a single point from the coarse grid is scattered to eight points in the fine grid.

1.2.1 Stencil Coefficient Types

Stencil computations sweep through grids, performing a weighted sum of points read from an input grid (array). If the weights used are constant scalars, it is termed a **constant coefficient** stencil. However if the coefficient or weights change from one grid point to another, they are not constants and have to be stored in separate grids. These type of stencil computations are classified as **variable coefficient** stencils.

Stencils in Figures 1.2 (a)(b)(c)(d) are constant coefficient stencils, the color at each stencil point in the figures represents the weight or coefficient. Figure 1.2 (e) represents a 3D variable coefficient 7-point stencil. Like the stencil in Figure 1.2 (a), this stencil also reads in 7 points, but does not use scalar constants for weight; instead it reads the weights from arrays `beta_i`, `beta_j` and `beta_k`. Code Listings 1.1 and 1.2 show simplified code for the 7-point constant coefficient and variable coefficient stencil operators, respectively. In variable coefficient stencils, in addition to the input and output grids, the grids corresponding to the coefficients are also accessed, creating higher traffic throughout the memory hierarchy.

```
1    double w1,w2;
2
3    for (k=0;j<N;k++)
4      for (j=0;j<N;j++)
5        for (i=0;i<N;i++){
6
7          phi_out[k][j][i] = w1 * phi_in[k][j][i] +
8                             w2 * (  phi_in[k][j][i+1] + phi_in[k][j][i-1]
9                                   + phi_in[k][j+1][i] + phi_in[k][j-1][i]
10                                  + phi_in[k+1][j][i] + phi_in[k-1][j][i]);
11
12     }
```

Listing 1.1: Out-of-place grid sweeps for Jacobi iterations with a 3D 7-point constant coefficient stencil operator. The code snippet corresponds to the stencil in Figure 1.2(a). The constant coefficients are `w1` and `w2`.

```
1
2    int sweep_color;
3
4    for (k=0;j<N;k++)
5      for (j=0;j<N;j++)
6        for (i=0;i<N;i++){
7
8           if((i+j+k+sweep_color) % 2 ==0){
9          phi[k][j][i] =
10           beta_i[k][j][i+1]*( phi[k][j][i+1] -phi[k][j][i] )
11          -beta_i[k][j][i]  *( phi[k][j][i]  -phi[k][j][i-1])
12          +beta_j[k][j+1][i]*( phi[k][j+1][i]-phi[k][j][i] )
13          -beta_j[k][j][i]  *( phi[k][j][i]  -phi[k][j-1][i])
14          +beta_k[k+1][j][i]*( phi[k+1][j][i]-phi[k][j][i] )
15          -beta_k[k][j][i]  *( phi[k][j][i]  -phi[k-1][j][i]));
16       }
17     }
```

Listing 1.2: In-place grid sweeps for Gauss-Seidel Red-Black iterations with a 3D 7-point variable coefficient stencil operator. The code snippet corresponds to the stencil in Figure 1.2(e). The variable coefficients are `beta_i`, `beta_j` and `beta_k`.

1.2.2 Radius of Stencils

In stencil computations, at each grid point, a set of neighboring points are read. The radius of the stencil is the offset of the farthest point read in each dimension. The 7- and 27-point stencils have a radius of 1, as they read ±1 points in each of the `i,j,k` dimensions. Similarly, the 13- and 125-points stencils read ±2 points in each of the `i,j,k` dimensions and have a radius of 2. Stencils with a larger radius need to read in more grid points, and have a larger working set, and also perform an increased number of floating-point operations.

1.2.3 Common Stencil Iterations

Stencil iterations can be either out-of-place grid sweeps or in-place grid sweeps. In an out-of-place sweep the grid being updated with computed values is different from the grids being read. In an in-place sweep, the grid being read and written is the same. The three common types of stencil iterations are Jacobi, Gauss-Seidel, and Gauss-Seidel Red-Black (GSRB).

1.2.3.1 Jacobi

Figure 1.3(a) illustrates Jacobi iterations with a 2D 5-point stencil, and Listing 1.1 shows code for Jacobi iterations with a 3D 7-point constant coefficient stencil. The grid being updated is different from the grid being read. All output grid points can be updated independently, making Jacobi iterations an embarrassingly parallel computation.

1.2.3.2 Gauss-Seidel and Gauss-Seidel Red-Black

Gauss-Seidel iterations are in-place grid sweeps, where the grid being updated is also being read. This means that at a given grid sweep, some point would have been updated and others would not. This mean that there is a dependence on the grid points being updated, which limits parallelism.

Gauss-Seidel Red-Black iterations aim to increase the parallelism in Gauss-Seidel iterations by partitioning the read/write grid into red and black points. The red points are surrounded by black points and vice-versa. A single Gauss-Seidel iteration then gets divided into two iterations: a red iteration followed by a black one. The red iteration updates the red grid points by reading a red point and its neighboring

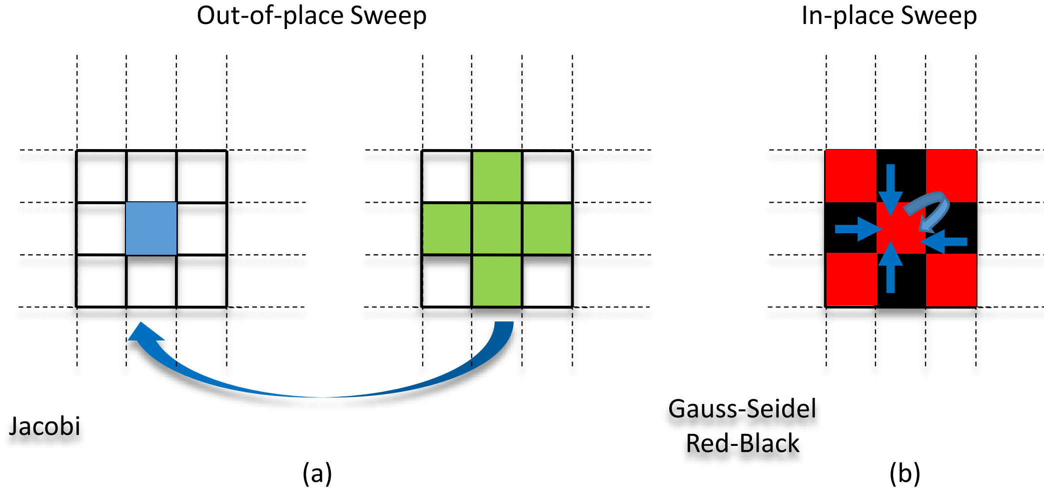


Figure 1.3: Illustration of (a) Jacobi iterations and (b) Gauss-Seidel Red-Black (GSRB) iterations with a 2D 5-point stencil on grid. Jacobi is an out-of-place sweep. To compute a stencil it reads in 5 points from an input grid and writes out a computed values to an output grid. GSRB partitions the grid points into either red or black points such that each red point is surrounded by black points and vice versa. Each GSRB sweep updates either the red or black points. It is an in-place operation, as the grid being updated is also read. The figure illustrates the update of a red point which requires its own value and values from the black neighbors.

black points, and the black iteration updates the black points similarly. The red and black iterations are embarrassingly parallel where all writes are independent.

Figure 1.3(b) illustrates the use of a 2D 5-point stencil in GSRB iterations, where a red point is being updated with values from itself and the neighboring black points. Listing 1.2 shows GSRB iterations for a 3D, variable-coefficient 7-point stencil, where `phi` is being read and updated. The if-condition which guards the statement ensures that only red or black points get updated based on the value of `sweep_color`.

1.2.4 Arithmetic Intensity of Stencils

Arithmetic intensity of a stencil computation is the ratio of floating-point operations (flops) performed to the memory (in bytes) that needs to be read in from the DRAM. By quantifying the balance between computation and communication, arithmetic intensity identifies a stencil as either being limited by memory bandwidth or floating-point intensity, and thus guides the selection of required optimization.

The arithmetic intensity for the stencil discussed in this dissertation is shown in Table 1.1. To compute a stencil for Jacobi iterations of the 7-, 13-, 27- and 125-point

constant coefficient stencils (rows 3-6), we need to read 8 bytes, write allocate 8 bytes and write 8 bytes, a total of 24 bytes per stencil. The 7-, 13-, 27- and 125-point stencils need 8, 15, 32 and 134 flops, and thus they have approximate arithmetic intensities 0.33, 0.63, 1.33 and 5.58, respectively.

Variable coefficient (VC) stencils read in more arrays (grids) and thus have much higher requirements, as can be seem from rows 1 and 2 in Table 1.1. The Jacobi iterations for the 7-point VC stencils read (`phi`, `beta_i`,`beta_j`,`beta_k`) in 32 bytes, write allocate 8 bytes and write back 8 bytes for a total of 48 bytes per stencil. The computation executes 17 flops to compute each stencil output. The GSRB iterations only compute half the grid points in each grid sweep, thus need two sweeps and double the data movement to update all grid points. Computing each stencil in GSRB iteration needs 80 bytes per stencil (not $48 \times 2 = 96$, as the same grid is read and updated, thus write allocation is excluded) and executes 17 flops per stencil.

1.2.5 Summary of Stencils Optimized

In this dissertation the stencils optimized have been illustrated in Figure 1.2, and their features listed in Table 1.1. Table 1.1 classifies the stencils based on their shape (radius and #points), size (#points), types of coefficients, iterations, and finally, the demands they place on the memory bandwidth and floating-point units of the processor.

In addition to these stencils, the 8-point restriction operation used in Geometric Multigrid is also a target. This stencil is a reduction operation which takes as input a larger N^3 grid and outputs a smaller $(N/2)^3$ grid; it averages values of 8 points from the larger grid and writes it to a single point in the smaller one.

1.2.5.1 Higher-Order Methods

In addition to the features listed in Table 1.1, the 7-, 13-, 27- and 125-point stencils can be classified by the ***order*** of the PDE solver they are used in. The order of a PDE solver denotes how fast the error computed decreases as the grid spacing decreases (or grid size increases). If a solver computes an error ϵ using a grid of size 64^3, as we decrease grid spacing by half, the grid size increases to 128^3 and the error drops to $\epsilon^{1/p}$, where $\mathbf{p}$ is the ***order*** of the method. Thus error drops exponentially as the

order of a method increases. In this dissertation, the 7-point stencils are second-order stencils used in second-order solvers. The 13-, 27- and 125-point stencils are used in higher-order PDE solvers, and they are fourth-, sixth-, and tenth-order, respectively.

1.2.5.2 Grid Boundary Conditions

Stencil computations are applied on structured grids with boundaries which can be commonly classified into periodic or nonperiodic. Application of a 2D 5-point stencil with a periodic boundary condition is illustrated in Figure 1.4. Points on the boundary of the grid have neighbors which wrap around the grid. As illustrated in the figure, the top-left and top neighbors of the upper-left grid points are the bottom-left and top-right points, respectively. In this dissertation only periodic boundary conditions have been considered.

1.3 Approaches to Reducing Data Movement

Traditional stencil computations execute far less than a flop for every byte of data they need, thus their performance is limited by their heavy memory demands. In the past on machines with smaller caches, operations on large structured grids could easily be bound by capacity misses in cache, leading to a variety of studies on blocking and tiling optimizations [3, 4, 5, 6, 7, 8, 9]. In recent years, numerous efforts have focused on improving memory bandwidth by increasing temporal locality. Automated and manual optimizations have attempted to increase locality by fusing multiple stencil sweeps through techniques like cache-oblivious, time-skewing, wavefront, or overlapped tiling [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. Almost all of these have concentrated their efforts on isolated stencil computations and have not considered an entire solver such as GMG. Solvers use a number of stencil computations and require several optimizations to be applied in sequence. This presents the twin challenges of developing optimizations that can be composed, and coming up with the best sequence of optimizations.

To reduce the heavy memory demands of stencil compuations, applied mathematicians are adopting higher-order methods for solvers [22, 23]. Higher-order methods achieve desired accuracy while working on much smaller grids (lower resolution). Smaller grids lower memory capacity requirements, which leads to re-

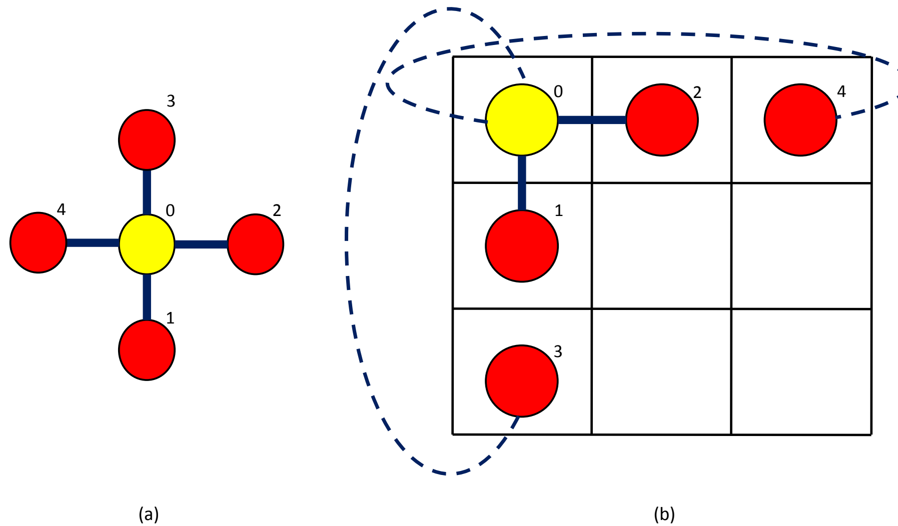


Figure 1.4: Visualization of periodic boundary conditions for structured grids. Figure (a) shows a 2D 5-point stencil and (b) illustrates application of the stencil on the upper-left corner of a 2D grid. Periodic boundary conditions mean points outside the boundary wrap around the grid. For the 2D 5-point stencil which reads point 0-4, the points 3 and 4 wrap around the grid as illustrated by the dashed lines.

duced data movement when computations stream through the grids. Higher-order methods achieve increased accuracy by using stencil computations with a higher number of floating-point operations. To update a grid point, higher-order stencils operate on a larger neighborhood of points; this often leads to performance being limited by the intensity of floating-point computation rather than memory bandwidth. Optimizing compute-intensive higher-order stencils have received far less attention than the memory-bound stencils. Manual optimizations [12, 24] and automated approaches [25, 26] alleviate performance bottlenecks of compute-intensive stencils by reducing loads and/or computation. Like optimizations for memory bound stencils, these techniques have only targeted isolated stencil computations on a single large grid, and not in the context of a solver, and importantly, these optimization efforts have ignored the interplay between optimizations to reduce computation intensity and to reduce memory traffic.

1.4 Optimization Challenges

The wide variety of stencils presents a spectrum of optimization challenges. With arithmetic intensities that range from 0.2 to over 5 (Table 1.1), stencil computations stress different subsystems on a node. This section outlines the various optimization challenges presented by stencils and GMG.

- **Memory bandwidth optimizations**

  Traditionally stencil computations have had very low arithmetic intensities. This makes the performance of stencil computations notoriously limited by the memory bandwidth of the machines. To improve memory use, optimizations often fuse multiple stencil sweeps into one, and trade off redundant floating-point computation for data movement.

- **Difference between stencil applications and smooths**

  Stencil computations can be expressed as $y = Ax$, where A is the stencil being applied to the grid x. Smooths, represented as ($x_{new} = x + wD^{-1}(b - Ax)$), require two more arrays b and D^{-1}. This means more data movement and makes smooths even more memory bandwidth-limited than stencil applications. Thus, optimizing smooths may require even more aggressive bandwidth optimization than simple stencil applications.

- **Managing floating-point computations**

  Stencils in Table 1.1 with arithmetic intensity greater than one have high intensities of floating-point operations. In fact, the 125-point stencil executes over 5 flops per byte moved. In such cases, the stencil computation may not even achieve performance corresponding to the DRAM bandwidth because it is limited by the large number of floating-point operations. Optimizing these compute-bound stencils requires identifying and exploiting reuse of computation to reduce floating-point operations. Furthermore, to achieve high floating-point performance on modern machines, it is essential for stencil codes to effectively use the SIMD (single instruction multiple data) instructions available on modern architectures. SIMD instructions process multiple data elements simultaneously

and can potentially improve floating-point performance by several integer factors (by the width of the SIMD unit).

- **Composing a sequence of optimizations**

  Optimizations which reuse and reduce floating-point operations and target compute-bound stencils must be designed to work with optimizations that target memory bandwidth-limited stencils. Reducing floating-point intensity of a compute-bound stencil will improve its performance, and may make it limited by the memory bandwidth. At this point, the optimization needs to be combined with memory bandwidth optimizations to push the performance even higher.

- **Optimization across GMG operators**

  Geometric Multigrid has five principal operations, of which four (smooth, residual, restriction, and interpolation) involve calculation of stencils. In addition to optimizing each of these operators in isolation, it is possible to optimize across them. For example, fusing two or more of these operations will reduce the number of times grids must be streamed into memory and improve memory bandwidth optimization. Unfortunately, such fusion will also involve trade offs such as increased working set and redundant computation. Finally, effectiveness of such optimizations will depend on the stencil, iteration, and architecture. Thus optimizing GMG means searching a larger space of possible optimizations compared to optimizing a single stencil computation.

- **Mapping computation to parallel threads**

  The increasing number of hardware threads on modern architectures makes mapping parallelism in stencil codes to threads crucial to performance. There is a delicate balance of parallelism, locality, redundant computation and working set in stencil computations. Increasing one of them may adversely affect the other. To achieve high performance, we must be able to generate and search many possible parallel variants which implement the same stencil computation.

1.5 Domain-Specific Optimization Techniques

Even though a large number of optimizations for stencils are known to the compiler community, current commercial compilers fail to generate highly optimized stencil code. Static compiling techniques cannot anticipate all possible execution environments, such as architecture, problem input size, and trade-offs between optimizations. Thus, state of the art commercial compilers do not typically risk potential slowdown by applying aggressive optimizations. To address the lack of support from commercial compilers, several domain-specific compilers and tools have been developed [27, 28, 29, 30, 31, 26].

These automated approaches tailor their optimizations and code generation to stencil computations. With the exception of the domain-specific language *Halide* [31], all other domain- or application-specific approaches have concentrated on isolated stencil applicatons. They have not applied their techniques to optimizing an entire solver with several stencil computations, and they have concentrated efforts on traditional memory-bound stencils with limited or no emphasis on higher-order stencils.

Another domain-specific approach to optimizing scientific codes presented in [12, 32], builds application-specific code generators. These are then used to generate many optimized code variants implementing the same computation; these variants are then searched to find the optimal code. This approach generates very high-quality code, targets both stencils in isolation and in solvers, and has optimized compute-intensive stencils. Unfortunately, this approach requires code generators to be rewritten for each new application without much reuse of optimizations across application. This requires significant human effort and hurts the productivity of the optimization expert.

1.6 Domain-Specific Extensions and Autotuning

Autotuning has been used to remedy this situation. Autotuning a computation involves generating and searching a space of possible implementations of the input computation to find the optimal code for a given execution context.

This dissertation presents research that extends a compiler framework with known and novel domain-specific optimizations targeting stencil computations and GMG. This approach of adding novel optimizations in a compiler framework and leveraging

the new and known optimization through autotuning is referred to as compiler-directed autotuning. Building domain-specific optimizations into a compiler framework [33] allows reuse of known optimizations across applications and greatly reduces the effort of writing application-specific code generators. A domain-specific autotuner is then created to drive the augmented compiler framework to generate multiple code variants for a computation, and the best code variant for a given execution is then empirically found.

The novel transformations for stencils and Geometric Multigrid are built into the CHiLL [33] compiler framework. CHiLL is a code transformation framework which supports dependence analysis, loop transformation and code generation. CHiLL allows composition of optimizations and is developed to support autotuning. The compiler framework has a scripting language interface; the scripts (also called transformation recipes) are sequences of optimizations which direct compiler application of transformations [34]. Figure 1.5 illustrates this approach. An autotuner generates

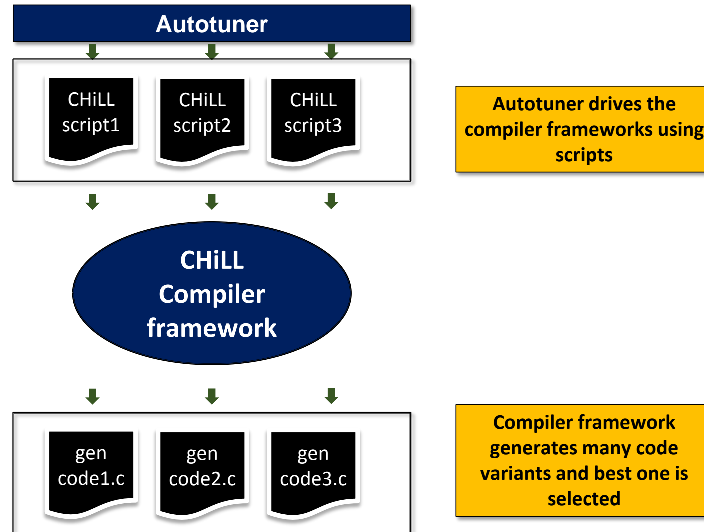


Figure 1.5: Autotuning based on the CHiLL compiler framework. CHiLL has a scripting language interface. Novel transformations were built into CHiLL. The autotuners implemented in research presented here generated CHiLL scripts to direct optimization. CHiLL ran the scripts to generate code variants.

multiple scripts which are used by CHiLL to generate multiple code variants for a computation, and the best performing variant is finally selected.

The benefit of compiler directed autotuning has been demonstrated in this research by applying the technique to optimizing stencils in isolation and in the context of Geometric Multigrid. Novel domain-specific compiler optimizations for memory limited stencils and compute intensive higher-order stencils have been developed. Considerable interactions between optimizations for memory and compute bounds stencils is seen, highlighting the need to compose sequences of optimizations. Furthermore, to target more complex and realistic structured grid codes seen in applications, optimizations are presented for a GMG benchmark (*miniGMG* [32]) that proxies block structured AMR codes. Block structured AMR codes such as CHOMBO [35] are commonly seen in modern scientific applications. They partition a grid into smaller sub-grids on which stencil computations are applied, and present more optimization challenges to the compiler than a stencil being applied to a single grid.

1.7 Contributions

This dissertation presents novel research which uses compiler optimizations and autotuning to optimize stencil computations and Geometric Multigrid. The contributions of this dissertation are outlined below.

1. **Communication-avoiding optimizations**

   This research develops domain-specific compiler optimizations and uses autotuning to reduce inter- and intra-node communication. The generated code achieves up to 4x speedup for smooths, 3x for the GMG solver, and matches or betters highly optimized manually tuned code.

2. **Optimize compute-intensive higher-order stencils**

   Higher-order GMG solvers are implemented to illustrate the high accuracy obtained by these methods and identify increased intensity of floating-point operations as the performance bottleneck. Research presented in this dissertation develops and implements a novel optimization, *partial sums*, which reuses computation to reduce floating-point operations. By using partial sums

in conjunction with communication-avoiding optimizations, speedups up to $4x$ for higher-order smooths are achieved.

3. **Parallel code generation for many- and multicore architectures**

   Research presented in this dissertation uses a compiler framework to generate optimized OpenMP and CUDA parallel code for stencils to target many- and multicore architectures. The parallel code generation capability is used to explore different threading strategies.

1.8 Thesis Structure

The remaining chapters in this disseration are:

- **Chapter 2** introduces the CHiLL compiler framework. It describes the basic compiler abstractions and known compiler optimizations in CHiLL which have been leveraged in this dissertation.

- **Chapter 3** describes the *m*iniGMG benchmark which was used to implement and optimize Geometric Multigrid. The chapter expains in detail the principal operations in GMG and their implementation in the benchmark.

- **Chapter 4** describes of communication-avoiding optimizations used to optimize memory bandwidth operations in GMG. The chapter first introduces the optimizations, then explains their implementation details and finally presents performance results and analysis.

- **Chapter 5** motivates the use of higher-order stencils and then identifies performance bottlenecks associated with these compute-intensive methods. Motivations for a new optimization targeting higher-order stencils is presented, followed by its description and implementation. The performance of the higher-order stencil is finally presented and the efficacy of higher-order methods is shown.

- **Chapter 6** describes code generation for graphics processing units (GPUs). The chapter explores parallel decomposition strategies to map GMG to the large number of hardware threads available on these platforms.

- **Chapter 7** discusses research related to optimizing stencil computations and GMG.
- **Chapter 8** presents conclusions and outlines future research.

CHAPTER 2

CHILL

The novel transformations presented in this dissertation are built into the CHiLL compiler framework [36, 33]. CHiLL is a loop transformation and code generation framework with a scripting language interface. CHiLL was designed to support autotuning by allowing sequences of transformations to be composed and applied. CHiLL also exposes parameters of the transformations for autotuning. The input to CHiLL is a source code written in C (or Fortran), and a transformation script. The script describes the set of transformations to be composed to optimize the provided source [34]. In research presented here, the script is either generated by an autotuner or written by an expert programmer, but it can also be derived automatically by a compiler decision algorithm [37]. After applying optimizations, CHiLL generates optimized C (or Fortran) code. To target CUDA code generation for NVIDIA GPUs, we use CUDA-CHiLL [34, 37]. It is a thin layer built on top of CHiLL to generate CUDA code. This chapter describes fundamental abstractions in CHiLL and CUDA-CHiLL, and gives examples of how they are used.

2.1 Organization of CHiLL

At the heart of CHiLL is a polyhedral framework that composes complex transformation sequences. Internally CHiLL uses Omega+, an enhanced version of Omega [38], and Codegen+ [39]. A polyhedral model represents each statement's execution in the loop nest as a lattice point in the space constrained by loop bounds, known as the iteration space. Then a loop transformation can be simply viewed as a mapping from one iteration space to another. CHiLL manipulates iteration spaces derived from the original program, using a dependence graph as an abstraction to reason about the safety of the transformations under consideration [40]. In CHiLL, iteration spaces are represented as integer sets, and loop transformations are linear mappings applied to

these integer sets. Omega+ is used to represent the integer sets as linear mappings, apply the mappings to the integer sets, and compute data dependences.

In a polyhedral model, after the relations representing loop transformations have been applied to input iteration spaces, optimized code is generated from the rewritten iteration spaces. Code generation involves scanning the polyhedra representing the iteration spaces of an optimized loop nest. The quality of the generated code directly impacts performance. Therefore, CHiLL uses a code generator called CodeGen+ that has advanced the state of the art in polyhedral scanning.

In the remainder of this chapter we describe the iteration space of a statement in a loop nest, relations used to tranform iteration spaces, data dependence, and legality of transformations. We use these concepts to illustrate four well-known loop transformations implemented in CHiLL: loop permutation, loop fusion, loop tiling, and loop skewing. Finally we introduce CUDA-CHiLL, a CUDA code generation tool built on top of CHiLL.

2.2 Integer Sets and Relations

Iteration spaces and mapping between iteration spaces are represented mathematically using Omega+. Omega+ provides interfaces to represent integer sets and mappings using Presburger formulas [41]. Presburger formulas are constructed by combining affine (equality or inequality) constraints on integer variables with the logical operations $\neg$, $\wedge$, and $\vee$, and the quantifiers $\forall$ and $\exists$. For example, let$S$ be the set of integers between 0 and 64, and S_{even} is the set of even integers in that range. These sets are defined as follows:

$$S := \{[i] : 0 < i < 64\}$$

$$S_{even} := \{[i] : \exists\alpha : (0 < i < 64 \;\&\&\; i = 2 * \alpha)\}$$

$$R := \{[i]- > [i'] : i' = i + 1\}$$

Relations are used to map between sets. They are also expressed using Presburger arithmetic. The relation R, shown above, adds 1 to each element of the set it is applied to. Thus, if R is applied to S, it will map it to a new set of integers, where the numbers will be betweem 0 and 65.

2.3 Iteration Spaces

In a polyhedral model, loops surrounding a statement can be described as a polyhedron in an integer linear space. Thus we can represent a loop nest's iteration space with a set of inequalities on loop index variables in the affine domain, where these variables have only integer coefficients. This representation of the iteration space is suitable for perfectly nested loops (all assignment statements in the innermost loop). For the loop nest in Listing 2.1, the iteration space of the statement can be simply represented as:

$$IS0 := \{[i, j] : 0 <= i < N \&\& 0 <= j < N\}$$

For imperfectly nested loops as shown in Listing 2.2, additional information is needed to capture ordering of statements and represent the loop structure. For the imperfect loop nest, the statements S1, S2, and S3 are at different levels of nesting. To reason about the effect of transformations on imperfect loop nests, the iteration space representation must capture this loop structure.

To capture ordering constraints between statements in an imperfect loop nest, we add an auxiliary loop to each loop level, with an additional auxiliary loop as the last dimension. These auxiliary loops *sink* all statements to the loop level of the innermost loop. This means the iteration spaces of all the statements in the loop nest have the same dimensionality. Furthermore, the auxiliary loops are added, ensuring that the order in which the statements are executed is preserved, in other words, the *lexicographic order* of the statements is correct [33]. Thus for an n-deep loop nest, we have (2n + 1)-dimension iteration spaces, and all statements, irrespective of their nesting depth, have the same number of dimensions in their iteration space. Although auxiliary loops carry special meaning during loop transformations, auxiliary loops and other loops are treated equivalently during code generation. Auxiliary loop iteration spaces are always constant valued and do not show up in the final code. For the code Listing 2.2, the iteration spaces of the statements are:

$$IS1 := \{[i, j] : 0 <= i < N \&\& j = 1\}$$

$$IS2 := \{[i, j] : 0 <= i < N \&\& 0 <= j < i\}$$

$$IS3 := \{[i, j] : 0 <= i < N \&\& j = i - 1\}$$

From the 2D iteration space of the statements it can be clearly seen that all the statements have the same loop nesting level (iteration of same dimensionality). Internally, the 2D space is stored as a 5D space with the help of auxiliary loops.

$$r1 := \{[i, j]- > [0, i, 0, j, 0]\}$$

$$r2 := \{[i, j]- > [0, i, 1, j, 0]\}$$

$$r3 := \{[i, j]- > [0, i, 2, j, 0]\}$$

2.4 Relations for Loop Transformations

CHiLL uses Omega+ to represent iteration spaces as sets, and uses relations to map between iteration spaces to represent loop transformations. The use of relations to represent loops is illustrated with the example of loop shifting. For example, the iteration space for code Listing 2.1 is shown below as $IS0$. The relation R, which shifts the iteration space in each dimension by 1, is applied to $IS0$, the output iteration space is $IS0'$.

$$IS0 := \{[i, j] : 1 <= i < N \ \ \&\& \ \ 1 <= j < N\}$$

$$R := \{[j, i]- > [j', i'] : i' = i + 1 \ \ \&\& \ \ j' = j + 1\}$$

$$IS0' := R(IS0) := \{[j, i] : 2 <= i < N + 1 \ \ \&\& \ \ 2 <= j < N + 1\}$$

The loop nest corresponding to $IS0'$ is shown in Listing 2.3. It can be easily seen that the loop bounds have been shifted by unity in each dimension. In addition, CHiLL adds a negative shift to the array references to counter the shift in the iteration space.

In the remaining discussion, relations will be abbreviated to: R := {[i] -> [i']]}, unless further details are necessary. Each loop transformation from an n-deep loop nest to a new m-deep loop nest is represented as a relation *map*. The loops marked as c_i's in *map* are the auxiliary loops. Ignoring the auxiliary loop we can rewrite *map* as *map'*.

$$map := \{[c_1, l_1, , , , , c_n, l_n, c_{n+1}]- > [c'_1, l'_1, , , , , c'_m, l'_m, c'_{m+1}]\}$$

$$map' := \{[l_1, , , , , l_n]- > [l'_1, , , , , l'_m]\}$$

```
1  #define N 64
2
3  void func0(){
4
5   double A[N+2][N+2], B[N+2][N+2];
6   int i,j;
7
8   for(i=1; i<N; i++){
9    for(j=1; j<N; j++){
10   //Statement S0
11   //iteration space IS0
12   A[j][i] = B[j][i-1]
13            +B[j][i]+B[j][i+1];
14  }}
15 }
```

Listing 2.1: Simple perfectly nested loop.

```
1  #define N 64
2  void func1(){
3
4   double Sum[N], A[N][N], B[N];
5   int i,j;
6
7   for(i=0; i<N; i++){
8    //Statement S1
9    //iteration space IS1
10   Sum[i] = 0;
11   for(j=0; j<i; j++){
12
13    //Statement S2
14    //iteration space IS2
15    Sum[i] = Sum[i]
16              +A[j][i]*B[j];}
17
18   //Statement S3
19   //iteration space IS3
20   B[i] = B[i]-Sum[i];
21   }
22 }
```

Listing 2.2: Loop nest with three statements at different nesting levels.

```
1  #define N 64
2  void func0(){
3   double A[N+2][N+2], B[N+2][N+2];
4   int i,j;
5
6   //Loop bounds have been shifted by (+1)
7   for(i=2; i<=N; i++){
8    for(j=2; j<=N; j++){
9     //Statement S0
10    //Array indices have been shifted by (-1)
11    //Iteration space IS0'
12    A[j-1][i-1] = B[j-1][i-1-1]+B[j-1][i-1]+B[j-1][i+1-1];}}
13 }
```

Listing 2.3: Loop nest after shifting.

2.5 Dependence Graph and Legality of Transformations

Dependence analysis [40, 38] is a key component of any compiler framework which ensures correctness of generated code. CHiLL uses Omega+ for data dependence (flow, antidependence, output and input dependence) analysis [38]. Using the data dependence information from Omega+, CHiLL creates a dependence graph for the statements in the input loop nest. The dependence graph is a standard component of loop restructuring compilers [40], and in CHiLL, the dependence graph is used to test for legality of loop transformations.

2.6 Loop Transformations in CHiLL

CHiLL implements a wide range of loop transformation algorithms which transform a loop nest from one state of representation to another, more suitable one. The representation of a loop nest includes the statements, iteration spaces, and the dependence graph. Common loop transformations implemented in CHiLL are listed in Table 2.1, for a complete list refer to the CHiLL user manual [36]. The following subsection briefly outlines the known loop transformations which have been used in the research presented in this dissertation.

Table 2.1: A subset of transformations available in CHiLL.

Transformation	**Purpose**
Loop permutation	Change the order of loops.
Loop tiling	Partition the iteration space to small blocks and iterate through blocks in sequence.
Loop unrolling	Similar to tiling in changing the iteration order, but uses explicit unrolled loop body.
Loop fusion	Fuse distinct loops for different statements into one.
Loop distribution	Distribute different statements in a single loop nest into distinct subloops each enclose a separate statement.
Loop peeling	Unrolls a number of iterations from the beginning or the end of a loop.
Loop splitting	Split the original loop to subloops each representing a disjoint part of the original iteration space.
Loop shifting	Adjusts the index of the loop by adding specified amount to what the the non transformed index.
Loop skewing	Modify the iteration space so that multiple dependences are carried by the same loop nesting level.

2.6.1 Loop Permutation

Loop permutation changes the ordering of the loops [40]. For a n-deep loop nest, given a permutation Π of the loop order, the relation for permutation can be expressed as:

$$permute := \{[l_1, , , , , l_n]- > [l_{\Pi_1}, , , , , l_{\Pi_n}]\}$$

CHiLL uses the dependence graph of the loop nest to make sure permutation is legal and does not violate data dependences. CHiLL also carefully updates auxiliary loops to reflect the change in loop order [33].

Permutation using CHiLL is illustrated in code Listings 2.4-2.6. Listing 2.4 shows the input code for a 2D stencil. The CHiLL script driving the transformation is shown in Listing 2.6. The first three lines direct CHiLL to the loop nest in the given file and procedure. Line 5, `original()`, initializes CHiLL, sets up the iteration space (with auxiliary loops), and collects dependence information for the statements in the loop nest. The command to permute is given in line 7. The command specifies the final ordering of the loop nest. The command can be represented by the relation permute', which is applied to the iteration space of the loop nest. Once the transformation is complete, polyhedra scanning generates the output code shown in Listing 2.5.

$$permute' := \{[l_1, l_2]- > [l_2, l_1]\}$$

2.6.2 Loop Skewing

Loop skewing changes the iteration space of a statement in a loop nest by adding an outer loop index value to an inner loop index. Skewing is illustrated using Listings 2.4, 2.7, and 2.8. In the generated code in Listing 2.7, the inner loop index `i` is now a linear function of outer loop index `j`. To account for the change is inner loop bounds, the array references have also been updated. In general, skewing can be expressed as the relation skew, where a_i terms are integer constants. Thus, the modified loop index is simply a linear combination of the other loop indices. The CHiLL script shown in Listing 2.8 uses the command `skew([0],2,[1,1])`. This directs CHiLL to skew loop level 2 (`i`) surrounding statement 0, such that the new loop level 1 is a linear combination of loop levels 1 and 2, with constant terms a_1 and a_2 set to 1. The mapping corresponding to this transformation is skew'.

$$skew := \{[l_1, l_2, ..., l_n] -> [l'_1, l_2, ..., l_n] : l'_1 = a_1 * l_1 + a_2 * l_2, ... + a_n * l_n\}$$

$$skew' := \{[i, j] -> [i', j] : i' = 1 * i + 1 * j\}$$

2.6.3 Loop Tiling

Loop tiling involves decomposing a single loop into two loops, one which executes a tile of consecutive iterations and one which iterates over the tiles. Listings 2.9 and 2.10 illustrate tiling of the `i` loop into loops `ii` and `i` with 1D tiles of width 16. The CHiLL command `tile(0,2,16,2,counted)` tiles loop level 2(`i`) for statement 0 into tiles of size 16, and tile controlling loop (`ii`) is at loop level 2 after the transformation is complete. For clarity of presentation, we assume that the loop bounds are from 0 to 64 instead of starting from 1, as in the other examples. This simple tile command for the given problem and tile size can be represented by the mapping $tile'_{16}$. Mappings for tiling in general are more complex and further details can be found in [40, 33]. Generated code where both loops `i` and `j` are tiled is shown in Listing 2.11, with, corresponding CHiLL script in Listing 2.12.

$$tile'_{16} := \{[i, j] -> [i', ii, j] : i' = 16 * ii + k \ \ \&\& \ \ 0 \leq k < 16 \ \ \&\& \ \ 0 \leq ii < 4\}$$

2.6.4 Loop Fusion

Combining statements in adjacent loops into a single loop nest is known as loop fusion [40]. Initially, CHiLL will have statements in a loop nest fused whenever possible. This automatic fusion falls out of CHiLL's algorithm to add auxiliary loops to ensure all statements have the same dimensionality of iteration space. The automatic fusion is illustrated in Listings 2.13 and 2.14. CHiLL will automatically transform the input code in Listings 2.13 to the fused in Listing 2.14 by making a call to the `original()` command. In addition, CHiLL also provides explicit fuse commands for optimization purposes. Fusion algorithm takes a set of statements and the loop level as parameters.

In addition to loop transformations such as tiling and fusion, CHiLL can also be used to compute properties of loop nest computations. The next section uses Listing 2.15 to illustrate how CHiLL computes data footprint of an array in a loop.

```
1   #define N 64
2
3   void stencil2D()
4   {
5           double A[N+2][N+2], B[N+2][N+2];
6           int i,j;
7
8           for(j=1; j<N; i++){
9                   for(i=1; i<N; i++){
10                          A[j][i] = B[j][i]
11                                  + B[j][i+1] + B[j][i-1]
12                                  + B[j+1][i] + B[j-1][i];
13          }}
14  }
```

Listing 2.4: Input code for 2D stencil computation.

```
1   #define N 64
2   void stencil2D()
3   {
4     double A[66UL][66UL];
5     double B[66UL][66UL];
6     int i;
7     int j;
8     for (i = 1; i <= 63;  i+= 1)
9      for (j = 1; j <= 63; j += 1)
10      A[j][i] = B[j][i] + B[j][i + 1]
11         + B[j][i - 1] + B[j+ 1][i]
12         + B[j - 1][i];
13  }
```

Listing 2.5: Generated code after loop permutation.

```
1   source: stencil2D.c
2   procedure: stencil2D
3   format : rose
4
5
6   original()
7   permute([2,1])
```

Listing 2.6: CHiLL script for loop permutation.

```
1
2  #define __rose_lt(x,y) ((x)<(y)?(x):(y))
3  #define __rose_gt(x,y) ((x)>(y)?(x):(y))
4  #define N 64
5
6  void stencil2D()
7  {
8   double A[66UL][66UL];
9   double B[66UL][66UL];
10  int i;
11  int j;
12  for (j = 1; j <= 63; j += 1)
13   for (i = j; i <= j + 63; i += 1)
14   A[j][-j + i] = B[j][-j + i]
15     + B[j][-j + i + 1]+ B[j][-j + i - 1]
16     + B[j + 1][-j + i]+ B[j - 1][-j + i];
17 }
```

Listing 2.7: Generated code after loop skewing.

```
1  source: stencil2D.c
2  procedure: stencil2D
3  format : rose
4
5
6  original()
7  skew([0],2,[1,1])
```

Listing 2.8: CHiLL script for loop skewing.

```
1  #define N 64
2
3  void stencil2D()
4  {
5    double A[66UL][66UL];
6    double B[66UL][66UL];
7    int i, ii;
8    int j;
9    for (j = 0; j <= 63; j += 1)
10    for (ii = 0; ii <= 3; ii += 1)
11     for (i = 16 * ii; i <= 16 * ii
          + 15; i += 1)
12       A[j][i] = B[j][i]
13         + B[j][i + 1] + B[j][i - 1]
14         + B[j + 1][i] + B[j - 1][i
             ];
15 }
```

Listing 2.9: Generated code after loop tiling.

```
1  source: stencil2D.c
2  procedure: stencil2D
3  format : rose
4
5
6
7  original()
8  tile(0,2,16,2,counted)
```

Listing 2.10: CHiLL script for loop tiling.

```
1  #define N 64
2
3  void stencil2D(){
4  double A[66UL][66UL];
5  double B[66UL][66UL];
6  int i, ii;
7  int j, jj;
8  for (jj = 0; jj <= 3; jj += 1)
9   for (ii = 0; ii <= 3; ii += 1)
10   for (j = 16*jj; j<=16*jj+15; j+= 1)
11   for (i = 16*ii; i<=16*ii+15; i+= 1)
12
13      A[j][i] = B[j][i]
14       + B[j][i + 1] + B[j][i - 1]
15       + B[j + 1][i] + B[j - 1][i];
16
17  }
```

Listing 2.11: Code after tiling loops `i`, `j`.

```
1  source: stencil2D.c
2  procedure: stencil2D
3  format : rose
4
5
6
7  original()
8  tile(0, 2, 16, 1, counted)
9  tile(0, 2, 16, 1, counted)
```

Listing 2.12: CHiLL script for tiling loops `i` and `j`.

```
1  #define N 64
2
3  void stencil1D()
4  {
5   double A[N+2];
6   double B[N+2];
7   double C[N+2];
8   int i;
9   int t;
10  for (t = 1; j <2; t++){
11   for (i = 1; i < 64; i ++)
12    A[i]= B[i-1]+B[i]+B[i+1];
13  for (i = 1; i < 64; i ++)
14    C[i]= B[i-1]+B[i]+B[i+1];
15   }
16 }
```

Listing 2.13: Input code to CHiLL for the fusion example.

```
1  #define N 64
2
3  void stencil1D()
4  {
5   double A[N+2];
6   double B[N+2];
7   double C[N+2];
8   int i;
9   int t;
10  for (t = 0; j<=1; t+=1){
11   for (i = 1; i <= 63; i+=1)
12    A[i]= B[i-1]+B[i]+B[i+1];
13    C[i]= B[i-1]+B[i]+B[i+1];
14   }
15 }
```

Listing 2.14: Automatic loop fusion in CHiLL after invoking the command `original()`.

```
1  #define N 64
2  for (i = 0; i < N; i++)
3   // Statement S0
4   b[i] = a[i-1] + a[i] + a[i+1];
```

Listing 2.15: Simple loop nest used to compute data footprint.

2.7 Computing Footprint of Array References

CHiLL can compute a footprint space for each array reference in a loop by using the iteration space of the loop in conjunction with the array reference. Footprint computation is explained using Listing 2.15. The statement in the loop has four array references: `b[i], a[i]`, `a[i-1]`, and `a[i+1]`. Each array reference generates a linear mapping which maps a point in the iteration space of the loop to a point in the data footprint space, which is simply a transformed integer set.

The linear mappings generated by array references `a[i-1]`, `a[i]`, and `a[i+1]` are ref_{-1}, ref_0 and ref_{+1}, respectively, and can be expressed as:

$$ref_{-1} := \{[i]- > [i'] : i' = i - 1\}$$

$$ref_0 := \{[i]- > [i'] : i' = i\}$$

$$ref_1 := \{[i]- > [i'] : i' = i + 1\}$$

The footprint spaces of these array references are the application of the linear mapping to the iteration space (IS) of the loop. Thus the footprint spaces of these references can be represented as:

$$footprint_{-1} = ref_{-1}(IS) := \{[i] : -1 <= i < 63\}$$

$$footprint_0 = ref_0(IS) := \{[i]- > [i'] : 0 <= i < 64\}$$

$$footprint_1 = ref_1(IS) := \{[i]- > [i'] : 1 <= i <= 64\}$$

The union of the footprint spaces gives the footprint accessed by array `a`. Computing the union can often result in an overapproximation. Further details of computing array footprints can be found in [33], and their use in transformations such as *datacopy* and array data flow analysis can be found in [33] and [42], respectively.

2.8 Extending Polyhedral Technology in CHiLL

CHiLL is not merely a polyhedral framework, it also allows manipulating the intermediate representation (IR) of the loops and statements. In fact, loop unrolling in CHiLL is not a polyhedral transformation [33]. The novel transformations added to CHiLL as part of this dissertation involve modifying the IR of the input statements,

and are not polyhedral transformations. To maintain composability of transformations in CHiLL, the approach taken is to rebuild the polyhedral representation of the program after the new transformation is applied. This involves correctly updating iteration spaces of the modified statements and their lexicographic order, and rebuilding the dependence graph.

2.9 CUDA-CHiLL

Graphics processing unit (GPU) accelerators have become a common hardware target for scientific computing, as they offer high computing power (teraflops per node) with better power efficiency than traditional multicores. Chapter 6 in this dissertation explores compiler optimizations and code generation for smooth on NVIDIA GPUs.

NVIDIA GPUs are programmer using the CUDA programming model. Nvidia GPU architectures organize the parallelism on a node in a two-level hierarchy, with a number of streaming multiprocessors (SMs), each of which has a SIMD unit with several cores. CUDA reflects two-level hierarchy. A CUDA program (called a CUDA kernel) describes a computation decomposition into a one- to three-dimensional space of thread blocks called a grid, where a block is mapped to one of the SMs. Each thread block defines a one- to three-dimensional space. A kernel thread program is executed for each point in the grid.

To help program NVIDIA GPUs, this dissertation uses CUDA-CHiLL. CUDA-CHILL is a layer on top of CHiLL which generates parallel CUDA code from sequential code. Once loop transformations and other optimizations are applied using existing machinery in CHiLL, parallel CUDA code is generated using CUDA-CHiLL.

2.9.1 Parallel Decomposition

GPUs are a tiled architecture where each streaming multiprocessor (SM) represents a separate tile. Parallel code should be partitioned across SMs so that each thread operates on mostly independent, localized data. Subdividing the iteration space of a loop into blocks or tiles with a fixed maximum size has been widely used when constructing parallel computations [43, 44, 45]. The shape and size of the

tile can be chosen to take advantage of the target parallel hardware and memory architecture.

Tiling was described in Section 2.6.3. Here we use it to create thread and block loop in CUDA. After tiling, CUDA-CHiLL is used to map one, two or three loop levels to block indices for grid dimensions, and to map up to three loops to thread indices. This is illustrated with an example of matrix vector multiplication in Listing 2.16-2.19.

The input sequential code is shown in Listing 2.16. The problem size, and thus loop trip count is N=1024. The statement in the loop body is called statement `S0`. The inputs are an NxN matrix and N-wide vector. CUDA-CHiLL has a **lua** scripting language interface. Listing 2.17 shows the lua script which directs CUDA-CHiLL to generate a CUDA kernel for matrix vector multiply.

For statement `0` in the input code, line 12 in Listing 2.17 tiles the `i,j` loops. The tile sizes for loops `i` and `j` are TI=32 and TJ=64, respectively. The tile controlling loops for `i` and `j` are `ii` and `k` respectively. The final order of the loops is given by `{ii,k,i,j}`. Listing 2.18 illustrates the loop restructuring effected by the `tile_by_index` command. Loop `i` has a trip count of TI=32, and the tile controlling loop `ii` has a trip count of N/TI = 32. Similarly, loop `j` has a trip count of TJ=64, and the tile controlling loop `k` has a trip count of N/TJ=16.

After tiling, the loop levels are assigned to CUDA blocks and threads using the `cudaize` command. Line 17 in Listing 2.17 uses the `cudaize` command to assign loop `ii` to a block dimension, and loop `i` to a thread dimension. As can be seen in Listing 2.18, the loops are marked to be assigned to blocks and threads, and Listing 2.19 shows the generated CUDA kernel execute by each thread. Loops `ii` and `i` have been replaced with per thread and per block identifiers `bx` and `tx`, and corresponding array references have been updated by CUDA-CHiLL.

2.10 Summary

This chapter describes the CHiLL compiler framework and the CUDA-CHiLL extension. CHiLL was designed to support autotuning by allowing easy and correct composition of transformations. CHiLL leverages polyhedral technology and internally uses Omega+ and Codegen+ to mathematically represent and manipulate loops

```
1  #define N 1024
2
3  void normalMV(float c[N][N], float a[N], float b[N]) {
4    int i, j;
5
6    for (i = 0; i < N; i++)
7      for (j = 0; j < N; j++)
8        a[i] = a[i] + c[j][i] * b[j];
9  }
```

Listing 2.16: The input sequential code for matrix vector multiply

```
1  TI=32
2  TJ=64
3  N=1024
4
5  tile_by_index(0, {"i","j"},
6                  {TI,TJ}, {l1_control="ii", l2_control="k"},
7                  {"ii", "k", "i", "j"})
8
9  cudaize(0, "mv_GPU", {a=N, b=N, c=N*N},
10         {block={"ii"}, thread={"i"}}, {})
```

Listing 2.17: CUDA-CHiLL script for matrix-vector multiply.

```
1  // ~cuda~ preferredIdx: bx
2  for(ii = 0; ii <= 31; ii++) {
3    for(k = 0; k <= 15; k++) {
4   // ~cuda~ preferredIdx: tx
5      for(i = 32*ii; i <= 32*ii+31; i++) {
6        for(j = 64*k; j <= 64*k+63; j++) {
7          s0(i,j);
8   }}}}
```

Listing 2.18: Tiled Code with candidate loops for CUDA blocks and threads.

```
1
2  __global__ void mv_GPU(float *a,float (*c)[1024],float *b)
3  {
4    int j;
5    int k;
6    int bx;
7    bx = blockIdx.x;
8    int tx;
9    tx = threadIdx.x;
10   for (k = 0; k <= 15; k += 1)
11     for (j = 64 * k; j <= 64 * k + 63; j += 1)
12       a[tx + 32 * bx] = a[tx + 32 * bx] + c[j][tx + 32 * bx] * b[j];
13 }
```

Listing 2.19: Generated CUDA kernel for matrix vector multiply

and to generate output code, respectively. Transformations described in this dissertation were built into CHiLL. This enabled composition of novel transformations with known compiler techniques developed over many decades of research.

CHAPTER 3

THE miniGMG BENCHMARK

This chapter describes Geometric Multigrid (GMG), a family of algorithms used to accelerate the convergence of iterative solvers. The basic operations in a Geometric Multigrid are essentially stencil computations or a mix of stencils and pointwise updates. In the past most compiler research in optimizing stencils concentrated on stencil computations in isolation. In contrast this dissertation focuses on optimizing a linear solver that uses multiple stencils.

To that end, optimized stencil kernels are generated for the the **miniGMG** benchmark. miniGMG is a compact Geometric Multigrid benchmark which proxies multigrid solvers in Adaptive Mesh Refinement (AMR) applications; it has over 2000 lines of C code with a dozen performance-critical functions. Compiler techniques are used to optimize important stencil kernels in miniGMG which dominate runtime.

The following sections present the baseline implementation of the miniGMG benchmark and highlight the stencil computations in the context of the overall solver. miniGMG has five principal operations: smooth, residual, restriction, interpolation, and ghost zone exchange executed in a sequence known as the V-cycle. The next section presents details of the V-cycle in miniGMG and describes the data decomposition and parallelism used. Code skeletons are used to make our discussion concrete. We end the chapter by describing challenges to optimizing miniGMG.

3.1 V-cycle

Multigrid methods provide a powerful technique to accelerate the convergence of iterative solvers for linear systems and are therefore used extensively in a variety of numerical simulations. Conventional iterative solvers operate on data at a single resolution and often require too many iterations. Multigrid simulations create a hierarchy of grid levels and use corrections of the solution from iterations on the

coarser levels to improve the convergence rate of the solution at the finest level. Geometric multigrid (GMG) begins with a structured mesh, where each progressively coarser grid contains half the grid points in each dimension. Given the fact that the operators are the same irrespective of grid spacing, this exponential reduction in grid sizes can bound multigrid's computational complexity to O(N), where N is the number of variables. When performance is highly correlated to computational complexity, the time spent on the finer grids will dominate the run time.

Figure 3.1 visualizes the structure of a multigrid V-cycle for solving $Lu^h = f^h$, in which L is the operator, u is the solution, f is the right-hand side, and superscripts represent grid spacings. At each grid spacing, multiple smooth operators reduce the error in the solution. The smooth can be a simple relaxation such as Jacobi, or something more complex, like a Gauss-Seidel, Red-Black (GSRB).

The right-hand side of the next coarser grid is defined as the *restriction* of the *residual* ($f^h - Lu^h$). Eventually, the grid (or collection of grids) cannot be coarsened any further using geometric multigrid. At that point, most algorithms switch to a bottom solver that can be as simple as multiple relaxations or as complicated as algebraic multigrid, a Krylov iterative solver, or a direct sparse solver. Once the coarsest grid is solved, the multigrid algorithm applies the solution (a correction) to progressively finer grids. This requires an interpolation of u^{2h} onto u^h. Smooth at the finer grid resolution is applied on the new correction.

3.2 Domain-Decomposition and Parallelism

miniGMG executes the V-cycle on a 3D domain. As shown in Figure 3.2, the benchmark creates a global 3D domain, and partitions it into subdomains of sizes similar to those found in multigrid solvers in real-world AMR applications such as CHOMBO [35]. Our configuration of miniGMG fixes the domain (problem) size to a 256^3 discretization on each multicore or GPU node, and uses subdomains of size 64^3. Thus at the finest level of the V-cycle, the 256^3 domain is decomposed into a list of 64 boxes or subdomains[1] of size 64^3. We subsequently use the terms subdomains and boxes interchangeably.

[1] miniGMG supports varying the subdomain (box) size; usually AMR applications use sizes from 32^3 to 128^3.

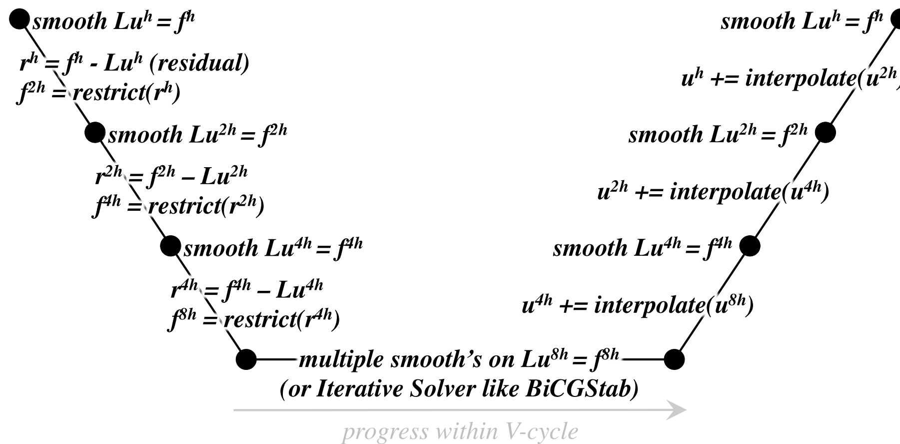


Figure 3.1: The multigrid V-cycle for solving $Lu^h = f^h$. Note, superscripts denote grid spacing.

Figures 3.2 and 3.3 presents the problem sizes, parallel decomposition, and the V-cycle that was used for the research presented here. The global 256^3 domain was decomposed into 64^3 boxes or subdomains at the finest resolution. Smooth ($Lu^h = f^h$), the residual ($f^h - Lu^h$) is computed on these 64^3 boxes and then they are coarsened to 32^3 boxes using the restrict operation (Section 1.1). This sequence is continued till we reach the bottom level of the V-cycle. In our implementation we use a truncated V-cycle where restriction stops at the 4^3 level (the bottom level). Thus our configuration of miniGMG has 5 ($64^3, 32^3, 16^3, 8^3, 4^3$) levels for the V-cycle. As this dissertation is focused on optimizing the multigrid V-cycle on a single node, a simple relaxation scheme using the smooth applied on the finer grid is also used at the bottom level. The simple relaxation scheme at the bottom level is sufficient to attain single-node multigrid convergence.[2] After smooths are applied at the coarsest 4^3 level, the boxes are interpolated to finer 8^3 boxes. The sequence of applying smooths and interpolations is continued until we reach the finest 64^3 boxes. At the finest level further smooths are applied and the V-cycle is complete.

The baseline implementation of miniGMG uses the MPI+OpenMP model to express parallelism on traditional multicore architectures. MPI is a library specification

[2]miniGMG includes both CG and BiCGStab bottom solvers to enable scalable multigrid implementations. For experiments where we have scaled to a larger number of nodes the bottom solver has been modified to use BiCGStab.

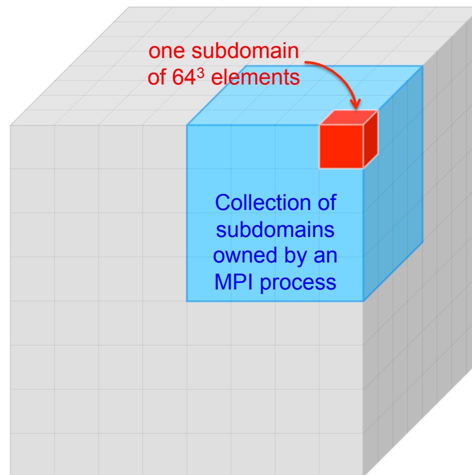


Figure 3.2: Visualization of the domain/process/subdomain hierarchy in miniGMG.

for message-passing [46]. miniGMG decomposes the global domain (256^3) into a list of subdomains which is then partitioned among MPI processes. For example, as illustrated in Figure 3.3, on a machine with four sockets we may have four MPI processes. Each process gets a (256×128×128) chunk of the global domain containing 16 boxes of size 64^3. Inside a MPI process the list of boxes is processed in parallel, each box is computed by an OpenMP thread. As we go down the V-cycle moving from larger, finer grids to smaller, coarser ones, the number of boxes remains the same and the work per thread decreases, thus reducing parallelism.

Applying a stencil on the boundary of a structured grid requires points exterior to the grid, as illustrated in Figure 3.4. The figure shows a 2D 5-point stencil being applied to an interior and a boundary point on a 4×4 grid. To apply the stencil on the boundary, an extra layer of points called a *ghost zone* [3] (gray colored points) is required. Thus to compute the illustrated stencil on a 4×4 grid, a larger 5×5 grid needs to be allocated.

When a grid is geometrically decomposed or tiled into smaller grid tiles, each grid tile needs a ghost zone. This is illustrated in Figure 3.5, where a 12×12 grid is decomposed into 4×4 tiles. Each grid tile or subdomain has a ghost zone which is a

---

[3] Ghost zones are also known as halo regions.

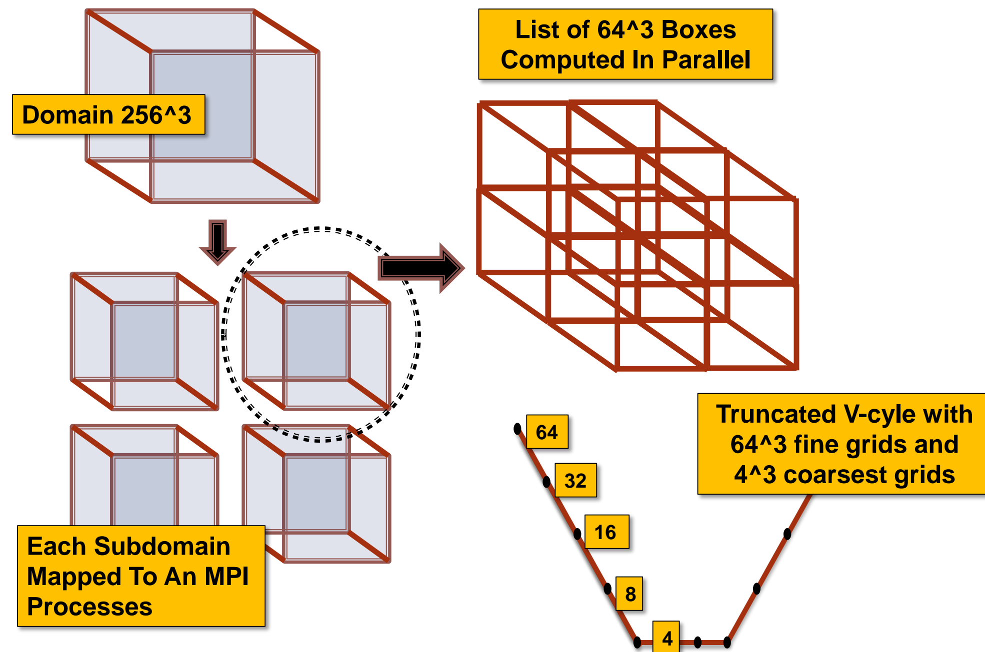


Figure 3.3: Execution of the miniGMG V-cycle. A node is assigned a 256^3 domain which is decomposed into a list of 64^3 subdomains (boxes). The list of subdomains is partitioned into MPI processes. The subdomains owned by each MPI process are then computed in parallel by a single OpenMP thread.

copy of the boundary points of its neighboring subdomains. This can be seen in the color coding used in the figure. The tile at the center has blue interior points, and colors of its ghost zone points correspond to the interior grid points of its neighbors. After a stencil computation sweeps through the entire domain and updates each point on the grid, the neighboring subdomains must exchange ghost regions to avoid having stale values. The ghost zone points are read but not updated during the stencil computation, and thus their exchange is necessary to ensure correctness. The size of the ghost zone and data exchange pattern depends on the shape of the stencil used.

The global 3D domain (256^3 at the finest level) corresponds to equally-sized grids. The grids represent the correction, right-hand side, residual, and stencil coefficients and each grid is stored as a separate array. It is important to note that the grids or arrays corresponding to the global domains are not allocated as contiguous chunks or memory. Instead miniGMG allocates the subdomains within a level as equally sized

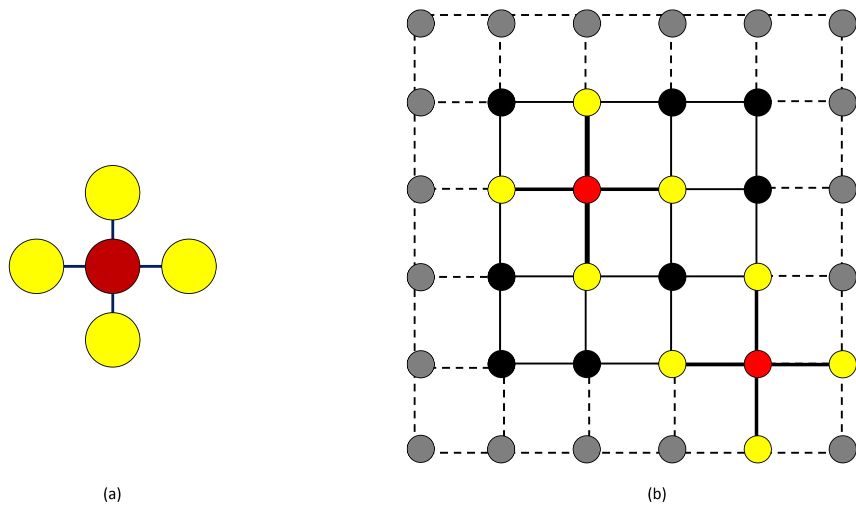


Figure 3.4: Application of a 2D 5-point stencil of a 4x4 2D grid. (a) Shape of the stencil. (b) Application of this stencil of an interior and boundary point of the grid. Ghost zone is shaded gray.

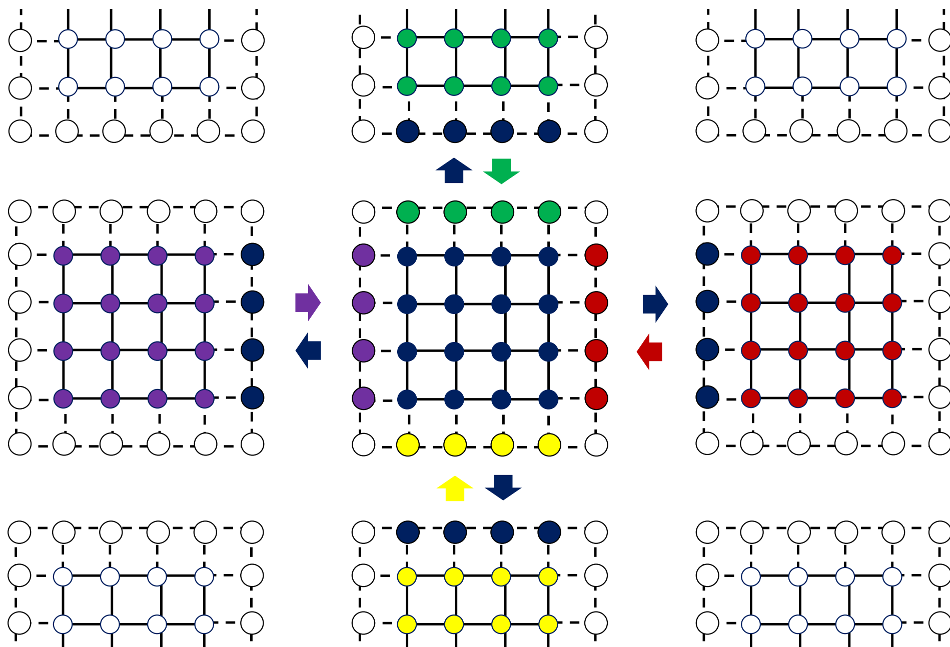


Figure 3.5: Visualization of ghost zones and data exchange between subdomains. A 12×12 domain (grid) is geometrically decomposed to 9, 4×4 subdomains (tiles). Each subdomain needs a ghost zone which must be exchanged between neighboring tiles after a stencil computation is applied to the entire domain.

grids (arrays), each grid contiguous in memory. This means at the finest level of the V-cycle a list of 64^3 grids is allocated instead of 256^3 grids.

The miniGMG benchmark allocates contiguous memory for the grids and includes the extra memory to support ghost zones needed by the suddomains. Thus, at each level of the V-cycle, the memory allocated for a grid of a subdomain is $(subdomain_size+2*ghost_zone)^3$. At the finest level where we use 64^3 subdomains with a one deep ghost zone, the baseline allocates the 64^3 grids as 66^3 contiguous elements.

The benchmark implements optimized routines which exchange the ghost zones via buffers. The data exchange does not use floating-point operations but still needs significant time. The shape of the stencil determines the data exchange pattern between subdomains, and the data exchange is required after every stencil sweep of the domain is complete.

3.3 V-cycle and Operator Code Skeletons

This section uses code skeletons extracted from the miniGMG benchmark to make the smooth, residual, and restriction/interpolations concrete. It start with code for the V-cycle to highlight the sequence in which these operations and the ghost zone exchanges are invoked.

Lines 1−20 in Listing 3.1 illustrate going down the miniGMG V-cycle and lines 24−35 show the bottom solves. Code for going back up the V-cycle has been omitted for brevity.

Application of multiple smooths at each level (0 being the finest and NumLevel being the bottom) is shown in lines 2−15. Lines 10−14 show the smooth being applied to the subdomains (boxes) in parallel, and lines 6−7 highlight the ghost zone exchange required prior to and between applications of each smooth. Application of residual (line 16) is similar to smooth. Residual is also applied in parallel on the boxes and there is a ghost zone exchange prior to the residual computation.

Unlike smooth, residual is only applied once. Restriction (line 19) follows residual, it is computed once and applied in parallel on the boxes. No ghost zone exchange is required prior to applying restriction. Going back up the V-cycle is similar, but the residual is not applied, and restrict is replaced by interpolation. Lines 24−35 show

```
1  for(level=0; level<NumLevel; level++){
2          for ( smooth=0; smooth<NumSmooths; smooth++ ){
3  
4                  // communication phase...
5                  // the boxes exhange boundaries with neighbors
6                  exchange_boundary_phi();
7  
8                  // apply smooth on each box in parallel
9                  # pragma omp parallel for private (box)
10                 for (box=0; box<NumBoxInSubdomain; box++){
11                  color=smooth;
12                  gsrb_smooth_function(Domain->SubDomain[box],phi,rhs,color);
13                 }
14         }
15         compute_residual();
16         // restrict to form the coarse and smaller grid
17         // We go down the v-cycle, ie. from a 64^3 grid to a 32^3 grid
18         compute_restriction();
19 }//down
20 
21 // bottom solve....
22 
23 for (smooth=0; smooth<NumBottomSmooths; smooth++){
24 
25         exchange_boundary_phi();
26 
27         // apply smooth on each box in parallel
28         # pragma omp parallel for private (box)
29         for (box=0; box<NumBoxInSubdomain; box++){
30                 color=smooth;
31         gsrb_smooth_function(Domain->SubDomain[box],phi,rhs,color);}
32         }//bottom solve
33 }
34 // back up the v-cycle.....
```

Listing 3.1: miniGMG V-cycle

the bottom solve being applied with the ghost zone exchanges between each application. The bottom smooth is applied many more times than the number of smooths at other levels.

3.3.1 Smooth

Smooths using a number of stencils have been optimized in this dissertation. miniGMG has been configured to use smooths with Gauss-Seidel Red-Black (GSRB) and Jacobi iterations of variable-coefficient stencils, and Jacobi iterations of constant-coefficient stencils. This section presents a smooth using a variable-coefficient stencil. Listing 3.2 shows code for a smooth ($Lu^h = f^h$), where $L = a\vec{\alpha}I - b\nabla\vec{\beta}\nabla$.

Programmers often wish to maintain flexibility and thus create smooth operators by composing multiple simpler operators, as illustrated in Listing 3.2. The smooth operator calculates the Laplacian, Helmholtz, and a Gauss-Seidel relaxation in sequence. The first loop nest (lines 2−13) calculates $\nabla\vec{\beta}\nabla u$, storing it to a temporary array. This loop executes the variable-coefficient stencil. It reads in points from the grid *phi* and multiplies it with appropriate coefficients from *beta_i*, *beta_j* and *beta_k*. The next loop nest (lines 15−20) updates that temporary by calculating $a\vec{\alpha}u - b\nabla\vec{\beta}\nabla u$. The final loop-nest(lines 22−29) performs the GSRB relaxation using the temporary array.

The same variable-coefficient stencil can be used with Jacobi relaxation. Listings 3.3 and 3.4 compare the loop structure corresponding to GSRB and Jacobi relaxes respectively. For clarity, the statements for Laplacian, Helmholtz and the Relaxation from Listing 3.2 have been represented by $S0$, $S1$ and $S2$. Jacobi iterations have statements $even_S0$, $even_S1$, and $even_S2$, odd_S0, odd_S1, and odd_S2. Statement odd_S0 executes the same stencil as $even_S0$, except in odd_S0 the array temp is read and phi is updated. Similarly in odd_S1 and odd_S2, temp and phi are interchanged. Thus, odd-numbered smooth applications with Jacobi relaxations update temp, and even-numbered smooths update phi. The crucial difference between the two relaxation schemes is the complex if-condition used in GSRB (line 26 in Listing 3.2). Jacobi updates every point on the grid and does not have this condition, and has a simpler if-condition to check whether phi or temp gets updated.

```
// Laplacian(phi) = b div beta grad phi
  for (k=0;j<N;k++)
    for (j=0;j<N;j++)
      for (i=0;i<N;i++)
       // statement S0
       temp[k][j][i] =b*h2inv*(
       beta_i[k][j][i+1]*( phi[k][j][i+1] -phi[k][j][i] )
      -beta_i[k][j][i]  *( phi[k][j][i]  -phi[k][j][i-1])
      +beta_j[k][j+1][i]*( phi[k][j+1][i]-phi[k][j][i] )
      -beta_j[k][j][i]  *( phi[k][j][i]  -phi[k][j-1][i])
      +beta_k[k+1][j][i]*( phi[k+1][j][i]-phi[k][j][i] )
      -beta_k[k][j][i]  *( phi[k][j][i]  -phi[k-1][j][i]));

// Helmholtz(phi) = (a alpha I - laplacian)*phi
  for (k=0;j<N;k++)
    for (j=0;j<N;j++)
      for (i=0;i<N;i++)
       // statement S1
       temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i]-temp[k][j][i];

// GSRB relaxation: phi = phi - lambda * (helmholtz-rhs)
  for (k=0;j<N;k++)
    for (j=0;j<N;j++)
      for(i=0;i<N;i++){
       // color is 0 for Red pass, 1 for Black
       if((i+j+k+color) % 2 ==0)
        // statement S2
        phi[k][j][i] = phi[k][j][i]-lambda[k][j][i]*(temp[k][j][i]-rhs[k][j
            ][i]);
       }
```

Listing 3.2: Smooth operator with Gauss-Seidel Red-Black relaxations.

```
1
2
3    for (k=0;j<N;k++)
4      for (j=0;j<N;j++)
5        for (i=0;i<N;i++)
6         //Laplacian
7         S0();
8
9    for (k=0;j<N;k++)
10     for (j=0;j<N;j++)
11       for (i=0;i<N;i++)
12        //Helmholtz
13        S1();
14
15   for (k=0;j<N;k++)
16     for (j=0;j<N;j++)
17       for(i=0;i<N;i++)
18       if((i+j+k+color)/2==0)
19         // GSRB update
20         S2();
```

Listing 3.3: GSRB relaxation.

```
1  if(smooth_application % 2 == 0)
       {
2
3    for (k=0;j<N;k++)
4      for (j=0;j<N;j++)
5        for (i=0;i<N;i++)
6         even_S0();
7
8    for (k=0;j<N;k++)
9      for (j=0;j<N;j++)
10       for (i=0;i<N;i++)
11        even_S1();
12
13   for (k=0;j<N;k++)
14     for (j=0;j<N;j++)
15       for(i=0;i<N;i++)
16        even_S2();
17  } else{
18
19    for (k=0;j<N;k++)
20     for (j=0;j<N;j++)
21       for (i=0;i<N;i++)
22        odd_S0();
23
24   for (k=0;j<N;k++)
25     for (j=0;j<N;j++)
26       for (i=0;i<N;i++)
27        odd_S1();
28
29   for (k=0;j<N;k++)
30     for (j=0;j<N;j++)
31       for(i=0;i<N;i++)
32        odd_S2();
33  }
```

Listing 3.4: Jacobi relaxation.

3.3.2 Residual

Residual uses the same stencil as smooth. The code is illustrated in Listing 3.5. For brevity of presentation the residual computation has been shown as one loop nest instead of separate loop nests similar to the baseline smooth. Like smooth, residual uses a stencil and requires ghost zone data, and thus ghost zones are exchanged before computing the residual. The residual is computed once per multiple application of smooth and contributes far less to the overall solve time.

3.3.3 Restriction and Interpolation

Restriction and interpolation are integral operations to the Geometric Multigrid. The restriction operation is applied when going *down* the multigrid V-cycle. Restriction takes as input a grid corresponding to the residual and computes a coarser-grained grid from it. The code for restriction is shown in Listing 3.6. The piecewise constant restriction used here is common to finite-volume methods. It is a constant-coefficient stencil which reads in eight points from the input fine-resolution grid, computes an average and writes it to a coarser output grid. The output grid is half the size of the input grid in each dimension, and this leads to the nonunit loop strides, and the indexing of the coarse grid involves a division by the constant coarsening factor. The difference in size between input and output grids for the restriction operation differentiates it from the other stencil operations which use equal-sized grids. The other feature unique to the stencil used in the restriction operation is that it does not read the ghost zone points of the input fine grid. Thus no ghost zone exchange is required prior to applying the restriction.

Interpolation is a scatter operation which performs the inverse of restriction. It maps a point from a coarse grid to eight points in the fine grid when going *up* the V-cycle. Interpolation is not an optimization target in this dissertation.

3.4 Optimization Challenges

Optimizations for miniGMG need to address three broad performance challenges on current and future architectures:

- **Reducing data movement.**

  Stencil computations are commonly memory-bandwidth limited, and thus,

```
1  // Input: Residual Operator
2  for(k=0;k<K;k++)
3    for(j=0;j<J;j++)
4      for(i=0;i<I;i++)
5        // statement S3 : Compute residual
6        res[k][j][i] = rhs[k][j][i]
7          - a * alpha[k][j][i] * phi[k][j][i]
8          + b*h2inv*(
9             beta_i[k][j][i+1]*( phi[k][j][i+1]-phi[k][j][i]  )
10           -beta_i[k][j][i]  *( phi[k][j][i]  -phi[k][j][i-1])
11           +beta_j[k][j+1][i]*( phi[k][j+1][i]-phi[k][j][i]  )
12           -beta_j[k][j][i]  *( phi[k][j][i]  -phi[k][j-1][i])
13           +beta_k[k+1][j][i]*( phi[k+1][j][i]-phi[k][j][i]  )
14           -beta_k[k][j][i]  *( phi[k][j][i]  -phi[k-1][j][i])
15         );
```

Listing 3.5: Residual operator

```
1  // Input: Restriction Operator
2  for(k=0;k<K;k+=2)
3    for(j=0;j<J;j+=2)
4      for(i=0;i<I;i+=2)
5        coarser_res[k/2][j/2][i/2] = 0.125 *(
6          res[k  ][j  ][i] + res[k  ][j  ][i+1] +
7          res[k  ][j+1][i] + res[k  ][j+1][i+1] +
8          res[k+1][j  ][i] + res[k+1][j  ][i+1] +
9          res[k+1][j+1][i] + res[k+1][j+1][i+1]
10        );
```

Listing 3.6: Restriction operator

minimizing data traffic is the single most important factor in optimizing them for modern architectures. Geometric Multigrid uses a sequence of stencil computations, and thus optimizations for GMG must aim to increase locality for both individual stencils and across stencil computations.

- **Parallel code generation.**

  With the increasing number of threads on a node, expressing parallelism in the generated code is crucial to getting good performance. Parallel code generation for miniGMG is particularly interesting, as parallelism decreases on descending the V-cycle and threading strategies need to be tailored to adapt to the level of the V-cycle.

- **Managing floating-point computation.**

  Smooths in Geometric Multigrid can use compute-intensive stencils leading to floating-point computations being the bottleneck. In such cases optimizations must aim to reuse computation to reduce floating-point operations. In addition to computation reuse, code generation must efficiently use architectural features such as SIMD-units which boost the performance of floating-point operations.

There is considerable interaction between these optimizations, for example, locality increasing optimizations for the miniGMG may increase computation and decrease parallelism. To explore these trade-offs, automatic code-tuning frameworks must compose and apply sequences of optimizations. Generating high-performance code will require searching the space of such compositions to pick the one best suited for a given input stencil and target architecture.

CHAPTER 4

COMMUNICATION-AVOIDING OPTIMIZATIONS

The principal operations in the Geometric Multigrid commonly execute less than one floating-point operation for every byte of data. This low arithmetic intensity, coupled with the fact that data movement is far more expensive than floating-point operations, makes the performance of miniGMG memory bandwidth limited. Thus, the key to achieving high performance is to reduce data movement.

In this chapter we introduce the types of data movement in the miniGMG benchmark, followed by the description of optimizations targeting data traffic. The description of the optimizations is followed by the details of their implementation in the compiler. The final part of the chapter presents the performance results and analysis of the generated code.

As the focus of this chapter is on reducing data movement, the variable-coefficient 3D 7-point stencil is used in this chapter. Variable-coefficient stencils read in many more arrays when compared to constant-coefficient ones, and consequently require higher volumes of data movement. Thus, applying the optimizations developed here on variable-coefficient stencils highlights the efficacy of the optimizations.

4.1 Types of Communication

Data movement in the miniGMG can be classified as either horizontal communication or vertical communication:

- **Vertical Communication**

  The Geometric Multigrid operators smooth, residual, restrict and interpolation (Listings 3.2, 3.5 and 3.6), are all three deep loop nests which sweep over 3D grids. Each 3D grid is stored as an array, and sweeping or streaming

through them generates data movement through the memory hierarchy. This memory traffic generated by each application of an operator is termed *vertical communication*.

- **Horizontal Communication**

  Sections 3.1 to 3.3 describe how miniGMG decomposes the global domain into subdomains which are processed in parallel using threads (OpenMP) and processes (MPI). After an operation such as a smooth is applied to the subdomains, a ghost zone exchange between neighboring subdomains is required. This data movement between threads and processes to update the ghost zones is called Horizontal communication.

4.2 Communication-Avoiding Optimizations

Vertical communication is required for each operator application as it sweeps through 3D grids, and horizontal communication is required between operator applications to update ghost zones. This chapter presents optimizations to reduce both types of data movement. Table 4.1 lists the optimizations and the type of data movement they target. The optimizations are presented in the context of miniGMG used with a variable-coefficient smooth and residual, Listings 3.2 and 3.5, respectively. The variable-coefficient stencils used in these operations have a very low arithmetic intensity and highlight the data-movement optimizations.

4.2.1 Fusing Components of Smooth

Programmers often wish to maintain flexibility and thus create smooth operators by composing multiple simpler operators, as illustrated in Listing 3.2 and 3.4. The smooth operator calculates the Laplacian, Helmholtz, and either a Gauss-Seidel or Jacobi relaxation in sequence. Each of these simpler operators is a loop nest which sweeps through the grids. The finest level of the miniGMG V-cycle has a 256^3 domain per node decomposed into 64^3 subdomains. Each subdomain stores the grids for `phi, temp, alpha, beta_i, beta_j, beta_k, lambda` and `rhs` as 66^3 arrays; the net memory requirement of these arrays is more than 1GB of memory and does not fit into last-level caches. Since the finer grids do not fit into the last-level cache, each

Table 4.1: The table illustrates the classification of optimizations described in this chapter as reducing vertical communication, horizontal communication, or both. Parallel code generation has been been left out as it is not a communication-avoiding optimization but helps improve wavefront, which reduces vertical communication.

Optimization	Vertical	Horizontal
Smooth Operator Fusion	✓	-
Deep Ghost Zones	-	✓
Wavefront	✓	-
Residual Restriction Fusion	✓	-
Smooth-Residual-Restriction Wavefront	✓	✓

loop nest or grid sweep results in data movement between DRAM and the last-level cache.

The three separate loop nests in smooth generate high DRAM traffic as the grids are streamed into cache three times. To reduce this data movement, the compiler fuses the multiple smooth operators together. Fusion is itself a vertical communication-avoiding optimization, since the results computed by one operator will remain in cache when used as input by the next operator.

The output of fusion for the GSRB and Jacobi smooths is outlined in Listings 4.1 and 4.2, respectively. Loop fusion for the GSRB smooth requires the if-condition that previously guarded just the GSRB update statement `S2()` to now guard the execution of all three statements `S0()`, `S1()`, and `S2()`.

An additional communication-avoiding optimization for the GSRB smooth is to replace the array `temp` with a scalar and not write it back to memory on completion. The first two statements write to `temp`, and the last statement uses the value written to update `phi`. Replacing `temp` with a scalar saves vertical communication associated with accessing `temp`, this is not possible in the Jacobi smooth where `temp` is used across the two `(k,j,i)` loop nests.

4.2.2 Deep Ghost Zones

Smooth contains a stencil computation which requires ghost zones. Stencil computations performed on the boundary of the subdomains read these ghost zone points but do not compute values to update them. Thus, after a stencil computation sweeps through the grids, the ghost zone values become stale and they need to be updated

```
1         for (k=0;j<N;k++){
2                 for (j=0;j<N;j++){
3                         for (i=0;i<N;i++){
4
5                         if((i+j+k+color)%2==0){
6
7                                 S0(k,j,i); /*Laplacian*/
8                                 S1(k,j,i); /*Helmholtz*/
9                                 S2(k,j,i); /*GSRB relaxation*/
10
11                        }/*end if*/
12        }}}
```

Listing 4.1: Fused GSRB smooth.

```
1   if(sweep % 2 == 0) {
2
3         for (k=0;j<N;k++){
4                 for (j=0;j<N;j++){
5                         for (i=0;i<N;i++){
6
7                         even_S0(k,j,i); /*Laplacian*/
8                         even_S1(k,j,i); /*Helmholtz*/
9                         even_S2(k,j,i); /*Jacobi relaxation*/
10        }}}
11
12  }else if(sweep %2 == 1){
13
14        for (k=0;j<N;k++){
15                for (j=0;j<N;j++){
16                        for (i=0;i<N;i++){
17
18                        odd_S0(k,j,i); /*Laplacian*/
19                        odd_S1(k,j,i); /*Helmholtz*/
20                        odd_S2(k,j,i); /*Jacobi relaxation*/
21        }}}
22
23 }/*end if*/
```

Listing 4.2: Fused Jacobi smooth.

by data exchange with neighboring subdomains. Deeper ghost zones reduce ghost zone exchanges by performing redundant computations to update ghost zones. This is illustrated in Figure 4.1, where a 2D 5-point stencil is applied to a 4x4 grid with a two-deep ghost zone. The first stencil application computes values for a 6x6 grid, (blue points are updated), the following stencil sweep uses the 6x6 grid to compute the values for the final 4x4 grid (red points). The second stencil sweep was possible without a prior data exchange because the first sweep updated a 6x6 grid, which meant the ghost zone to compute the 4x4 grid has updated values. The blue points shown in Figure 4.1(b) are computed redundantly but they reduced horizontal communication with each consecutive stencil sweep working on a smaller grid. Thus, deeper ghost zones reduce the frequency of horizontal communication.

Deeper ghost zones exchange fewer messages but change the pattern of communication between neighboring subdomains. Figure 4.2 shows the communication pattern for a two-deep ghost zone used by a 2D 5-point stencil. When compared to Figure 3.5, it is evident that a larger message corresponding to deeper ghost zones must be exchanged. In addition to the larger messages, the number of neighbors involved also increases, as corner points in the ghost zone also need to be updated.

The output code for GSRB and Jacobi smooths with deeper ghost zones are shown in Listings 4.3 and 4.4, respectively. A new `t`-loop has been added to apply smooth multiple times. The loop bounds for the `k,j`, and `i`-loops are now functions of `t` to ensure that with each application of smooth the region of the grid which is computed shrinks in all dimensions and uses only valid data. In addition to the loop bounds, the if-condition used in GSRB also has to include the loop index `t`, since consecutive smooths update either the red or black points.

Deeper ghost zones reduce horizontal communication at the expense of redundant computation. As one goes down the V-cycle, the coarser grids have much-reduced computation and cannot afford redundant computation. Thus ghost zone depth must be tuned for each level of V-cycle to find the optimum balance.

4.2.3 Wavefront Computation

Deep ghost zones allow multiple smooths to be applied before a ghost zone exchange. As each smooth application is a grid sweep, the multiple grid sweeps generate

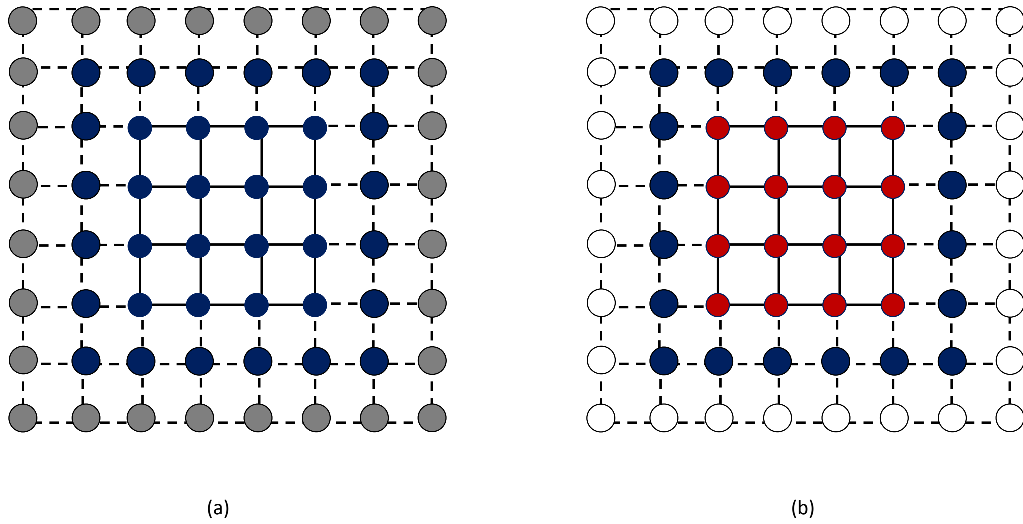


Figure 4.1: Two applications of a 3D 5-point stencil on a 4x4 grid with a two-deep ghost zone. (a) The stencil is first applied on a 8x8 grid to compute the values for the 6x6 grid (blue points). The outer layer of grid points shaded grey are read and used as the ghost zone. (b) The second stencil sweep uses the 6x6 grid to compute the output of the 4x4 grid (red points). For the second stencil sweep, the blue points are used as the ghost zone.

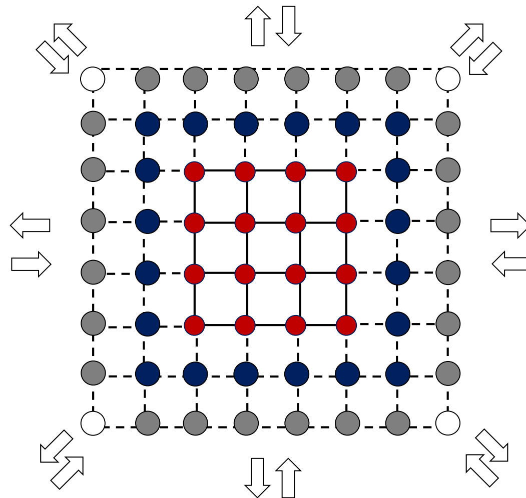


Figure 4.2: Deeper ghost zone changes the communication pattern and volume between a subdomain and its neighbors. A one-deep ghost zone requires exchange with left, right, top, and bottom neighbors. An additional layer of ghost zone now requires exchange with neighbor on the corners as well. In addition, a larger volume of data must be exchanged.

```
1      /* d = ghost zone depth */
2      for (t=0;t<d;t++){
3              for (k=t-(d-1);j< N +(d-1);k++){
4                      for (j=t-(d-1);j< N +(d-1);j++){
5                              for (i=t-(d-1);i< N +(d-1);i++){
6
7                              if((i+j+k+t+color)%2==0){
8
9                              S0(t,k,j,i); /*Laplacian*/
10                             S1(t,k,j,i); /*Helmholtz*/
11                             S2(t,k,j,i); /*GSRB relaxation*/
12
13                     }/*end if*/
14         }}}}
```

Listing 4.3: Fused GSRB smooth with overlapped ghost zones.

```
1      /* d = ghost zone depth */
2      for (t=0;t<d;t++){
3
4              if(t %2 == 0) {
5
6              for (k=t-(d-1);j< N +(d-1);k++){
7               for (j=t-(d-1);j< N +(d-1);j++){
8                for (i=t-(d-1);i< N +(d-1);i++){
9
10                      even_S0(t,k,j,i); /*Laplacian*/
11                      even_S1(t,k,j,i); /*Helmholtz*/
12                      even_S2(t,k,j,i); /*Jacobi relaxation*/
13                     }}}
14
15             }else if(t %2 == 1){
16
17             for (k=t-(d-1);j< N +(d-1);k++){
18              for (j=t-(d-1);j< N +(d-1);j++){
19               for (i=t-(d-1);i< N +(d-1);i++){
20
21                      odd_S0(t,k,j,i);/*Laplacian*/
22                      odd_S1(t,k,j,i); /*Helmholtz*/
23                      odd_S2(t,k,j,i); /*Jacobi relaxation*/
24                     }}}
25
26         }}/*end if*//*end t*/
```

Listing 4.4: Fused Jacobi smooth with overlapped ghost zones..

high vertical communication between the caches and DRAM. A wavefront computation is used to reduce this vertical communication. A wavefront fuses multiple grid sweeps into one, thereby reducing DRAM traffic.

Wavefront computation is explained in terms of GSRB smooth, which uses a 3D 7-point stencil, and then extended to the Jacobi smooth. The following discussion assumes that a four-deep ghost zone is used, which means four GSRB smooths are applied: red (R1), black (B1), Red (R2), and finally, a black (B2) sweep. GSRB partitions the grid into red and black points, where a red point has only black neighbors and vice-versa. A red sweep updates red points and a black sweep updates black ones.

A sweep through a 3D N×N×N grid can be visualized as streaming through N $\{k = 0, ..., N-1\}$ 2D N×N `ij`-planes. A Wavefront computation fuses the four grid sweeps R1-B1-R2-B2 into one by computing values for a set of four `ij`-planes at a time. This is illustrated in Figure 4.3(a), which shows the cross section of a 3D grid and illustrates how a wavefront works on `ij`-planes. The first red sweep, R1, on plane $k = z + 3$ is computed, followed by B1 on plane $k = z + 2$, R2 on $k = z + 1$, and B2 on plane $k = z$. At this stage, plane $k = z$ has gone through all four sweeps and the wavefront now performs the same sequence on planes $k = z+4$ to $k = z+1$, as shown in Figure 4.3(b). An `ij`-plane goes through all the four sweeps R1, B1, R2, and B2 as the wavefront progresses through it, and in one sweep of the grid, four smooths are applied. The wavefront described here processes four planes, and is thus termed four deep.

Jacobi smooth uses an out-of-place stencil computation where every odd-numbered smooth application read `phi` and updates `temp`, and every even-numbered application reads `temp` and updates `phi`. Figure 4.4 illustrates a four-deep wavefront for the Jacobi smooth. The wavefront needs a four-deep ghost zone and fuses four smooths into one. The figure highlights the ping-pong between `temp` and `phi`.

Jacobi iterations like GSRB use a 3D 7-point stencil, and thus read in three `ij`-planes: top, center, and bottom, and outputs a single `ij`-plane. In the wavefront shown in Figure 4.4, the first plane output is updated to `temp`. Plane $k = z + 7$ of `temp` is computed using planes $k = z + 8$ (top), $k = z + 7$ (center), and the bottom

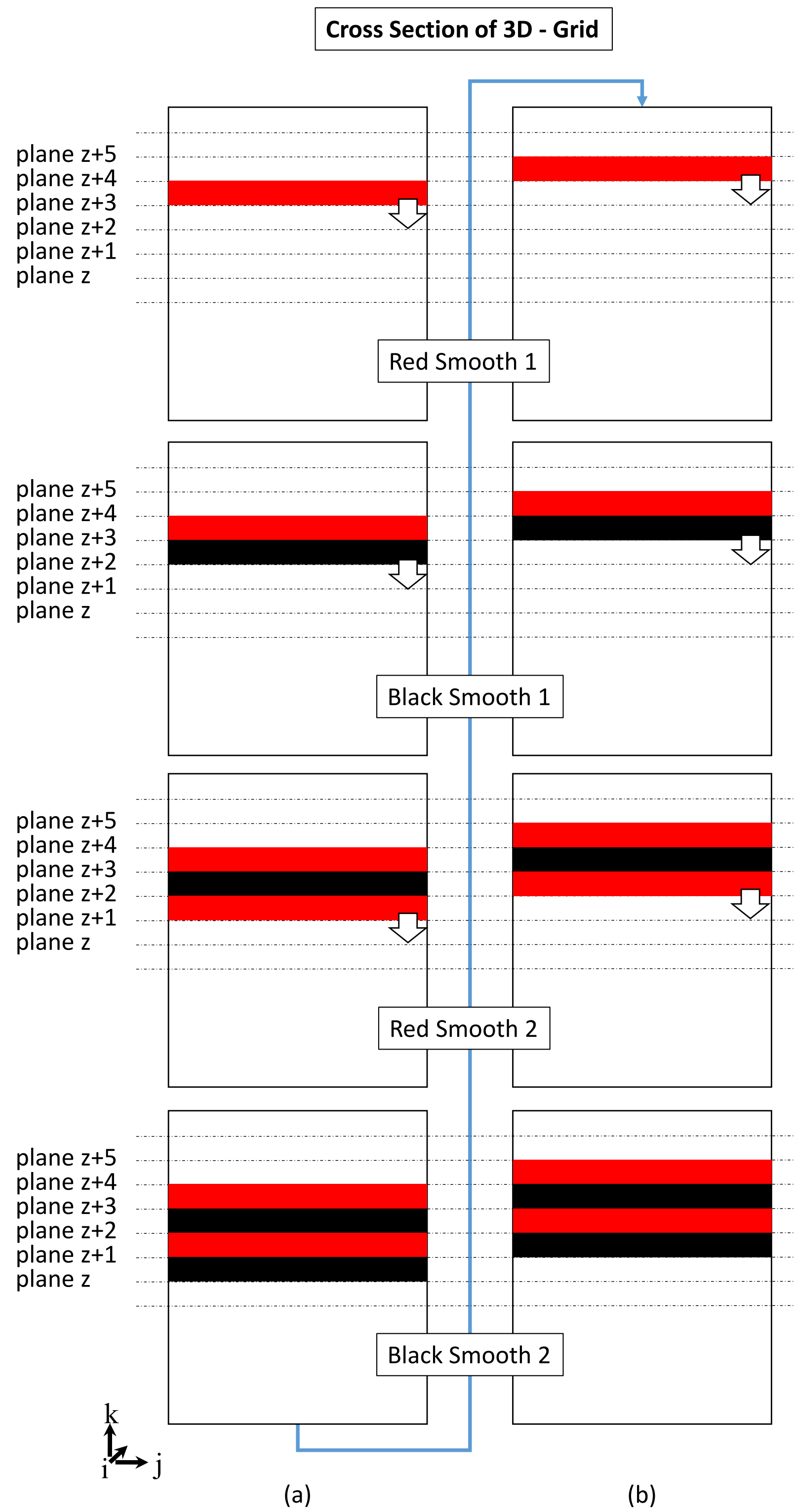


Figure 4.3: Progress of the GSRB wavefront.

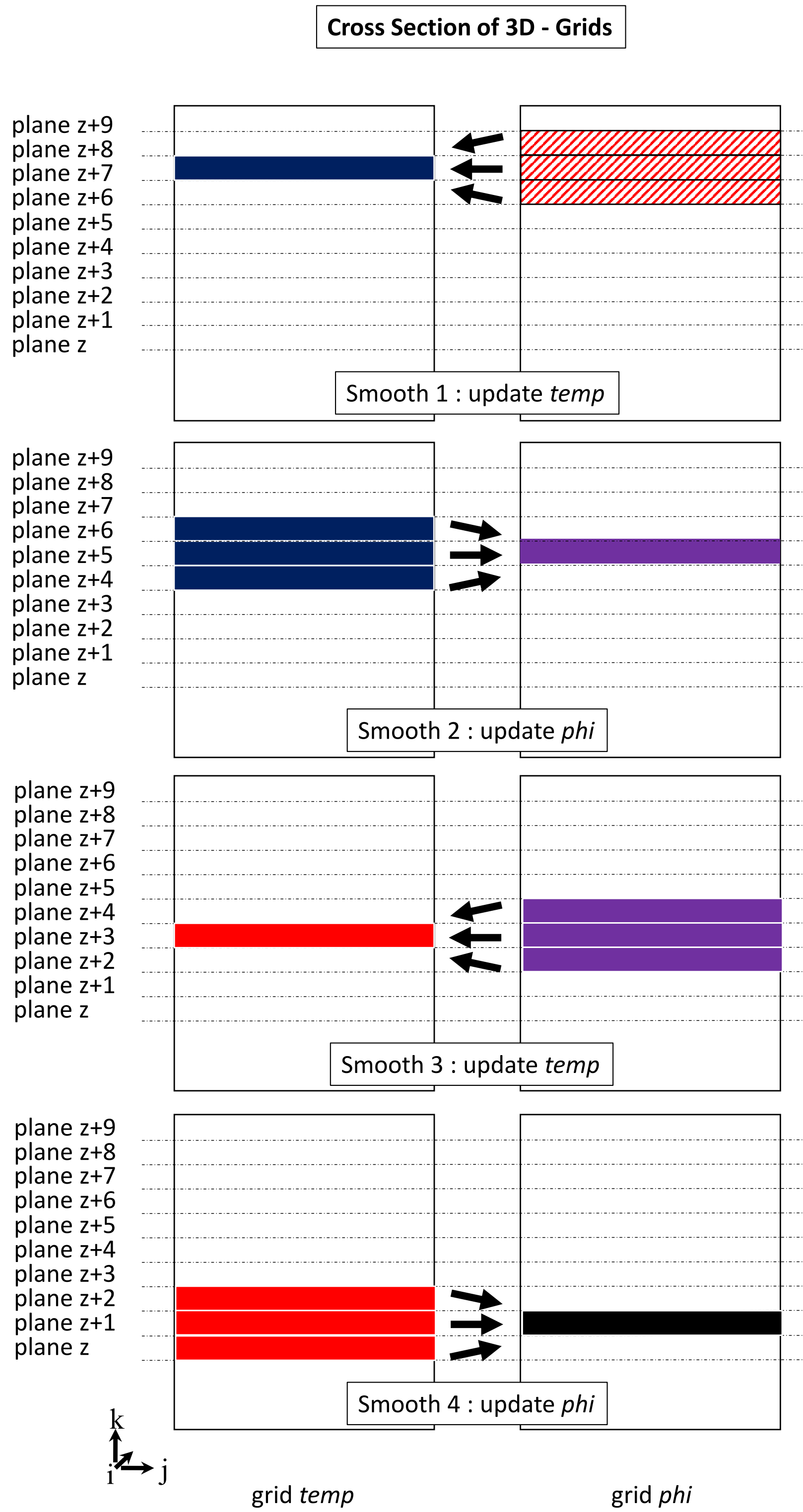


Figure 4.4: Jacobi wavefront.

plane $k = z + 6$ of `phi`. The next step of the wavefront updates phi and computes plane $k = z + 5$ of `phi` using planes $k = z + 6$, $k = z + 5$, and $k = z + 4$ from `temp`. The next two updates proceed similarly, as shown in the figure.

In the wavefront for Jacobi iterations, there is a difference of 2 between the planes that are updated; for example, in Figure 4.4 planes $z + 7$, $z + 5$, $z + 3$, and finally $z + 1$ are updated in sequence. The GSRB wavefront, on the other hand, updates 4 consecutive planes: $z + 3$, $z + 2$, $z + 1$, and z. This crucial difference between wavefronts for GSRB and Jacobi iterations arises from data dependences and means that many more planes need to be read and be held in memory for Jacobi iterations, leading to a larger working set.

Code Listings 4.5 and 4.6 show the skeleton code for the four-deep GSRB and Jacobi wavefronts applied on a 64^3 box, respectively. It is important to note that in both code fragments the `k`- and `t`-loops have been interchanged (or permuted). The time step `t`-loop which was previously outermost and was responsible for applying multiple grid sweeps is now nested inside the outermost `k`-loop. This means the k-dimension is scanned or iterated through only once and corresponds to a single grid sweep. The two innermost `j`- and `i`-loops are not changed, as we do not modify how points in an `ij`-plane are updated/traversed.

The modified indexing used in the statements `(S0, S1, ...)` in the two code listings determines the order in which the `ij`-planes are updated in the wavefront computations. The indexing of the statements prior to creating a wavefront were of the form `(t,k,j,i)` i.e., `S0(t,k,j,i)` (Listing 4.3). In the GSRB wavefront the indexing changes to `(t,k-t,j,i)`, and for Jacobi it is `(t,k-2*t,j,i)`. The modified array index expression for the `k`-dimension means that at each iteration of the `t`-loop, the `ij`-plane that is updated is one (GSRB) or two (Jacobi) planes lower than the one updated in the previous iteration of `t`. As seen previously, for four iterations of the t-loop (from 0 to 3), this translates to planes $k = z + 3$, $k = z + 2$, $k = z + 1$, and $k = z$ getting updated for GSRB. And for Jacobi, these are planes $k = z + 7$, $k = z + 5$, $k = z + 3$, and $k = z + 1$.

```
1    for (k = -3; k <= 66; k++) {
2      for (t = 0; t <= min(3,intFloor(t+3,2)); t++) {
3        for (j = t-3; j <= -t+66; j++) {
4          for (i= t-3+intMod(-k-color-j-(t-3),2); i<=-t+66; i+=2)
5          {
6                    S0(t,k-t,j,i); /* Laplacian */
7                    S1(t,k-t,j,i); /* Helhmoltz */
8                    S2(t,k-t,j,i); /* GSRB      */
9          }}}}
```

Listing 4.5: GSRB Wavefront for a 64^3 box with a four-deep ghost zone.

```
1
2    for (k = -3; k <= 70; k++) {
3      for (t = max(0,k-66); t <= min(3,intFloor(t+3,3)); t++) {
4       if(t % 2 == 0){
5        for (j =t-3; j <= 66-t; j++) {
6          for (i= -3; i<=66; i++)
7          {
8                    even_S0(t,k-2*t,j,i); /* Laplacian */
9                    even_S1(t,k-2*t,j,i); /* Helhmoltz */
10                   even_S2(t,k-2*t,j,i); /* GSRB      */
11         }
12      }}
13     if(t % 2 == 1){
14         for (j =t-3; j <= 66-t; j++) {
15         for (i= -3; i<=66; i++)
16         {
17                   odd_S0(t,k-2*t,j,i); /* Laplacian */
18                   odd_S1(t,k-2*t,j,i); /* Helhmoltz */
19                   odd_S2(t,k-2*t,j,i); /* GSRB      */
20         }
21      }
22     }
23   }}
```

Listing 4.6: Jacobi Wavefront for a 64^3 box with a four-deep ghost zone.

4.2.4 Parallel Code Generation

Wavefronts hold multiple planes in memory, thus increasing the working set. This may lead to spilling out of the faster caches (L1/L2). We generate nested multi-threaded code via OpenMP to share planes across threads and reduce the working set per thread.

Figure 4.5 illustrates how an `ij`-plane in a box gets shared between a number of OpenMP threads for the GSRB smooth with a four-deep wavefront. The four threads tile the iteration space of the `j`-loop. Each thread processes the four R-B-R-B planes of the wavefront before needing to synchronize.

Code Listings 4.7 and 4.8 show skeleton codes for a four-deep GSRB and Jaobi wavefront applied on a 64^3 box, respectively. In Listing 4.5 there is a single thread processing the box, and in Listing 4.7 there are 12 OpenMP threads collaboratively processing the box. The `j`-loop in Listing 4.5 which performed 70 iterations (-3 to 66) has been tiled, and each tile is assigned to a thread. With 12 threads, each thread gets $\lceil 70/12 \rceil = 6$ iterations. Line 16 shows where the threads working on a box need to synchronize. The synchronization point is after the completion of the `t`-loop. This means that threads can process all four sweeps R-B-R-B before synchronizing.

The OpenMP barrier in line 16 of Listing 4.7 means that all the threads working on a box wait for all other threads to finish processing a set of 4 planes before proceeding. This is expensive and not required in this case, as each thread only needs to sync with its immediate neighbors. For example, in Figure 4.5 thread 2 needs to sync with thread 1 and thread 3. To ameliorate the effect of the expensive barrier we followed the strategy of expert manual tuners in [32] and generated code to implement spinlocks. An array of locks depending on the number of collaborating threads was created, and each thread only waited on its two neighbors. Spinlocks improve performance, but unfortunately this is not a portable approach and breaks programming (OpenMP) abstractions.

Collaborative threading with multiple threads per box for the Jacobi wavefront is shown in Listing 4.8. The generated code is simpler, but the threading is more expensive. This is because, due to data dependences, the threads must synchronize after each plane is processed. This is more expensive than GSRB, where multiple planes can be processed before synchronization. For the Jacobi, the OpenMP parallel

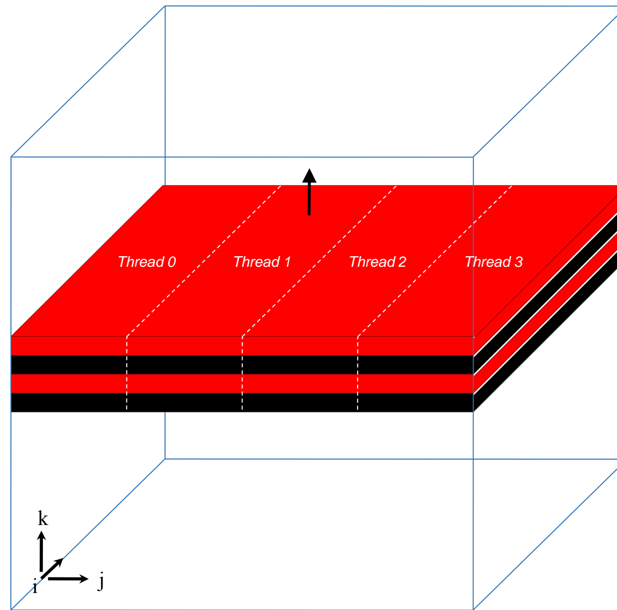


Figure 4.5: Multiple threads working collaboratively to process a subdomain/box

for directive was used rather than creating an OpenMP parallel region, as in the case of GSRB.

Generating code where multiple threads process a box creates three strategies for thread decomposition. As illustrated in Figure 4.6, we can have interbox parallelism, nested parallelism and intrabox parallelism. Each box is processed by a single thread in interbox parallelism, and in intrabox parallelism a single box is processed with all threads working on it. Nested parallelism has multiple boxes with multiple threads working inside each box and leverages nested parallelism in OpenMP. As shown in Figure 4.6, on a system with six threads we can have six boxes being processed in parallel by a single thread, or x boxes being processed with y threads in them such that $x \times y = 6$, or have a single box with all 6 threads working on it. Larger boxes have a bigger working set than the smaller boxes down the V-cycle, which suggests that the system should assign more threads per box for the larger grids, and fewer threads for the smaller grids — ultimately the thread distribution is optimized via autotuning.

```
1  #pragma omp parallel private (...) num_threads(12)
2  {
3    tid=omp_get_thread_num();
4
5    for (k = -3; k <= 66; k++) {
6      for (t = 0; t <= min(3,intFloor(t+3,2)); t++) {
7        for (j = 6*tid-3; j <= min(6*tid+2,66); j++) {
8          for (i= t-3+intMod(-k-color-j-(t-3),2); i<=-t+66; i+=2)
9          {
10                   S0(t,k-t,j,i); /* Laplacian */
11                   S1(t,k-t,j,i); /* Helhmoltz */
12                   S2(t,k-t,j,i); /* GSRB      */
13         }
14       }
15     }
16     #pragma omp barrier (or explicit locks)
17   }
18 }
```

Listing 4.7: Threaded GSRB Wavefront for a 64^3 box with a four-deep ghost zone.

```
1  #pragma omp parallel private(....) num_threads(3)
2
3    for (k = -3; k <= 70; k++) {
4      for (t = max(0,k-66); t <= min(3,intFloor(t+3,3)); t++) {
5       if(t % 2 == 0){
6
7       #pragma omp for
8        for (j =t-3; j <= 66-t; j++) {
9          for (i= -3; i<=66; i++)
10         {
11                  even_S0(t,k-2*t,j,i); /* Laplacian */
12                  even_S1(t,k-2*t,j,i); /* Helhmoltz */
13                  even_S2(t,k-2*t,j,i); /* GSRB      */
14         }
15      }
16    }
17    if(t % 2 == 1){
18
19        #pragma omp for
20        for (j =t-3; j <= 66-t; j++) {
21         for (i= -3; i<=66; i++)
22         {
23                  odd_S0(t,k-2*t,j,i); /* Laplacian */
24                  odd_S1(t,k-2*t,j,i); /* Helhmoltz */
25                  odd_S2(t,k-2*t,j,i); /* GSRB      */
26         }
27      }
28    }
29
30  }
31 }
```

Listing 4.8: Threaded Jacobi Wavefront for a 64^3 box with a four-deep ghost zone.

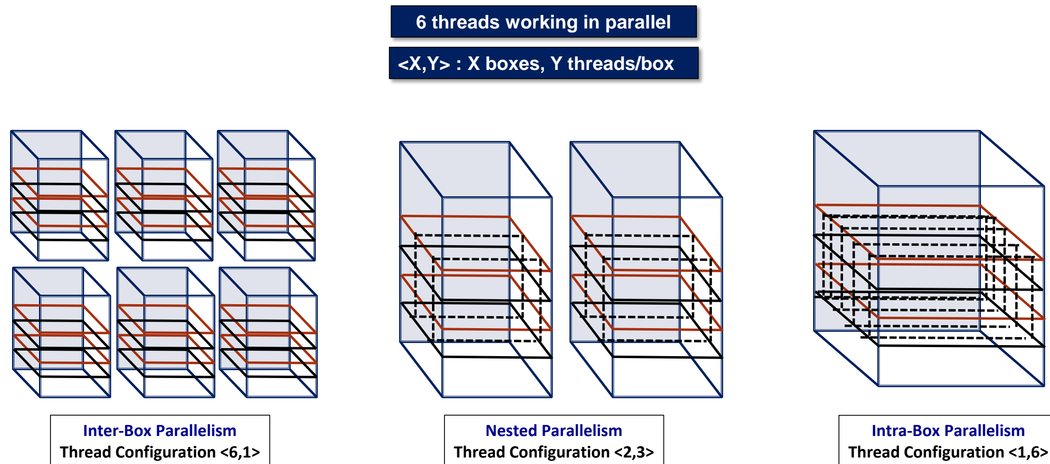


Figure 4.6: Example parallel decompositions on Hopper, which has 6-cores per socket. All the boxes in a subdomain may work in parallel, or all the threads may work on one box collaboratively, or nested parallelism may be used.

4.2.5 Residual-Restriction Fusion

Wavefront computation introduced in Section 4.2.3 fuses multiple sweeps of a box/grid into one and reduces vertical communication. We extend the same strategy of fusing multiple grid sweeps into one by fusing the residual and restriction computations into one sweep.

Unfortunately there exists a data dependence between residual (Listing 3.5) and restriction (Listing 4.9) which prevents this fusion. The dependence arises because every iteration of the triply nested `ijk`-loop residual updates a single point on the output (finer) grid. Restriction needs to read 8 points from the finer grid and restrict it to a single output point of the coarser grid. If the loop were naively fused, restriction would read points on the input finer grid before they were correctly updated.

To break this data dependence, a new compiler transformation was designed to fuse restriction with the other preceding operators. A novel compiler transformation was developed which converts the restriction stencil into an accumulation. This conversion *breaks* the data dependence and enables the sequence of loops to be fused.

The transformation converts restriction, an 8-point out-of-place stencil, to an accumulation. Listing 4.10 shows the output of this transformation. In line 12 of the output code each point from the input grid is read, multiplied by coefficient, and

scattered to the correct output point in the coarse grid. Instead of requiring to read 8 points, restriction now reads a single point from the fine grid, and thus the data dependence is broken. Since we are generating an accumulation, care must be taken to zero out the values in the output grid points before accumulating to them, and this is performed in line 6 of Listing 4.10.

The output of the residual-restriction fusion is shown in Listing 4.11. Line 6 zeroes out planes of the coarse grid before accumulating. The nested for-loop (lines 10-14) computes the residual (S4) and immediately scatters it to the correct point in the output coarse grid. Thus residual-restriction is now complete in one grid sweep performed by the common outermost `k`-loop.

4.2.6 Smooth-Residual-Restriction Wavefront

We extend the wavefront strategy further to create an even deeper wavefront from fused smooths, residual, and restriction. This fuses six grid sweeps (4 smooths, residual and restriction) into one sweep. To eliminate the ghost zone exchange required prior to residual computation we have to increase the ghost zone depth to five from the previously used four. The deeper ghost zone reduces horizontal communication further. The wavefront created then reduces vertical communication. The deeper wavefront means a larger working set. The working set is managed by generating nested parallel code.

The smooth-residual-restriction wavefront is illustrated in Figure 4.7. The cross-section of a box shows a wavefront that is six planes deep. The first four planes compute 2 GSRB sweeps, the fifth plane computes the residual, and the last plane computes the restriction and writes to the coarser output grid.

Listing 4.12 illustrates the code generated for this wavefront for the GSRB smooth applied to a 64^3 box with a five-deep ghost zone. There are three threads working collaboratively inside the box, and they synchronize using spin locks (lines 36-38). The code illustrates that the `k`-loop was skewed against the time `t`-loop, and then they are permuted, making `k` the outer loop and giving a single grid sweep (in the `k`-dimension). The smooths (lines 9-16) are followed by initializing a plane of the output coarser grid (lines 18-23), and then the computation of residual and restriction

```
1  /* Input: Restriction Operation */
2  for(k=0;k<K;k+=2)
3    for(j=0;j<J;j+=2)
4      for(i=0;i<I;i+=2)
5        coarser_res[k/2][j/2][i/2] = 0.125 *(
6          res[k  ][j  ][i] + res[k  ][j  ][i+1] +
7          res[k  ][j+1][i] + res[k  ][j+1][i+1] +
8          res[k+1][j  ][i] + res[k+1][j  ][i+1] +
9          res[k+1][j+1][i] + res[k+1][j+1][i+1]
10        );
```

Listing 4.9: Restriction operation.

```
1  /* Output: Restriction as a Scatter Operation */
2  for(k=0;k<K;k+=2)
3    for(j=0;j<J;j+=2)
4      for(i=0;i<I;i+=2)
5        /* statement S4 : Initialize coarse_res*/
6        coarser_res[k/2][j/2][i/2] = 0;
7
8  for(k=0;k<K;k++)
9    for(j=0;j<J;j++)
10     for(i=0;i<I;i++)
11       /* statement S5 : Restrict fine_res to coarse_res */
12       coarser_res[k/2][j/2][i/2] += 0.125* res[k][j][i];
```

Listing 4.10: Restriction as an accumulation.

```
1  /* Output: Restriction as a Scatter Operation */
2  for(k=0;k<K;k++){
3   if(k%2 == 0){
4     for(j=0;j<J;j+=2){
5      for(i=0;i<I;i+=2){
6      S4();/* statement S4 : Initialize coarse_res*/
7      }}
8    }
9
10  for(j=0;j<J;j++){
11   for(i=0;i<I;i++){
12       S3();/* Compute residual */
13       S5();/* statement S5 : Restrict fine_res to coarse_res */
14  }}
15
16 }/* End K */
```

Listing 4.11: Fused residual-restriction.

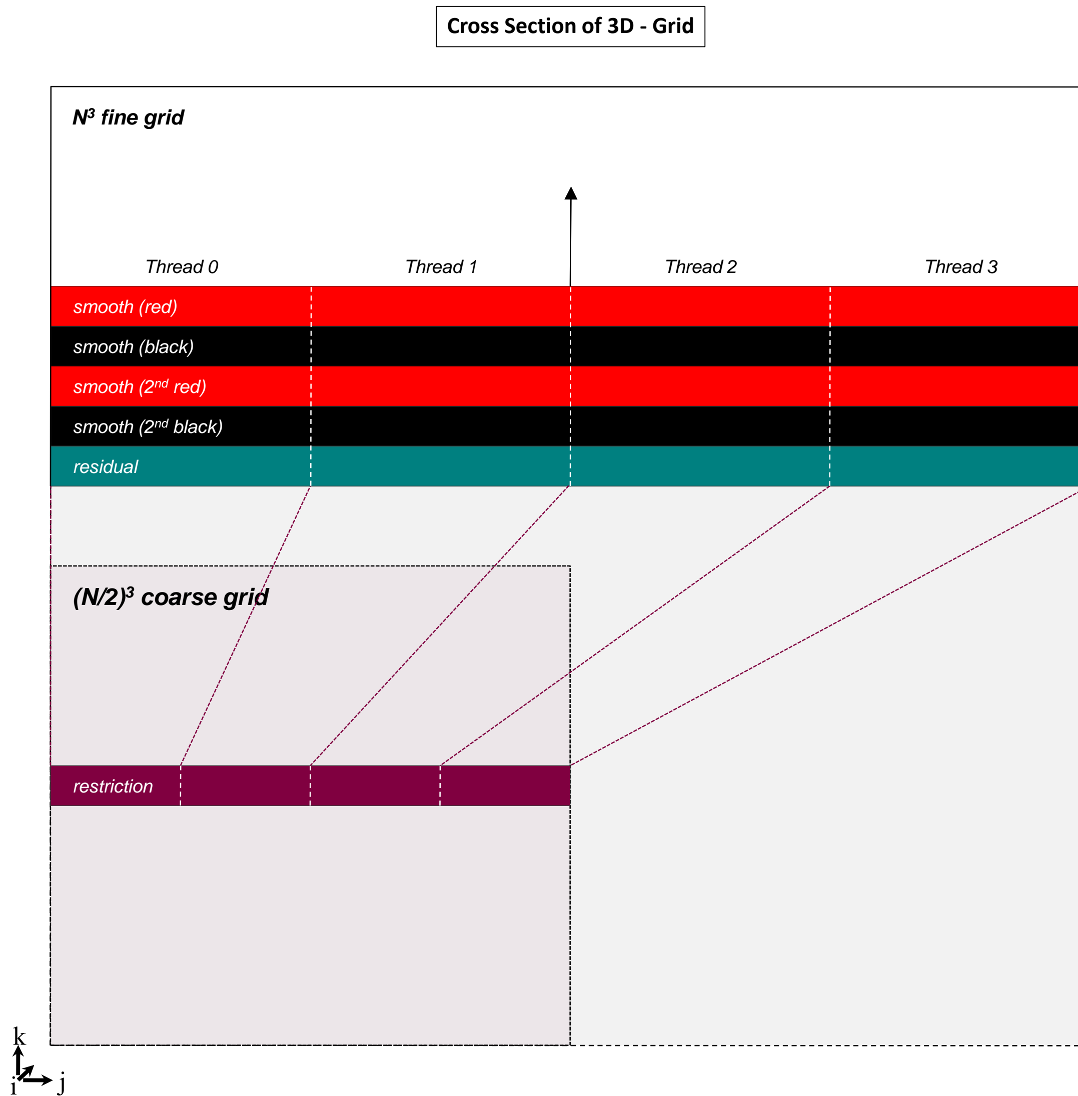


Figure 4.7: Wavefront applying smooths, residual and restriction.

```
1  #pragma omp parallel  private(...)\
2                       shared(locks) num_threads(3)
3  {
4  tid = omp_get_thread_num();
5  num_threads = omp_get_num_threads();
6  left = min(tid - 1,0);
7  right = max(tid + 1,num_threads - 1);
8
9  for(k = -4; k <= 67; k++){
10   for(t = 0; t <= min(3,intFloor(k+4,2)); t++){
11     for(j = 24*tid-4; j <= 24*tid + 19; j++){
12       for(i= -4+intMod(-k-color-j-(t-4),2); i<=67; i+=2) {
13         S0(t,k-t,j,i); /* Laplacian */
14         S1(t,k-t,j,i); /* Helhmoltz */
15         S2(t,k-t,j,i); /* GSRB      */
16       }}} /* End t,j,i */
17
18   if(4<=k && intMod(k,2) == 0){
19     for(j = max(24 * tid - 4,0);
20          j <= min(62,24 * tid + 18); j += 2){
21       for(i = 0; i <= 62; i += 2 ) {
22       S4(t,k-t,j,i); /* Initialize coarse_res  */
23       }}} /* End if */
24
25   if(4<=k){
26     for(j = max(24 * tid - 4,0);
27          j <= min(62,24 * tid + 18); j ++){
28       for(i = 0; i <= 63; i ++ ) {
29       S3(t,k-t,j,i); /* Compute residual  */
30       S5(t,k-t,j,i); /* Restrict fine_res to coarse_res */
31       }}} /* End if */
32
33   /* After computing the 5-deep wavefront */
34   /* Threads sync with their right and left neighbors */
35
36   locks[tid] = k;
37   if(left!= tid){ while (locks[left] < k) pause();}
38   if(right!= tid){ while (locks[right] < k) pause();}
39
40     }/* End k */
41
42 }/* End OMP region */
```

Listing 4.12: Simplified generated code for threaded wavefront with GSRB and residual and restriction fused. The code is specialized for a 64^3 box, with a five-deep ghost zone and 3-threads working inside a box.

to the coarser grid (lines 25-31). The threads synchronize after completing multiple smooths, residual, and the restriction. This deep wavefront is applied when going down the V-cycle. On the way back up, residual and restriction are replaced by the interpolation operation. Hence we use a four-deep wavefront going back up the V-cycle, but the ghost zone depth is still fixed at five, resulting in excess communication.

In a similar manner, wavefront computations can be generated for Jacobi style stencils, but Jacobi stencils present additional challenges to the memory system. Jacobi reads and writes to different arrays, leading to an even larger working set. Collaborative threading for Jacobi stencils is less effective, since due to dependences, threads must synchronize after computing each plane, unlike GSRB, where all the planes in the wavefront can be computed before the threads need to synchronize. Thus, deeper wavefront may not always help Jacobi smooths.

4.3 Autotuning Opportunities

The optimizations described in the previous sections create an interesting space of possible optimization sequences. Generating optimal code for each type of smooth at each level of the V-cycle requires us to select the most appropriate sequence of optimizations from this space. The following list outlines the optimizations which are candidates for autotuning.

- **Ghost zone depth**

  The ghost zone depth governs the amount of redundant computation performed, the frequency of MPI communication, and controls the depth of a wavefront computation. As memory bandwidth is a key limitation only for larger box sizes, the optimal value for ghost zone depth varies for different box sizes in the V-cycle.

  Ghost zone depth depends on the radius of the stencil used. Ghost zones are integer multiples of the stencil radius. The 3D 7-point stencil which has a radius of 1 can have ghost zone depths 1,2,3,4, ... A stencil of radius 2, such as the 3D 13-point stencil, can have ghost zones of depth 2,4,6, ... In general, the range of possible values of ghost zone depth is from $stencil_radius$ to $stencil_radius \times number_of_stencil_applications$. For the 7-point stencils,

where smooth can be applied up to four times, the ghost zone depth can vary from 1 to 4.

- **Wavefront**

  Wavefront computations dramatically reduce DRAM traffic but involve code with complex loop bounds. Creating a wavefront after using deep ghost zones is a binary decision. In the current code generation strategy, the ghost zone depth and number of planes in the wavefront (depth of wavefront) are identical.

- **Smooth-residual-restriction fusion and wavefront**

  Fusing smooth-residual-restriction requires an even larger ghost zone and creating a wavefront computation involving many planes. The decision to apply this optimization is binary and depends on the type of stencil, iteration, level of V-cycle, and architecture.

- **Thread decomposition**

  Threading is nested at two levels. As shown in Figure 4.6, miniGMG already uses a set of OpenMP threads for each box. The final generated code also introduces multiple threads per box. Since the box size varies across V-cycles in a GMG computation, the optimal number of threads per box also varies with the level of the V-cycle and type of stencil computation. The space of possible thread decompositions on a machine with `n` threads is <x, y> where $x \times y = \texttt{n}$.

4.4 Compiler Implementation

This section describes the implementation of the transformations described in Section 4.2. The transformations were implemented in the CHiLL compiler framework which was described in Chapter 2. The compiler abstractions introduced in that chapter — iterations spaces, dependence graphs, and data dependences, are used here to describe the implementation.

The discussions that follow explain the implementation details using the GSRB smooth as an example. The GSRB smooth has a more complex iteration space and presents more challenges. If handling Jacobi iterations demands additional attention, the implementation details regarding that are highlighted.

The order of optimizations presented in this section is different from Section 4.2. Instead of presenting the optimizations in the order they are applied, first the novel domain-specific compiler transformations are presented (Sections 4.4.1 to 4.4.3). This is followed by a subsection of how the data dependence graph of the input code needs to manipulated. Sections 4.4.5 and 4.4.6 describe optimizations which are implemented using known compiler transformations already built into CHiLL. The final subsection describes how we generated parallel OpenMP code.

4.4.1 Fusing Components of Smooth

The input code for GSRB and Jacobi iterations shown in Listings 3.2 and 3.4 consist of three statements, `S0`, `S1`, and `S2`, that correspond to the three smooth operators, Laplacian, Helmholtz, and GSRB, respectively. Once parsed by CHiLL, the iteration spaces corresponding to these operators are as follows:

$$IS_{S0} := \{[k, j, i] : 0 \leq k < N\&\&0 \leq j < N\&\&0 \leq i < N\};$$

$$IS_{S1} := \{[k, j, i] : 0 \leq k < N\&\&0 \leq j < N\&\&0 \leq i < N\};$$

$$IS_{S2} := \{[k, j, i] : 0 \leq k < N\&\&0 \leq j < N\&\&0 \leq i < N\&\&k+j+i+2\alpha+color = 0\};$$

Note that IS_{S2} has an additional term in its iteration space related to the if-condition in the GSRB code which checks the color of grid points. Iteration spaces for the statements do not have this additional term.

In the case of Jacobi, loop fusion falls out implicitly from the *iteration space alignment algorithm*, which attempts to carve out a unified iteration space for the imperfect loop nest of the original code [33]. The statements in Jacobi smooth have identical iteration spaces without fusion preventing dependences. Thus, CHiLL is able to automatically fuse the three loop nests.

In GSRB smooth, the iteration spaces for the three statements are not identical, and there exists a data dependence between the statements which prevents loop fusion. This data dependence is detected by CHiLL, and reported as a fusion-preventing dependence. The data dependence between S2 and S0 is related to the reads and writes of `phi` and does not allow fusion of the three loops. However, we make the observation that the iteration spaces for the Laplacian and Helmholtz operators

(statements `S0` and `S1`) compute values of `temp` that are never used by the `S2` of GSRB.

We use *array data-flow analysis* to analyze the iteration spaces and access expressions and derive a conservative approximation of the elements of `temp` defined in S0 and S1 and used in S2 [42]. Using techniques to compute data footprint associated with array references (described in Section 2.7), CHiLL determines that the array region read by S2 is a proper subset of the regions defined by S0 and S1. Since `temp` is a local variable redefined on every sweep, and it is not live after the smooth operator is completed, it is safe to contract the iteration spaces of S0 and S1 to match that of S2.

In the fused code, the compiler recognizes that array `temp` is a local variable, and does not need to be rewritten back to memory. Because there are no dependences on `temp` crossing iteration boundaries, *scalar replacement* is then employed to make this a scalar that is overwritten on each iteration of the loop.

4.4.2 Overlapping Ghost Zones

Once fused, the iteration spaces from the previous section end up with a combined iteration space that matches that of statement S2. We observe that introducing ghost zones as in the previous section is really just introducing a new loop t and changing the bounds for each of the loops in the fused loop nest to compute ghost regions and generate a hypertrapezoidal iteration space.

Due to the presence of the if-condition in the GSRB smooth, the iteration space is a hypertrapezoid with holes. The iteration space IS has two distinct parts, arising from the loop nest and also the relation (k+j+i+2α+color=0) which represents the if-condition; the iteration space is the conjunction of these terms. We added a new domain-specific transformation which maps the old iteration space with the new loop t using the mapping *map*. The mapping *map* will take IS, the iteration space of the input loop nest, and map it to the iteration space IS' of the modified loop nest. In addition to this, the variable `color` needs to get updated with every sweep of the grid. This mean the values of color will also be affected by the additional loop. For this purpose, we apply another mapping map' to update `color`.

$$map := \{[k,j,i] \rightarrow [t,k',j',i'] : 0 \leq t < d$$

$$\&\& \;\; k - d + 1 + t \leq k' < k + d - t$$

$$\&\& \;\; j - d + 1 + t \leq j' < j + d - t$$

$$\&\& \;\; i - d + 1 + t \leq i' < i + d - t\};$$

$$IS' := map(IS);$$

$$map' := \{[color] \rightarrow [color + t]\};$$

The application of map' will cause the value of color to be updated everywhere it appears, including within the statements. Although in our current implementation this relation is provided to the implementation, it could be derived automatically through analysis or domain knowledge. This gives a new relation for the if-condition as (k+j+i+2α+color+t=0). The conjunct of the new iteration space and the new term gives us the final modified iteration space.

4.4.3 Stencils as Accumulation

The accumulation transformation described in Section 4.2.5 targets constant-coefficient out-of-place stencils, such as the 8-point stencil of the restriction operator. As is standard with polyhedral compiler frameworks, we require that all subscript expressions are *affine*, or linear combinations of the loop indices and loop-invariant variables.

Examining the statement associated with a stencil computation, the compiler computes a bounding box of the points in the stencil statement, such that each dimension derives its lower and upper bound from the minimum and maximum values in that dimension.

Concretely, at every iteration $\vec{I} = \langle i_1, i_2, i_3 \rangle \in IS$, we compute a single output of the stencil computation. For a 2D 5-point stencil, for example, this requires loading $\{[i_1][i_2]$, $[i_1][i_2 \pm 1]$,$[i_1 \pm 1][i_2]$,$[i_1 \pm 1][i_2 \pm 1]\}$ from input array `in`, and writing the weighted sum of these points to output array `out`$[i_1][i_2]$ on every iteration. Using the above notation, we can rewrite an out-of-place constant-coefficient p-point stencil as a weighted sum of p points.

$$out[i_1][i_2][i_3] = \sum_{m=1}^{p} w_m * in[i_1 + o_1^m][i_2 + o_2^m][i_3 + o_3^m]$$

Each stencil point m is characterized by its constant-coefficient w_m, and the vector offset from the iteration $\vec{I}$, $\vec{O}_m = \langle o_1^m, o_2^m, o_3^m \rangle$. The bounding box is computed from the lower and upper bounds for each dimension.

$$lb_1 = min_m o_1^m,\ lb_2 = min_m o_2^m,\ lb_3 = min_m o_3^m$$

$$ub_1 = max_m o_1^m,\ ub_2 = max_m o_2^m,\ ub_3 = max_m o_3^m$$

$$BoundingBox = \{[b_1, b_2, b_3] : lb_1 \leq b_1 \leq ub_1, lb_2 \leq b_2 \leq ub_2, lb_3 \leq b_3 \leq ub_3\}$$

If the set of points of the stencil and its bounding box represent the identical volume, as in the case of the 8-point restriction operation, we call this a *full* stencil. In our implementation we will restrict this accumulation optimization to computations on full stencils. If we are willing to tolerate complex control flow to determine what points to execute, and therefore potentially inefficient code, more general stencils can be supported by the compiler using operations such as convex hull and set difference, which are supported in CHiLL/Omega+.

The key concept is that the compiler rewrites the statement, representing the stencil as an accumulation, and modifies the loop nest accordingly. From the perspective of a polyhedral compiler, it adds additional loops to the iteration space *IS* for the computation, and replaces the stencil statement with a different statement that reads and accumulates into the output variable. For correctness, the compiler must also create a new statement and iteration space to initialize the output variable. These steps are shown in Figure 4.8.

The example below illustrates this approach for the restriction operator, the iteration space *IS* is:

$$IS = \{[k, j, i] : \exists \alpha : (0 \leq k, j, i < N \ \&\& \ i, j, k = 2 * \alpha)\}$$

The original stencil statement is modified and two new statements are created, *S1* (initialization) and *S2* (accumulation) are as follows — the coefficient c_1 is the common

Initialize Output:

```
Assume stencil statement S0 has iteration space IS
Create initialization statement S1 with iteration space IS.
```

$\text{out}[l_1][l_2][l_3] = 0;$

```
Insert S1 lexicographically before S0.
```

Expand the iteration space for the accumulation:

```
Create a mapping M for IS to incorporate BoundingBox.
```

$$M := \{[l_1, l_2, l_3] \rightarrow [l'_1, b_1, l'_2, b_2, l'_3, b_3] : \\ l'_1 = l_1, l'_2 = l_2, l'_3 = l_3, \\ lb_1, lb_2, lb_3 \leq b_1, b_2, b_3 \leq ub_1, ub_2, ub_3\}$$

```
Apply M to IS to create new iteration space IS'
             IS' = M(IS)

/* In special case IS' can be simplified to       */
/* coalesce the loops and is described in the text */
```

Replace S0 with the new statement S2:

```
Create new statement S2 with iteration space IS'.
```

$\text{out}[l_1][l_2][l_3] \mathrel{+}= \text{in}[l_1{+}b_1][l_2{+}b_2][l_3{+}b_3];$

Figure 4.8: Code generation steps for accumulation transformation.

constant-coefficient of the stencil. If the stencil has unique coefficients, we will use $\text{coeff}[b_1][b_2][b_3]$ instead:

$$\textit{S1:}\;\; coarser_res[k/2][j/2][i/2] = \; 0;$$

$$\textit{S2:}\;\; coarser_res[k/2][j/2][i/2] \mathrel{+}= \; c_1 * res[k+b_1][j+b_2][i+b_3];$$

The iteration space for the initialization statement *S1* is the original iteration space for the restriction loop, which we call *IS*. The iteration for *S2* is modified. In Figure 4.8, the relation M maps the original iteration space *IS* to the new iteration space IS'. For the restriction computation, where there is a stride on the original

loops and the bounding box is contained within the stride, it is beneficial to apply an additional coalescing transformation to IS to eliminate the loops associated with the bounding box and convert the k, j, and i loops to unit stride accesses.

$$IS' = M(IS);$$

$$M' := \{[k, b_1, j, b_2, i, b_3] \rightarrow [k', j', i'] :$$

$$k - b_1 \leq k' \leq k + b_1$$

$$\&\& \;\; j - b_2 \leq j' \leq j + b_2$$

$$\&\& \;\; i - b_3 \leq i' \leq i + b_3\};$$

$$IS_{final} = M'(IS');$$

When the code generator scans the loop nests associated with IS and IS_{final}, the compiler then derives the code shown as the output of Listing 4.10. Although not strictly a polyhedral transformation, the new statements and loop iteration spaces are essentially converting subscript expressions to loop iteration spaces, and rely heavily on the polyhedral framework abstractions. A similar approach is used in CHiLL to introduce statements and modify the iteration space when performing loop unrolling [37].

4.4.4 Rebuilding the Dependence Graph

CHiLL maintains a dependence graph based on data dependences of the input code. The dependence graph is commonly used to check the legality of transformations being applied. The domain-specific transformations that were added to CHiLL and described in Sections 4.4.1 to 4.4.3 do not use the dependence graphs for legality checking. In fact, once these new transformations are applied, the dependence graph is rendered obsolete, as the data dependences have been modified. The data dependence graph is rebuilt after applying each of the new transformations to ensure that we do not stop the legal composition of transformations after the novel stencil-specific optimizations have been applied. The dependence graph for the modified program is rebuilt by running the dependence analysis over the intermediate representation of the transformed code.

4.4.5 Wavefronts

Wavefront computations are created by using known loop transformations *skewing* and *permute*. The motivation for this particular sequence of transformations is explained using the GSRB smooth with a 3D 7-point stencil and illustrated in Figure 4.9.

Figure 4.9(a) shows the iteration space for four stencil applications (t = 0,1,..,3) of the stencil on the grid. For clarity of presentation, we have represented this 4D iteration space as a 2D space by projecting out the `i` and `j` dimensions. This means a point in the `k`-axis represents an `ij`-plane. Thus, a point (z,T) in this space corresponds to the `ij`-plane with k=z having the t=T stencil sweep applied to it. As seen previously, computing an output plane for the 3D 7-point stencil requires reading in three planes. Thus computing plane (z,T) depends on reading planes (z-1,T-1), (z,T-1), and (z+1,T-1); that is, the top, center and bottom planes computed from the previous grid sweep/time step.

The four stencil applications which are grid sweeps are visualized in Figure 4.9(a) by the arrows showing the direction of loop traversal. The iteration space is scanned horizontally, which means the `ij`-planes are streamed from DRAM, creating high data traffic. Ideally we want to hold each `ij`-plane in memory and apply all four stencil applications on it before moving on to the next plane to reduce data movement. This means scanning the iteration space vertically, as shown in in Figure 4.9(b). Scanning the iteration space vertically, means permuting loops `t` and `k`; that is, in Figure 4.9(a) `t` was the outermost loop, but in Figure 4.9(b) `k` will be the outer loop, with `t` nested inside it.

Unfortunately there is a data dependence which prevents this desired loop permutation. The data dependence is illustrated in Figure 4.9(c). The point (2,2) depends on points (1,1), (2,1), and (3,1). Loop permutation would mean executing point (2,2) just after executing (2,1) but before (3,1); this is illegal and prevented by our compiler.

To break this data dependence, we skew the iteration space as shown in Figure 4.9(d). As can be seen, after skewing, loop permutation of vertical scanning of the iteration space is legal. The skewing and permutation for GSRB is performed

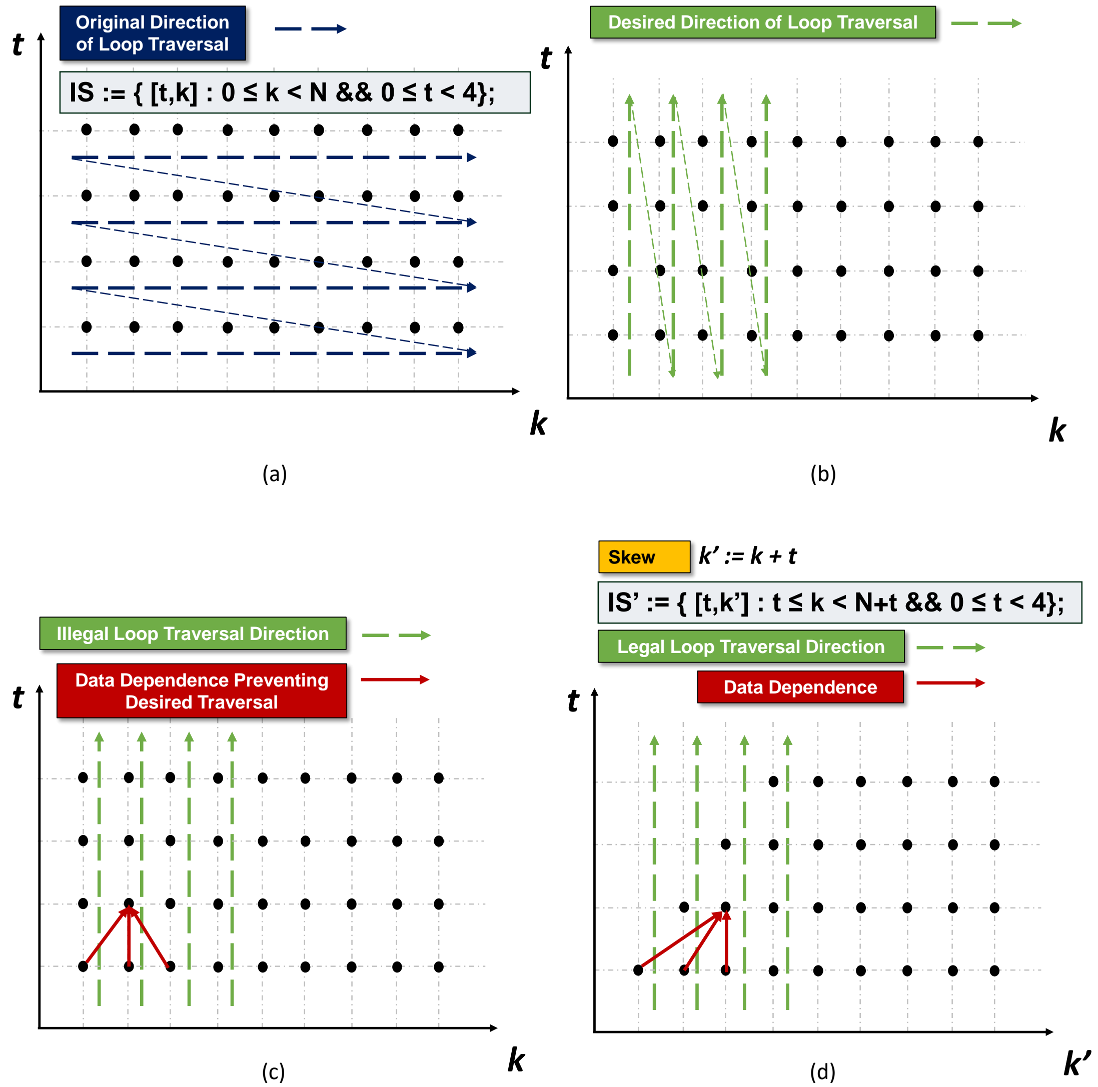


Figure 4.9: Loop skewing and loop permutation to create a wavefront.

in CHiLL by the mappings $skew_{gsrb}$ and *permute*. The skewing factor for GSRB as shown in $skew_{gsrb}$ is one. The skewing factor is governed by data dependences and depends on the iteration type (GSRB vs. Jacobi) and stencil radius. For Jacobi iteration using the same 3D 7-point stencil we need to skew `k` against `t` by a factor of two. The skewing relation for Jacobi is shown as $skew_{jacobi}$. The effect of the larger skewing factor can be seen in Figure 4.4; there is a separation of two between planes getting updated, whereas consecutive planes get updated in the GSRB wavefront (Figure 4.3).

$$skew_{gsrb} := \{[t, k, j, i] \rightarrow [t, k + t, j, i]\}$$

$$skew_{jacobi} := \{[t, k, j, i] \rightarrow [t, k + 2t, j, i]\}$$

$$permute := \{[t, k, j, i] \rightarrow [k, t, j, i]\}$$

4.4.6 Smooth Residual Restriction Fusion

A deep wavefront with smooth, residual, and restriction is created using the same technique of skew and permute. The input code to the compiler from which the deep wavefront is generated is a sequence of loops nests. The first triply nested loop nest is for smooth, followed by residual and restriction. The time step `t`-loop which controls the application of these stencils is added to surround the three loop nests. In the input code residual and restriction follow smooths as expected.

The first step is to convert the restriction to an accumulation. This is followed by the use of loop fusion to fuse residual and restriction together. The next step is to skew the `k`-loops of all the statements against the outermost `t`-loop, as done in Section 4.4.5, and then permute loops `k` and `t` to create the wavefront.

4.4.7 Parallel Code Generation

Parallel OpenMP code generation follows the compiler transformations. The parallel code generation strategies for Jacobi and GSRB smooth are considerably different. For GSRB, an OpenMP parallel region is used whereas for Jacobi iterations we use an OpenMP parallel-for construct.

For GSRB (Listing 4.7) we tile the `j`-loop and assign each each tile a thread. This involves injecting nodes into the AST to setup OpenMP threads correctly and to replace index variables in the code with thread identifiers.

The entire loop nest for GSRB is marked as an OpenMP parallel region and the loops and loop body are parsed to generate the variables in the `private` OpenMP clause (line 1). The number of threads in the `num_threads` clause is provided to CHiLL via a script, and a node for this clause is added to the AST.

The number of threads that is provided as input to CHiLL is used to tile the `j`-loop. The tiled loop is then removed from the AST and the loop index in the code is replaced by the thread identifier `tid`. Finally at the end of the `k`-loop an OpenMP barrier or spinlocks are added by injecting nodes into the AST.

The Jacobi iteration uses simpler OpenMP code. The entire loop nest is marked with `#pragma omp parallel`. A `private` clause and `num_threads` clause are added to this pragma in the same way as was done for the GSRB code. Tiling and barriers are not used for Jacobi; the `j`-loop is simply marked as parallel with `#pragma omp for`. All the OpenMP directives are nodes in the compiler AST. After all loop transformations are complete, they are added by injecting appropriate nodes into the AST of the generated code.

4.5 Putting It Together

Our approach to generating code variants and selecting the best version relies on using CHiLL through its script-based interface. We generate CHiLL scripts (also called transformation recipes) which are run to generate variants of smooth, residual, and restriction. In similar fashion to the autotuning work by Williams et al. [32], we stitched together variants of the functions generated by CHiLL using tables of pointers. Thus a call to the smooth routine in miniGMG would look up a table of function pointers and make the call to the appropriate variant of smooth.

As the goal of this work is to develop domain-specific optimization techniques for GMG, our autotuner was specialized for the miniGMG implementation. A driver program instantiates template transformation recipes that are then applied to the input code to generate the optimized code.

Listing 4.13 shows an example recipe or CHiLL script used to generate a variant of the GSRB smooth. Lines 3-6 specify the problem size, in this case we are generating code for a 64^3 box. Line 12 adds ghost zones which are four deep. Lines 16-17 create a wavefront and lines 21-22 generates code with 12 threads working inside the box. The commands in this recipe refers to applying a transformation to a statement at a particular loop level. In the script, S0 corresponds to all the statements of smooth: Laplacian, Helmholtz, and GSRB updates. Once they are fused, the same transformations are applied to the set of statements. The only differences for the recipe for the smooth including Jacobi is that there are two statements to which the transformations are applied, corresponding to odd/even iterations, and the dependence distance for skewing is different.

The CHiLL script in Listing 4.13 is an instantiated template of a script to generate a wavefront for the GSRB smooth. The template has been instantiated with a ghost zone depth of 4, a wavefront computation being set to true and the number of threads working in a box being set at 12. Through an external python script, autotuning varies the values of parameters to create multiple scripts which are used to generate GSRB smooth variants.

Listing 4.14 illustrates a complex script that was used to create a deep wavefront which fused smooths, residual, and restriction. S0 are the statements of smooth, S1 is residual. In this script, first restriction is converted to an accumulation (line 13), then residual, restriction, and smooth are fused (lines 17-24), and finally a wavefront is created (line 28-19) and parallel code generated (32-33). This script is an instantiated template to generate code for fused smooth, residual, and restriction with ghost depth set to 5, wavefront set to true, and 12 threads per box.

4.6 Results

This section presents an overview of the experimental platforms, the problem solved by miniGMG, and the configuration of miniGMG. This is followed by a detailed analysis of our performance results.

```
1  #code generation is
2  #specialized to problem size
3  known (K == 64)
4  known (J == 64)
5  known (I == 64)
6
7  #initializes CHiLL's
8  #abstraction of loops
9  original()
10
11 #ghost zone depth is d == 4
12 add_ghosts([S0], L1, d)
13
14 #skew followed by permute
15 #to create a wavefront
16 skew ([S0], 2, [1,1])
17 permute ([S2,S1,S3,S4])
18
19 #parallel code generation
20 #tile j-loop, then the tiled loop
21 #is assigned to a parallel region
22 tile(S0,3,6,2, counted)
23 gen_omp_parallel_region(0,0)
```

Listing 4.13: CHiLL script to create a four-deep wavefront for GSRB smooth with OpenMP threading

```
1  #code generation is
2  #specialized to problem size
3  known (K == 64)
4  known (J == 64)
5  known (I == 64)
6
7  #initializes CHiLL's
8  #abstraction of loops
9  original()
10
11 #ghost zone depth is d == 5
12 add_ghosts([S0,S1,S2], L1, d)
13
14 #convert restrict to
15 accumulation
16 stencil_to_restrction(S2)
17
18 #converting restriction to accumulation
19 #creates statements which are then fused together
20 fuse([S1,S2,S3],1)
21 fuse([S1,S2,S3],2)
22 fuse([S1,S2,S3],3)
23 fuse([S1,S2,S3],4)
24 fuse([S1,S2,S3],5)
25
26 #moves statement initializing coarse grid to zero
27 #to right position
28 distribute([S2,S3], 4)
29
30 #deep wavefront computation
31 skew ([S0,S1,S2,S3], 3, [0,1,1])
32 permute([S1,S3,S2,S4,S5])
33
34 #Generate OpenMP code
35 tile(S0, 4, 18, 3, counted)
36 gen_omp_parallel_region(0,0)
```

Listing 4.14: CHiLL script to create a wavefront for GSRB smooth, residual, and restriction with OpenMP threading

4.6.1 Problem Specification

We use a double-precision, finite-volume discretization of the variable-coefficient operator $L = a\vec{\alpha}I - b\nabla\vec{\beta}\nabla$ (Listing 3.2), with periodic boundary conditions as the linear operator within our test problem. The right-hand side f (in $Lu^h = f^h$) is $sin(2\pi x)sin(2\pi y)sin(2\pi z)$ on the [0,1] cubical domain. The u, f, and $\vec{\alpha}$ are cell-centered data, while the $\vec{\beta}$'s are face-centered.

4.6.2 miniGMG Configuration

In all our experiments the size of the global 3D domain at the finest level of the V-cycle is 256^3 per node. The global 3D domain is decomposed into a list of 64, 64^3 subdomains or boxes. At the bottom or coarsest level of the V-cycle we use 4^3 boxes, and global domain 64^3. The truncated V-cycle has 5 levels with box sizes 64^3, 32^3, 16^3, 8^3 and 4^3.

While going down the V-cycle (finer→coarser) we apply 4 GSRB (red-black-red-black) smooths. At the bottom level of the V-cycle (on 4^3 boxes) we apply 48 smooths, and while going back up the V-cycle (coarser→finer) 4 more smooths are applied.

Thus 48 smooths are applied to boxes at the bottom level, and at all other levels the boxes go through two phases of four consecutive smooths, hence 8 overall. For the Jacobi smooth the same number of smooth applications is used; 4 going down and up the V-cycle, and 48 bottom smooths.

4.6.3 Manually Optimized and Baseline miniGMG

The generated code was compared to the implementation of miniGMG presented in Chapter 3 and to a manually optimized version from Williams et al. [32]. In this chapter we refer to the implementation of miniGMG described in Chapter 3 as the baseline version.

The manually optimized version has fused the components of smooth, used deep ghost zones, and wavefronts. The code also uses multiple OpenMP threads working on a box or subdomain and fuses the restriction and residual computations. In addition, it has other optimizations, such as software prefetching and explicit SIMDization. The differences between the compiler-generated code and the manually optimized code are presented alongside analysis of the performance results.

4.6.4 Evaluated Platforms

We evaluate the benefits of our compiler technology on two commodity processor architectures similar to those used by Williams et al. in [32]. The machines used are the National Energy Research Scientific Computing Center (NERSC) supercomputers, Hopper and Edison. In both cases, we use the default Intel compiler available on the NERSC machines with `-O3 -fno-alias -fno-fnalias` and either `-xAVX` or `-msse3`.

Edison is a Cray XC30 at NERSC. Each node contains two 12-core Xeon Ivy Bridge chips, each with four DDR3-1600 memory controllers and a 30MB L3 cache [47]. Each core implements the 4-way AVX SIMD instruction set and includes both a 32KB L1 and a 256B L2 cache.

Hopper is a Cray XE6 at NERSC. Each node contains four 6-core Opteron chips, each with two DDR3-1333 memory controllers and a 6MB L3 cache [48]. Each core implements the 2-way SSE3 SIMD instruction set and includes both a 64KB L1 and a 512KB L2 cache.

The details of the machines are summarized in Table 4.2. On an Edison node we have used MPI processes to match the number of sockets. The 256^3 domain is divided into the two processes, with each getting a problem size $256 \times 256 \times 128$ ($x \times y \times z$). A four-socket Hopper node has four MPI process, each working on a problem size $256 \times 128 \times 128$.

4.6.5 Analysis of GSRB Smooth Performance

Figure 4.10 compares the performance of the generated code for the 7-point variable-coefficient GSRB smooth against the baseline smooth and the manually optimized code on Hopper. The x-axis shows the box sizes corresponding to the levels of the V-cycle, and the y-axis plots the speedups of the generated and manually tuned code against the baseline smooth. Table 4.3 tabulates the best-performing code variants selected by autotuning. Similarly, Figure 4.11 and Table 4.4 compare the performance of GSRB smooth on Edison, and tabulates the best-performing code variants.

In addition to plotting the speedup achieved by generated and manually optimized codes, Figures 4.10 and 4.11 also plot (the blue dots) the Roofline memory (DRAM)

Table 4.2: Overview of evaluated platforms.

Core Architecture	Intel Ivy Bridge	AMD Opteron
Clock (GHz)	2.40	2.1
DP GFlop/s	19.2	8.4
Data Cache (KB)	32+256	64+512
Chip Architecture	**Intel Xeon E5-2695v2**	**AMD Opteron 6172**
Cores	12	6
Last-level Cache	30 MB	5 MB
DP GFlop/s	230.4	50.4
Memory Capacity	32 GB	8 GB
System	**Cray XC30 (Edison)**	**Cray XE6 (Hopper)**
CPUs/Node	2	4
STREAM Bandwidth	88 GB/s	48 GB/s
Compiler	icc 14.0.0	icc 13.1.3

bounds [49] for GSRB smooth. The Roofline Model uses bound and bottleneck analysis to represent architecture performance as a function of requisite data movement and computation. The Roofline bounds show the maximum speedup that can be achieved over the baseline GSRB smooth when the three components of the smooth have been fused, and the performance bottleneck of the computation is the DRAM bandwidth. The speedup achieved by the generated code over the Roofline bound illustrates the effectiveness of the communication-avoiding wavefront optimization.

4.6.5.1 Computing Roofline Memory Bounds

The Roofline bounds are computed by dividing the total volume of data moved from the DRAM to the caches for a single grid sweep by the DRAM bandwidth. The data volume is computed for all the arrays: phi, the betas, lambda, and rhs. The bounds are computed assuming wavefront optimization is not applied, which means a single smooth is applied per grid sweep. Thus the data moved from DRAM to the caches can be computed as:

Data Moved = iterations * boxes * grid size * arrays (R+W) * size of double

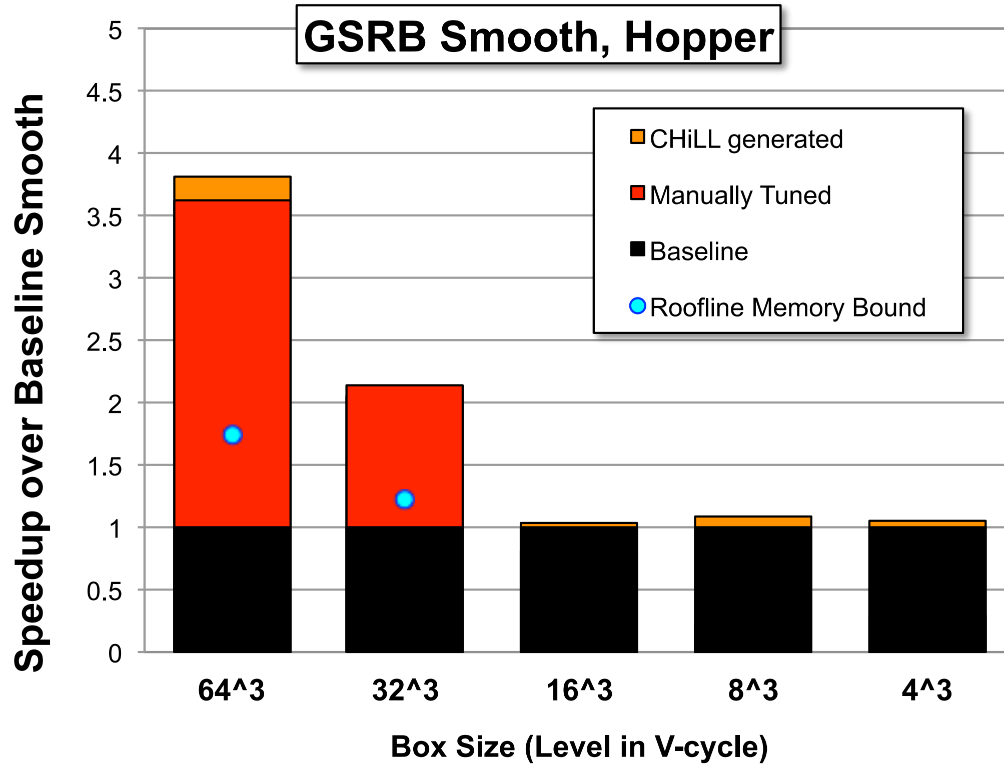


Figure 4.10: Speedups of CHiLL-generated and manually tuned GSRB smooth relative to the baseline code on Hopper. The speedups are shown for all levels of the V-cycle. Generated code outperforms expert-written code on all sizes except the 32^3 box, and always outperforms baseline code.

Table 4.3: Configurations of best-performing code variants for GSRB smooth on Hopper.

Box Size	Ghost Zone Depth	Thread Decomposition	Code Variant
64^3	4	<2,3>	wavefront
32^3	4	<2,3>	wavefront
16^3	2	<6,1>	wavefront
8^3	2	<6,1>	fused
4^3	4	<6,1>	fused

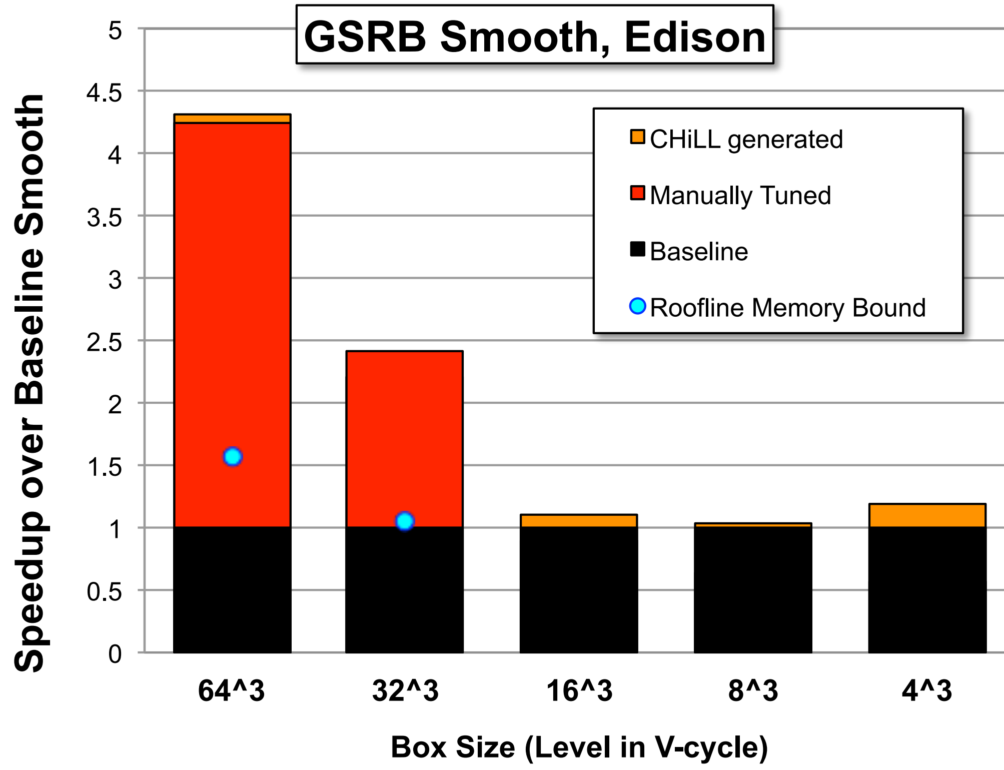


Figure 4.11: Speedups of CHiLL-generated and manually tuned GSRB smooth relative to the baseline code on Edison. The speedups are shown for all levels of the V-cycle. Generated code outperforms expert written code on all sizes except the 32^3 box, and always outperforms baseline code.

Table 4.4: Configurations of best-performing code variants for GSRB smooth on Edison.

Box Size	Ghost Zone Depth	Thread Decomposition	Code Variant
64^3	4	<4,3>	wavefront
32^3	4	<4,3>	wavefront
16^3	2	<12,1>	wavefront
8^3	2	<12,1>	fused
4^3	2	<12,1>	fused

The number of iterations is 80: 4 smooths going down the V-cycle, 4 smooths going back up the V-cycle, and there are 10 V-cycles. There are 64 boxes or subdomains, grid size including the ghost zone range from 66^3 for the finest boxes to 6^3 for the coarse boxes. There are 7 arrays read and 1 written, thus 8 arrays, and 8 bytes for each double precision number. This formula assumes ideal DRAM traffic, accounting for only compulsory cache misses. Based on the above formula, the ideal execution time of smooth is data moved divided by DRAM bandwidth (e.g. 88GB/s for Edison). The Roofline bounds plotted in Figures 4.10 and 4.11 are the speedup of the ideal execution time over the time taken by baseline smooth. The bounds for boxes smaller than 32^3 are not shown, as the total memory required for boxes at these levels fits into the last-level caches.

4.6.5.2 Optimizations and Smooth Performance

To understand the salient characteristics of performance, we examine each optimization at a time. The first optimization is fusing the components of smooth. Fusing the operators in smooth yields speedups on all grid sizes across all architectures. Though loop fusion does increase the working set, the benefit of reducing vertical data movement is far greater. In fact, even at the lower levels of the V-cycle where all the boxes fit into the last-level caches, fusion improves performance.

After fusion the next steps in optimization are adding deeper ghost zones and using the deep ghost zones to create a wavefront computation. Deep ghost zones reduce the number of horizontal messages sent at the cost of redundant computation and increased size of each grid. Wavefront computations reduce vertical communication but increases the working set and requires code with complex loop bounds. For optimal performance at each level of the V-cycle we first need to find the best ghost zone depth, and then we need to decide if a wavefront computation is beneficial.

The finer 64^3 and 32^3 grids do not fit into the last-level caches, and we expect wavefront computations with deep ghost zones to give a performance win. As the grids get smaller and coarser and fit into the last-level cache, the redundant computation and increased grid sizes become expensive. In fact, the grids at the coarsest subdomain for a four-deep ghost zone increases from approximately 12KB to nearly

100KB. Thus we expect the benefit of a wavefront and deep ghost zones to diminish as we go down the V-cycle.

This trend is clearly visible in Tables 4.3 and 4.4. Hopper and Edison both support a four-deep wavefront for 64^3 and 32^3 boxes and a two-deep wavefront for 16^3 boxes. On Hopper a four-deep wavefront for the finest grid achieves a speedup of 3.5x over the baseline implementation and outperforms the heavily optimized manually tuned code. On Edison and its large caches, increasing ghost zone depth allows for more on-chip locality and, thus, even higher speedups. At the finest level, Edison supports a four-deep wavefront and shows speedups close to 4.5x. The smaller 8^3 and 4^3 subdomains use a deep ghost zone, but cannot support a wavefront. On these coarse boxes the cost of the redundant computation was offset by the reduced messages sent to exchange ghost zones.

Next we look at parallel code generation and inter- and intra-box parallelism. On tradition multicore-based architectures miniGMG uses MPI+OpenMP to express parallelism. Each socket in Hopper and Edison runs a MPI process with 6 and 12 threads per process, respectively. The number of threads per socket/MPI process is set to the number of cores. We represent threading as $\langle \textit{boxes in parallel}, \textit{threads per box} \rangle$. Where *boxes in parallel* refers to the number of boxes being processed in parallel at a time in each MPI process, and *threads per box* refers to the number of threads working on a box. The total number of threads in a process is set to the number of cores thus *boxes in parallel* * *threads per box* = #cores. Baseline miniGMG has interbox parallelism, where each box is processed by an OpenMP thread. Thus the baseline configurations for Hopper and Edison are $\langle 6, 1 \rangle$ and $\langle 12, 1 \rangle$, respectively.

Wavefront computations increase locality and reduce vertical communication to DRAM, but increase the working set. To manage the larger working set we use thread blocking, where multiple threads work on a box creating intrabox parallelism. Since OpenMP threads have an overhead associated with them, the finer grids with larger working sets benefit from multiple threads per box. As the grids become coarser and the working set decreases, mulitple threads inside a box become expensive to sustain, and degrade performance. The thread decomposition for each level of the V-cycle shown in Tables 4.3 and 4.4 clearly illustrates this trend for the GSRB smooth on

both Hopper and Edison. The 64^3 and 32^3 boxes use multiple threads, but boxes coarser than 32^3 are processed by a single thread. On Hopper 64^3 and 32^3 boxes have three threads process each box and two such boxes are processed in parallel, hence the configuration $\langle 2, 3\rangle$. On Edison for these box sizes the configuration is $\langle 4, 3\rangle$. Boxes coarser than 32^3 cannot support multiple threads and they are processed in parallel by a single thread on both the architectures. After tuned application of the above optimizations, generated code for GSRB smooth outperforms manually tuned code for all V-cycle levels except 32^3 box sizes. The manually tuned smooth was highly optimized for the finer grids, since the time spent on smooth on the finest 64^3 grid dominates runtime. The compiler-generated, problem-size-specific code that was tuned for each level of the V-cycle. As can be in seen in Figures 4.10 and 4.11, specialization for problem size meant the generated code clearly outperformed manually tuned code on the smaller grids.

The manually optimized smooth performs communication-avoiding optimization with a four-deep wavefront. It uses intrabox parallelism where all the threads (Hopper-6, Edison-12) in a MPI process work on a single box. In addition, it performs explicit software prefetching from DRAM into caches as an additional memory-bandwidth optimization. The manually optimized code also unrolled loops and generated explicity SIMD instructions based on the architecture; SSE on Hopper, AVX on Edison.

The generated code outperforms manually tuned on 64^3 boxes by specializing for problem size and searching different threading configurations. The tuning process selected nested parallelism, with three threads inside each box and multiple boxes being processed in parallel, instead of having all threads work collaboratively on a single box. The nested parallelism provides better load balancing and significantly improves performance. The lack of loop unrolling and explicit SIMD code generation did not hurt the performance of generated code, since smooth is heavily DRAM memory-bandwidth limited. Smooth is limited by memory bandwidth even after a wavefront, and Unrolling and SIMD code generation, which improve floating-point performance, do not provide a benefit in this memory limited scenario.

Figure 4.12 quantifies the overall speedup on the MG solver attained via the CHiLL compiler as a function of optimizations employed for the 7-point variable-coefficient

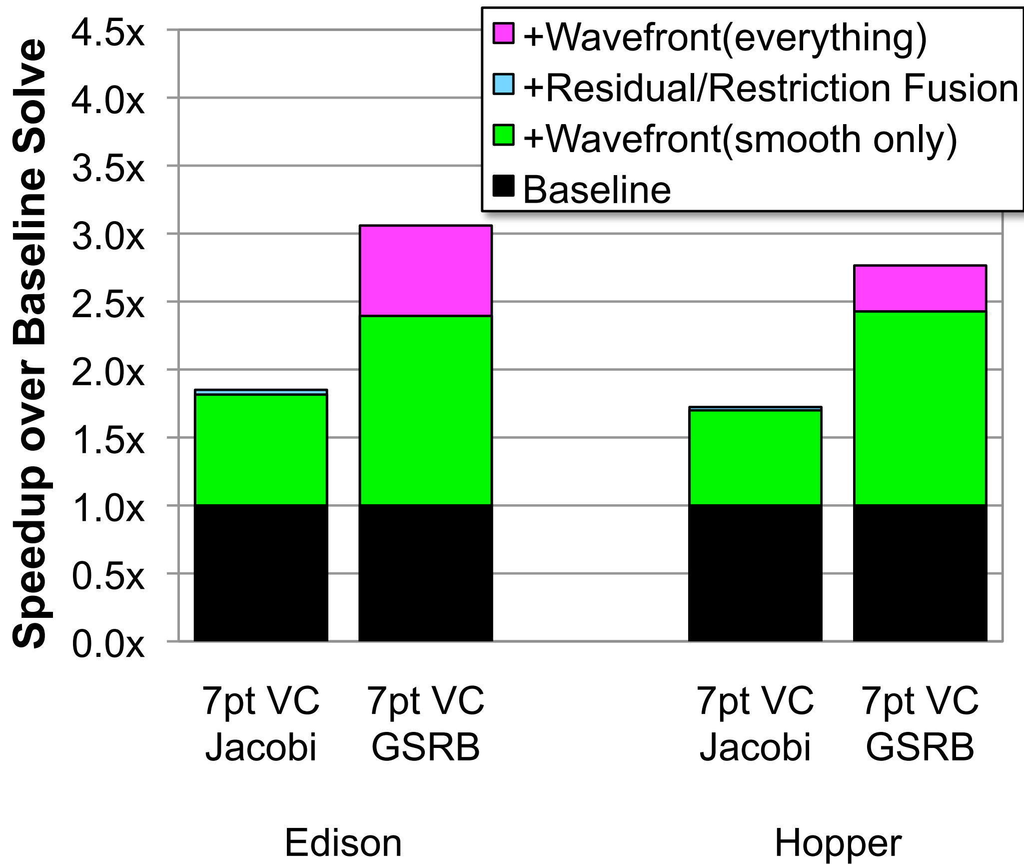


Figure 4.12: Speedup for the MG Solver attained from incrementally enabled optimizations in the CHiLL compiler. The performance results are categorized by type of smooth and architecture. "VC" is variable-coefficient.

operator using either the Jacobi or the GSRB smooth. We observe that just application of the best communication-avoiding wavefront smooth (requiring 4 ghost zones) can improve performance of the entire solver by up to 2.4×.

4.6.6 Smooth, Residual and Restriction Fusion

After optimizing smooth, stencil reordering is used to fuse the residual and restriction operations. This fusion significantly accelerates these two operations, but as shown in Figure 4.12, the overall benefit for the solver is relatively small. This is expected, as the total data movement eliminated by the residual-restriction fusion is a small fraction of the overall MG solver, which uses 4 smooths going down the V-cycle (presmooths) and 4 when going back up (postsmooths).

To further reduce both horizontal and vertical communication we fuse all operations of presmoothing into a single wavefront ("+Wavefront(everything)" in Figure 4.12) that performs all smooths, residual, and restriction with a single communication phase requiring the exchange of a five-deep ghost zone. The current version of miniGMG supports a single ghost depth at a given level of V-cycle. This means we use a five-deep ghost zone for the 64^3 boxes to support a five-deep smooth-residual-restriction wavefront when going down the V-cycle. On the way back up, we can only have a four-deep wavefront, and this results in extra communication cost for the wider-than-required ghost zone.

The performance benefit of the deep wavefront is highly dependent on architecture and smooth. A communication-avoiding Jacobi smooth requires a significantly larger per-cache working set than a communication-avoiding GSRB smooth using multiple threads per box. As GSRB is more likely to maintain a working set in the L2, it should be no surprise that we see a more significant benefit from a communication-avoiding GSRB smooth — up to 2.5× on Edison. The working set continues to balloon as one fuses the residual and restriction calculations. At this point, it is unlikely that the working set will be maintained in the L2, thus resulting in reduced bandwidth to the L3 and a differentiation of Hopper and Edison, with the latter attaining a significant net speedup over 3× for the variable-coefficient GSRB.

4.7 Summary and Conclusion

In this chapter we have described compiler technology and autotuning to automate communication-avoiding optimizations for the smooth, residual, and restriction operators in a geometric multigrid computation. We have tested our optimizations on computations with variable-coefficient stencil which stress the memory systems heavily.

Our results show that the optimizations lead to speedups as high as $4\times$ for smooths and over $3\times$ for the entire solver, and that different optimization strategies are needed at different levels of the V-cycle. Gains also vary for the two smooth operators GSRB and Jacobi, and for the two different architectures, thus pointing to the value of autotuning in finding the best set of optimizations for a given execution context. In addition a novel compiler transformation was developed to convert a stencil computation into an accumulation which enables loop fusion and further reduces communication and improves performance.

For the GSRB smooth we are able to match or better the performance of manually tuned codes written by experts. The compiler-generated code avoids low-level optimizations performed by manual tuners, but it is still able to match performance, as it can search a richer space of optimizations, such as different parallel thread decompositions. In addition, the optimizations added to the compiler framework take the wavefront strategy further by fusing all operators, smooth, residual, and restriction into a single wavefront. The manually tuned code has a wavefront for smooth, and fuses residual and restriction but does not fuse everything together into a single wavefront. Finally, we show that such a deep wavefront works for a GSRB smooth but not Jacobi.

The approach to optimizing the GMG illustrated in this chapter also applies to other GMG implementations that use different smooths. In the following chapter we use smooths which use stencils of varying size and shape. The stencil size and shape determine ghost zone depths and dependence distance, and we apply the optimizations presented here by tailoring them to the stencil.

CHAPTER 5

OPTIMIZATIONS FOR COMPUTE-INTENSIVE STENCILS

Finite-Volume/Finite-Difference solutions for partial differential equations (PDE) have, at their base, computations of *stencils*. In this context, a stencil approximates the application of a differential operator to a function evaluated on a rectangular grid. The error in the approximation is proportional to some integer power p of the grid spacing (p is referred to as the *order of accuracy* of the discretization).

For a given partial differential operator, the number of points required in a stencil typically increases as a polynomial in the order of accuracy. This is in contrast to the exponential dependence on p in the reduction of the error. Furthermore, larger stencils, while requiring larger numbers of floating-point operations, can be organized to require a comparable degree of main memory data movement as their lower-order counterparts. Thus, with processor architectures becoming more compute-intensive, high-order schemes are increasingly important, as they can achieve greater accuracy with less data movement. This property, combined with the need for computational scientists to minimize the memory capacity required to obtain a given level of error in their simulations, has been motivating the effort to increase the order of accuracy of stencil-based algorithms for PDE solvers.

For higher-order methods, the high arithmetic intensity of larger stencils limits performance. Thus, higher-order methods may not even achieve performance corresponding to the DRAM bandwidth. To address these performance bottlenecks, this chapter introduces and evaluates a novel compiler transformation, *partial sums*. Partial sums works by recognizing that for almost all stencils, there is data and operation reuse between neighboring points. This reuse is more significant for higher-order stencils, which examine many more neighboring input points to compute each output

point. Partial sums is used to exploit this reuse to reduce loads, while removing floating-point operations that are redundant across multiple output calculations, thus improving both computation and memory costs of the higher-order stencils. The transformation recognizes that stencil computation can be reordered, and therefore computes and reuses partial results in *partial sums*.

Finally, this chapter explores combining partial sums with the communication-avoiding and thread-scheduling optimizations presented in Chapter 4. By composing transformations we can take a higher-order smooth, remove its computation bottleneck, and make it bandwidth-limited. We can then further improve performance by applying communication-avoiding optimizations.

5.1 Stencil Definitions and Accuracy

A stencil is a linear transformation of the form: $L(a)_{\mathbf{i}} = \sum_{\mathbf{j}} \alpha_{\mathbf{j}} a_{\mathbf{i}+\mathbf{j}}$. This chapter considers a collection of test problems for stencil calculations, all of which are discretizations of Poisson's equation $\Delta(\phi) = f$, where $\Delta\phi \equiv \sum_{i=0}^{d} \frac{\partial^2 \phi}{\partial x_i^2}$. Here f is some given function, and we want to solve for ϕ. For this study, we assume that ϕ and f are periodic functions on $[0,1]^d$, i.e., $\phi(\mathbf{x}) = \phi(\mathbf{x}+\mathbf{q})$ for all integer tuples $\mathbf{q}$, and similarly for f. We compute approximate solutions on a rectangular lattice $\phi^h_{\mathbf{i}} \approx \phi(\mathbf{i}h)$ by solving the stencil equations (superscripts denote grid spacing) $L^h \phi^h = f^h$, where $f^h = f(\mathbf{i}h)$, $h = 1/N$ for some integer N. In that case, the periodicity of the exact solution translates into periodicity on the lattice: $\phi^h_{\mathbf{i}+\mathbf{q}1} = \phi^h_{\mathbf{i}}$, and similarly for f^h. Thus, we will evaluate our operator and solve our equations on the grid $[0, N-1]^d$, and use periodicity to evaluate stencil dependences that are not on that grid.

In this work, we examine the four representative stencils operators shown in Figure 5.1. Points with the same color have the same value for the coefficient. Stencils with the high degree of symmetry shown here are a consequence of the use of centered-difference approximations on rectangular grids, which is ubiquitous in discretizations of Poisson's equation and other constant-coefficient, elliptic operators on such grids. The standard approximation for explicit integrations use either the well-known 7-point [22] or 13-point stencils [50], with second- and fourth-order accuracy, respectively. The sixth-order (27-point) and tenth-order (125-point stencils) in our study are what are known as Mehrstellen stencils. They achieve their stated

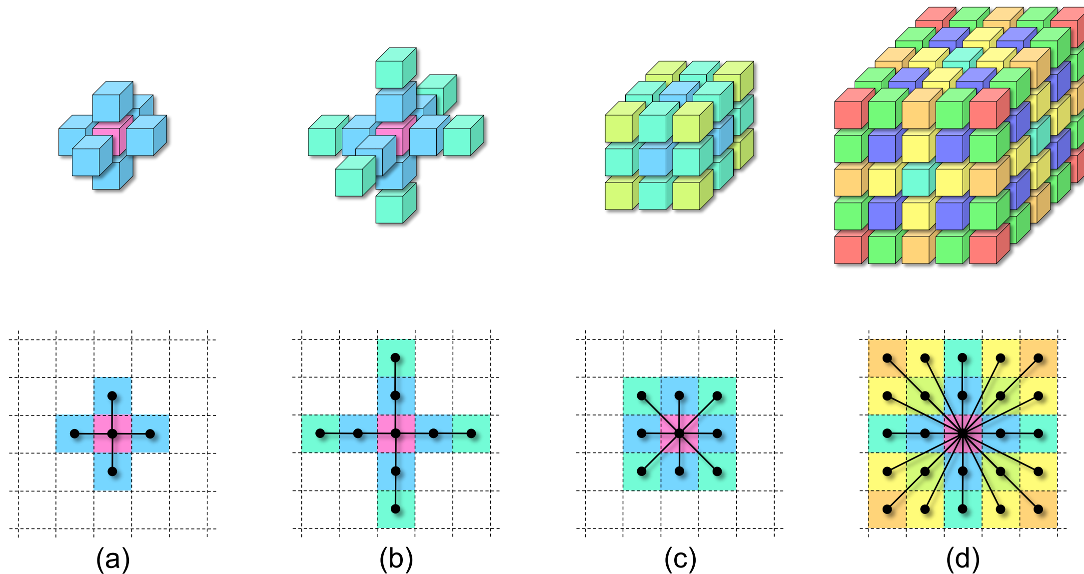


Figure 5.1: (top) Visualizations of the discretized 3D Laplacian operators (stencils) used in this chapter. (bottom) 2D cross sections through the centers of 3D stencils. Color is used to denote the coefficient associated with that point. The 27- and 125-point stencils have complex symmetries that we exploit.

accuracy only if the right-hand side is preprocessed. Rather than solving $L^h\phi^h = f^h$, one solves $L^h\phi^h = Mf^h$, where M is a stencil operator whose coefficients sum to 1. This is a one-time operation applied to the right-hand side, prior to applying whatever solution algorithm is used (e.g., geometric multigrid (GMG), as is used here), and hence does not have significant impact on the performance of the solver. The 27-point stencil and the associated Mehrstellen correction stencil are classical, and can be found in [22]. The 125-point stencil and its associated Mehrstellen correction stencil are new, and will be published elsewhere [23]. For the purposes of this paper, it is only necessary to know the symmetries of the operator, which are summarized in Figure 5.1. The accuracy claims for this method will be verified in Section 6.3.

5.2 Stencil Reordering: Partial Sums

The partial sum transformation described in this chapter targets constant-coefficient, out-of-place stencils[1]. This section illustrates the partial sum transformation, and

[1]defined in Chapter 1.

details of the compiler implementation are presented in Section 5.3, and performance impact as well as interaction with other optimizations is described in Section 5.4.

For an illustration of computing stencils via partial sums, consider the 9-point 2D stencil in Listing 5.1. The 9-point stencil is the 2D analog of the 27-point 3D stencil of Figure 5.1(c). Figure 5.2 illustrates the application of this stencil for three consecutive iterations <j,i>, <j,i+1> and <j,i+2> of the inner loop `i`. When the stencil is applied at <j,i> to compute `out[j][i]`, inputs `in[j][i+1]`, `in[j-1][i+1]` and `in[j+1][i+1]` are reused in the next two iterations of the i loop. The rectangle in bold in Figure 5.2 highlights the points that are reused. If we conceptualize the stencil as a box, these points are the *right edge* for iteration $\langle j, i\rangle$, the *center* for iteration $\langle j, i+1\rangle$ and the *left edge* for iteration $\langle j, i+2\rangle$.

Therefore, we employ an optimization that avoids loading all nine inputs of the 2D 9-point stencil, but instead loads only the right edge while reusing data from the previous two iterations of the i loop. At iteration $\langle j, i\rangle$, the right edge points `in[j][i+1]`, `in[j-1][i+1]` and `in[j+1][i+1]` are loaded. We capture the contribution these points make to the outputs at $\langle j, i\rangle$, $\langle j, i+1\rangle$, and $\langle j, i+2\rangle$ by calculating the weighted sum of the loaded edge with coefficients corresponding to the right, left, and center edge, as illustrated in Figure 5.3. The compiler constructs an array of coefficients to be used in the partial sum transformation. If we visualize the array of coefficients as a box as in Figure 5.3, with its entries corresponding to coefficients of the stencil, then the weighted sum of input array points with the right edge of the array of coefficients computes B0, the center computes B1, and the left edge computes B2. B0 is used to compute the output at $\langle j, i\rangle$, while B1 and B2 are buffered for outputs at the next two iterations of the i loop.

The use of buffers is illustrated in Figure 5.4 (top). The arrow marked 1 for the entry `R[i]` corresponds to B0. Arrows marked 2 and 3 for entries `C[i+1]` and `L[i+2]` correspond to B1 and B2, respectively. Finally, the output at <j,i>, `out[j][i]`, is computed as the sum `R[i]+C[i]+L[i]`. The buffer entry `R[i]` was computed in the current iteration <j,i>, whereas `C[i]` and `L[i]` were computed and entered in the buffer in previous iterations <j,i-1> and <j,i-2>, respectively.

As mentioned earlier, symmetries in stencil coefficients are ubiquitous, and we exploit symmetry to reduce floating-point operations in computing B0, B1, and B2.

The colors in the array of coefficients in Figure 5.3 represent symmetry, with each color representing a unique coefficient. The code implementing partial sums in Figure 5.4 (bottom) can now be explained using Figure 5.3 and Figure 5.4 (top). Symmetry about the $j-$axis means B0 and B2 (entries for R[i] and L[i+2]) are equal and do not have to be recomputed (line 3 in Figure 5.4). Symmetry about the $i-$axis means `in[j-1][i+1]` and `in[j+1][i+1]` will always be multiplied by the same coefficient, and thus we store their sum in $r2$; `in[j][i+1]` is loaded to $r1$. B0 is the weighted sum of $r1$ and $r2$ with coefficients $w1$ (blue) and $w2$ (green) (line 1 in Figure 5.4), respectively, and B1 with coefficients $w3$ (red) and $w2$ (green) (line 2).

In a similar manner, for 3D stencils we compute partial sums of 2D planes instead of 1D edges, and the array of coefficients in three dimensions instead of two. Figure 5.5 shows a 2D cross-section of a 3D stencil, and the symmetries present. With increasing number of symmetries the number of unique coefficients decreases. The 3D 125-point stencil we optimize in this dissertation exhibits symmetries about the `i,j,` and `k`-axes and diagonals, as shown in Figure 5.5(d).

Figure 5.6 shows a 2D plane of the 3D array of coefficients for the 125-point stencil. Each 2D plane has 6 unique coefficients labeled from 0-5. When computing partial sums with the 125-point stencil, the leading 2D plane of 25 points is loaded and used to compute buffer entries. As shown in Figure 5.6, the 25 points are then decomposed into six factors, `r0` to `r5`, which are multiplied by the unique coefficient and then summed to compute the buffer entries. This is similar to the 2D 9-point stencil discussed above, where the leading edge had two unique coefficients and was thus factored and stored in `r1` and `r2`.

Listing 5.2 shows simplified generated code for the 3D 27-point stencil. Three buffers are allocated, for the left, center, and right planes; and three scalars are created for three unique coefficients in each 2D plane. In the general case, the number of buffers required is twice the *radius* of the stencil plus one (radius is 1 for 7- and 27-point stencils, 2 for 13- and 125-point stencils). Partial Sums can be thought of as computing the output in two sweeps on the innermost `i` loop. As shown in the listing, the first loop computes the entries in the the three buffers, and the second loop sums these entries to compute the final output. In the compiler-generated code

```
1
2  for (j=0; j<N; j++)
3    for (i=0; i<N; i++)
4          out[j][i] = w1*(
5                  in[j-1][i  ] + in[j+1][i  ] +
6                  in[j  ][i-1] + in[j  ][i+1]
7          ) +
8                  w2*(
9                  in[j-1][i-1] + in[j+1][i-1] +
10                 in[j-1][i+1] + in[j+1][i+1]
11         ) +
12         w3* (in[j  ][i  ] );
```

Listing 5.1: The input code for a 2D 9-point stencil

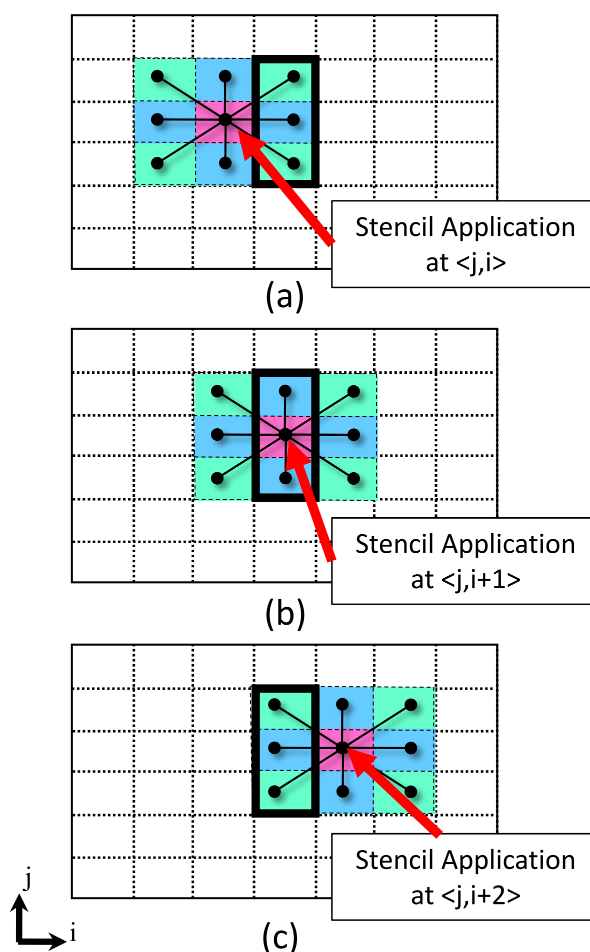


Figure 5.2: Visualization of 2D 9-point stencil application on a 2D grid. Figure shows stencil operator being applied on three consecutive iterations of the inner loop (j,i), (j,i+1) and (j, i+2). The edge of points {(j+1, i+1), (j, i+1), (j-1, i+1)}, bound by the bold rectangle, get reused by the three iterations. The coefficients of the stencil are color coded, blue = w1, green = w2, and red = w3.

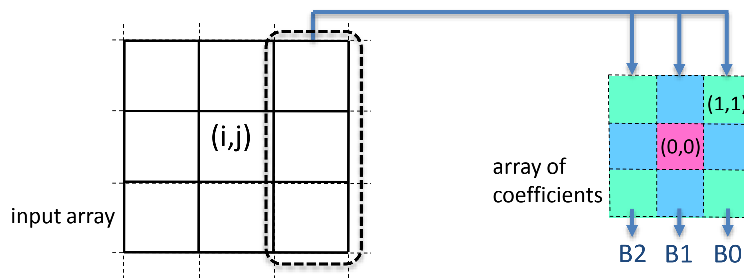


Figure 5.3: Illustration of deriving partial sums. The right edge from the input array is loaded and multiplied by weights stored in the array of coefficients. The sum of products of the loaded points and the right, center, and left edges of the array of coefficients are BO, B1, and B2, respectively.

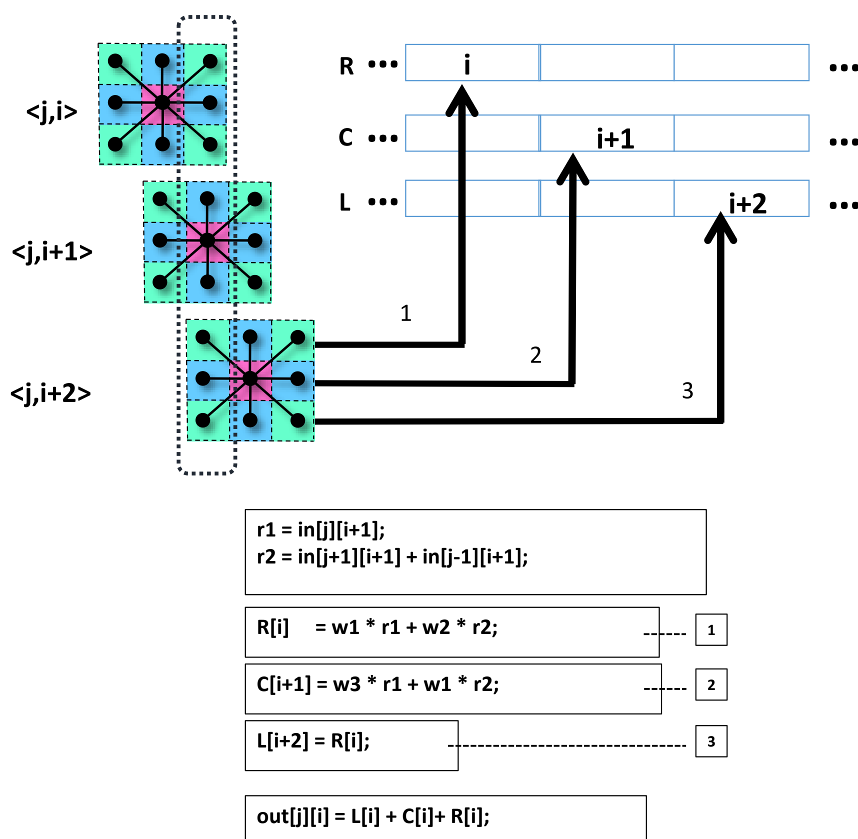


Figure 5.4: The reuse of the leading edge of points loaded at iteration <j,i> gets captured in three buffers R, C, and L (top). Buffer entires R[i] (1), C[i+1] (2), L[i+2] (3) correspond to B0, B1, and B2 from Figure 5.3. The loaded edge is factored into `r1` and `r2` based on symmetry of the color-coded coefficients. The factors are used to compute B0, B1, and B2. The final output `out[j][i]` is the sum of buffer entries.

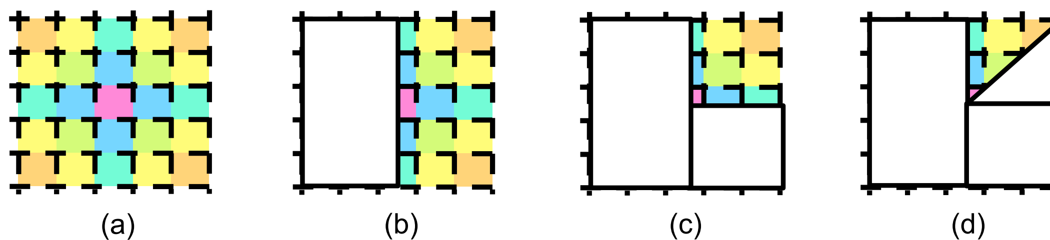


Figure 5.5: Increasing symmetries in coefficients allows us to increasingly reduce floating-point computation. Symmetry about the $j-$axis (in b) permits discarding half the coefficients, and symmetry about the $i-$axis (c) and the diagonal (d) lets the compiler consider even fewer coefficients.

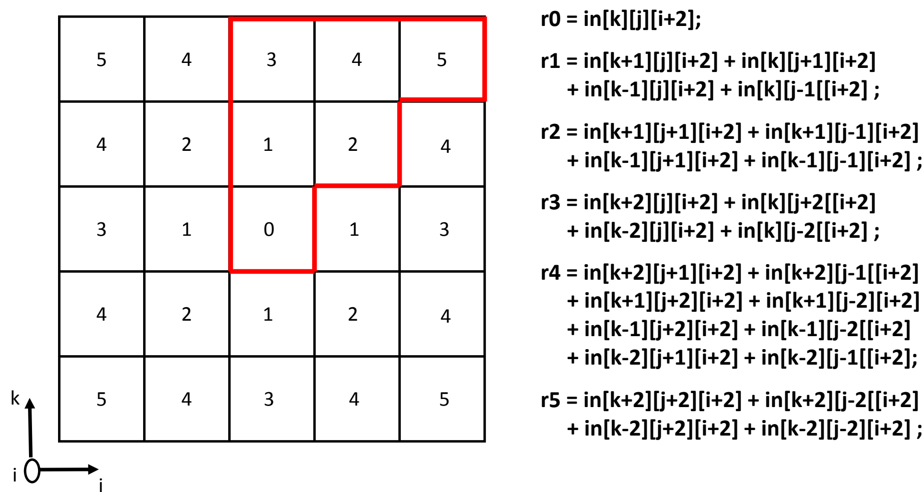


Figure 5.6: Visualization of a 2D plane from the 3D array of coefficients for the 125-point stencils (left). As shown in Figure 5.5(d), there are 6 unique coefficients 0-5. When applying the 125-point stencil at iteration <j,i> using partial sums, the leading 2D plane of 25 points is loaded and factored according to the unique coefficients (right). The six factors `r0` - `r5` are multiplied by appropriate coefficients to compute partial sums which are then buffered. The factors are created by summing loaded points which are multiplied by the same coefficient, and coeffient `0` corresponds to loaded point `in[k][j][i+2]`.

```
1  // distance of farthest stencil point
2  // from origin per dimension
3  int radius = 1;
4
5  // allocate 2*radius+1 buffered partial sums,
6  // N is grid (box) dimension
7  // create (radius+1)*(radius+2)/2 temporaries
8
9  double B0[N], B1[N], B2[N];
10 double r1, r2, r3;
11
12 for (k=0; k<N; k++){
13   for (j=0; j<N; j++){
14
15     // preamble code sets up the pipeline
16     ....
17
18     // steady state computation
19     // computation of the buffer entries
20     for(i=0; i<(N-radius); i++){
21
22       r1 = phi[k][j][i+1];
23       r2 = phi[k+1][j  ][i+1] + phi[k-1][j  ][i+1] +
24            phi[k  ][j-1][i+1] + phi[k  ][j+1][i+1];
25       r3 = phi[k+1][j+1][i+1] + phi[k+1][j-1][i+1]+
26            phi[k-1][j+1][i+1] + phi[k-1][j+1][i+1];
27
28       B2[i] = w1*r1 + w2*r2 + w3*r3;
29       B1[i+1] = w2*r1 + w3*r2 + w4*r3;
30       B0[i+2] = B2[i];
31     }
32
33     // summing the buffer entries
34     // In actual code these two loops get fused
35     for(i=0; i<(N-radius); i++)
36       phi_new[k][j][i] = B0[i] + B1[i] + B2[i];
37   ...
38   // cleanup code to avoid extra computation
39   ...
40   }
41 }
```

Listing 5.2: Output code for 3D 27-point stencil, optimized using partial sums.

we fuse the loops together using loop fusion.

While a reference implementation of the 9-point stencil necessitates 11 floating-point operations (8 adds, 3 multiplies) to compute each output point, our approach requires only 9 floating-point operations (3 adds and 4 multiplies for the partial sums, plus 2 adds to sum the symmetric partial results). The reduction in floating-point operations becomes more significant with the size and dimension of the stencil operator; for example, the 125-point stencil has 124 adds, but once optimized it has only 38 for a more-than-3× reduction.

An additional advantage of this approach is that it results in code amenable to SIMD code generation, AVX and SSE for our target architectures. Intuitively, SIMDization is enabled because the calculations in the loop, including the partial sum calculations, have unit stride across the innermost i dimension. While we are unable to isolate the SIMDization benefits, we performed an experiment to illustrate the differences between the 125-point stencil code before and after the partial sum optimization. Using Intel Architecture Code Analyzer (IACA), we profiled both versions of the code on an Intel Westmere i5-540M platform.[2] In the partial-sum-optimized code, all floating-point adds use SIMD instructions, whereas only about two-thirds of the adds in the baseline code use SIMD instructions. However, L1 and shuffle bandwidth can limit the ultimate benefit from increased SIMDization.

5.2.1 Composition of Optimizations

The partial sum optimization reduces floating-point operations in compute-bound kernels to make them more memory-bound. Once memory-bound, communication-avoiding optimizations introduced in Chapter 4 can be used to further improve performance.

5.3 Compiler Implementation

The partial sum transformation described in the previous section has been implemented in CHiLL, and extends previous communication-avoiding optimizations in CHiLL targeting GMG described in Chapter 4. Building new transformations into a polyhedral framework easily allows for composition of transformations as

[2] IACA does not run on our target machines.

long as data dependences are not violated. We compose partial sums with loop transformations to target both computation and communication bottlenecks. This section describes the abstractions used by the compiler in the automation of the partial sum transformation.

We make the following assumptions about the input code to our framework. As is standard with polyhedral compiler frameworks, we require that all subscript expressions are *affine*, or linear combinations of the loop indices and loop-invariant variables. The partial sum optimization requires that the subscript expressions are *separable*, such that each dimension references just a single loop index. Partial Sum applies to out-of-place stencils. Out-of-place updates are loop nest computations where the right-hand sides are read-only matrices per stencil sweep (e.g. Jacobi). Currently we are limited to constant-coefficient, out-of-place stencils.

5.3.1 Background Review

In most representative applications, stencils are implemented as multidimensional loop nest computations (all stencils considered are 3D). As described in Chapter 2, in CHiLL, we represent this loop nest by an iteration space *IS*, which mathematically describes polyhedra corresponding to points in the 3D iteration space:

$$IS = \{[l_1, l_2, l_3] : 0 \leq l_1, l_2, l_3 < N\} \tag{5.1}$$

By convention, l_3 is the innermost loop of a 3D loop nest. It is standard to normalize iteration spaces to start at 0. Bounds' constraints can be far more complex, but for simplicity of explanation, we show an upper bound that is a constant or variable.

5.3.2 Abstractions for Partial Sums

Examining the statement associated with a stencil computation, the compiler builds four abstractions to perform the partial sum optimization: (1) *StencilPoints* refers to the set of points that comprise the stencil, offset from a specific iteration in the 3D iteration space; (2) *BB* is the axis-aligned bounding box of *StencilPoints*,

such that each dimension derives its lower and upper bound from the minimum and maximum values in that dimension; (3) *Coeff* is a 3D array the same size as *BB* to hold the coefficients for the points in the stencil; and, (4) *Buffer* is a set of arrays that are used to hold the partial sums in the generated code. This subsection describes how these abstractions are derived automatically by the compiler and used by the code generator to produce code analogous to the example in Listing 5.2. For simplicity, we assume a unit stride for the loop iteration spaces, but extensions for nonunit-stride loops are straightforward.

5.3.3 Deriving StencilPoints and BoundingBox (BB)

At every iteration $\vec{I} = \langle i_1, i_2, i_3 \rangle \in IS$, we compute a single output of the stencil. We can rewrite an out-of-place, constant-coefficient p-point stencil as a weighted sum of p points, with w_m representing the coefficient for point m. A vector offset from iteration $\vec{I}$ for each point m is then $\vec{O}_m = \langle o_1^m, o_2^m, o_3^m \rangle$. The compiler computes BB from lower and upper bounds for each dimension of these offsets (i.e., $lb_1 = min_m o_1^m$, $ub_1 = max_m o_1^m$, $\ldots$). This notation gives rise to the following definitions:

$$out[i_1][i_2][i_3] = \sum_{m=1}^{p} w_m * in[i_1 + o_1^m][i_2 + o_2^m][i_3 + o_3^m]$$

$$StencilPoints = \bigcup_{m=1}^{p} \vec{O}_m$$

$$BB = \{[b_1, b_2, b_3] : \quad lb_1 \leq b_1 \leq ub_1, lb_2 \leq b_2 \leq ub_2, lb_3 \leq b_3 \leq ub_3\}$$

5.3.4 Deriving Coefficients (Coeff)

If *StencilPoints* and *BB* represent the identical volume, as in the 27-point stencil, we call this a *full* stencil. The star-shaped 7-point stencil operators of Figure 5.1(a) are not full. A set difference mathematically determines the holes in the stencil. The compiler creates an array *Coeff*, the same size as *BB*, to store the coefficients for the partial sum. For simplicity of explanation, we will assume *Coeff* is centered at $\langle 0, 0, 0 \rangle$ and allows negative indices. Points $\vec{B} = \langle b_1, b_2, b_3 \rangle \in BB$ which belong to *StencilHoles*, are set to zero in the array of coefficients, and others are assigned

appropriate constant values. We can then rewrite the stencil computation at each output point $\vec{I}$ using the following equations:

$$StencilHoles = BB - StencilPoints$$
$$Coeff[b_1][b_2][b_3] = \begin{cases} 0 \quad : & (\vec{B}) \in StencilHoles \\ w_m : & (\vec{B} = \vec{O}_m) \in StencilPoints \end{cases}$$
$$out[i_1][i_2][i_3] = \sum_{\vec{B} \in BB} Coeff[b_1][b_2][b_3] * in[i_1 + b_1][i_2 + b_2][i_3 + b_3]$$

5.3.5 Deriving Partial Sums and Buffers

We form partial sums for subsets of BB, planes for a 3D stencil or similarly, lines for a 2D stencil as in Figure 5.4. $Plane_k$ is the p_1, p_2 plane inside the BB at $p_3 = k$. The BB can be partitioned into $(ub_3 - lb_3 + 1)$ such planes (corresponding to the stencil radius plus 1); there is a corresponding plane in *Coeff* such that points in BB are array indices for elements in *Coeff*. We can compute an output point $\vec{I}$ as a sum of partial sums PS_k at each plane $k \in BB$, which is staged in a buffer. There are $(ub_3 - lb_3 + 1)$ buffers, each as wide as the trip count of the inner loop. Within the innermost loop, the values in the buffers are summed to compute the output. These rewrites of the stencil are captured as follows:

$$Plane_k = \{[p_1, p_2, p_3] : lb_1, lb_2 \leq p_1, p_2 \leq ub_1, ub_2; p_3 = k\}$$
$$PS_k(\vec{I}) = \sum_{\vec{P} \in Plane_k} Coeff[p1][p2][p3] * in[i_1 + p1][i_2 + p2][i_3 + p3]$$
$$Buffer_k[i_3] = PS_k(\vec{I})$$
$$out[i_1][i_2][i_3] = \sum_{k=lb_3}^{ub_3} Buffer_k[i_3]$$

5.3.6 Exploiting Reuse in Partial Sums

The optimizations derived from partial sums recognize that loads associated with a plane have reuse in the third dimension. At iteration $\vec{I}$, the rightmost plane $Plane_r(\vec{I})$, where $r = ub_3$, is of particular importance. This is the leading plane in the bounding box of the stencil (in the direction of increasing third dimension), and it corresponds to the right edge of the 2D stencil described in Section 5.2. We load only this plane, and reuse it for other output points. Let $\vec{I^k} = \langle i_1, i_2, i_3 + k\rangle \in IS, 0 \leq k \leq (ub_3 - lb_3)$. $Plane_r(\vec{I})$ is the same set of points as $Plane_{r-1}(\vec{I^1})$, and $Plane_{r-k}(\vec{I^k})$ in general. $Plane_r(\vec{I})$ can thus be used to compute $(ub_3 - lb_3 + 1)$ partial sums. The partial sums are computed by using points from $Plane_r(\vec{I})$ and sweeping through the planes of *Coeff*; the results are buffered in $Buffer_{r-k}[i_3 + k]$, for the range of values of k.

Buffer and output at $(\vec{I})$ are computed from PS as in the previous step.

We can now derive the partial sums from just the load of $Plane_r(\vec{I})$. The following is computed $\forall k, 0 \leq k \leq (ub_3 - lb_3)$:

$$PS_{r-k}(\vec{I^k}) = \sum_{\vec{P} \in Plane_r} Coeff[p_1][p_2][ub_3 - k] * in[i_1 + p1][i_2 + p2][i_3 + p3]$$

5.3.7 Exploiting Symmetry to Reduce Floating-Point Operations

Figure 5.5 shows that several coefficients in each plane of Coeff have the same value. This means multiple points in $Plane_r$ use the same coefficient to compute partial sums. For each unique coefficient, we sum the points in $Plane_r$ that use it. This sum is stored, and multiplied by the appropiate weight to calculate the partial sums, and does not have to be recomputed. In our current implementation we check for symmetry in BB about the axes and diagonals in a plane and across planes before implementing our floating-point reducing optimization. This technique can be easily extended to work with fewer degrees of symmetry. When all the symmetries are present, BB is a cube such that upper and lower bounds are the same, lb = -ub, and the coefficients in each $\langle p1, p2\rangle$ plane are symmetric about the $p2$ and $p3$ axes and diagonals.

In any 2D plane of *Coeff*, the coordinates of unique coefficients are defined as $UC = \{\langle p1, p2\rangle : 0 \leq p1 \leq p2 \leq ub\}$, corresponding to the colored octant in Figure 5.5(d). From reflections about the axes and diagonals, for any point $\langle p1, p2\rangle \in UC$, the set of reflected points $\mathit{Ref}(p1, p2) = \{\langle \pm p1, \pm p2, ub\rangle, \langle \pm p2, \pm p1, ub\rangle\}$ in $Plane_r$ are weighted with the same coefficient *Coeff* [p1][p2][ub-k]. The sum of the points in the input `in`, corresponding to $\langle p1, p2\rangle$, is $R_{p_1p_2}$. We rewrite partial sums using the factored terms to exploit symmetry. For stencils with zero-valued coefficients in *Coeff*, like the 7-point stencil, we take care not to generate redundant factored terms. The following is computed $\forall_k, 0 \leq k \leq (ub - lb)$:

$$
\begin{aligned}
R_{p_1p_2} &= \sum_{\langle x,y\rangle \in \mathit{Ref}(p1,p2)} in[x][y][ub] \\
PS_{r-k}(\vec{I^k}) &= \sum_{\langle p1,p2\rangle \in UC} \mathit{Coeff}[p_1][p_2][ub-k] * R_{p_1p_2} \\
\mathit{Buffer}_{r-k}[i_3 + k] &= PS_{r-k}
\end{aligned} \tag{5.2}
$$

5.3.8 Code Generation

Once the compiler has performed the rewriting steps described above, it must generate the transformed code. The steps of code generation are described in detail in Figure 5.7. The compiler must create the buffer objects, and compute their values using the rewriting shown in Eqn.(2) directly above. It must also modify the original stencil statement to refer to the buffers rather than the input code. As we near the upper bound while iterating through the inner-loop, there will be no need to calculate all the partial sums, and we will go off the end of the allocated buffers. The figure shows the restricted iteration space, and loop peeling [40] to address this. We also generate a code variant where *IS'* equals the original *IS*, which does extra computation, allocates buffers longer than the loop bound, but has no cleanup code. Both the code variants perform similarly.

Allocate buffer objects:
Assume stencil statement $S0$ has iteration space IS defined in Eqn.5.1.
Create $ub_3 - lb_3 + 1$ ($2ub$+1, when symmetrical) buffers, $Buffer_{lb_3}, \ldots, Buffer_{ub_3}$.
Each buffer is an array of length N (from IS).
Create (1+ub)(2+ub)/2 scalars to hold sums $\mathbf{R_{p_1p_2}}$ for each unique coefficient in a plane.

Insert statements to compute buffers.
Create a new compound statement $S1$ that has (1+ub)(2+ub)/2 statements to compute the sums $\mathbf{R_{p_1p_2}}$.
To $S1$ append $ub_3 - lb_3 + 1$ statements to compute the buffers.
Each statement is of the form in Eqn.5.2
Since all values in *Coeff* are constant, these are copied directly into the statement (not an array reference).
Insert $S1$ lexicographically before $S0$.

Update *S0* to use buffers.
Create a new statement $S2$ that computes the output at $\vec{I}$ from the sum of all buffers.
Replace $S0$ with $S2$.

Update *IS*.
Decrease the number of iterations of IS to avoid going off the end of the buffers.
Create new iteration space
$IS' = \{[l_1, l_2, l_3] : 0 \leq l_1, l_2 < N \&\& \ 0 \leq l_3 < N - (ub_3 - lb_3) \ \}$
Peel off remaining iterations and use *S0*.

Figure 5.7: Code generation steps for partial sum transformation.

5.4 Experimental Results

To investigate the efficacy of partial sums, we apply it to various discretizations of Poisson's equation, with orders of accuracy p ranging from 2 to 10. Numerical solutions to Poisson's equation are ubiquitous in simulations of a variety of physical problems, including fluid dynamics, astrophysics, electromagnetics, and plasma physics. We evaluate our approach in the context both of applying the operators in a stand-alone fashion, and within a Geometric Multigrid (GMG) method for solving Poisson's equation using these discretizations.

In this section we present performance results and analysis using our optimizing compiler technology. Additionally, the Roofline performance model is used to help

quantify attainable performance. The Roofline model provides a nominal upper bound to attainable performance.

5.4.1 Review of Evaluated Platforms

We evaluate the benefits of our compiler technology using the systems described in Chapter 4, and detailed in Table 4.2. The following is a brief summary of the two platforms.

Edison is a Cray XC30 at NERSC. Each node contains two 12-core Xeon Ivy Bridge chips, each with four DDR3-1600 memory controllers and a 30MB L3 cache. Each core implements the 4-way AVX SIMD instruction set and includes both a 32KB L1 and a 256B L2 cache. With a high flop-to-byte ratio, we expect this machine to be memory-limited for most operations without communication-avoiding optimizations.

Hopper is a Cray XE6 at NERSC. Each node contains four 6-core Opteron chips, each with two DDR3-1333 memory controllers and a 6MB L3 cache. Each core uses the 2-way SSE3 SIMD instruction set and includes both a 64KB L1 and a 512KB L2 cache. Hopper's lower machine balance may result in the 125-point operator being compute-limited.

On each platform, we used the installed Intel compiler with flags: `-O3 -fno-alias -fno-fnalias` and either `-xAVX` or `-msse3`.

5.4.2 Problem Solved on miniGMG

Multigrid is a fast linear solver that uses an iterative and recursive approach to solve elliptic PDEs. Each iteration of a multigrid solve requires performing a V-Cycle. As shown in Figure 3.1, a V-Cycle involves performing stencil operations (smooths) on progressively coarser (smaller) grids, solving a coarse grid problem, and then using that coarse-grid solution to correct the fine-grid solution.

The implementation described in this chapter uses the miniGMG benchmark introduced in 3. The miniGMG configuration used in the chapter is described in the previous chapter, Section 4.6.2. The benchmark creates a block structured 3D grid partitioned into subdomains (boxes) which are distributed among processes. In all experiments, our finest grid in miniGMG is 256^3 cells. This 256^3 grid is decomposed into disjoint 64^3 boxes (requiring ghost zone exchanges) distributed among multiple

processes on one compute node. To optimize for NUMA, we run with one MPI process per NUMA node (2 per Edison node and 4 per Hopper node). miniGMG implements a multigrid V-Cycle that is terminated when each box reaches 4^3 cells. At each level, we apply four weighted Jacobi smooths using the relevant stencil shown in Figure 5.1. Our experiments run a fixed 10 V-Cycles with a point relaxation bottom solver that performs 48 smooths.

As in Chapter 4, the smooths are decomposed into three separate loop nests: the first applies one of the stencils to an array and writes to the temporary array, the second reads the temporary array and forms either the Poisson or Helmholtz operator, while the third performs a weighted Jacobi update of the current solution.

We construct a manufactured solution:

$$u_{true} = sin^{13}(2\pi x)sin^{13}(2\pi y)sin^{13}(2\pi z)+sin^{13}(6\pi x)sin^{13}(6\pi y)sin^{13}(6\pi z) \text{ on } [0,1]^3$$

and symbolically apply the Laplacian to it to find a nominal right-hand side f. We then apply the appropriate Mehrstellen correction and solve $L^h u^h = Mf^h$. We compare u^h and u_{true} under the l_2 norm to calculate error.

5.4.3 Experimental Methodology

The input to CHiLL is source code written in C or Fortran and a *script* describing the set of transformations to be composed to optimize the provided source. Listing 5.3 shows the loop structure of input code for Jacobi smooth with various stencils. Listing 5.4 illustrate a CHiLL script to optimize a 125-point smooth and create a wavefront computation with partial sums.

The first few line of the CHiLL script initializes the compiler framework, then line 14-15 creates a wavefront computation. A new script command for the partial sum transformation called *partial_sums* has been implemented in CHiLL, which identifies the stencil statement to which this transformation should be applied. After creating a wavefront, *partial_sums* is applied to the two stencil operations. Partial sums creates multiple statement, thus loop fusion is needed to fuse all statements together. Finally OpenMP parallel code is generated using the final CHiLL command in the script. The command directs the compiler to insert an OpenMP parallel-for surrounding the third loop in the loop nest (outermost loop is number one), and set the number of threads to 4.

```
1    if(even_sweep){
2
3        for (k=0;j<N;k++)
4          for (j=0;j<N;j++)
5            for (i=0;i<N;i++)
6                //stencil operator
7                even_statement_S0()
8
9        for (k=0;j<N;k++)
10         for (j=0;j<N;j++)
11           for (i=0;i<N;i++)
12               //form Poisson or Helmholtz operator
13               even_statement_S1()
14
15       for (k=0;j<N;k++)
16         for (j=0;j<N;j++)
17           for (i=0;i<N;i++)
18               //Jacobi update
19               even_statement_S2()
20
21   } else {
22
23       for (k=0;j<N;k++)
24         for (j=0;j<N;j++)
25           for (i=0;i<N;i++)
26               //stencil operator
27               odd_statement_S0()
28
29       for (k=0;j<N;k++)
30         for (j=0;j<N;j++)
31           for (i=0;i<N;i++)
32               //form Poisson or Helmholtz operator
33               odd_statement_S1()
34
35       for (k=0;j<N;k++)
36         for (j=0;j<N;j++)
37           for (i=0;i<N;i++)
38               //Jacobi update
39               odd_statement_S2()
40   }
```

Listing 5.3: Loop structure of the input Jacobi smooth. The even and odd statements are identical, excpect that the input and output grids/arrays are swapped. Statement 0 applies the 7,13,27,125-point operators. The code for the stencil operators are listed in Appendix A.

```
1  #code generation is
2  #specialized to problem size
3  known (K == 64)
4  known (J == 64)
5  known (I == 64)
6  known (ghost_zone == 4)
7
8  #initializes CHiLL's
9  #abstraction of loops
10 original()
11
12 #skew followed by permute
13 #to create a wavefront
14 skew ([0,1,2,3,4,5], 2, [3,1])
15 permute ([2,1,3,4])
16
17 #apply partial sums
18 #to the two stencil operations
19 #for the Jacobi smooth
20 partial_sums(0)
21 partial_sums(5)
22
23 #fuse the statements created
24 #by the partial_sums transformation
25 fuse([0,1,2,3,4,5,6,7,8,9], 4)
26
27 #parallel code generation
28 #add omp parallel for to loop level 3 (j)
29 #with num_threads set to 4
30 omp_par_for(3, 4)
```

Listing 5.4: CHiLL script to create a 2-deep wavefront with partial sums for Jacobi smooth with 125-point stencil.

CHiLL scripts are created to vary the depth of the ghost zone (thus the depth of the wavefront), which depends on the radius of the stencil. The number of threads is also varied, and scripts are created with partial sums turned on and off.

5.4.4 Computing Roofline Memory Bounds

Tables 5.1 and 5.2 provide the Roofline memory bounds for ApplyOp and smooth, respectively. Roofline memory bounds place an upper limit on the number for stencils that can be computed if the bottleneck is DRAM bandwidth. The results are expressed in terms of Million Stencils per second. Thus computing a stencil sweep on a 256^3 grid in a second means $256^3 \div 10^6$ or 16.78 million stencils per second.

The analysis required to compute these bounds is explained with the concrete example of the 7-point stencil used in ApplyOp on Edison. We apply the stencil on a 256 domain which has been split into 64 64^3 subdomains. Each subdomain is a 66^3 grid or array to account for the one deep ghost zone. Computing ApplyOp requires reading in and writing out 64, 66^3 double precision arrays. As we do not perform explicit cache bypass we generate memory traffic for reading in the arrays, write allocation for the arrays and finally write back. The total memory traffic is D = $3 \times 8 \times 64 \times\ 66^3$ bytes. Edison's observed stream bandwidth of B = 88 GB/s, and the time required to stream in and write out the two arrays is T = D/B. Thus if we are bottlenecked by memory bandwidth, in time T, the number of stencils computed would be S = 64×64^3, and the number of millions stencils computed per second would simply be S /(T$\times 10^6$) or 3343. The other entries in the two tables are computed similarly. Stencils with a larger radius require deeper ghost zones and need more data movement and have lower Roofline bounds.

Smooths ($x_{new} = x + wD^{-1}(b - Ax)$) require two more arrays than ApplyOp ($y = Ax$). The right-hand side b and the inverse of the diagonal D^{-1} need to be streamed in from DRAM. The data traffic for the Jacobi smooth with a seven point is D'= $5 \times 8 \times 64 \times\ 66^3$ bytes. Smooths have higher memory movement compared to ApplyOp, and thus have lower Roofline bounds.

Table 5.1: The table shows the Roofline memory (DRAM) bounds for ApplyOp computation with four different stencils. ApplyOp reads in a grid, applies the stencil operator to it and writes out the computed values to a different output grid. The bounds are expressed in terms of Million Stencils per Second that can be computed when DRAM bandwidth is the bottleneck. The stencil was applied to a 256^3 domain which was split into a list of 64^3 sudomains. To account for ghost zones a 64^3 sudomain translates to 66^3 and 68^3 grids for stencils with radius 1 and 2 respectively. Stencils with larger radius require larger data volumes and have lower Roofline memory bounds.

Roofline Memory Bound (256^3 domain per node)

	Stencil	**MStencils per second**	
Stencils	**Radius**	**Hopper**	**Edison**
7, 27-point	1	1824	3343
13, 125-point	2	1667	3057

Table 5.2: The table shows the Roofline memory (DRAM) bounds for the Jacobi smooths using four different stencils. The bounds are expressed in terms of Million Stencils per Second that can be computed when DRAM bandwidth is the bottleneck. In addition to input and output grids, smooths require reading in two more grids (arrays) rhs and lambda. This extra data movement means smooths have lower Roofline bounds than ApplyOp.

Roofline Memory Bound (256^3 domain per node)

	Stencil	**MStencils per second**	
Stencils	**Radius**	**Hopper**	**Edison**
7, 27-point	1	1094	2006
13, 125-point	2	1000	1834

5.4.5 Stencil Performance

To highlight the memory bottleneck on modern multicore processors as well as differentiate ApplyOp ($y = Ax$) characteristics from smooth characteristics, Figure 5.8 presents ApplyOp (stencil in isolation) performance on the finest (256^3) grid, using either the baseline implementation or CHiLL with the Partial Sums optimization. We provide a memory bound based (blue circle) on the Roofline performance model derived from the size of the arrays (including ghost zones) and the bandwidths of the target machines listed in Table 4.2.

As shown in Figure 5.8, baseline performance is close to the Roofline bound for most cases. The use of the partial sums optimization to eliminate unnecessary

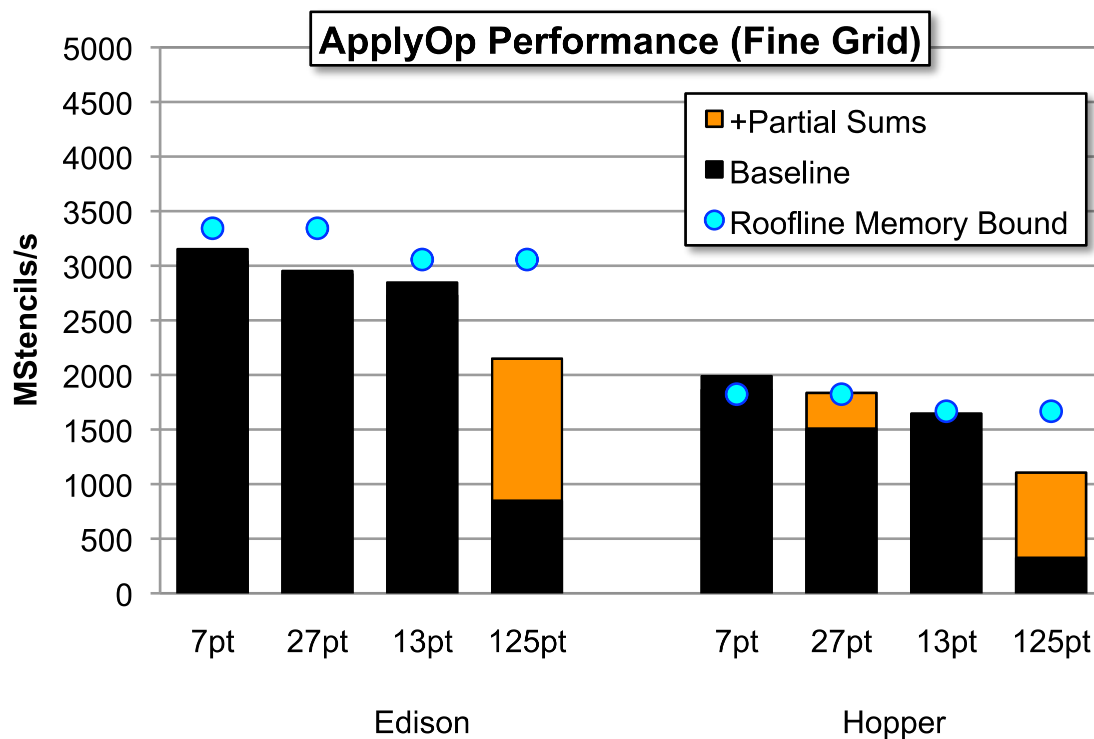


Figure 5.8: ApplyOp (y=Ax) stencil performance attained with the CHiLL compiler by optimization, operator, and platform. The Roofline memory bound is for the noncommunication-avoiding implementation. Partial sums can move compute-limited operations towards a memory-limited state.

floating-point adds and regularize the computation for efficient SIMDization significantly improved performance and ensured performance came close to the Roofline bound. Note, however, that the compute-intensive 125-point stencil fell well below its Roofline bound.

5.4.6 Smooth Performance on the Fine Grid

The core operation in a multigrid solver is a smooth. It is applied multiple times in sequence on each level of the multigrid V-Cycle. As such, we may apply a wavefront or time-skewing technique to overcome the memory bandwidth limit for ApplyOp. We use a fixed 256^3 problem for all experiments. Although this ensures we may compare smooth performance when using different discretizations of the Laplacian (different stencils), it also implies that the result using a 125-point operator will be more accurate. Please note that unlike a simple stencil operation ($y = Ax$), a smooth ($x_{new} = x + wD^{-1}(b - Ax)$) requires significantly more data movement, including reading arrays for the right-hand side b and the inverse of the diagonal D^{-1}. As a result, our smooth is nominally memory-limited for all operators.

Figure 5.9 presents smooth performance on the finest grid (256^3) as a function of operator, platform, and optimization. Once again we have included a Roofline bound (blue circle) to indicate the nominal performance bound of a smooth prior to any kind of communication-avoiding algorithmic change. Observe that the baseline implementation of the smooth is well below the Roofline bound in all cases. This is a result of the smooth being applied in two stages — application of the linear operator, and using that result to correct x. Enabling the "Fusion" optimization in CHiLL allows the compiler to automatically fuse these operations and eliminate the access to the intermediate temporary array. This significantly reduces data movement and allows the more memory-intensive 7- and 13-point smooths to reach their Roofline bounds.

Unfortunately, the more compute-intensive 27- and 125-point stencils demand efficient SIMDization to reach their Roofline bounds. Enabling the "Partial Sums" optimization in CHiLL allows the compiler to automatically restructure these stencils to eliminate superfluous additions and regiment the computation for SIMDization by

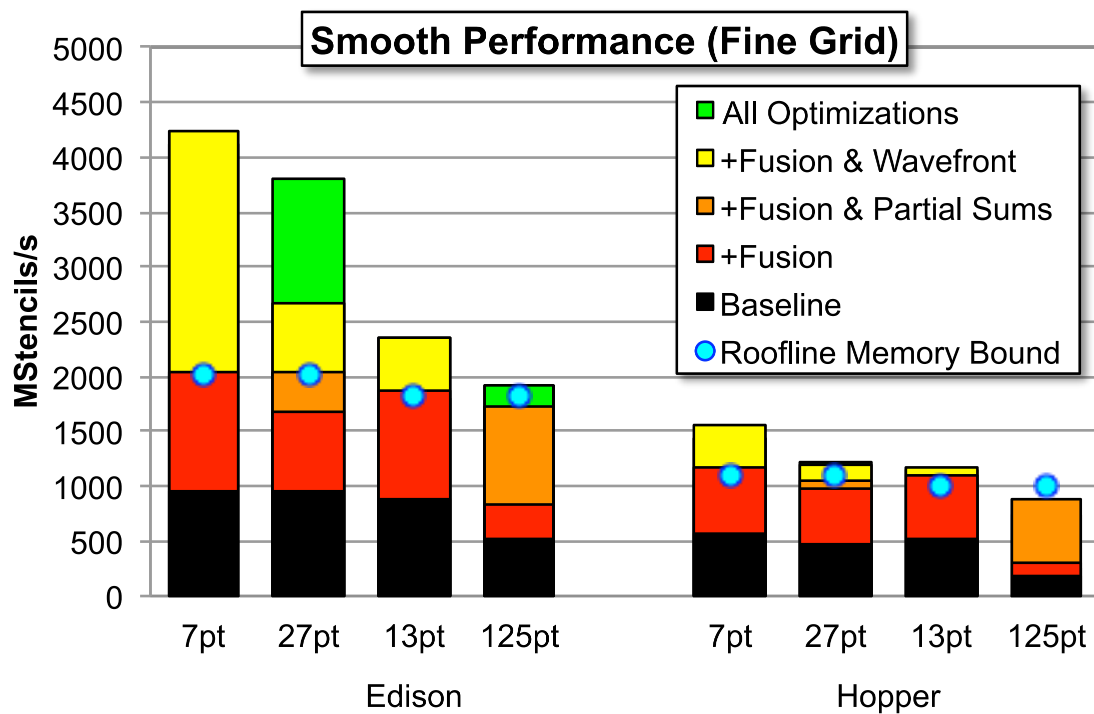


Figure 5.9: Jacobi smooth performance attained with the CHiLL compiler by optimization, operator, and platform. The Roofline memory bound is for the noncommunication-avoiding (no wavefront) implementation and is lower than ApplyOp due to additional data movement like the RHS. The wavefront transformation allows CHiLL to exceed this limit.

the backend compiler. The benefit is clear — a more than doubling of 125-point smooth performance and performance near the Roofline bound for all operators. Naively, one may conclude that reaching the Roofline-bound represents the upper end of performance. However, this simply implies that a new set of algorithmic optimizations are required to further improve performance.

As smooths are applied in sequence within the multigrid V-Cycle, it is possible to view their execution as a quadruply nested loop. Manually reordering these loops is beneficial, but unproductive. Conversely, our additions to CHiLL allow the compiler to automatically add ghost zones and restructure the loops into a communication-avoiding "wavefront" without loss of accuracy (the result is bit identical) or productivity. As seen in Figure 5.9, our approach attains roughly a 2× performance boost for the 7- and 27-point smooth on Edison. (Note, "All Optimizations" include tuned nested OpenMP.) Whereas Edison is heavily memory-limited, Hopper is not. As such, the benefit of a communication-avoiding algorithm is limited on Hopper. Communication-avoiding 13- and 125-point smooths suffer on two axes. First, generating wavefronts for these operators requires skewing loops by the larger stencil radius. This larger skew factor increases the working set, increases cache pressure and makes it difficult to fit the working set in the fastest caches. Second, the 125-point operator is likely compute-bound on Hopper and nearly compute-bound on Edison. Thus, the potential benefit from communication-avoiding is small.

5.4.7 Smooth Performance Throughout the V-Cycle

Unlike simple explicit methods that only need to attain high performance for a stencil on a large grid, multigrid requires high performance on grids of exponentially varying size. In Figure 5.10, we explore the performance of the 27- and 125-point smooths on Edison as they operate on coarser (smaller) grids. Examining the baseline implementation for the 27-point operator shows the expected rise in performance when moving to the coarser grids, which nominally fit in ever lower levels of cache. Note, the first smooth at each level will inevitably read from DRAM. As such, high cache bandwidths only amortize this slow initial smooth. On small grids, efficient 12-way OpenMP multithreading becomes impossible and performance drops. The

125-point smooth sees a similar behavior but to a lesser degree, as it is ultimately compute-limited.

As optimizations are enabled in CHiLL, we see the compiler can nearly sustain constant performance for the 256^3, 128^3, and 64^3 levels for the 27-point operator by automatically tuning for the optimal optimizations. Similarly, Table 5.3 shows the compiler continually shifts the set and parameterization of the optimizations employed for the 125-point smooth at each level of the V-Cycle. A manually optimized implementation would likely only target the fine grid and would thus deliver lower performance on the coarse grids, while significantly increasing programmer overhead.

The partial sums optimization requires a two-pass approach in which the first creates a few auxiliary results. The cost of this initial pass is amortized on large arrays but becomes an impediment on small arrays. Thus the benefit of partial sums decreases on the smaller grids.

5.4.8 miniGMG Solver Performance and Error

Figure 5.11 presents the performance of the miniGMG multigrid solver using either the existing Intel compiler (baseline) or our optimizing CHiLL compiler as a function of discretization and platform. Performance is expressed in millions of degrees of freedom solved per second (DOF/s). For this scalar problem, fine-grid cell is one degree of freedom. Thus, solving a 256^3 grid in 1 second would equate to computing 16.78 million DOF/s. Generally, the CHiLL compiler can provide an overall speedup of about 2× using all available optimizations. The attained speedup was a bit less on the 13-point operator, as it did not benefit from partial sums, and achieving high performance on a communication-avoiding wavefront is particularly challenging.

When comparing raw performance (MStencil/s), results show that the 7- and 27-point operators were comparable with the 125-point operator, attaining about half the throughput. However, this only conveys half the story. As highlighted in Figure 5.12, miniGMG with the Mehrstellen correction attains roughly 2,370× and 377,000× better accuracy for our test problem using the 27- or 125-point operators (respectively) than miniGMG attained using the 7-point operator. As we move to ever finer domains, the benefit increases. Thus, if the goal were to solve to a particular

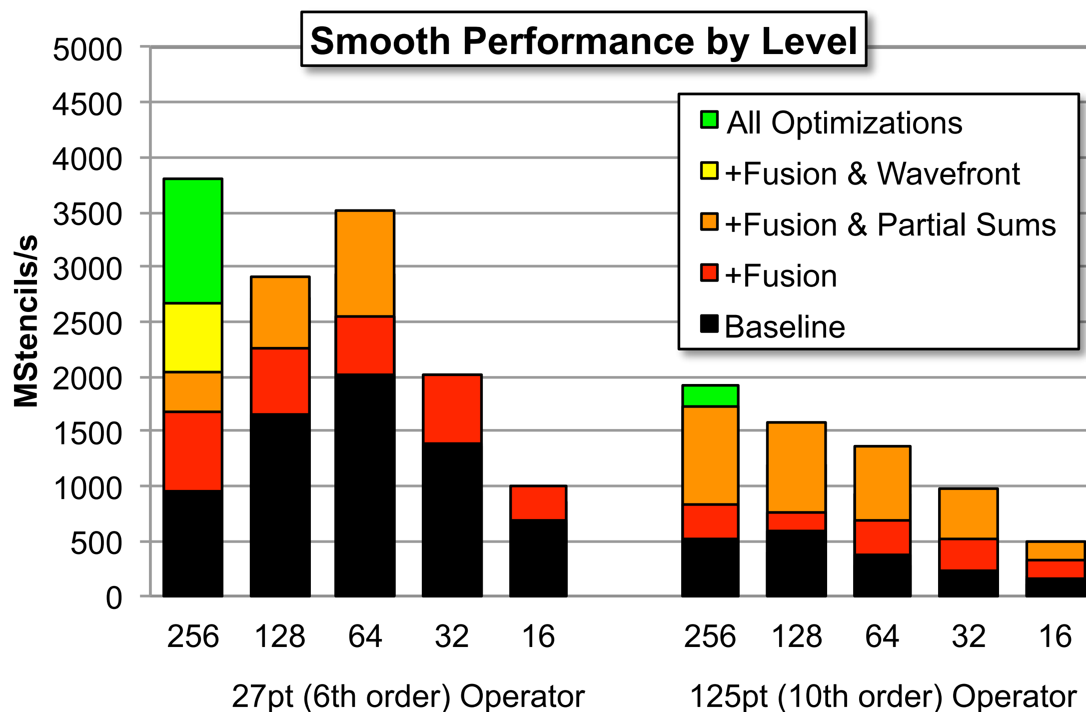


Figure 5.10: Jacobi smooth performance on Edison attained with the CHiLL compiler as a function of level in the V-Cycle (256^3 fine grids down to 16^3 coarse grids) for the 27- and 125-point operators. Observe that the reference implementation of the memory-limited 27-point operator receives a cache boost on the coarser levels, while the compute-limited 125-point does not.

Table 5.3: CHiLL was able to select optimizations uniquely for each multigrid level and platform. <#,#>denotes the number of inter- and intra-box threads with nested OpenMP.

125-point smooth on Edison

Level	256^3	128^3	4^3	32^3	16^3
Box Size	64^3	32^3	16^3	8^3	4^3
Operator Fusion	✓	✓	✓	✓	✓
Partial Sums	✓	✓	✓	✓	✓
Wavefront Depth	2	-	-	-	-
Nested OpenMP	<4,3>	<12,1>	<12,1>	<12,1>	<12,1>

125-point smooth on Hopper

Level	256^3	128^3	4^3	32^3	16^3
Box Size	64^3	32^3	16^3	8^3	4^3
Operator Fusion	✓	✓	✓	✓	✓
Partial Sums	✓	✓	✓	✓	✓
Wavefront Depth	1	-	-	-	-
Nested OpenMP	<6,1>	<6,1>	<6,1>	<6,1>	<6,1>

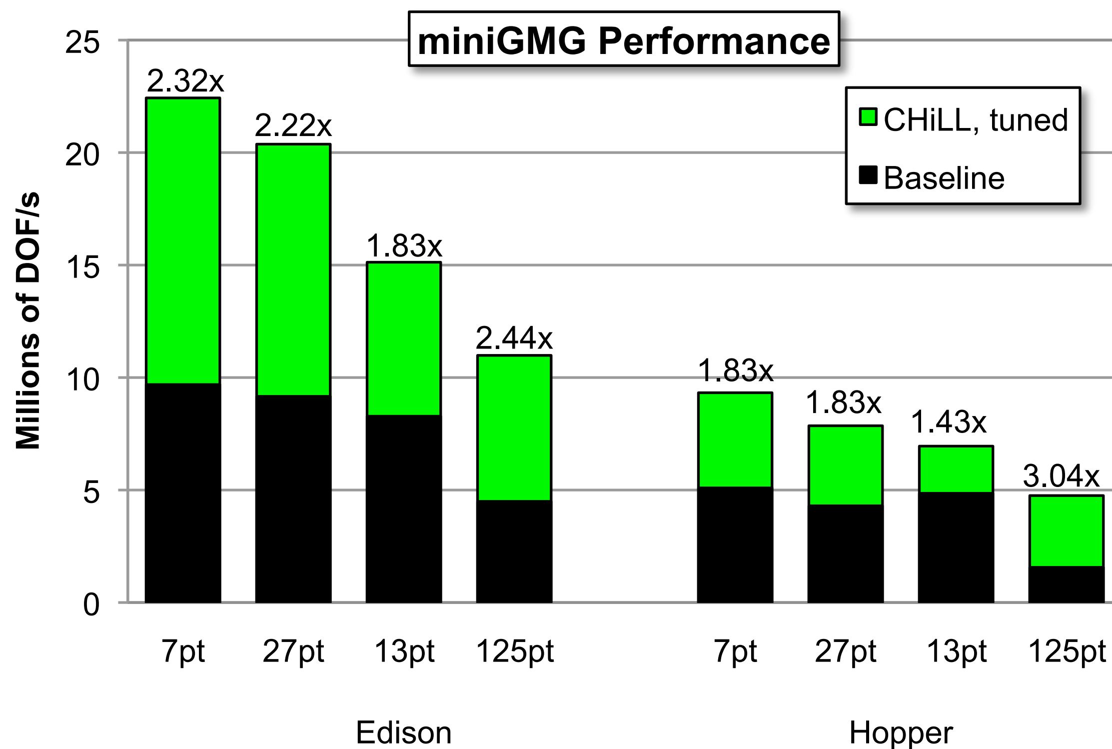


Figure 5.11: miniGMG performance (millions of degrees of freedom solved per second) using either the Intel compiler (baseline) or the CHiLL compiler. The labels indicate the overall solver speedup attained via CHiLL. The performance of the tenth-order 125-point solver is within a factor of 2 of the second-order 7-point solver, but provides nearly a million times lower error.

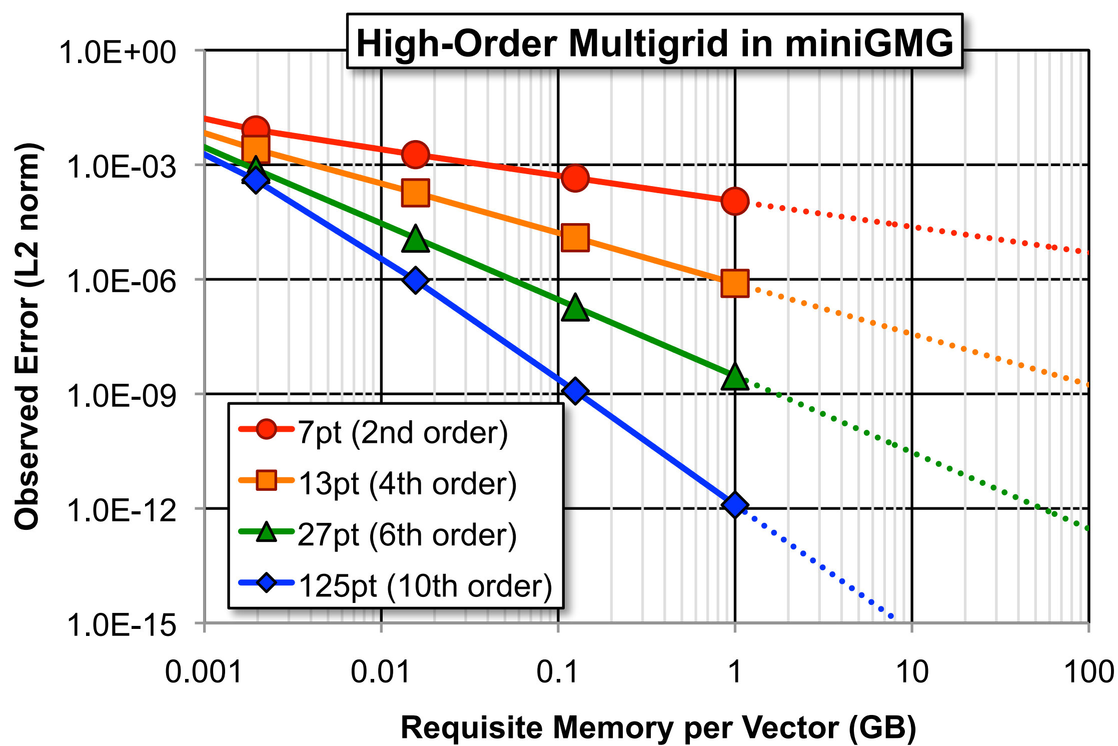


Figure 5.12: Error attained in miniGMG as a function of operator and grid size. A 1GB vector represents a single 512^3 grid. Multigrid will require several of these grids. Observe that the tenth-order method delivers a three-digit increase in accuracy for every $8\times$ increase in memory.

numerical error, using the Mehrstellen correction with these discretizations reduces total data movement by *several orders of magnitude.*

5.4.9 Distributed Memory Results

Distributed memory experiments were run to illustrate that the performance benefit of CHiLL-generated code is not lost with weak scaling. Figure 5.13 presents multigrid solver time to solution using the 27-point constant coefficient operator and the matrix-free BiCGStab bottom solver when weak-scaled from one compute node to 1331 nodes (31,944 cores) on Edison and Hopper, with each node assigned a 256^3 domain partitioned into 64^3 boxes. Each NUMA node runs an MPI process of either 8 threads (Edison) or 6 threads (Hopper). Observe that the CHiLL compiler successfully provides a speedup of 1.27x–1.45x on Edison and 1.54x–1.70x on Hopper over our baseline code at scale.

The increase in time-to-solution in this MG+Krylov hybrid solver is attributable to a few factors. First, MPI send/recv message time increases with scale. Second, BiCGStab is not an O(N) algorithm, and thus the bottom solver requires more and more iterations as the problem scales. Third, the performance of MPI AllReduce degrades with scale despite the high-performance Aries interconnect. Thus the net performance benefit of CHiLL is amortized by these two inefficiencies. It should be noted that at large scales, 8 V-Cycles were required to converge, while at small scales, 9 V-Cycles were required (8 V-Cycles barely missed the convergence criterion), thus amortizing the impact of the degradation in network performance.

5.5 Conclusions

High-order discretizations of the Laplacian often result in compute-intensive stencils that perform more than an order of magnitude more floating-point operations per point than the traditional second-order discretization. The paradigm shift from compute-limited architectures to bandwidth-limited architectures has revitalized interest in these methods. In this chapter, we explored several novel augmentations to the CHiLL compiler designed to improve the computational performance of these stencils. Using the miniGMG multigrid benchmark, we showed that the compiler could nearly quadruple the performance of the 125-point Jacobi smooth on Edison

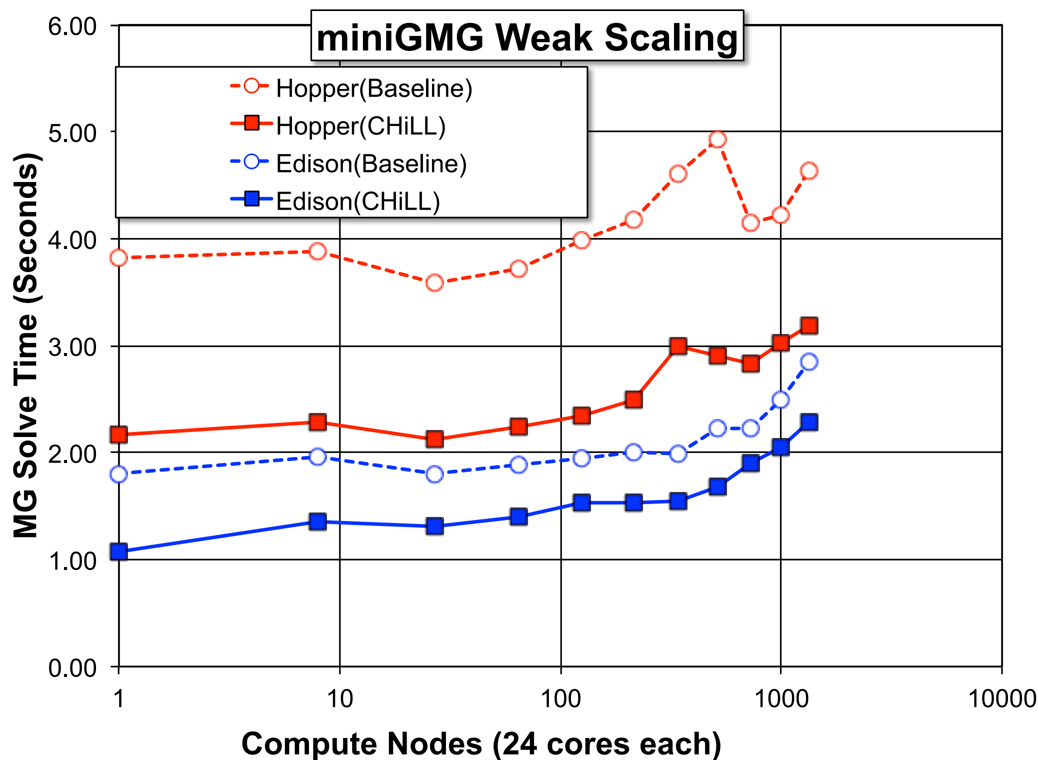


Figure 5.13: The performance benefit of CHiLL is not lost as one weak scales miniGMG with a 27-point stencil to 31,944 cores (1331 nodes). The above figure illustrates weak-scaled miniGMG time to solution using either the baseline code or the CHiLL compiler. Four weighted Jacobi relaxations are used at each level, with BiCGStab as the bottom solver, and a 256^3 domain per node.

and reach the Roofline performance bound. Moreover, we showed the compiler could tune optimizations independently by platform and multigrid level.

To highlight the true potential of high-order methods, we examined the reduction in error as we increase resolution. We show that our tenth-order method using the compact 125-point stencil in conjunction with a Mehrstellen correction attains tenth-order accuracy and provides a $1000\times$ increase in accuracy for every $8\times$ increase in memory. This has the tremendous potential of reducing total data movement and total energy (across a supercomputer) by many orders of magnitude compared to the second-order multigrid solver.

CHAPTER 6

GEOMETRIC MULTIGRID ON GPUs

Graphics processing unit (GPU) accelerators are a popular hardware target for scientific codes. This dissertation concentrates on programming NVIDIA GPUs using the CUDA prgoramming model [51]. The CUDA programming model exposes parallelism via a hierarchy of parallel threads. CUDA threads are grouped into thread blocks, and the thread blocks are arranged in a grid.

Unfortunately, to target CUDA, the original stencil codes written in C must be completely rewritten to map to the hierarchical parallelism exposed by CUDA. Furthermore, when mapping computation to CUDA threads, the programmer has to consider *memory coalescing* [51]. When consecutive CUDA threads access consecutive memory locations in memory, the accesses can be *coalesced* into a single access, improving memory bandwidth use.

Rewriting stencil codes to target GPUs places a significant programming burden on application scientists. Furthermore, several parallel implementations of the same computation need to be maintained, thus increasing the complexity of scientific applications. Ideally, the programmer should be able to use automated tools to generate optimized and architecture-specialized stencil codes from a single naive implementation.

To address the productivity challenges highlighted above, research presented in this dissertation uses CHiLL and CUDA-CHiLL. CUDA-CHiLL extends code generation capabilities of CHiLL to generated CUDA code, and enables exploring different strategies for mapping code to CUDA threads and blocks. By using CHiLL or CUDA-CHiLL, OpenMP or CUDA code can be generated from the same naive sequential input. This chapter demonstrates that CUDA-CHiLL improves programmer productivity without sacrificing performance. Code generated by CUDA-CHiLL

matched code written by expert programmers and achieves 80% of the Roofline performance bound.

6.1 Gauss-Seidel Red-Black (GSRB) Smooth on GPUs

In this chapter we focus on using CUDA-CHiLL to generate high-performance GSRB smooth (Listing 3.2). GSRB smooth uses a 7-point variable coefficient stencil and places very high demands on memory bandwidth. In similar fashion to the previous chapters we optimize smooth in context of the *miniGMG* benchmark.

In this chapter we run miniGMG on a single GPU card. Similar to the problem size on a multicore node, we use a 256^3 domain on a single GPU. At the finest level of the V-cycle, the domain is split into a list of 64 boxes of size 64^3 each. This discussion focuses on automatic high-performance CUDA code generation from C, and does not perform communication-avoiding optimizations, such as wavefronts.

6.1.1 Strategy for Parallel Decomposition of Smooth

The parallelism in smooth is mapped to the two-level parallelism hierarchy in CUDA, which is expressed in terms of a grid of thread blocks. As illustrated in Figure 6.1, smooth is implemented in miniGMG using a 3D grid of 2D thread blocks.

The rationale behind 2D thread blocks is that the 2D blocks are used to tile the `i` and `j` dimensions of each box (`i` is the dimension of unit stride), and the `k` dimension is not tiled (`k` dimension has the largest stride). Not tiling the dimension of maximum stride is known as Rivera Tiling [4], and has been shown to improve the performance of 3D scientific codes. A dimension of the 3D grid is used to map to each box or subdomain, the other two dimensions of the 3D grid control the number of thread blocks working on each box. Using a 3D grid allows exploring a larger space of parallel decompositions, as the 3D grid can easily be reduced to a 2D grid by setting one of the two dimensions of the grid to unity (the third dimension of the grid is always set to the number of boxes).

To make the explanation concrete, we use an example thread decomposition of a 256^3 domain which is split into a list of 64, 64^3 subdomains. The 3D CUDA grid of thread blocks has dimensions {BX=2 (dimension `i`), BY= 4 (dimension `j`), BZ=64 (number of boxes)}. Each 2D thread block in this grid has dimensions

{TX=32(dimension `i`), TY= 16 (dimension `j`)}. The BZ dimension of the grid maps to boxes (subdomains), each `ij`-plane {BX=2, BY= 4} of thread blocks in the grids is assigned to a single 64^3 box or subdomain. Thus, there are eight thread blocks working inside a box, and 64 {BZ=64} planes of 2D thread blocks, with each plane working on box. This block and grid arrangement is shown in Figure 6.1.

Figure 6.2 illustrates how the 2D CUDA thread blocks process each 64^3 box. The thread blocks encompass the `ij`-planes of the box, but not the `k`-dimension. In terms of memory layout, `i` is the fastest-changing dimension and `k` the slowest. As mentioned earlier, in this decomposition, the dimension `k` is not tiled. Thus, each CUDA thread computes 64 output grid points (in the `k` dimension), and each thread block computes 32(TX)*16(TY)*64 = 32,768 output points. The optimized smooth in miniGMG uses a 3D grid of dimension {BX=2, BY= 16, BZ=64}, and 2D thread blocks {TX=32, TY= 4} [52].

6.2 Code Generation with CUDA-CHiLL

CUDA-CHiLL parallelizes a sequential C implementation of GSRB smooth and generates parallel CUDA code. The input C code for smooth is shown in Listing 6.1. To be able to effectively parallelize the computation inside each box and across boxes, our input code has a four-deep loop nest, and we start with the three statements fused into one loop nest. The outermost loop iterates through all the boxes in the domain. Furthermore, our arrays (grids) are four-dimensional, reflecting the outermost `box` loop.

To correctly accommodate four-dimensional array references, the grid creation and allocation routines in miniGMG benchmark had to be modified. Each 64^3 subdomain (box) has a number of associated 66^3 grids. Thus for the entire domain (256^3), each *grid*: `phi`, `beta_i`, `beta_j`, `beta_k`, `rhs`, `lambda`, `alpha`, is a list of 66^3 grids. In the miniGMG benchmark, each 66^3 grid is allocated a contiguous chunk of memory, but consecutive grids for a component, e.g., `phi`, are not contiguous in memory. The memory allocation was changed such that each grid is still a contiguous chunk of memory, but grids for each component are also contiguous.

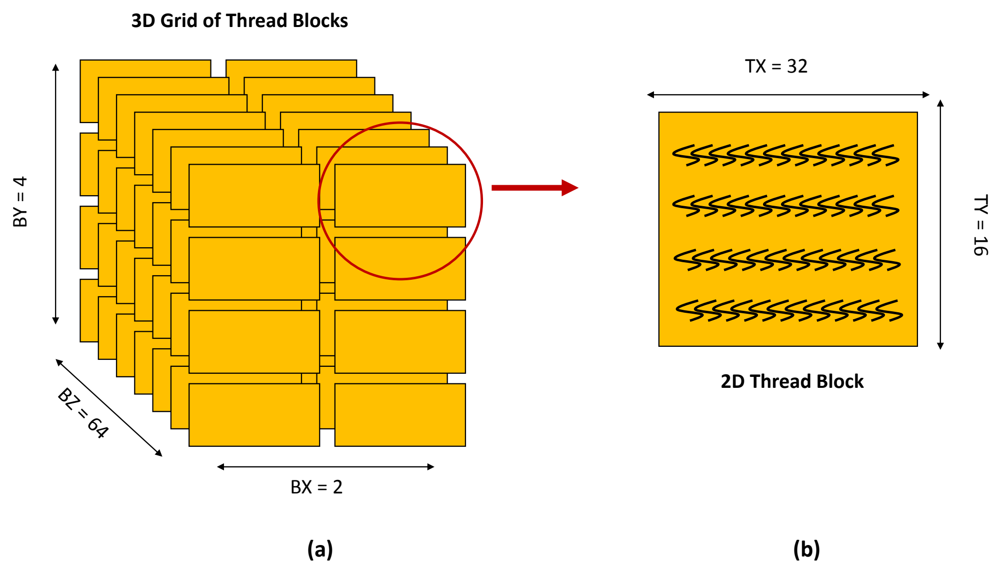


Figure 6.1: Organization of CUDA threads into a 3D (2,4,64) grid with 2D (32,16) thread blocks. Each 2D (X,Y) plane in the 3D grid has 8 2D thread blocks of dimension (32,16). Each 2D plane in the 3D grid works on a single subdomain (box) in miniGMG, and there are 64 such planes to process 64 subdomains.

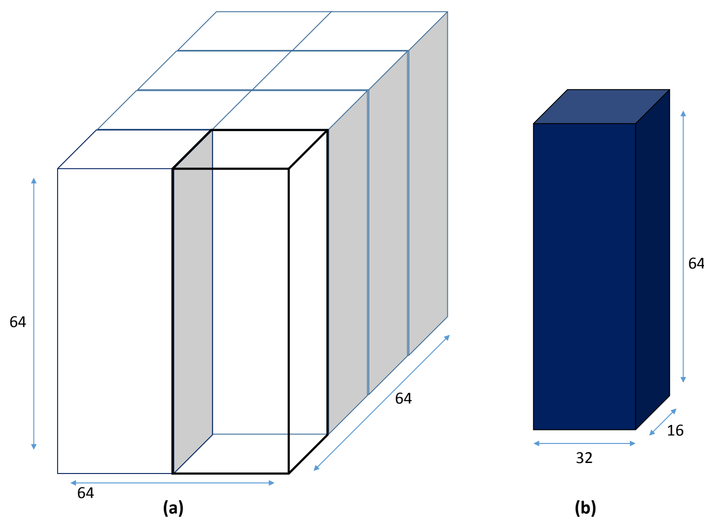


Figure 6.2: Illustration of work done by each thread block. Figure (a) shows 8, (32,16) thread blocks working on a 64^3 subdomain (box), with each thread computing a column of 64 output points. The blue column in figure (b) represents the 32*16*64 = 32,768 output grid points computed by each thread block.

```
#define PR_SIZE 64
#define NUM_BOXS 64

void smooth_GSRB(double a, double b, double h, int sweep){

double _phi[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
double _rhs[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
double _alpha[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
double _beta_i[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
double _beta_j[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
double _beta_k[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
double _lambda[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];

double h2inv = 1.0/(h*h);
int i,j,k;
int box;
int color = sweep;
double _t;

for(box=0; box<NUM_BOXS; box++){
 for(k=1; k<=PR_SIZE; k++){
  for(j=1; j<=PR_SIZE; j++){
   for(i=1; i<=PR_SIZE; i++){
   if(( i+ j + k + (color) ) % 2 == 1 ) {

 _t =  b*h2inv*(
  _beta_i[box][k][j][i+1] *( _phi[box][k][j][i+1]-_phi[box][k][j][i] )
  -_beta_i[box][k][j][i]  *( _phi[box][k][j][i]-_phi[box][k][j][i-1] )
  +_beta_j[box][k][j+1][i]*( _phi[box][k][j+1][i]-_phi[box][k][j][i] )
  -_beta_j[box][k][j][i]  *( _phi[box][k][j][i]-_phi[box][k][j-1][i] )
  +_beta_k[box][k+1][j][i]*( _phi[box][k+1][j][i]-_phi[box][k][j][i] )
  -_beta_k[box][k][j][i]*( _phi[box][k][j][i]-_phi[box][k-1][j][i] ));

 _t =  a*_alpha[box][k][j][i]*_phi[box][k][j][i] - _t;

 _phi[box][k][j][i] = _phi[box][k][j][i] -
          _lambda[box][k][j][i]*(_t -_rhs[box][k][j][i]);

  }}}}}
}
```

Listing 6.1: Input C code for GSRB smooth with 4D arrays

6.2.1 Mapping to Blocks and Threads

The parallelization strategy for miniGMG used by expert programmers described in Section 6.1.1 is also used to generate CUDA code with CUDA-CHiLL. CUDA-CHiLL is used to tile the input four-deep loop nest to create two more loop levels. To create a 3D grid with 2D thread blocks, three out of these six loops are then mapped to blocks in the 3D grid, and two loop levels are mapped to threads in a 2D block.

Listing 6.2 illustrates the script[1] that drives CUDA-CHiLL to generate a CUDA kernel for GSRB. The `tile` commands in line 12 of the script tile the loops in the input loop nest. The first argument to tile is the statement number (CUDA-CHiLL treats the entire loop body as a single statement). The next argument is the set of input loops to be tiled. This is followed by the tile sizes, the names of the tile controlling loops, and the final order of resulting tiled and tile controlling loops.

Listing 6.3 shows the structure of the loop nest after tiling. The tile controlling loop `bb` and `kk` have a single iteration and are not generated by CUDA-CHiLL. This effectively means that loops `box`, and `k` have not been tiled. The `cudaize` command in line 17 of Listing 6.2 then marks the candidate loops for block and thread dimensions. Loops `box`, `jj` and `ii` are marked as dimensions of the 3D grid. Loops `i`, and `j` are marked as the dimensions of the 2D thread blocks. During CUDA code generation the loops marked as dimensions for grids and blocks are removed, and array references to those loop indices are replaced with block indices (`bx, by, bz`) or thread indices (`tx, ty`).

6.2.2 Extensions to Code Generation in CUDA-CHiLL

CUDA-CHiLL was built on top of CHiLL to generate CUDA code. It uses CHiLL to apply loop transformations (such as tiling), and then leverages the code generation capabilities of Codegen+. Once Codegen+ scans the polyhedra representing the iteration space of the loop nest, it creates an intermediate representation or abstract syntax tree (AST) representation of the ouptut code. CUDA-CHiLL works on this AST and modifies it to generate CUDA code by mapping loops to thread and block indices. In the current implementation, only normalized loops can be mapped to a

[1]Section 2.9 introduces CUDA-CHiLL scripts written in the Lua scripting language.

```
1  init("gsrb.cu", "gsrb",0,0)
2  dofile("cudaize.lua")
3
4  --tile size
5  N=64
6  TI=32
7  TJ=16
8  TK=64
9  TZ=64
10 --end tile sizes
11
12 tile_by_index(0,
13   {"box","k","j", "i"},{TZ,TK, TJ, TI},
14   {l1_control="bb", l2_control="kk", l3_control="jj", l4_control="ii"},
15   {"bb","box","kk","k","jj","j","ii","i"})
16
17 cudaize(0, "kernel_GPU",{},
18   {block={"ii","jj","box"},
19    thread={"i","j"}},{})
```

Listing 6.2: CUDA-CHiLL Lua script for GSRB smooth.

```
1  // ~cuda~ preferredIdx: bz
2  for(box = 0; box <= 63; box++) {
3
4   // ~cuda~ preferredIdx: k
5    for(k = 1; k <= 64; k++) {
6
7   // ~cuda~ preferredIdx: by
8      for(jj = 0; jj <= 3; jj++) {
9
10   // ~cuda~ preferredIdx: ty
11       for(j = 0; j <= 15; j++) {
12
13     // ~cuda~ threadLoop preferredIdx: bx
14         for(ii = 0; ii <= 1; ii++) {
15
16      // ~cuda~ preferredIdx: tx
17          for(i = intMod(-j-k-color-1,2); i <= 31; i += 2) {
18
19             S0();
20
21  }}}}}}
```

Listing 6.3: Tiled code with candidate loops for CUDA blocks and threads.

thread/block index, which means candidate loops must have a lower bound of 0, and a unit stride.

The complex if-condition involving a modulo operation in GSRB presents a code generation challenge to CUDA-CHiLL. This is because the if-condition which guards the statement in GSRB gets fused into the innermost for-loop once Codegen+ creates an output. This can be seen in the innermost loop in Listing 6.3, which has a complex conditional with a modulo condition as a lower bound, and the loop has strided access. Thus, the loop is not normalized, and cannot be mapped to a thread or block index. To remedy the situation, CUDA-CHiLL was extended to handle such strided loops. The modulo condition will be removed from the lower bounds of the loop, and the loop stride will be reduced to one. The if-condition with the modulo would be pushed back into the loop body by creating a new AST node for the if-condition and wrapping the statement in the loop inside the AST node. This change checks to see if the stride of the loop matches the right hand side of the modulo in the if-condition. The output CUDA code in Listing 6.4 illustrates how the if-condition gets pushed back into the body of the loop. In this dissertation, CUDA-CHiLL was also extended to support creation of 3D grids introduced in CUDA 4.0.

6.2.3 Space of Generated GSRB Variants

A simple autotuner was written in Python to generate CUDA-CHiLL scripts to create variants of GSRB smooth. The generated code variants were then run in the miniGMG framework. Code variants were created by varying the dimensions of the 2D thread block. If the size of a thread block is <TX,TY>, the dimension of the 3D grid is <64/TX, 64/TY, 64>, where each suddomain is 64^3. The variants were created by varying TX from 8 to 64, and TY from 4 to 64 such that TX*TY is at least 32 (warp size), and TX*TY $\leq$ 1024 (maximum number of threads per block).

6.3 Experimental Results

As done in previous chapters, GSRB smooth was optimized in the context of miniGMG. However, miniGMG was completely rewritten to target GPUs [52]. This section first describes miniGMG on GPUs, then presents the experimental methodology used, and finally presents Roofline bounds and performance results.

```
1  __global__ void __smooth_GSRB( double a, double b, double h, int sweep){
2
3  double _phi[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
4  double _rhs[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
5  double _alpha[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
6  double _beta_i[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
7  double _beta_j[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
8  double _beta_k[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
9  double _lambda[NUM_BOXS][PR_SIZE+2][PR_SIZE + 2][PR_SIZE + 2];
10
11 int color = sweep;
12 double h2inv = 1.0/(h*h);
13 int k;
14 int by; by = blockIdx.y;
15 int bz; bz = blockIdx.z;
16 int tx; tx = threadIdx.x;
17 int ty; ty = threadIdx.y;
18 double _t;
19
20 for (k = 1; k <= 64; k += 1)
21  if ((tx - (-ty - k - color - 1)) % 2 == 0) {
22
23  _t = b * h2inv * (_beta_i[bz][k][ty + 16 * by + 1][tx + 1 + 1] *
24    (_phi[bz][k][ty + 16 * by + 1][tx + 1 + 1]
25    -_phi[bz][k][ty + 16 * by + 1][tx + 1])
26    - _beta_i[bz][k][ty + 16 * by + 1][tx + 1]
27    * (_phi[bz][k][ty + 16 * by + 1][tx + 1]
28    - _phi[bz][k][ty + 16 * by + 1][tx + 1 - 1])
29    + _beta_j[bz][k][ty + 16 * by + 1 + 1][tx + 1]
30    *(_phi[bz][k][ty + 16 * by + 1 + 1][tx + 1]
31    - _phi[bz][k][ty + 16 * by + 1][tx + 1])
32    - _beta_j[bz][k][ty + 16 * by + 1][tx + 1]
33    * _phi[bz][k][ty + 16 * by + 1][tx + 1]
34    - _phi[bz][k][ty + 16 * by + 1 - 1][tx + 1])
35    + _beta_k[bz][k + 1][ty + 16 * by + 1][tx + 1]
36    *(_phi[bz][k + 1][ty + 16 * by + 1][tx + 1]
37    - _phi[bz][k][ty + 16 * by + 1][tx + 1])
38    - _beta_k[bz][k][ty + 16 * by + 1][tx + 1]
39    * (_phi[bz][k][ty + 16 * by + 1][tx + 1]
40    - _phi[bz][k - 1][ty + 16 * by + 1][tx + 1]));
41
42  _t = a * _alpha[bz][k][ty + 16 * by + 1][tx + 1]
43  * _phi[bz][k][ty + 16 * by + 1][tx + 1] - _t;
44
45  _phi[bz][k][ty + 16 * by + 1][tx + 1] = _phi[bz][k][ty + 16 * by + 1][tx +
       1]
46    - _lambda[bz][k][ty + 16 * by + 1][tx + 1]
47    * (_t - _rhs[bz][k][ty + 16 * by + 1][tx + 1]);}
48 }
```

Listing 6.4: Generated CUDA code for GSRB smooth. The generated code is for 2D thread blocks (TX=64, TY=16) and 3D grid (BX=1, BY=4,BZ=64).

6.3.1 miniGMG on GPUs

The GPU (device) is connected to the CPU (host) via the PCIe bus. Because GPU and CPU have separate memories, data transfer between the two chips has to be over the PCIe bus, which has far lower bandwidth than DRAM bandwidths. To avoid data transfer over the PCIe bus, the grid creation routines in miniGMG were modified to allocate data directly on the GPU. The grid allocation on the GPU was similar to the CPU implementation. Instead of allocating single large grids for the entire domain, the domain was decomposed into subdomains or boxes, which were were allocated as contiguous chunks of memory.

In addition to the grids, buffers for ghost zone exchanges were also allocated directly on the GPU device. Routines to initialize grids, exchange ghost zone data and the operations smooth, residual, restrict, and interpolations were all rewritten in CUDA to reflect the allocation of grids on the device.

6.3.2 Experimental Methodology

The miniGMG benchmark with optimized GSRB smooth was run on a single NVIDIA Kepler K20c card. The relevant details of the card/chip are listed in Table 6.1. The problem solved on the GPU is described in Section 4.6.1.

The miniGMG benchmark is used to solve a 256^3 domain on a single GPU node. The domain is decomposed into a list of 64^3 subdomains at the finest level of the GMG V-cycle. The experiments were performed by running 480 iterations of GSRB smooth (240 Red, 240 Black iterations) on the finest grids. Figures 6.3 and 6.4 show the performance of code generated by CUDA-CHiLL. Execution times of smooth

Table 6.1: Overview of evaluated NVIDIA GPU.

NVIDIA Kepler K20c	
Shared Multiprocessor (SMX) per Chip	13
CUDA cores per SMX	192
CUDA cores per Chip	2496
DP GFlop/s	1170
Global Memory	5 GB
Effective DRAM Bandwidth (SHOC)	147.77 GB/s
Shared Memory + L1 cache per SMX	64KB
Thread Private Registers per SMX	256KB

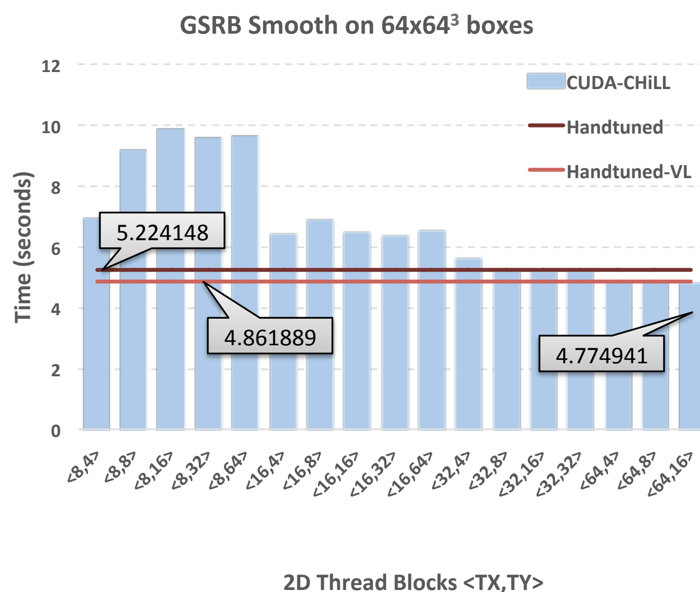


Figure 6.3: Execution times for 480 iterations of GSRB (240 Red, 240 Black) smooth on 64^3 boxes. Lower bar means better performance. The horizontal lines corresponds to performance of manually tuned codes. The x-axis are the 2D thread block sizes (TX,TY). The grid corresponding to each thread block is (64/TX, 64/TY, 64). Best performance was achieved with thread block (64, 16) and grid (1,4,64).

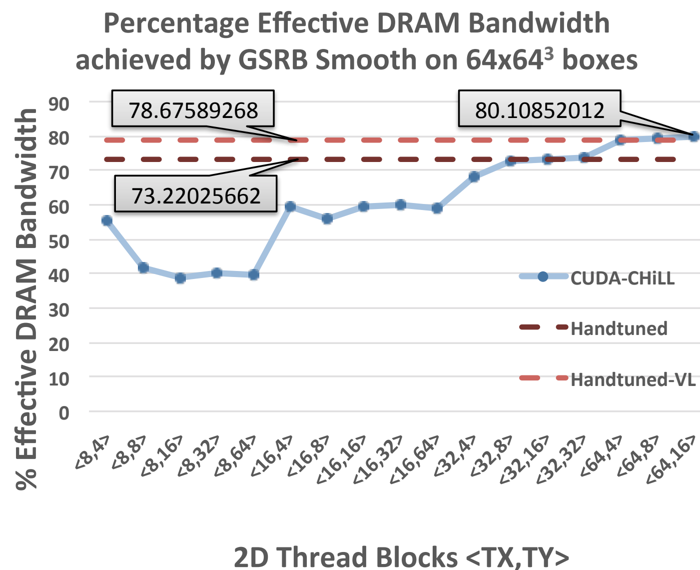


Figure 6.4: The fraction of the effective DRAM bandwidth achieved by GSRB smooth. Higher values correspond to better performance.

generated by CUDA-CHiLL and handtuned smooth are plotted in Figure 6.3. Figure 6.4 plots the percentage of effective DRAM bandwidth achieved by generated and handtuned codes.

6.3.3 Optimizations in Handtuned Code

The manually tuned code for GSRB smooth [52] is highly optimized. There are three variants of handtuned code: the first, labeled "Handtuned" in the plots, relies on caching only, while the second relies on explicit use of shared memory. The third variant, labeled Handtuned-VL, achieved the best performance, but required extensive code restructuring. The expert programmer collapsed the `i` and `j` loops, and then divided the iterations of the collapsed loop across (448 wide) vectors.

The expert programmer also searched thread and block configurations, and selected (BX=2, BY=16, BZ=64) as dimensions of the 3D grid, and (TX=32, TY=4) as the block size. All manually tuned versions replaced the if-condition in GSRB with a ternary operator. Use of shared memory for smooth degraded performance and is not discussed further.

6.3.4 Performance of Smooth

Figure 6.3 plots the execution time for 480 iterations of GSRB smooth. The generated code is able to slightly better the performance achieved by the highly tuned, manually written code variants. Interestingly, autotuning using CUDA-CHiLL selects a different thread and block configuration than manual tuning as the best performing code variant. The performance trend highlighted in Figure 6.3 indicates that memory coalescing was the most important factor in achieving performance. Thus, having thread blocks with larger `i` dimensions improved performance.

To get a better understand of the performance of this memory-bandwidth-bound computation, the percentages of DRAM bandwidth achieved by the manually tuned and generated codes are plotted in Figure 6.4. The effective bandwidth achieved is calculated as the data moved by 480 iterations of GSRB divided by execution time. The SHOC benchmark measures the effective DRAM bandwidth (with ECC turned on) achieved on this chip as 147.77 GB/s [53]. Data moved is computed in similar fashion to Section 4.6.5.1, and is computed as:

Data Moved = iterations * boxes * grid size * arrays (R+W) * size of double

The number of iterations is 480, there are 64 boxes or subdomains, grid size including ghost zones in 66^3. There are 7 arrays read and 1 written, thus 8 arrays, and 8 bytes for each double precision number. This formula assumes ideal DRAM traffic, accounting for only compulsory cache misses. Based on the above formula and observed execution times, we see that the generated CUDA-CHiLL kernel achieves a very high fraction, i.e., 80%, of the effective DRAM bandwidth on the chip.

6.4 Conclusions

Research presented in this dissertation uses CUDA-CHiLL to address the productivity and performance challenges of optimizing stencil computations on GPUs. CUDA-CHiLL is used to generate high-performance CUDA codes for GSRB smooth from a simple sequential C input. Autotuning is used to drive code generation to create a space of possible CUDA thread and block decompositions.

This chapter demonstrates that the CUDA-CHiLL-generated GSRB smooth achieves impressive performance. The generated code betters the performance of highly tuned manually written code, and achieves 80% of the performance bound computed by the Roofline model. The generated code does not resort to low-level optimizations, instead, CUDA-CHiLL achieves high performance by searching a richer space of parallel decompositions. This clearly shows the power of autotuning used in combination with code generation to generate high-performance stencil codes.

This chapter presents performance results for optimized smooth for the finest box sizes. Tuning smooth for all levels of V-cycle and generating and optimizing CUDA code for residual and restriction is left as future work. Furthermore, an important observation regarding the read-only arrays in the generated CUDA code smooth was that the native CUDA compiler (nvcc) was not able to map the read-only arrays: $beta_i$, $beta_j$,..., to the read-only data caches on the GPU. Future work can also look at using CUDA-CHiLL to explicitly map read-only arrays to the read-only data caches.

CHAPTER 7

RELATED WORK

Due to their importance in scientific computation, stencil computations have a rich history of both manual and automated optimization techniques. This chapter briefly describes past and current research in optimizations for stencil codes. The description of related work is divided into four categories. 1) First, we describe optimizations to reduce data movement. We look at past techniques to reduce capacity misses, followed by more current optimizations to improve memory bandwidth use, and trade redundant computation for reduced message passing. 2) Next, we look at stencil reordering optimizations. These optimizations generally target compute-intensive stencils, but in this dissertation they have been used for both compute- and bandwidth-limited stencils. 3) Most optimization efforts targeting stencils usually focus on constant-coefficient stencils. There are a few exceptions. This section describes research efforts that optimize solvers, and not just stencils in isolation. 4) The final sections describe both automated and manual optimizations targeting stencils on GPUs.

7.1 Optimizations to Reduce Data Movement

In the past, operations on large structured grids could easily be bound by capacity misses in cache, leading to a variety of studies on blocking and tiling optimizations [3, 4, 5, 6, 7, 8, 9]. However, a number of factors have made such approaches progressively obsolete on modern platforms. On-chip caches have grown by orders of magnitude and are increasingly able to capture sufficient locality for the fixed box sizes associated with typical MG methods. The rapid increase in on-chip parallelism has also quickly out-stripped available DRAM bandwidth resulting in bandwidth-bound performance.

Thus, in recent years, numerous efforts have focused on increasing temporal locality by fusing multiple stencil sweeps through techniques like cache-oblivious, time-

skewing, wavefront, or overlapped tiling [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Most of these efforts have been concentrated on 2D stencils, which are not as common as 3D stencils in scientific computing, and almost none of them have targeted variable-coefficient stencils. Variable-coefficient stencils require much larger volumes of data movement, and hence need more aggressive bandwidth optimizations. Furthermore, most research optimizes out-of-place stencil sweeps such as Jacobi iterations. Optimizing GSRB is less common, but one exception is the related work from Treibig. They implement a 2D GSRB on SIMD architectures by separating and reordering the red and black elements [54], additionally a 3D multigrid on an IA-64 (Itanium) is implemented via temporal blocking.

The most closely related work consists of domain-specific compilers for parallel code generation from a stylized stencil specification [27, 28, 29] or from a code excerpt [30]. Pochoir uses a cache-oblivious strategy, which limits the control over the code generation [28]. The other compilers introduce parallelism and ghost zones through tiling and expanding both the data set and the tile size, rather than starting with already parallel code [30, 29]. These tiling approaches do not produce the hyper-trapezoidal loop nests presented in this dissertation, but rather compute and then ignore some incorrect results. None of these approaches appear capable of supporting the optimization of a collection of operators, particularly if GSRB is included.

7.2 Stencil Reordering Optimizations

The novel transformation described in Chapter 5 reorders stencil computation to exploit reuse and reduce floating-point operations. A similar transformation, array common subexpression elimination [25], was built into the ZPL compiler. The next subsection details the difference between these two approaches. This is followed by a description of other stencil-reordering techniques, both manual and automated.

7.2.1 Comparison with Array Common Subexpression Elimination

A compiler formulation of a related reordering transformation called array common subexpression elimination was described in [25]. Array common subexpression elimination is implemented using an abstraction called a *tablet*, which records the

structure of the stencil inputs and their coefficients. To capture reuse, redundancy conditions, heuristics and benefit functions are used to generate *subtablets* of the tablet. The subtablets are used to compute partial sums which are reused via scalar temporary variables. This method of exploiting reuse through the subtablets is more complex but more general than the partial sums method. The subtablet construction allows for reuse of points other than a plane and enables handling of multistatement stencils. However, exploiting reuse in scalar registers introduces a scalar dependence across loop iterations and inhibits instruction-level and SIMD parallelization by native backend compilers.

In contrast, the partial sums approach always picks the *leading plane* of points, and it buffers the computed partial results in vectors. Picking the leading plane and explicitly looking for symmetry to reduce redundant computation is simpler than using heuristics, redundancy conditions, and cost functions to discover reuse and symmetry from subtablets. Further, partial sums avoids introducing dependences and generates code easily vectorized by the native backend compiler.

7.2.2 Other Stencil Reordering Techniques

Manual optimization of stencil computations have developed techniques such as *semi-stencils* to reduce loads [24], and using array common subexpression elimination after unrolling to reduce floating-point computations [12, 5]. In [12, 5], the authors unroll the loops of the stencil computation to expose array common subexpressions and reorder computation to reduce floating-point operations using a stencil-specific code generator. Our approach is automated, and does not rely on unrolling.

Polyhedral techniques for reconfigurable computing construct custom storage structures to exploit reuse [55], but are limited to reuse between consecutive iterations and do not consider higher-order stencils.

Recent work reorders stencil computations from the direct specification of the stencil as an update from a set of input points [26]. Stencils are converted to an accumulation, and then loop shifting exposes register reuse of the same input, contributing to different output. Their approach does not reduce floating-point computations.

Though data layout transformation (DLT) [56] is not a stencil reordering transformation, it was developed to improve SIMD code generation for stencil codes, and thus improve floating-point performance. DLT was designed to eliminate the data stream alignment problem in generating SIMD code for stencil computations. DLT transposes stencil inputs so that multiple computations can be performed in SIMD registers without the costly shifting of data across iterations of the innermost loop. Partial sums access aligned planes and buffer them in separate arrays; thus, we have addressed stream alignment without requiring the data transpose.

7.3 Optimizations for Solvers

Most efforts in automatic optimization of stencil computations have concentrated on isolated stencil computations, where a stencil operator is applied to a structured grid. Far less attention has been paid to optimizing a solver, where there are multiple stencils and more data movement involved. In the recent past Olschanowsky et al. used a semiautomatic approach [57] to optimize a PDE solver. The domain-specific language Halide [31], which has generally focused on optimizing image processing pipeline, has also been used to optimize miniGMG. Finally, Chan et al. used autotuning to improve precision and performance of multigrid by switching algorithms at different levels (grid resolutions) of the solver [58].

Olschanowsky et al. [57] study block-structured AMR code, where the domain has been decomposed into subdomains or boxes. This paper uses loop shifting and fusion with wavefront techniques to reduce data movement. They, however, do not take into account the stencil computation being applied multiple times, and hence do not consider data movement incurred when ghost zone exchange take place. They also use inter- and intra-box threading, but do not use nested parallelism.

When optimizing a complex proxy application such as miniGMG, we have to optimize several stencil operators and how they work together. To the best of our knowledge, the DSL Halide [31] which focuses on image processing pipelines is the only other auomated framework to do so. Halide is now being considered to optimize scientific stencil kernels as well. It has recently been used to optimize miniGMG and HP-GMG. Research from Halide has not considered distributed memory experiments,

communication-avoiding optimizations such as increasing ghost zones, and the use of higher-order stencils.

Chan et al. explored how, using an autotuned approach, one could restructure the MG V-cycle (to switch from an iterative to direct method depending on the MG level) to improve time-to-solution in the context of a 2D, constant-coefficient Laplacian [58]. This approach uses autotuning to choose algorithms for each level of the multigrid. This technique is orthogonal to our implemented optimizations and can be incorporated in future work.

7.4 Stencil Computations on GPUs

There has been a significant body of work on optimizing stencil computations to target GPUs. This section divides the related work into two broad categories: manually optimized stencil codes and domain-specific languages and tools. These two techniques are further divided into optimization efforts that only optimize single grid sweeps, and those that use wavefront or temporal tiling (blocking) optimizations to fuse multiple grid sweeps into one.

7.4.1 Manual Optimization Efforts

Initial work on manual or semiautomatic optimizations for stencils was done by Datta et al. [12]. They achieved an unprecedented 36 Gflops (Double Precision) on an NVIDA GTX 280 card. Their work optimized constant-coefficient stencils, and did not use wavefront optimizations on the GPU.

Micikevicius [59] manually optimized higher-order stencil (orders 6 to 12) in isolation and in a solver. These higher-order stencils are shaped like the 13-point stencil in Chapter 5. This optimization effort used shared memory in addition to the other optimizations presented in [12].

Nguyen et al. [11] explored using larger ghosts and temporal tiling in an approach they termed 3.5D-tiling. They were not able to improve performance for 3D stencils using these approaches on GPUs. This was because their technique increased redundant computation, and the older GPUs (GTX285) became compute bound easily.

Recent work from Maruyama and Aoki [60] manually optimizes a 3D, 7-point, constant-coefficient stencil. They used a number of optimization techniques, including

use of shared memory with warp specialization, and exploiting read-only caches using a compiler intrinsic. They also used temporal tiling (similar to the wavefront computation discussed in Chapter 4). In fact, this is the only research effort to date that has successfully used temporal tiling with 3D stencils. They ran their optimized code on the NVIDIA Kepler K20x. On this architecture they achieve around 80% of the Roofline performance. With temporal tiling, they are able to achieve 20% further improvement in performance. The CUDA code generated by CUDA-CHiLL was run on an NVIDIA K20c GPU. The K20c and K20x are very similar GPUs, with k20x being more powerful with an additional SMX and higher memory bandwidth. On the K20c GPU, the generated code also achieves 80% of the Roofline performance.

7.4.2 Domain-Specific Automated Optimization Efforts

There have been many domain-specific approaches to optimizing stencils on GPU accelerators. These efforts can be classified as programming language extensions to target GPUs, domain-specific languages that optimize stencil computations for both GPUs and multicores, and finally, code generators for stencils on GPU.

Mint [61] is a programming language extension for stencils on a GPU. It lets the programmer optimize stencils on a GPU by decorating code with pragmas. The generated code was slightly slower than hand-tuned code, and did not explore temporal tiling.

Most domain-specific languages [62, 27, 31] for stencil computations target both multicores and GPUs. Of these, [62, 27] do not support shared memory or temporal tiling or large ghost zones on GPUs. Halide [31], as discussed earlier, is a mature DSL which is designed primarily for image processing pipeline. It has been used to optimize miniGMG on GPUs, but their details have not been published.

Stencil-specific code generators have been used to generate and autotune stencil code on GPUs [30, 29]. These techniques target shared memory. Temporal tiling and overlapped tiling are used in [30], but these techniques are shown to work only with 2D-stencils.

7.5 Summary and Conclusions

In summary, research presented in this dissertation optimizes a wide spectrum of stencil computations of varying arithmetic intensities, and it optimizes these stencils in isolation and in the context of a solver. This is in contrast to most research published on stencils which does not look at solvers. Furthermore, this dissertation focuses on developing compiler transformations that allow composing sequences of compiler transformations. CHiLL's ability to compose transformations allows it to take a higher-order smooth operator, remove its computation bottleneck with partial sums and make it bandwidth-limited, and then further apply DRAM bandwidth-reducing optimizations. No prior work has combined reducing floating-point operations with communication-avoiding optimization for higher-order stencils. Furthermore, no prior research in automated tools has used stencil reordering to enable loop fusion as was done in Section 4.4.3. Converting stencils to an accumulation is a form of stencil reordering, and it enables fusion of the residual and restriction operations.

On modern GPUs, we are able to achieve performance comparable to highly tuned stencil codes when we do not account for temporal tiling. Temporal tiling or wavefront is a powerful technique, and is important future work for both miniGMG and CUDA-CHill. As shown by [60], in the future, as GPUs become more powerful and can support redundant computation, temporal tiling will be feasible for 3D stencils. It must be also be noted that, with the exception of research presented here, and the two domain-specific tools, Halide [31] and Mint [61], no other research effort has optimized variable-coefficient stencils on GPUs.

CHAPTER 8

FUTURE RESEARCH AND CONCLUSIONS

This dissertation presents research to address the programmer productivity and performance challenges of stencil computations. The optimization challenge posed by the wide variety of stencil computations and the increasing complexity of computer architectures places a huge burden on domain scientists. Compiler frameworks and autotuners can greatly reduce programming effort by optimizing and mapping an architecture-agnostic implementation to different platforms. The research presented in this dissertation aims to free application scientists from architecture-specific code tuning by using autotuners which leverage domain-specific compiler optimizations.

8.1 Contributions

The benefit of compiler-based autotuning has been demonstrated in my research by optimizing stencils and Geometric Multigrid. Novel domain-specific compiler transformations for communication-avoiding and optimizations for higher-order stencils were developed and built into the CHiLL compiler framework. The novel optimizations implemented in CHiLL were designed to work with existing loop transformations such as loop tiling, permutation, and unrolling. The autotuner uses CHiLL to generate code that is run in the miniGMG framework. MiniGMG is a mini-app developed by Williams et al. to proxy adaptive mesh refinement (AMR). The compiler framework has been extended to generate optimized OpenMP code to target multicores, and CUDA for GPUs. My research on compiler-directed autotuning has been published in [63], and autotuning and code generation for GPUs has appeared in [37].

Communication-avoiding optimizations reduce horizontal communication (between processors) and vertical communication (between a processor and its memory hierarchy) by generating overlapped ghost zones and parallel wavefront, respectively. These optimizations improve the performance of stencil computations with low arithmetic

intensity. Higher-order stencils require smaller grids and thus incur lower data movement, but these stencils have increased floating-point computations, thus are compute limited. Optimizations for higher-order stencils reduce the number of floating-point operations and improve register reuse. Communication-avoiding optimization and parallel code generation for multicores appears in [64, 65], and the novel optimization for higher-order stencils is introduced in [66]. The effectiveness of compiler-directed autotuning is illustrated in [66], as the optimization for compute-intensive stencils is composed with communication-avoiding optimizations and parallel code generation to tune higher-order solvers.

8.1.1 Optimizing Memory Bandwidth Limited Stencils

Stencil computations and variable coefficient stencils, in particular, are traditionally memory-bandwidth bound with a low arithmetic intensity. With data movement costs dwarfing computation, reducing communication and effective use of the memory hierarchy is critical for high performance. To reduce horizontal communication we introduced a novel domain-specific compiler transformation which creates overlapped ghost zones (overlapped tiling). Vertical communication was addressed by creating a parallel wavefront computation to reduce DRAM traffic. Wavefront generation requires many levels of loop tiling, skewing, and permutation. Parallel code generation was then used to reduce working set and improve load balance.

Using these communication-avoiding optimizations, our autotuner generated a tuned parallel wavefront with (over 4x) speedups over code generated by the state-of-the-art *icc* compiler and matched the performance of highly tuned code. The generated code is further optimized by fusing smooth, residual, and restriction operators and generating a nested parallel wavefront. To enable such aggressive fusion, a novel compiler transformation which converts a stencil computation into an accumulation was required. The wavefront generated used even deeper ghost zones, further reducing communication and taking performance beyond that of manually tuned code.

8.1.2 Optimizing Compute-Intensive Higher-Order Stencils

Higher-order stencils have high arithmetic intensity, hence their performance is limited by increased floating-point operations and poor register reuse. A novel trans-

formation called *partial sums* was designed to optimize higher-order stencils. This transformation aims to remove the bottlenecks of high arithmetic intensity and poor register reuse and achieve performance corresponding to the memory bandwidth. Partial Sums reorders stencils to improve register use and improve SIMDization. It computes sums of a subset of array points accessed and buffers their reuse of following computations. Symmetry in the stencil coefficients (when present) is exploited to reduce flops. Once the transformed stencil computation is memory bound, communication-avoiding optimization can be then applied to further optimize the code.

To test the efficacy of higher-order stencils and *partial sums*, sixth- and tenth-order (27- and 125-point stencils respectively) smooths were implemented in miniGMG. Unlike previous efforts to optimize higher-order stencils, which only considered flops or register reuse, compiler-based autotuning allowed us to explore optimizations to address both the computation and memory bottlenecks. Generated tuned code achieved high performance and relied on considerable interplay between Partial Sums, communication-avoiding optimizations, and parallel code generation. Optimized codes showed speedups over 4x for smooths and up to 3x improvements for the solver. This translates to optimized tenth-order smooth running at half the speed of optimized second-order smooth, but giving many orders of magnitude better accuracy.

8.1.3 Parallel Code Generation

The efficient mapping of computation to parallel processing elements on a node is critical in achieving high performance on current and emerging architectures. Unfortunately, writing high-performance parallel code is very challenging, and not performance portable across architectures. To improve productivity without sacrificing the performance of manually optimized codes, this dissertation uses CHiLL to generate high-performance parallel code targeting modern multicores and GPUs from the same sequential input. CHiLL was used to generate optimized stencil codes using either OpenMP or CUDA, from the same naive sequential C input.

To target multicores, the code generation capabilities of CHiLL are extended to generate OpenMP code. OpenMP code generation in CHiLL supports two types of

parallelism: flat and nested parallel OpenMP. Nested parallel OpenMP is used to manage the working set and improve load balance.

This disseration uses CUDA-CHiLL to target NVIDIA GPUs. CUDA-CHiLL is a thin layer built on top of CHiLL to generate CUDA code. This dissertation improves CUDA-CHiLL to enable handling complex loop bounds present in Gauss-Seidel Red-Black (GSRB) smooth. CUDA-CHiLL is then used to generate CUDA code variants with different parallel decomposition, and autotuning is used to pick the best variant. The CUDA-CHiLL-generated code outperforms manually tuned code from experts, and achieves 80% of the computation's Roofline performance bound.

8.2 Future Work

Even though optimizing stencil computations is a mature research area[1], the growing diversity in architectures and stencils, combined with a lack of code-optimizing tools, still places a huge optimization burden on the programmer (domain scientist). This dissertation attempts to remedy this situation by using compiler optimizations, both novel and known, combined with autotuning to improve programmer productivity and code performance. The promising results shown in this dissertation can be extended to address other challenges common to computations on structured grids. This section briefly describes a few research areas where compiler optimizations and autotuning can be developed to benefit stencil computations.

8.2.1 Nonperiodic Boundary Conditions

All the stencil computations that are optimized in this dissertations have periodic boundary conditions. Unfortunately, stencil computations that are used in scientific applications such as AMR frequently have complex nonperiodic boundary conditions. To optimize such real-world scientific codes, the optimizations presented in this thesis must be extended to handle complex boundary conditions.

Periodic boundary conditions simplified application of communication-avoiding optimizations. For example, a wavefront computation involves fusing multiple smooths (stencil sweeps of a grid) into one grid sweep. This was done using the compiler

[1]The earliest stencil specific compiler I came across in my research was the Connection Machine convolution compiler [67] from 1991.

transformations skew and permute. In the case of nonperiodic boundary conditions, the smooth is preceded by a grid sweep to update the boundaries. Creating a communication-avoiding wavefront computation may then involve creating a wavefront of smooths interleaved with boundary updates. Generating such a complex wavefront will require a wide variety of loop transformations, and new domain-specific techniques may be required to enable aggressive loop fusion.

8.2.2 Target Emerging Many-Core Architectures

In addition to GPUs, the emergence of many-core nodes from Intel, such as the Xeon Phi, will necessitate revisiting code optimizations for stencil computations. Targeting the Intel many-core architecture will require managing the larger number of hardware threads, and using the wide SIMD units efficiently.

Single instruction multiple data (SIMD) instructions work on multiple data elements simultaneously, thus can improve the performance of floating-point operations by integer factors. As the SIMD units on many-core are wider than traditional multicores, the efficient use of the SIMD units on Xeon Phi will be critical to performance. The vectorization of GSRB codes will have to be investigated, as native compilers have not been able to vectorize loops with strides. New compiler transformations may be necessary to improve the performance of GSRB on these machines. Furthermore, code generation for higher-order stencils on the Xeon Phi will be challenging. Thus, the effectiveness of Partial Sums and SIMD code generation has to be revisited for the wider SIMD units. In addition to wide SIMD units, generated code has to use the fine-grained parallelism of many-core architectures efficiently.

8.2.3 Code Generation for Emerging Runtimes

Optimizing stencil computations to target the increaing parallelism on future architectures will require code generation to leverage emerging runtimes. To efficiently use the increasing number of hardware threads, code generation can explore task-based parallelism in runtimes such as Open Community Runtime (OCR) [68] and Habanero [69]. In addition, code generation can use these runtimes to express point-to-point communication, which is currently not supported in OpenMP. The lack of point-to-point communication hindered the implementation of a threaded wavefront

in Chapter 4, and forced the compiler to generate low-level spinlocks. Using emerging runtime to address such drawbacks and to effectively map code to parallel hardware will be critical to generating high-performance stencil codes on futures architectures.

8.3 Conclusion

Optimizing stencil computations on modern machines presents a daunting challenge to application programmers. To improve productivity and performance, this dissertation presents a compiler-directed autotuning technique specialized for stencils. This approach allows us to exploit domain-specific knowledge via novel stencil-specific optimization, while letting us leverage decades of compiler research. Autotuning then creates sequences of optimizations, which then drive the compiler to generate code variants. Finally the best variant is selected for the given execution context. This ability to compose and apply sequences of optimizations, combined with autotuning, will be critical to making code optimization tools that can target complex real-world scientific applications.

APPENDIX

STENCILS

The following code snippets illustrate the constant-coefficient stencils used in Chapter 5.

A.1 Seven Point Stencil

```
1  #define ALPHA (-6.0)
2  #define BETA (1.0)
3
4  _out[k][j][i] = ALPHA * _in[k][j][i]
5                 +  BETA * ( _in[k][j+1][i] + _in[k][j-1][i]
6                 +  _in[k-1][j][i] + _in[k+1][j][i]
7                 +  _in[k][j][i-1] + _in[k][j][i+1]  );
```

Listing A.1: 7-point stencil.

A.2 Thirteen Point Stencil

```
1  #define ALPHA (-90.0/12.0)
2  #define BETA (16.0/12.0)
3  #define GAMMA (-1.0/12.0)
4
5  _out[k][j][i] = ALPHA * _in[k][j][i]
6
7                 +  BETA * (     _in[k-1][j][i] + _in[k+1][j][i] +
8                 + _in[k][j-1][i] + _in[k][j+1][i]+
9                 + _in[k][j][i-1] + _in[k][j][i+1])
10                + GAMMA * (     _in[k-2][j][i] + _in[k+2][j][i] +
11                + _in[k][j-2][i] + _in[k][j+2][i]+
12                + _in[k][j][i-2] + _in[k][j][i+2]);
```

Listing A.2: 13-point stencil.

A.3 Twenty Seven Point Stencil

```
1  #define ALPHA (-128.0/30.0)
2  #define BETA (14.0/30.0)
3  #define GAMMA (3.0/30.0)
4  #define DELTA (1.0/30.0)
5
6  _out[k][j][i] = ALPHA * _in[k][j][i]
7      +  BETA * (         _in[k-1][j][i] + _in[k][j-1][i]
8          + _in[k][j+1][i] + _in[k+1][j][i]
9          + _in[k][j][i-1] + _in[k][j][i+1])
10         + GAMMA * (     _in[k-1][j][i-1] + _in[k][j-1][i-1]
11         + _in[k][j+1][i-1] + _in[k+1][j][i-1]
12         + _in[k-1][j-1][i] + _in[k-1][j+1][i]
13         + _in[k+1][j-1][i] + _in[k+1][j+1][i]
14         + _in[k-1][j][i+1] + _in[k][j-1][i+1]
15         + _in[k][j+1][i+1] + _in[k+1][j][i+1])
16         + DELTA * (     _in[k-1][j-1][i-1] + _in[k-1][j+1][i-1]
17         + _in[k+1][j-1][i-1] + _in[k+1][j+1][i-1]
18         + _in[k-1][j-1][i+1] + _in[k-1][j+1][i+1]
19         + _in[k+1][j-1][i+1] + _in[k+1][j+1][i+1]);
```

Listing A.3: 27-point stencil.

A.4 Hundred Twenty Five Point Stencil

```
double c0= -1.0/7560;
double c1= 2.0/14175 ;
double c2 = -11.0/16200;
double c3= 4.0/2025;
double c4= -16.0/14175;
double c5= -11.0/2100;
double c6= 64.0/1575;
double c7= 256.0/2835;
double c8 = 776.0/1575;
double c9= -6848.0/1575;

_out[k][j][i] = c9 * _in[k][j][i]+
        +c8 * ( _in[k-1][j+0][i+0]+     _in[k+0][j-1][i+0]+
                        _in[k+0][j+0][i-1]+     _in[k+0][j+0][i+1]+
                        _in[k+0][j+1][i+0]+     _in[k+1][j+0][i+0])
    +c5 * ( _in[k-2][j+0][i+0]+ _in[k+0][j-2][i+0]+
                        _in[k+0][j+0][i-2]+ _in[k+0][j+0][i+2]+
                        _in[k+0][j+2][i+0]+ _in[k+2][j+0][i+0])
    +c3 * ( _in[k-2][j-1][i+0]+ _in[k-2][j+0][i-1]+
                        _in[k-2][j+0][i+1]+ _in[k-2][j+1][i+0]+
                        _in[k-1][j-2][i+0]+     _in[k-1][j+0][i-2]+
                        _in[k-1][j+0][i+2]+ _in[k-1][j+2][i+0]+
                        _in[k+0][j-2][i-1]+ _in[k+0][j-2][i+1]+
                        _in[k+0][j-1][i-2]+ _in[k+0][j-1][i+2]+
                        _in[k+0][j+1][i-2]+ _in[k+0][j+1][i+2]+
                        _in[k+0][j+2][i-1]+ _in[k+0][j+2][i+1]+
                        _in[k+1][j-2][i+0]+ _in[k+1][j+0][i-2]+
                        _in[k+1][j+0][i+2]+ _in[k+1][j+2][i+0]+
                        _in[k+2][j-1][i+0]+ _in[k+2][j+0][i-1]+
                        _in[k+2][j+0][i+1]+ _in[k+2][j+1][i+0] )
    +c7 * (     _in[k-1][j-1][i+0]+ _in[k-1][j+0][i-1]+
                        _in[k-1][j+0][i+1]+ _in[k-1][j+1][i+0]+
                        _in[k+0][j-1][i-1]+ _in[k+0][j-1][i+1]+
                        _in[k+0][j+1][i-1]+ _in[k+0][j+1][i+1]+
                        _in[k+1][j-1][i+0]+ _in[k+1][j+0][i-1]+
                        _in[k+1][j+0][i+1]+ _in[k+1][j+1][i+0] )
    +c2 * ( _in[k-2][j-2][i+0]+ _in[k-2][j+0][i-2]+
                        _in[k-2][j+0][i+2]+ _in[k-2][j+2][i+0]+
                        _in[k+0][j-2][i-2]+ _in[k+0][j-2][i+2]+
                        _in[k+0][j+2][i-2]+ _in[k+0][j+2][i+2]+
                        _in[k+2][j-2][i+0]+ _in[k+2][j+0][i-2]+
                        _in[k+2][j+0][i+2]+ _in[k+2][j+2][i+0])
    +c6 * ( _in[k-1][j-1][i-1]+ _in[k-1][j-1][i+1]+
                        _in[k-1][j+1][i-1]+ _in[k-1][j+1][i+1]+
                        _in[k+1][j-1][i-1]+ _in[k+1][j-1][i+1]+
                        _in[k+1][j+1][i-1]+ _in[k+1][j+1][i+1] )
        +c4 * ( _in[k-2][j-1][i-1]+ _in[k-2][j-1][i+1]+
                        _in[k-2][j+1][i-1]+ _in[k-2][j+1][i+1]+
                        _in[k-1][j-2][i-1]+ _in[k-1][j-2][i+1]+
                        _in[k-1][j-1][i-2]+ _in[k-1][j-1][i+2]+
                        _in[k-1][j+1][i-2]+ _in[k-1][j+1][i+2]+
                        _in[k-1][j+2][i-1]+ _in[k-1][j+2][i+1]+
                        _in[k+1][j-2][i-1]+ _in[k+1][j-2][i+1]+
                        _in[k+1][j-1][i-2]+ _in[k+1][j-1][i+2]+
                        _in[k+1][j+1][i-2]+ _in[k+1][j+1][i+2]+
                        _in[k+1][j+2][i-1]+ _in[k+1][j+2][i+1]+
                        _in[k+2][j-1][i-1]+ _in[k+2][j-1][i+1]+
                        _in[k+2][j+1][i-1]+ _in[k+2][j+1][i+1] )
   +c1 * (  _in[k-2][j-2][i-1]+ _in[k-2][j-2][i+1]+
                        _in[k-2][j-1][i-2]+ _in[k-2][j-1][i+2]+
                        _in[k-2][j+1][i-2]+ _in[k-2][j+1][i+2]+
                        _in[k-2][j+2][i-1]+ _in[k-2][j+2][i+1]+
                        _in[k-1][j-2][i-2]+ _in[k-1][j-2][i+2]+
                        _in[k-1][j+2][i-2]+ _in[k-1][j+2][i+2]+
                        _in[k+1][j-2][i-2]+ _in[k+1][j-2][i+2]+
                        _in[k+1][j+2][i-2]+ _in[k+1][j+2][i+2]+
                        _in[k+2][j-2][i-1]+ _in[k+2][j-2][i+1]+
                        _in[k+2][j-1][i-2]+ _in[k+2][j-1][i+2]+
                        _in[k+2][j+1][i-2]+ _in[k+2][j+1][i+2]+
                        _in[k+2][j+2][i-1]+ _in[k+2][j+2][i+1] )
        +c0 * ( _in[k-2][j-2][i-2]+     _in[k-2][j-2][i+2]+
               _in[k-2][j+2][i-2]+      _in[k-2][j+2][i+2]+
                        _in[k+2][j-2][i-2]+     _in[k+2][j-2][i+2]+
                        _in[k+2][j+2][i-2]+     _in[k+2][j+2][i+2] );
```

Listing A.4: 125-point stencil.

REFERENCES

[1] K. Asanovic, R. Bodik, B. Catanzaro *et al.*, "The landscape of parallel computing research: A view from Berkeley," EECS, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[2] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial.* Society for Industrial and Applied Mathematics, 2000.

[3] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, 2004.

[4] G. Rivera and C. Tseng, "Tiling optimizations for 3D scientific computations," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2000.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2008.

[6] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice Boltzmann simulation optimization on leading multicore platforms," in *Proc. Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, 2008.

[7] S. Williams, L. Oliker, J. Carter, and J. Shalf, "Extracting ultra-scale lattice boltzmann performance via hierarchical and distributed auto-tuning," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[8] M. Kowarschik and C. Weiß, "Dimepack - a cache-optimized multigrid library," in *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2001.

[9] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, "Cache optimization for structured and unstructured grid multigrid," *Elect. Trans. Numer. Anal*, vol. 10, pp. 21–40, 2000.

[10] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the Cell processor for scientific computing," in *Proc. Conference on Computing Frontiers*, 2006.

[11] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[12] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review*, vol. 51, no. 1, pp. 129–159, 2009.

[13] M. Frigo and V. Strumpen, "Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations," in *Proc. ACM International Conference on Supercomputing (ICS)*, 2005.

[14] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[15] D. Wonnacott, "Using time skewing to eliminate idle time due to memory bandwidth and network limitations," in *Proc. Interational Conference on Parallel and Distributed Computing Systems*, 2000.

[16] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," Department of Computer Science, Rutgers University, Tech. Rep. DCS-TR-379, 1999.

[17] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *Proc. International Computer Software and Applications Conference*, 2009.

[18] T. Zeiser, G. Wellein, A. Nitsure, K. Iglberger, U. Rude, and G. Hager, "Introducing a parallel cache oblivious blocking approach for the lattice Boltzmann method," *Progress in Computational Fluid Dynamics*, vol. 8, 2008.

[19] P. Ghysels, P. Kłosiewicz, and W. Vanroose, "Improving the arithmetic intensity of multigrid with the help of polynomial smoothers," *Numerical Linear Algebra with Applications*, vol. 19, no. 2, pp. 253–267, 2012.

[20] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2012.

[21] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2007.

[22] L. Collatz, *The numerical treatment of differential equations.* Springer, 1960.

[23] P. Colella, C. Kavoulklis, P. McCorquodale, and B. V. Straalen, "A high-performance implementation of the method of local corrections for constant-coefficient elliptic pde," unpublished.

[24] R. De La Cruz, M. Araya-Polo, and J. M. Cela, "Introducing the semi-stencil algorithm," in *Proc. International Conference on Parallel Processing and Applied Mathematics (PPAM).*

[25] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Eliminating redundancies in sum-of-product array computations," in *Proc. International Conference on Supercomputing (ICS)*, 2001.

[26] K. Stock, M. Kong, T. Grosser, F. Rastello, L.-N. Pouchet, and J. R. P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[27] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[28] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proc. ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2011.

[29] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2012.

[30] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proc. ACM international conference on Supercomputing (ICS)*, 2012.

[31] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[32] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Optimization of geometric multigrid for emerging multi- and manycore processors," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[33] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," University of Southern California, Technical Report 08-897, Jun 2008.

[34] M. W. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Proc. International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Oct 2009.

[35] B. C. Applied Numerical Algorithms Group (ANAG), Lawrence Berkeley National Laboratory, "Chombo website," [Online], Available: http://seesar.lbl.gov/ANAG/software.html.

[36] Compiler Technologies to Optimize Performance (CTOP), University of Utah, Salt Lake City, UT, "Chill website," [Online], Available: http://ctop.cs.utah.edu/ctop.

[37] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013.

[38] W. Pugh, "The omega test: A fast and practical integer programming algorithm for dependence analysis," in *Proc. ACM/IEEE Conference on Supercomputing (SC)*, 1991.

[39] C. Chen, "Polyhedra scanning revisited," in *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2012.

[40] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kauffman, 2002.

[41] G. Kreisel and J. L. Krivine, *Elements of Mathematical Logic.* North Holland Publishing Company, 1967.

[42] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, 1991.

[43] M. Wolfe, "More iteration space tiling," in *Proc. Conference on Supercomputing (SC)*, 1989.

[44] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988.

[45] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in *Proc. International Conference on Supercomputing (ICS)*, 1992.

[46] Compiler Technologies to Optimize Performance, University of Utah, "Chill website," [Online], Available: http://ctop.cs.utah.edu/ctop.

[47] National Energy Research Scientific Computing Center (NERSC), "Edison website," [Online], Available: http://www.nersc.gov/users/computational-systems/edison.

[48] ——, "Hopper website," [Online], Available: http://www.nersc.gov/users/computational-systems/hopper.

[49] S. Williams, A. Watterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the ACM*, April 2009.

[50] Q. Zhang, H. Johansen, and P. Colella, "A fourth-order accurate finite-volume method with structured adaptive mesh refinement for solving the advection-diffusion equation," *SIAM J. Sci. Comput.*, vol. 34, no. 2, 2012.

[51] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers Inc., 2010.

[52] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Optimization of geometric multigrid for emerging multi- and manycore processors," Lawrence Berkeley National Laboratory, Technical Report 6676E, 20012.

[53] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proc. Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[54] J. Treibig, "Efficiency improvements of iterative numerical algorithms on modern architectures," Ph.D. dissertation, University Erlangen, 2008.

[55] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013.

[56] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Proc. International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC/ETAPS)*, 2011.

[57] C. Olschanowsky, M. M. Strout, S. Guzik, J. Loffeld, and J. Hittinger, "A study on balancing parallelism, data locality, and recomputation in existing pde solvers," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[58] C. Chan, J. Ansel, Y. L. Wong, S. Amarasinghe, and A. Edelman, "Autotuning multigrid with petabricks," in *Proc. Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[59] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proc. Workshop on General Purpose Processing on Graphics Processing Units*, 2009.

[60] N. Maruyama and T. Aoki, "Optimizing Stencil Computations for NVIDIA Kepler GPUs," in *Proc. International Workshop on High-Performance Stencil Computations*, 2014.

[61] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing cuda performance in 3d stencil methods with annotated c," in *Proc. International Conference on Supercomputing (ICS)*, 2011.

[62] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *Proc. Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.

[63] P. Basu, M. W. Hall, M. M. Khan, S. Maindola, S. Muralidharan, S. Ramalingam, A. Rivera, M. Shantharam, and A. Venkat, "Towards making autotuning mainstream," *International Journal of High Performance Computing Applications*, vol. 27, no. 4, pp. 379–393, 2013.

[64] P. Basu, S. Williams, A. Venkat, B. Van Straalen, M. Hall, and L. Oliker, "Compiler generation and autotuning of communication-avoiding operators for geometric multigrid," in *Proc. High Performance Computing Conference (HIPC)*, 2013.

[65] P. Basu, S. Williams, B. V. Straalen, L. Oliker, and M. Hall, "Converting stencils to accumulations for communication-avoiding optimization in geometric multigrid," in *Proc. Workshop on Stencil Computations (WOSC)*, 2014.

[66] P. Basu, M. W. Hall, S. Williams, B. van Straalen, L. Oliker, and P. Colella, "Compiler-directed transformation for higher-order stencils," in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[67] M. Bromley, S. Heller, T. McNerney, and G. L. Steele, Jr., "Fortran at ten gigaflops: The connection machine convolution compiler," *SIGPLAN Not.*, vol. 26, no. 6, May 1991.

[68] Open Community Runtime (OCR), "Open community runtime website," [Online], Available: https://01.org/open-community-runtime.

[69] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Tasirlar, Y. Yan, Y. Zhao, and V. Sarkar, "The habanero multicore software research project," in *Proc. ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009.