Auto-tuning Stencil Codes for Cache-Based Multicore Platforms

Kaushik Datta

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-177
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-177.html

December 17, 2009

Auto-tuning Stencil Codes for Cache-Based Multicore Platforms

by

Kaushik Datta

B.S. (Rutgers University) 1999
M.S. (University of California, Berkeley) 2005

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Katherine A. Yelick, Chair
Professor James Demmel
Professor Jon Wilkening

Fall 2009

The dissertation of Kaushik Datta is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Auto-tuning Stencil Codes for Cache-Based Multicore Platforms

Abstract

Auto-tuning Stencil Codes for Cache-Based Multicore Platforms

by

Kaushik Datta

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine A. Yelick, Chair

As clock frequencies have tapered off and the number of cores on a chip has taken off, the challenge of effectively utilizing these multicore systems has become increasingly important. However, the diversity of multicore machines in today's market compels us to individually tune for each platform. This is especially true for problems with low computational intensity, since the improvements in memory latency and bandwidth are much slower than those of computational rates.

One such kernel is a stencil, a regular nearest neighbor operation over the points in a structured grid. Stencils often arise from solving partial differential equations, which are found in almost every scientific discipline. In this thesis, we analyze three common three-dimensional stencils: the 7-point stencil, the 27-point stencil, and the Gauss-Seidel Red-Black Helmholtz kernel.

We examine the performance of these stencil codes over a spectrum of multicore architectures, including the Intel Clovertown, Intel Nehalem, AMD Barcelona, the highly-multithreaded Sun Victoria Falls, and the low power IBM Blue Gene/P. These platforms not only have significant variations in their core architectures, but also exhibit a 32× range in available hardware threads, a 4.5× range in attained DRAM bandwidth, and a 6.3× range in peak flop rates. Clearly, designing optimal code for such a diverse set of platforms represents a serious challenge.

Unfortunately, compilers alone do not achieve satisfactory stencil code performance on this varied set of platforms. Instead, we have created an automatic stencil code tuner, or *auto-tuner*, that incorporates several optimizations into a single software framework. These optimizations hide memory latency, account for non-uniform memory access times, reduce the volume of data transferred, and take advantage of

special instructions. The auto-tuner then searches over the space of optimizations, thereby allowing for much greater productivity than hand-tuning. The fully auto-tuned code runs up to 5.4× faster than a straightforward implementation and is more scalable across cores.

By using performance models to identify performance limits, we determined that our auto-tuner can achieve over 95% of the attainable performance for all three stencils in our study. This demonstrates that auto-tuning is an important technique for fully exploiting available multicore resources.

Professor Katherine A. Yelick
Dissertation Committee Chair

To Sorita and my family.

Contents

List of Figures

List of Tables

List of symbols

AMR	Adaptive Mesh Refinement
AST	Abstract Syntax Tree
ATLAS	Automatically Tuned Linear Algebra Software
BGP	Blue Gene/P
CFD	Computational Fluid Dynamics
CFL	Courant-Friedrichs-Lewy Stability Condition
CMT	Chip MultiThreading
CPU	Central Processing Unit
CSE	Common Subexpression Elimination
CSR	Compressed Sparse Row
DLP	Data-Level Parallelism
DP	Double Precision
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
ELL	Efficiency-Level Language
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FLAME	Formal Linear Algebra Method Environment
FLOP	Floating Point Operation
FMA	Fused Multiply-Add
FMM	Fast Multipole Method
FSB	Frontside Bus
GPGPU	General Purpose Graphics Processor Unit

GSRB	Gauss-Seidel Red Black
HPC	High Performance Computing
HT	HyperTransport
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
KCCA	Kernel Canonical Correlation Analysis
MCH	Memory Controller Hub
MCM	Multi-chip Module
NUMA	Non-Uniform Memory Access
OSKI	Optimized Sparse Kernel Interface
PDE	Partial Differential Equation
PERL	Practical Extraction and Report Language
PLL	Productivity-Level Language
QPI	QuickPath Interconnect
SEJITS	Selective Embedded Just-In-Time Specialization
SIMD	Single Instruction Multiple Data
SML	Statistical Machine Learning
SMP	Symmetric Multiprocessor
SMT	Simultaneous Multithreading
SPIRAL	Signal Processing Implementation Research for Adaptable Libraries
SVM	Support Vector Machines
TLB	Translation Lookaside Buffer
TLP	Thread-Level Parallelism
UMA	Uniform Memory Access
VF	Victoria Falls

Acknowledgments

First, I would like to thank my advisor, Kathy Yelick. From the struggles at the beginning of my graduate career until now, she has always supported me, and for that I am ever thankful. Her wonderful guidance and knowledge have led me to where I am now.

I owe an equally large debt of gratitude to Jim Demmel, whose incisive questions and attention to detail have always forced me to think deeply about the subject at hand. I hope to carry this mode of thinking into my future endeavors.

I also thank Jon Wilkening and Ras Bodik for serving on my qualifying exam committee, despite the fact that it was rendered almost useless by the impending and unforeseen multicore revolution. Well, unforeseen by me at least. I appreciate that Jon was on my thesis committtee as well.

Sam Williams has been one of the truly inspirational figures during the latter half of my graduate career. His combination of knowledge, diligence, enthusiasm, and humility is rare. I thank him for all his wonderful suggestions and for serving as an unofficial thesis reader. This thesis is that much better for it.

I also appreciate the friendship of Rajesh Nishtala, who has not only served as a peer with whom to discuss research, but also a friend who brought some much needed levity to our office. Ultra high school will have to live on without us.

I can't thank the members of the Bebop group enough, especially Shoaib Kamil, Karl Fuerlinger, and Mark Hoemmen. I have learned so much from all of you, and am constantly amazed by the level of the discussions and the quality of the research. I often wonder about how I entered the group.

The scientists at Lawrence Berkeley National Laboratory propelled my career by teaching me how to write high-quality papers quickly. Certainly, Lenny Oliker, John Shalf, and Jonathan Carter are masters of this craft, and I appreciate that they took me under their wing. Thanks also go to Terry Ligocki and Brian Van Straalen, who took time from their busy schedules to teach me the basics of AMR. I hope that my work points you in the right direction for tuning these codes for the manycore era.

The Berkeley Par Lab has also been instrumental to my achievements by providing me with a wider forum of people with whom to discuss parallel computing. The

feedback that I've received from other members, as well as at the retreats, has been invaluable. In particular, I would like to thank Jon Kuroda and Jeff Anderson-Lee for maintaining (and resurrecting) hardware so that I could collect the data in this thesis. I am sure they will be as happy as I am when I submit it. I also thank Archana Ganapathi, who introduced me to the now ubiquitous world of machine learning.

In a simpler world, when we talked about mega-flops instead of giga-flops, I was a member of the Berkeley Titanium project. This is where I was introduced to the wonderful world of high-performance computing and learned much about many of the topics in this thesis. While I am indebted to everyone in the group, special thanks go to Paul Hilfinger, Dan Bonachea, and Jimmy Su.

On a more personal front, I am forever grateful to my parents and my family for their continued love and support, despite the extended duration of my California "trip". Thank you for your patience.

Finally, I wish to thank Sorita. You, more than anyone, have spurred me to finish– even when the finish line looked like a dot. You are my love, my light, and my life.

I would like to thank the Argonne Leadership Computing Facility at Argonne National Laboratory for use of their Blue Gene/P cluster. That laboratory is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. I would like to also express my gratitude to Sun for its machine donations. Finally, thanks to Microsoft, Intel, and U.C. Discovery for providing funding (under Awards #024263, #024894, and #DIG07-10227, respectively) and usage of the Nehalem computer used in this study.

Chapter 1

Introduction

1.1 The Rise of Multicore Microprocessors

Until recently, the computer industry produced single core chips with ever-increasing clock rates. Furthermore, advances such as multiple instruction issue, deep pipelines, out-of-order execution, speculative execution, and prefetching also increased the throughput per clock cycle, but at the cost of building more complex and power-inefficient chips. However, these two trends continued to ensure that buying a new computer would result in better performance while preserving the sequential programming model [2].

As we see in Figure 1.1, these two trends stopped in 2004. A paradigm shift occurred in the microprocessor industry, as it was forced to completely redesign its chips due to heat density and power constraints. This *power wall* meant that the path toward even higher clock rate, more complicated single core chips was impeded. As a result, the industry retreated towards having multiple simpler, lower frequency cores on a chip. While these "multicore" chips were able to address the issues resulting from the power wall, the multiple cores forced the software industry to switch from a sequential to a parallel programming model. This was a drastic shift, and one that the industry is still grappling with today. However, now that explicit parallelism is being exposed in software, it has become easier to start thinking about the much higher degrees of parallelism that will be required in the approaching "manycore" era. The machines in the manycore realm will have hundreds to thousands of threads, but will

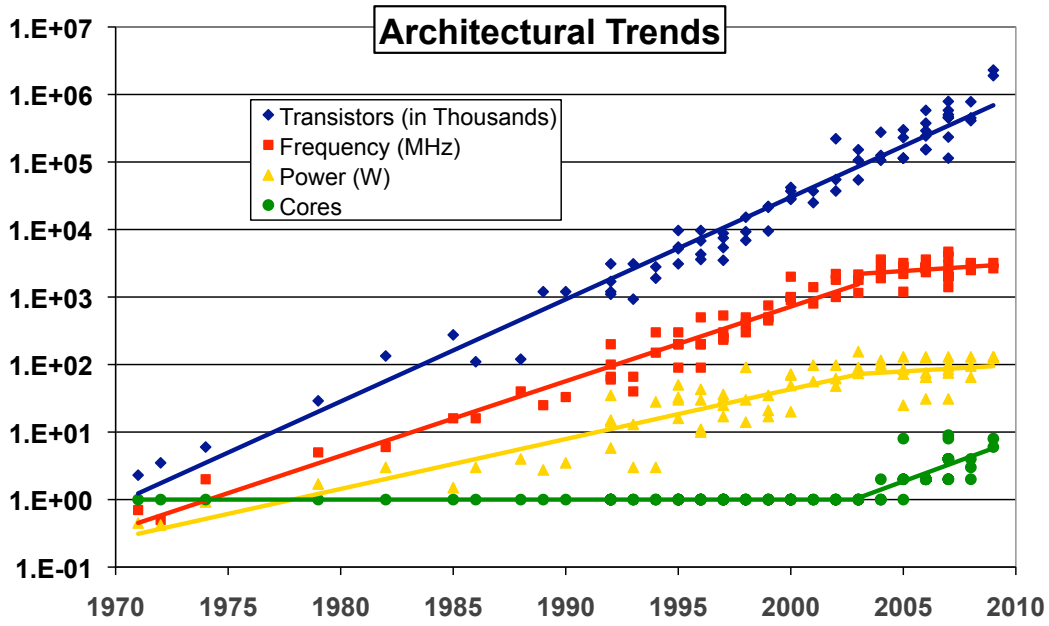


Figure 1.1: This graph shows that while the growth in transistors (and thus Moore's Law [36]) continues unabated, clock speed and power stabilized in 2004. This is the same time that multicore chips first started appearing. This data was collected by Herb Sutter, Kunle Olukotun, Christopher Batten, Krste Asanović, and Katherine Yelick.

utilize much simpler, lower-frequency cores to attain large power and performance benefits. Thus, we need to start designing codes now that can handle increasing amounts of parallelism [44].

1.2 Performance Portability Challenges

Unfortunately, merely writing explicitly parallel code is not enough. We will see in Chapters 7 and 8 that naïvely threaded programs, even for very simple kernels, fail to effectively utilize a multicore machine's resources. This can be largely attributed to the complexity of current hardware and software and the inability of general purpose compilers to handle that complexity. These compilers have two major limitations that cause codes to underperform. First, in order to hasten compilation time, they almost always rely on heuristics, not experiments, to determine the optimal values of needed parameters. We will see in Section 3.1 that today's multicore platforms are

sufficiently diverse and complex that heuristics no longer suffice. Second, software is often written with complex data and control structures so that these compilers also fail to infer the legal domain-specific transformations that are needed to produce the largest performance gains. However, this is to be expected, given some of the challenging algorithmic and data structure changes that may be required.

One solution to this problem is to hand-tune the relevant computational kernel for a given platform; this would allow the programmer to specify the domain-specific transformations that the compiler alone was unable to exploit. While this may work in the short term, as soon as the tuned code needs to be ported, possibly to a different architecture with more cores, it will likely need to be retuned. In Sections 7.5 and 8.4, we show that that the best code and runtime parameters for one multicore machine often do not correspond to good performance on other machines. In fact, a stencil code tuned for one platform, if ported to another platform without further tuning, may achieve less than half of the new platform's best performance. This assumes that there is enough parallelism in the code to keep all of the new platform's threads busy and load balanced. If this is not true, then the performance drops even further.

1.3 Auto-tuning

A better solution is to automatically tune, or *auto-tune*, the relevant kernel. This does require a significant one-time cost to the programmer, but once the auto-tuner is constructed, it shifts the burden of tuning from the programmer to the machine. As a result, it is highly portable. Moreover, it increases programmer productivity when the one-time cost to build the auto-tuner can be amortized by running across several different machines.

There are other advantages to auto-tuning. Auto-tuners can be designed to handle any core count, thereby making them scalable. This will be a valuable asset in the manycore era. Furthermore, auto-tuners can also be constructed for maximizing metrics other than performance (*e.g.* power efficiency), thus also making them flexible. Due to all of these reasons, auto-tuning has already had several previous success stories, including: FFTW [20], SPIRAL [41], OSKI [54], and ATLAS [55].

1.4 Thesis Contributions

In this thesis, we construct auto-tuners for codes that perform relatively simple, but common, nearest-neighbor, or *stencil*, computations. The following are our primary contributions.

- We have identified a set of new and known optimizations specifically for stencil codes, including: core blocking, thread blocking, register blocking, NUMA-aware data allocation, array padding, software prefetching, cache bypass, SIMDization, and common subexpression elimination. Collectively, they are designed to optimize the data allocation, memory bandwidth, and computational rate of stencil codes.

- Based on these optimizations, we have developed three stencil auto-tuners that can achieve substantially better performance than naïvely threaded stencil code across a wide variety of cache-based multicore machines. After full tuning, we realized speedups of 1.9×–5.4× over the performance of a naïvely threaded code for the 3D 7-point stencil, between 1.8×–3.8× for the 3D 27-point stencil, and between 1.3×–2.5× for the 3D Helmholtz kernel with a 2 GB fixed memory footprint.

- We have developed an "Optimized Stream" benchmark that uses many of the same optimizations as in the stencil auto-tuners to achieve the best possible streaming memory bandwidth from a given machine. It is able to achieve bandwidths between 0.3%–13% higher than those attained from the untuned "Stream" benchmark [35]. The Optimized Stream benchmark is also one piece of the Roofline model [58], and thus allows us to place a performance "roofline" on bandwidth-bound kernels.

- We have analyzed the performance of our auto-tuner against the bandwidth and computational limits of each machine. For the 7-point stencil, we are able to achieve between 74%–95% of the attainable bandwidth for the memory-bound machines, as well as 100% of the in-cache performance for the compute-bound IBM Blue Gene/P. For the 27-point stencil, we realized between 85%–100%

of the in-cache performance for the compute-bound architectures, 89% of the attainable bandwidth for the bandwidth-bound Intel Clovertown, and 65% of both attainable bandwidth and computation on the AMD Barcelona. Finally, for a single iteration of the Helmholtz kernel, we performed at between 89%–100% of the peak attainable bandwidth. Therefore, in almost all cases, we are able to effectively exploit either the bandwidth or computational resources of the machine.

- In Chapter 9, we no longer tuned a single large problem. Instead, we auto-tuned many small subproblems that mimicked the behavior of the Adaptive Mesh Refinement (AMR) code that this kernel was ported from. In order to tune these multiple subproblems well, we introduced the adjustable *threads per subproblem* parameter. Fewer threads per subproblem corresponded to coarse-grained parallelism, while more threads per subproblem meant fine-grained parallelism. We discovered that utilizing fewer threads per problem usually performed best, but also introduced load balancing issues. If future manycore architectures do not provide better support for fine-grained parallelism, load balancing will be an even larger issue than it is today.

1.5 Thesis Outline

The following is an outline of the thesis:

Chapter 2 gives an overall description of stencils, along with examples from some of the many applications from which they arise. Stencils are also described as a very specific, but efficient, form of sparse matrix-vector multiply.

Chapter 3 goes into depth about each of the cache-based multicore machines that were employed in this study. We also discuss some of the experimental details that are used in the later chapters, including the choice of compilers, the manner in which threads are assigned to cores, and how timing data is collected.

Chapter 4 introduces the stencil-specific optimizations that were used in this study. These optimizations are grouped into four rough categories: problem decomposition, data allocation, bandwidth optimizations, and in-core optimizations.

Chapter 5 subsequently explains how these optimizations were combined into a stencil auto-tuner, including details about code generation and parameter space searching.

Chapter 6 details how the Roofline model [58] provides performance bounds for stencil codes. The Roofline model is a general model that incorporates both bandwidth and computation limits into its performance predictions.

Chapters 7, 8, and 9 then discuss the tuning and resulting performance of the 7-point stencil, 27-point stencil, and the Helmholtz kernel, respectively. Together, these three stencils are representative of many real-world stencil codes. By understanding how we were able to achieve good performance for these stencils, similar techniques can be applied to many more stencil codes.

Chapters 10 discusses related and future work, and we conclude in chapter 11.

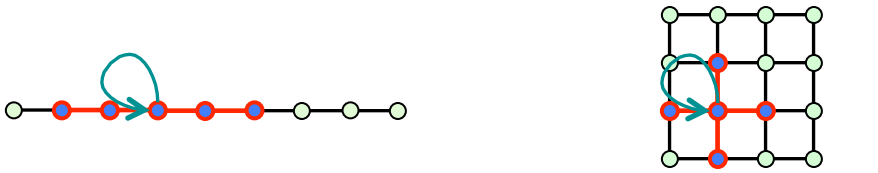
Chapter 2

Stencil Description

2.1 What are stencils?

Partial differential equation (PDE) solvers are employed by a large fraction of scientific applications in such diverse areas as diffusion, electromagnetics, and fluid dynamics. These applications are often implemented using iterative finite-difference techniques that sweep over a spatial grid, performing nearest neighbor computations called *stencils*. In a stencil operation, each point in a regular grid is updated with weighted contributions from a subset of its neighbors in both time and space– thereby representing the coefficients of the PDE for that data element. These coefficients may be the same at every grid point (a constant coefficient stencil) or not (a variable coefficient stencil). Stencil operations are often used to build solvers that range from simple Jacobi iterations to complex multigrid [6] and adaptive mesh refinement (AMR) methods [3].

Stencil calculations perform sweeps through data structures that are typically much larger than the capacity of the available data caches. In addition, the amount of data reuse is limited to the number of points in the stencil, which is typically small. The upshot is many (but not all) of that these computations achieve a low fraction of theoretical peak performance, since data from main memory cannot be transferred fast enough to avoid stalling the computational units on modern microprocessors. Reorganizing these stencil calculations to take full advantage of memory hierarchies has been the subject of much investigation over the years. These have principally



(a) 1D 5-Point Stencil **(b)** 2D 5-Point Stencil

Figure 2.1: A visualization of two lower dimensional stencils. In (a), we show a one-dimensional five-point stencil, where the center point is both being read and written. In (b), we show a two-dimensional five-point stencil, where again the center point is read and written.

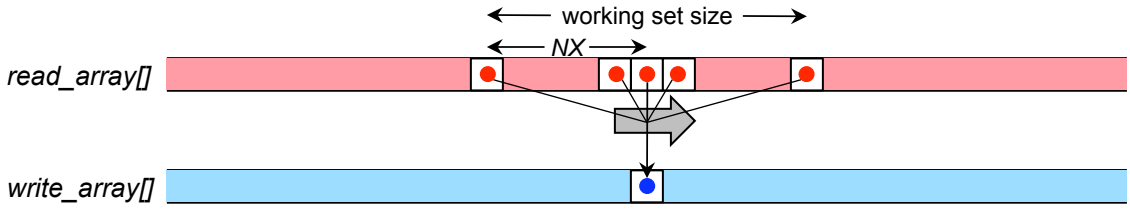
focused on tiling optimizations [43, 42, 32] that attempt to exploit locality by performing operations on cache-sized blocks of data before moving on to the next block. However, a study of stencil optimization [27] on (single-core) cache-based platforms found that these tiling optimizations were primarily effective when the problem size exceeded the on-chip cache's ability to exploit temporal recurrences. We will show shortly that for lower dimensional stencils, modern microprocessors have caches large enough to exploit these temporal recurrences. For stencils with dimensionality higher than two, however, tiling optimizations are still effective and needed.

2.1.1 Stencil Dimensionality

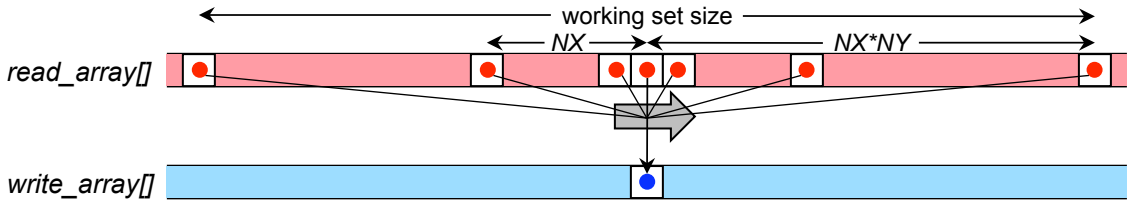
1D and 2D Stencils

In Figure 2.1, we show two examples of simple lower dimensional stencils. While lower dimensional stencils are fairly common, they are likely to be less amenable to tuning as well as heavily bandwidth-bound.

First, we know from previous work [28] that the lower the stencil dimensionality, the less likely it is to be affected by capacity cache misses [24]. This is because the required working set size is smaller. If we examine Figure 2.2(a), we see the conventional memory layout for a 2D 5-point stencil like the one in Figure 2.1(b). If NX is the unit-stride grid dimension of the 2D grid, then for the 5-point stencil, the first and last stencil points are separated by $(2 \times NX)$ doubles in the read array. When streaming through memory, by the time the last stencil point is read in, all the pre-



(a) Memory layout for the 2D 5-point stencil



(b) Memory layout for the 3D 7-point stencil

Figure 2.2: A visualization of the memory layout for two stencils of different dimensionality. In (a), we see that the first and last stencil points of the 2D 5-point stencil are separated by $(2 \times NX)$ doubles, where NX is the unit-stride dimension of the 2D grid. In (b), the distance between the first and last stencil points for the 3D 7-point stencil is much larger– specifically, $(2 \times NX \times NY)$ doubles, where NX and NY are the contiguous and middle dimensions of the 3D grid, respectively.

vious stencil points will still be in cache since the working set is miniscule compared to the last-level cache of current multicore processors. To make this concrete, the working set when performing a 2D five-point stencil on a 256^2 grid is about 4 KB. In comparison, the last-level caches of the machines in this thesis range from 2–8 MB. By avoiding capacity misses, 2D stencils will not benefit from cache tiling optimizations like *core blocking*, which we will introduce in Section 4.1.1. In this respect, 2D stencil codes have less headroom for performance improvement than codes that do incur capacity misses.

Second, the number of flops performed per point also usually decreases with lower stencil dimensionality. This is simply because there are fewer dimensions in which stencil points can occur. However, regardless of dimensionality, the memory traffic for each grid point remains constant. Therefore, for the same number of grid points, there are fewer flops performed for lower dimensional stencils. Consequently, these stencils are more susceptible to bandwidth limitations.

While the performance of lower dimensional stencil codes can be improved through

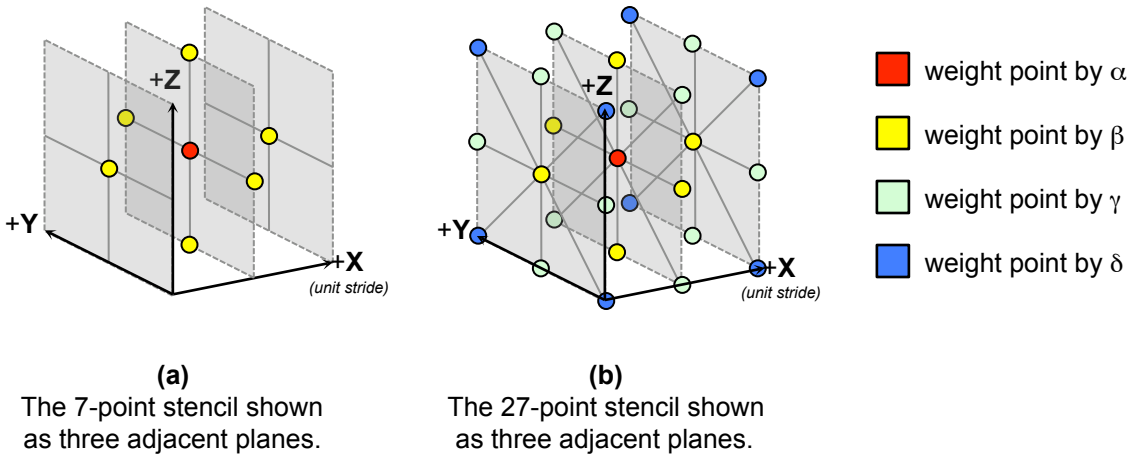


Figure 2.3: A visual representation of the 3D 7-point and 27-point stencils, both of which are shown as three adjacent planes. These stencils are applied to each point of a 3D grid (not shown), where the result of each stencil calculation is written to the red center point. Note: the color of each stencil point represents the weighting factor applied to that point.

tuning, they are heavily bandwidth-bound and typically do not benefit from cache tiling. As a result, the achievable speedup is fairly constrained compared to 3D stencils. In this thesis, we do not tune 1D and 2D stencils.

3D Stencils

The focus of this thesis is on 3D stencils, two of which are shown in Figure 2.3. The 7-point stencil, shown in Figure 2.3(a), weights the center point by some constant α and the sum of its six neighbors (two in each dimension) by a second constant β. Naïvely, a 7-point stencil sweep can expressed as a triply nested ijk loop over the following computation:

$$B_{i,j,k} = \alpha A_{i,j,k} + \beta(A_{i-1,j,k} + A_{i,j-1,k} + A_{i,j,k-1} + A_{i+1,j,k} + A_{i,j+1,k} + A_{i,j,k+1}) \quad (2.1)$$

where each subscript represents the 3D index into array A or B.

The 27-point 3D stencil, as shown in Figure 2.3(b), is similar to the 7-point stencil, but with additional points to include the edge and corner points of a 3×3×3 cube surrounding the center grid point. It also introduces two additional constants– γ, to weight the sum of the edge points, and δ, to weight the sum of the corner points.

These two stencils are the focus of our tuning in Chapters 7 and 8, and they will also be discussed in further detail later in this chapter.

Unlike the lower dimensional stencils that were just discussed, the required working set size for 3D stencils is much greater than 1D or 2D stencils. We see in Figure 2.2(b) that if NX and NY are the contiguous and middle dimensions of our 3D grid, respectively, then for the 3D 7-point stencil, the first and last points will be separated by $(2 \times NX \times NY)$ doubles in the read array. This is significantly larger than the $(2 \times NX)$ separation for the 2D 5-point stencil, since the distance is now in terms of *planes*, not *pencils*. For example, if we performed a sweep of the 3D 7-point stencil over a 256^3 grid, the working set is about 1 MB, which is approximately 250× the working set size for the 2D analog problem. Consequently, cache capacity misses are far more likely, but optimizations like core blocking should now be effective.

Furthermore, the number of flops per grid point will be higher in 3D codes as well. As we will discuss later, the 3D 7-point stencil in Figure 2.3(a) performs eight flops per point, since the center point is weighted by α and each of the six nearest neighbors is weighted by β. More generally, this stencil gives one weight (α) to the center point, and a separate weight (β) to each of the two neighboring stencil points in each dimension. Thus, the d-dimensional analog of this stencil will perform $(2d+2)$ flops per point, which increases linearly with dimensionality.

The increase in flop count is even more dramatic when we alter the dimensionality of the 3D 27-point stencil, displayed in Figure 2.3(b). This stencil has four weights associated with it– one each for the one center point (α), six face points (β), twelve edge points (γ), and eight corner points (δ). The full computation requires 30 flops per grid point. More generally, the d-dimensional analog of this stencil utilizes $(d+1)$ weights and performs $(3^d + d)$ flops per point. In this case, the flops per point rise exponentially with dimensionality.

Due to the occurrence of capacity misses and the greater number of flops per point, we anticipate that a larger number of transformations (including cache blocking and computational optimizations) will be effective for 3D stencils than for lower dimensional stencils. We also expect that the resulting speedups will be larger as well.

Higher-Dimensional Stencils

For higher dimensional stencils, the points that were made for three-dimensional stencils are further amplified. If we continue storing the structured grid data in the usual manner (shown in Figure 2.2), then the distance in memory between the first and last point will again be multiplied by the size of another dimension. If we naïvely stream through memory without any tiling optimizations, the required working set will almost definitely be larger than the available last-level cache. We should also expect more flops per point, given that there are more dimensions present.

Lattice Methods

Lattice methods are still structured grid codes, but instead of a single unknown per point, many can be present. Whether lattice methods should be considered very high dimensional stencils or a separate category unto themselves is debatable. However, we would be remiss if they were not mentioned. One example of a lattice method is a plasma turbulence simulation that was tuned by Williams et al [56]. This lattice Boltzmann application coupled computational fluid dynamics (CFD) with Maxwell's equations, resulting in a momentum distribution, a magnetic distribution, and three macroscopic quantities being stored *per point*. In total, each point required reading 73 doubles and updating 79 doubles, while performing approximately 1300 flops. This resulted in complex data structures and memory access patterns, as well as significant memory capacity requirements, so proper tuning (especially at the data structure level) was critical.

2.1.2 Common Stencil Iteration Types

This subsection goes into depth about three common stencil iteration types: Jacobi, Gauss-Seidel, and Gauss-Seidel Red-Black iterations.

Jacobi

As shown in Figure 2.4(a), Jacobi iterations are essentially out-of-place sweeps. In order to perform Jacobi iterations, there is at least one read grid and one write

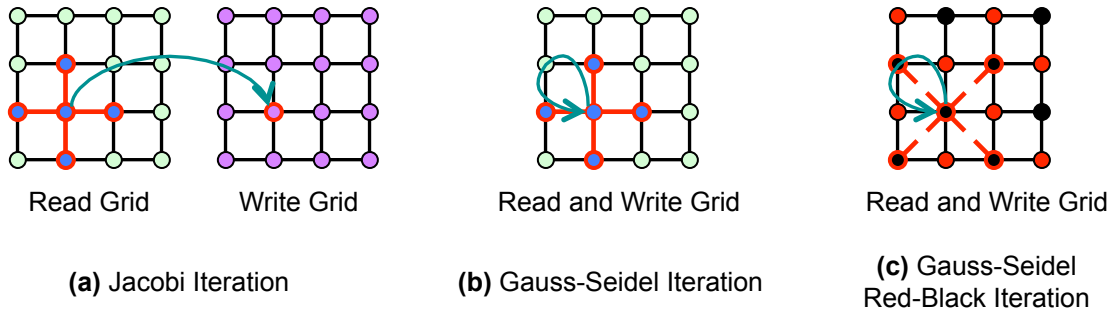


Figure 2.4: A visual representation of the some of the common iteration types that occur with stencil codes. In a Jacobi iteration, a stencil is applied to each point in the read grid, and the result is written to the corresponding point in the write grid. The Gauss-Seidel iteration is similar, except that there is only a single grid, and therefore the iteration is in-place. The Gauss-Seidel Red-Black iteration is also in-place, but all the points of one color are updated before the points of the other color are updated.

grid, but no grids are both read and written. We sweep through these grids by first incrementing the unit-stride index, then incrementing the middle index, and finally incrementing the least-contiguous index. Essentially, we stream consecutively through memory.

The power of Jacobi sweeps lies in the fact that it is easily parallelizable; any point in the write grid(s) can be computed independently of any other point, so they can be updated in an embarrassingly parallel fashion. Thus, since all partitioning schemes produce the same correct result, we are free to choose the one that performs best. Another benefit of the Jacobi iteration is that on x86 architectures, we can use the *cache bypass* optimization (discussed in Section 4.3.2) to reduce the memory traffic on a write miss by half by eliminating the cache line read. Most stencil codes are bandwidth-bound, so this has the potential to generate large performance speedups. The major drawback to the Jacobi iteration is that it requires that we store distinct read and write arrays, which increases both storage and bandwidth requirements.

When we perform 7-point stencil and 27-point stencil updates in Chapters 7 and 8, respectively, we will be performing a single Jacobi sweep. This allows us great flexibility in how we parallelize the problems.

Gauss-Seidel

Gauss-Seidel iterations, shown in Figure 2.4(b), are in-place sweeps. Zero or more read-only grids may be present, but the write grid must also be read from first. One consequence of this fact is that almost all the computed stencils will include some points that were already updated during the current sweep and others that were not. This means that there is an inherent dependency chain that needs to be respected if we wish to replicate the same final answer. Unfortunately, this significantly limits the amount of available parallelism in the sweep. It also causes different traversals through the grid to generate different (but valid) results. The major benefit to Gauss-Seidel sweeps is that the write grid is also read from, thereby reducing the need for additional arrays and the memory traffic associated with them.

Gauss-Seidel Red-Black

Gauss-Seidel Red-Black (GSRB) iterations are similar to Gauss-Seidel sweeps in that while read-only grids may be present, the write grid must be read from first. In order to deal with the limited parallelism that is exposed when consecutively updating each point (like in Gauss-Seidel sweeps), GSRB updates only every other point. For instance, in Figure 2.4(c), a black grid point will only be updated when the red points that it depends on have all been updated. Once the needed black points are ready, we can again start updating the red points, and so on. This type of sweep has similar parallelism characteristics to Jacobi, since any of the points of a single color can be updated independently of any other. Thus, the partitioning of this sweep among threads can be arbitrary.

If we define a GSRB sweep as performing a single update for both the red and black grid points, then a naïve GSRB implementation will sweep over the grid twice, likely requiring twice the memory traffic of a Jacobi or Gauss-Seidel sweep. However, we can minimize the required memory traffic by having a leading "wavefront" for the red points and a trailing wavefront for the black points. Assuming the last-level cache is sufficiently large, this would update both sets of points while only reading them from memory once.

The Helmholtz kernel, discussed in Chapter 9, will perform GSRB sweeps over

each subproblem.

Numerical Convergence Properties

While the numerical convergence properties of these iterative methods are generally out of the scope of this thesis, they are a major concern for tuning real-world stencil codes, and thus we touch upon them here. Out of the three iteration types just discussed, the Jacobi iteration is typically the slowest to converge. A superficial explanation for this is because every point is updated with old (*i.e.* not updated during the current iteration) grid point values. This is rectified by performing Gauss-Seidel sweeps, since most points are calculated from some old and some new values. Indeed, in most cases, Gauss-Seidel shows better convergence than Jacobi. However, the best convergence usually comes from GSRB; one GSRB step can decrease the error as much as two Jacobi steps. This is a general phenomenon for matrices arising from approximating differential equations with finite difference approximations [14].

It is important to note that when solving a linear system, it is possible that one step of Jacobi could reduce the error more than one step of GSRB. This is because the amount of convergence depends both on the problem and the iterative method. In most cases, though, GSRB will converge fastest and Jacobi slowest.

Ultimately, many iterative solvers attempt to reduce the problem error (or residual) below a certain threshold. In order to decide how to minimize the time needed to reach this level of convergence, we need to understand how many iterations a given solver will require (based on its numerical properties), as well as how long each iteration will take (based on numerical, hardware and software properties).

2.1.3 Common Grid Boundary Conditions

Another concern with structured grid codes is how to deal with boundary conditions. Here we discuss two common boundary conditions and how to deal with them.

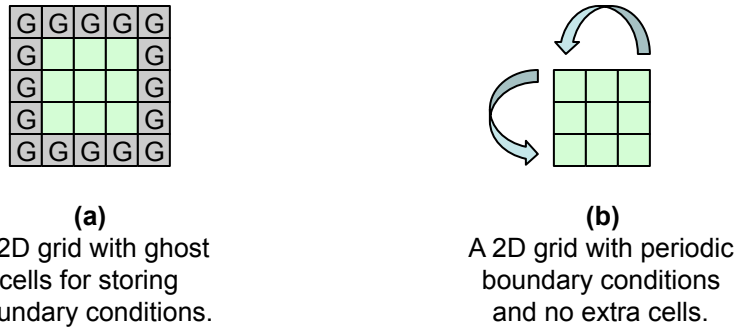


Figure 2.5: In (a), we show a 3×3 grid with surrounding ghost cells (marked with a "G") that are used to store boundary conditions. In (b), we show a 3×3 grid which does not require ghost cells because it has periodic boundary conditions.

Constant Boundaries

There are two main types of *constant boundary conditions*. In the first case, the points along the boundary do not change with time, but do change depending on position. In this case, we can use *ghost cells* (like in Figure 2.5(a)) to store these values before any stencil computations begin. Once the ghost cells are initialized, they do not need to be altered for the rest of the problem. In this thesis, all three 3D stencil kernels have this type of boundary condition. However, the ghost cells consume a non-trivial amount of memory for 3D grids. Suppose that we have an N^3 grid that is surrounded by ghost cells. Then, the resulting grid has $(N+2)^3$ cells. If $N = 16$, then ghost cells represent an astounding 30% of all grid cells. However, if $N = 32$, then the percentage drops to 17%.

The second case is if the boundary value does not change with time or position. In this case, the entire boundary can be represented by a single constant scalar throughout the course of the problem. Consequently, we no longer need to have individual ghost cells like the previous case.

Periodic Boundaries

Another common boundary condition is to have *periodic boundaries*, as shown in Figure 2.5(b). For points along the boundary, this means that they have additional neighbors that wrap around the grid. For example, the left neighbor of the upper left

point is the upper right point, while the upper neighbor is the lower left point. While periodic boundaries can be represented using ghost cells that are updated after each iteration, in many cases no ghost cells are used at all. Instead, the required values are merely read from the side of the grid. This helps lower the memory footprint of the grid as well as bandwidth requirements.

2.1.4 Stencil Coefficient Types

Constant Coefficients

In the stencils we have thus far discussed, we have usually assumed that the stencil coefficients are constant. For instance, in Figure 2.3, the color of each stencil point represents whether that point is weighted by the constant coefficient α, β, γ, or δ. Many finite difference calculations employ constant coefficient stencils like these.

When the coefficient values are constant scalars, they do not need to be continually read from memory for every new stencil. Instead, they can be hard-coded into the inner loop of the stencil code, and then kept in registers during the actual computation. This results in a large reduction in potential storage requirements and memory traffic. Furthermore, if these coefficients are simple integer values, the compiler may even be able to further optimize parts of the computation.

In many ways, the constant coefficient stencil is an ideal scenario. Consequently, if we can expose the appropriate properties in an iterative solver's underlying matrix so as to generate a constant coefficient stencil, we will always do so. In Section 2.2, we will specify these properties.

The 7-point and 27-point stencils that we tune in Chapters 7 and 8, respectively, both have constant coefficients.

Variable Coefficients

However, the stencil coefficients need not be constant. As we will examine in Section 2.2, the iterative solver's underlying matrix may not have the appropriate properties for it. In such a case, we may still be able to utilize a variable coefficient stencil, where the stencil weights change from one grid point to another. Unlike constant coefficient stencils, we now need to store these weights in separate grids.

When performing our calculations, we will stream through these grids along with our original grid of unknowns. This will create extra DRAM traffic, but we will show in Section 2.2.3 that this is still preferable to performing a sparse matrix-vector multiply.

The Helmholtz kernel that we will tune in Chapter 9 employs a variable coefficient stencil.

2.2 Exploiting the Matrix Properties of Iterative Solvers

Thus far, we have discussed various stencil characteristics, but we have not mentioned the origins of stencils in any detail. This section explains how both variable and constant coefficient stencils can arise from iterative solvers.

Imagine that we have a structured grid like the one shown in Figure 2.6. This grid is a simple two-dimensional grid with periodic boundary conditions in both dimensions. Now, let us suppose that we would like to apply an iterative solver to this grid. In most cases, this solver will require that every point in the structured grid be updated with some linear combination of other grid points. In order to generally represent this linear transformation, we can create a matrix that stores the weights that each point contributes to every point in the grid. In order to create this matrix, we first need to select an ordering of the grid points. In our case, the simplest way to proceed is to choose a *natural row-wise ordering*, where we first order the top row from left to right, then the second row in a similar manner, and so on. The numbers inside each grid point of Figure 2.6 show such an ordering [30].

We can now create a matrix A that represents the linear transformation performed by the iterative solver. The amount that point j will be weighted by when calculating the new value of point i is given by matrix element A_{ij}.

2.2.1 Dense Matrix

At this juncture, a critical question is how best to store this matrix. At the most general level, we can store A as a dense matrix. However, for a $n \times n$ square grid

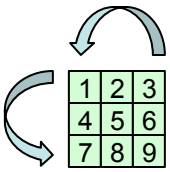


Figure 2.6: A 3×3 numbered grid with periodic boundary conditions in both the horizontal and vertical directions. The grid points are numbered in a natural row-wise ordering.

with no extra ghost cells, the resulting matrix A will have dimensions of $n^2 \times n^2$. Thus, the 3×3 grid from Figure 2.6 would be stored as a 9×9 matrix. Despite the large size of A in a dense matrix format, most iterative solvers only reference a few grid points in updating each point, so we expect A to be *sparse* (*i.e.* most of the elements are zero). Therefore, storing A in a dense matrix would be inefficient in terms of both storage and flops.

2.2.2 Sparse Matrix

A better choice would be some sort of sparse matrix format. Let us assume that there are nnz non-zeros in the A matrix, each of which will be stored as an 8 Byte double-precision number. In addition, the matrix indices will be stored as 4 Byte integers. Given this, if we choose the commonly-used Compressed Sparse Row (CSR) format [53] to store A, we will require about $12nnz+4n^2$ Bytes of storage and perform approximately $2nnz$ flops. This is much better than the $8n^4$ Bytes of storage and $2n^4$ flops required by the dense format. While the CSR format does perform indirect accesses into the matrix, it should still be orders of magnitude faster than the dense format for sufficiently large and sparse matrices.

2.2.3 Variable Coefficient Stencil

Sparse matrices are a general way to represent the linear transformation performed by an iterative solver. However, in many cases, these iterative solvers perform nearest neighbor operations on the structured grid. For instance, imagine that for every point grid in Figure 2.6, we only require the values of the points immediately

$$\left[\begin{array}{ccc|ccc|ccc}
2 & 7 & -13 & 8 & & & 4.2 & & \\
9 & 101 & 20 & & -31 & & & -2 & \\
-7 & 14 & 5.3 & & & 5 & & & 87 \\
\hline
-12 & & & -15 & 23 & 17 & 8 & & \\
 & 13 & & 1.3 & 7 & -9 & & 14 & \\
 & & -67 & 5 & 0.4 & 3 & & & 51 \\
\hline
-3 & & & 1 & & & 9 & 17 & 10 \\
 & 0.1 & & & 42 & & -41 & 2 & -1 \\
 & & 81 & & & 8 & 32 & 91 & 71
\end{array}\right]$$

Table 2.1: This sparse matrix has a very regular structure, but the non-zero values are unpredictable. For readability, the matrix is divided into 3×3 submatrices and only the non-zeros are shown.

above, below, to the left, and to the right of that point, as well as the value of the point itself. Thus, every point will be updated with the values from five grid points, but the weights associated with these grid points are not predictable.

The 9×9 matrix in Table 2.1 represents such a situation for the case of periodic boundary conditions. Due to its regular structure, we no longer explicitly need to store the matrix indices. Instead of storing this $n^2 \times n^2$ matrix in a sparse format, we can store it as five $n \times n$ grids. For a given point (i, j) from the grid in Figure 2.6, the corresponding (i, j) point in each of these grids represents the weight associated with point (i, j) or any of its four neighbors. More information about these variable coefficient stencils can be found in Section 2.1.4.

Thus, we now require $40n^2$ Bytes of storage and perform $9n^2$ flops. In a sparse matrix format, we would need $64n^2$ Bytes of storage, where the extra $24n^2$ Bytes is attributed to the unneeded indices of the CSR format. However, we have not altered the flop count by changing the data structure from a sparse matrix to a variable coefficient stencil.

2.2.4 Constant Coefficient Stencil

Finally, if we have a matrix with the same structure as Table 2.1, but also with predictable non-zeros, we no longer need to explicitly store the non-zeros either.

$$\left[\begin{array}{ccc|ccc|ccc}
-4 & 1 & 1 & 1 & & & 1 & & \\
1 & -4 & 1 & & 1 & & & 1 & \\
1 & 1 & -4 & & & 1 & & & 1 \\
\hline
1 & & & -4 & 1 & 1 & 1 & & \\
 & 1 & & 1 & -4 & 1 & & 1 & \\
 & & 1 & 1 & 1 & -4 & & & 1 \\
\hline
1 & & & 1 & & & -4 & 1 & 1 \\
 & 1 & & & 1 & & 1 & -4 & 1 \\
 & & 1 & & & 1 & 1 & 1 & -4
\end{array}\right]$$

Table 2.2: This is the matrix associated with the 2D Laplacian operator with periodic boundary conditions. It has a very regular structure and the non-zero values are predictable.

Instead of storing five separate arrays, we only need to store five separate scalar constants! Moreover, if we wish to apply an operator like the 2D Laplacian, we only need to store two constants. This is because the center point will always be weighted by -4, while all of four of its neighbors will be weighted by 1. The matrix associated with the 2D Laplacian operator with periodic boundary conditions is shown in Table 2.2. Now, we only need to store 16 Bytes worth of data– the weight of the stencil's center point and the weight of its four neighbors. Moreover, this reduces our flop count from $9n^2$ down to $6n^2$. We have finally succeeded in reducing the dense matrix representing our iterative solver down to a simple constant coefficient stencil.

2.2.5 Summary

The above arguments for reducing storage, bandwidth, and computation requirements were made for two-dimensional structured grids, and are summarized in Table 2.3. However, the corresponding savings are even more stark for three-dimensional stencils, which are the focal point of this thesis.

	Explicit Indices	Implicit Indices
Explicit Non-zeros	Usual Sparse Matrix-Vector Multiply	Variable Coefficient Stencil
Implicit Non-zeros	Example: Laplacian of a General Graph	Constant Coefficient Stencil

Table 2.3: This table displays the continuum from sparse matrix-vector multiply to variable-coefficient stencils and finally constant-coefficient stencils.

2.3 Tuned Stencils in this Thesis

This thesis focuses on two second-order finite difference operators as well as a finite volume operator. The finite difference operators are constant coefficient stencils, while the finite volume operator is a variable coefficient stencil. In this section, we describe each in more detail, including some information on where the stencils originate from.

2.3.1 3D 7-Point and 27-Point Stencils

The 3D 7-point and 27-point stencils, visualized in Figure 2.3, commonly arise from the finite difference method for solving PDEs [30]. The 7-point stencil performs eight flops per grid point, while the 27-point stencil performs 30 flops per point (without any type of common subexpression elimination). Thus, the *arithmetic intensity*, the ratio of flops performed for each Byte of memory traffic, is about 3.8× higher for the 27-point stencil than the 7-point stencil. We will see in Chapter 8 that the compute-intensive 27-point stencil will actually be limited by computation on some multicore platforms.

The 7-point 3D stencil is fairly common, but there are many instances where larger stencils with more neighboring points are required. One such stencil arises from T. Kim's work in optimizing a fluid simulation code [29]. By using a Mehrstellen scheme [10] to generate a 3D 19-point stencil (where δ equals zero in Figure 2.3) instead of the usual 7-point stencil, he was able to reach the desired error reduction in 34% fewer stencil iterations. Thus, larger stencils can reduce the number of iterations

needed to reach a desired threshold of convergence. In this thesis, we chose to examine the performance of the 27-point 3D stencil because it serves as a good proxy for many of these compute-intensive stencil kernels.

In general, though, the numerical properties of the 7-point and 27-point stencils are outside the scope of this work; we merely study and optimize their performance across different multicore architectures. Our results will hopefully allow the reader to judge as to whether these numeric/performance tradeoffs are worthwhile. As an added benefit, this analysis also helps to expose many interesting features of current multicore architectures.

2.3.2 Helmholtz Kernel

The final stencil that we tune is the Helmholtz kernel. This kernel is ported from Chombo [9], a software framework for performing Adaptive Mesh Refinement (AMR) [3].

The Helmholtz kernel that we tune in Chapter 9 attempts to solve for ϕ in the equation:

$$L(\phi) = rhs \tag{2.2}$$

where rhs is a given right-hand side and L is the linear Helmholtz operator:

$$L = \alpha\vec{A}\vec{I} - \beta\vec{\nabla} \cdot \vec{B}\vec{\nabla} \tag{2.3}$$

We can solve Equation 2.2 iteratively by calculating the residual, multiplying it by λ, and subtracting this quantity from our original ϕ (called ϕ^* below):

$$\phi_{new} = \phi^* - \lambda(L(\phi^*) - rhs) \tag{2.4}$$

If we perform enough iterations of Equation 2.4, we should converge (albeit slowly) to a ϕ whose residual is below a given threshold. However, to hasten this process, we can use solvers like multigrid [6, 52], where these iterations can be used to relax each multigrid level. In the case of AMR, where many small grids are present, multigrid is applied to the entire collection of subproblems [33], while a relaxation operator (like GSRB) is applied to each of the individual subproblems.

The power of the Helmholtz equation comes from its ability to solve time-dependent problems *implicitly* within a multigrid solver. Explicit time discretization schemes

Single Helmholtz Subproblem		
Subgrid	Read/Write	Dimensions
phi	Read and Write	$(NX+2) \times (NY+2) \times (NZ+2)$
aCoef0	Read Only	$NX \times NY \times NZ$
bCoef0	Read Only	$(NX+1) \times NY \times NZ$
bCoef1	Read Only	$NX \times (NY+1) \times NZ$
bCoef2	Read Only	$NX \times NY \times (NZ+1)$
lambda	Read Only	$NX \times NY \times NZ$
rhs	Read Only	$NX \times NY \times NZ$

Table 2.4: A description of the seven grids involved in a single variable-coefficient Helmholtz subproblem. The NX, NY, and NZ grid parameters are visually displayed in Figure 4.1.

place bounds on the size of the time step due to the Courant-Friedrichs-Lewy (CFL) stability condition. Implicit time discretization schemes, however, have no time step restriction, and are unconditionally stable if arranged properly.

One example of this is the parabolic heat equation. While this equation can be solved using an explicit forward Euler scheme, the CFL condition will keep our time steps short. The discrete Helmholtz equation, on the other hand, can apply several different implicit schemes merely by varying α and β in Equation 2.3. In particular, we can apply a backward Euler, Crank-Nicholson, or backward difference formula through the Helmholtz equation, all of which allow for much larger time steps than forward Euler.

A second example is the hyperbolic wave equation. Again, the CFL condition only limits the time steps of explicit methods. For most common implicit discretizations, each time step can again be solved implicitly using the discrete Helmholtz equation with an appropriately tuned α and β. These examples apply to more general time-dependent parabolic and hyperbolic PDEs as well. The beauty of this approach is that the larger the time steps, the more will be gained through an appropriate multigrid treatment [52].

Now, if we actually discretize the Helmholtz equation (Equation 2.4), the result is a variable coefficient stencil consisting of seven grids. Six of these grids are read only, while the `phi` grid is both read and written; the dimensions of each of these grids

Figure 2.7: This diagram shows that the cell-centered grid in (a) requires far fewer grid points than the face-centered grid in (b). The grid points in (b) are color-coded, where the points along a cell's horizontal edges are green and the points along the vertical edges are in blue.

are given in Table 2.4. As this is a variable coefficient stencil, we note that the `phi` array cannot be merely updated with scalar weights; there are five grids (other than `phi` or `rhs`) that need to be referenced in order to perform this stencil calculation.

Some of the arrays in Table 2.4 require further explanation. For instance, $\vec{B}$ is represented as three separate arrays– `bCoef0`, `bCoef1`, and `bCoef2`. This is because the stencil originates from a *finite volume*, not finite difference, calculation. So as to abide by certain conservation laws, B employs a face-centered, not cell-centered, discretization. As we can see in Figure 2.7, the face-centered grid in (b) requires many more grid points than cell-centered grid in (a). The primary reason for this is that the face-centered discretization requires that there be a grid point along each edge of a given cell. For instance, Figure 2.7(b) shows the points along each cell's horizontal edges in green and the points along the vertical edges in blue. For this calculation, the face-centered grid points along each dimension are stored separately. As this is a three-dimensional problem, $\vec{B}$ is thus stored as three separate arrays (`bCoef0`, `bCoef1`, and `bCoef2`). In addition, each of these arrays needs a single extra grid point in one dimension in a similar fashion to how there are four columns, but five rows, of green points in Figure 2.7(b).

The `phi` array deserves some explanation as well, since it has two extra cells in each dimension. The extra cells in this case are simply ghost cells that store boundary values. To be mathematically correct, these ghost cells should be updated after each iteration. In some cases, however, it is possible to perform multiple iterations without a ghost cell update while still preserving stability and accuracy. This is an area of

current research [31].

When executing a sweep of this variable coefficient stencil, we can choose any of the iteration types displayed in Figure 2.4, but GSRB was chosen for its convergence and parallelization properties. However, due to the number of arrays present and the fact that we are employing a GSRB stencil, the arithmetic intensity of this kernel is fairly low, despite performing 25 flops per point.

2.4 Other Stencil Applications

Thus far, we have discussed a few areas where stencils originate. The following section introduces other uses of stencils, but this is far from an exhaustive list.

2.4.1 Simulation of Physical Phenomena

Stencils are often found when modeling physical phenomena. For instance, Nakano et al used stencils as part of a molecular dynamics algorithm for realistic modeling of nanophase materials [38]. Specifically, in order to compute the Coulomb potential, which is very expensive due to its all to all nature, the authors decided to use the fast multipole method (FMM). In the FMM, the Coulomb potential is computed using a recursive hierarchy of cells. At each level of this hierarchy, the near-field contribution to the potential energy is calculated through nearest neighbor stencil calculations. The FMM algorithm is able to reduce this $O(N^2)$ all-to-all calculation down to a complexity of $O(N)$.

Stencils are also used in quantum mechanics simulations. Shimono et al [45] developed a hierarchical method that decomposed the spatial grid based on higher-order finite differences and multigrid acceleration [6]. This method also refines adaptively near each atom to accurately operate the ionic pseudopotentials on the electronic wave functions. This divide-and-conquer scheme is used to iteratively and quickly solve for the potential throughout the grid. As an added benefit, this approach provides simple and efficient parallelization of the problem due to the short-ranged operations involved.

A final simulation example comes from the area of earthquake modeling. Specifi-

cally, Dursun et al have tuned a seismic wave propagation code for x86 clusters [15]. The code employs a higher-order 3D 37-point stencil based off of the finite difference method. Such a stencil not only involves heavy computation but also large memory requirements (since the points are not clustered around the center point). The authors used spatial decomposition, multithreading, and SIMD (explained in Section 4.4.2) to achieve speedups of up to 7.9×. These optimizations are a subset of the ones we employ in this thesis, described in Chapter 4.

2.4.2 Image Smoothing

We now take a step away from simulations and instead focus on image smoothing, a fundamental operation in computer vision and image processing. This smoothing is often done through a bilateral filter, which tries to remove noise while not smoothing away edge features. Consequently, a bilateral filter uses a variable coefficient stencil, where the weights are computed as the product of a geometric spatial component and signal difference. Kamil et al tuned this stencil kernel [26] using many of the same techniques that we employed in this thesis.

2.5 Summary

This chapter has introduced stencils, as well as some of the issues that occur when confronting variations in dimensionality, iteration type, boundary condition, or coefficient type. Many of these issues will play out as we tune the three stencils in Chapters 7, 8, and 9.

Part of this chapter was also dedicated to explaining the origins of stencils that are used as iterative solvers. As we saw, we need to exploit the underlying matrix structure to generate variable coefficient stencils, but we also need to take advantage of predictable non-zeros if we wish to use constant coefficient stencils. Fortunately, this happens fairly often, as we discovered stencils being utilized as iterative solvers in material modeling, quantum mechanics, and seismic wave modeling codes. We also observed a non-simulation use of stencils– performing image smoothing using a bilateral filter. Thus, stencils are found in many scientific disciplines, and some

non-science ones as well. This ubiquity emphasizes the importance of achieving good stencil code performance on multicore platforms.

Chapter 3

Experimental Setup

This chapter details our experimental methodology at almost every level of the system stack. At the hardware level, we discuss specifics about the multicore platforms used in our evaluations and the thread mapping policy we implemented. At the software level, we justify our choice of parallel programming model, programming language, and compiler. Finally, we also consider how we data is collected and presented to ensure reproducibility.

3.1 Architecture Overview

We compiled data across a diverse array of cache-based multicore computers that represent the building blocks of current and near future ultra-scale supercomputing systems. This not only allows us to fully understand the effects of architecture, it also demonstrates our auto-tuner's ability to provide performance portability. Table 3.1 details the core, socket, and system configurations of the five cache-based computers used in this work. They are also discussed below.

3.1.1 Intel Xeon E5355 (Clovertown)

Displayed in Figure 3.1(a), Clovertown was Intel's first foray into the quad-core arena. Reminiscent of Intel's original dual-core designs, each socket consists of two dual-core Xeon chips that are paired into a multi-chip module (MCM). In order for a socket to communicate with other parts of the system, it is attached to a common

Core Architecture	Intel Core2	Intel Nehalem	AMD Barcelona	IBM PowerPC 450	Sun Niagara2
Type	superscalar ooo[†]	superscalar ooo	superscalar ooo	dual issue in-order	dual issue in-order
ISA	x86	x86	x86	PowerPC	SPARC
Threads/Core	1	2	1	1	8
Process	65nm	45nm	65nm	90nm	65nm
Clock (GHz)	2.66	2.66	2.30	0.85	1.16
DP GFlop/s	10.7	10.7	9.2	3.4	1.16
L1 D-cache	32KB	32KB	64KB	32KB	8KB
private L2 cache	—	256KB	512KB	—	—

Socket Architecture	Xeon E5355 Clovertown	Xeon X5550 Nehalem	Opteron 2356 Barcelona	Blue Gene/P Compute Chip	UltraSparc T5140 T2+ Victoria Falls
Cores/Socket	4 (MCM)	4	4	4	8
shared last-level cache	2×4MB (shared by 2)	8MB	2MB	8MB	4MB
memory parallelism	HW prefetch	HW prefetch	HW prefetch	HW prefetch	Multi-Threading

System Architecture	Xeon E5355 Clovertown	Xeon X5550 Nehalem	Opteron 2356 Barcelona	Blue Gene/P Compute Node	UltraSparc T5140 T2+ Victoria Falls
Sockets/SMP	2	2	2	1	2
NUMA	—	✓	✓	—	✓
DP GFlop/s	85.3	85.3	73.6	13.6	18.7
DRAM Pin Bandwidth (GB/s)	21.33(read) 10.66(write)	51.2	21.33	13.6	42.66(read) 21.33(write)
Flop:Byte DP Ratio	2.66	1.66	3.45	1.00	0.29
DRAM Size (GB)	16	12	16	2	32
DRAM Type	FBDIMM-667	DDR3-1066	DDR2-800	DDR2-425	FBDIMM-667
System Power (W) [§]	530	375	350	31[‡]	610
Compiler	`icc` 10.0	`icc` 10.0	`icc` 10.0	`xlc` 9.0	`gcc` 4.0.4

Table 3.1: Architectural summary of evaluated platforms. [†]out-of-order (ooo). [§]System power is measured with a digital power meter while under a full computational load. [‡]Power running Linpack averaged per blade. (www.top500.org)

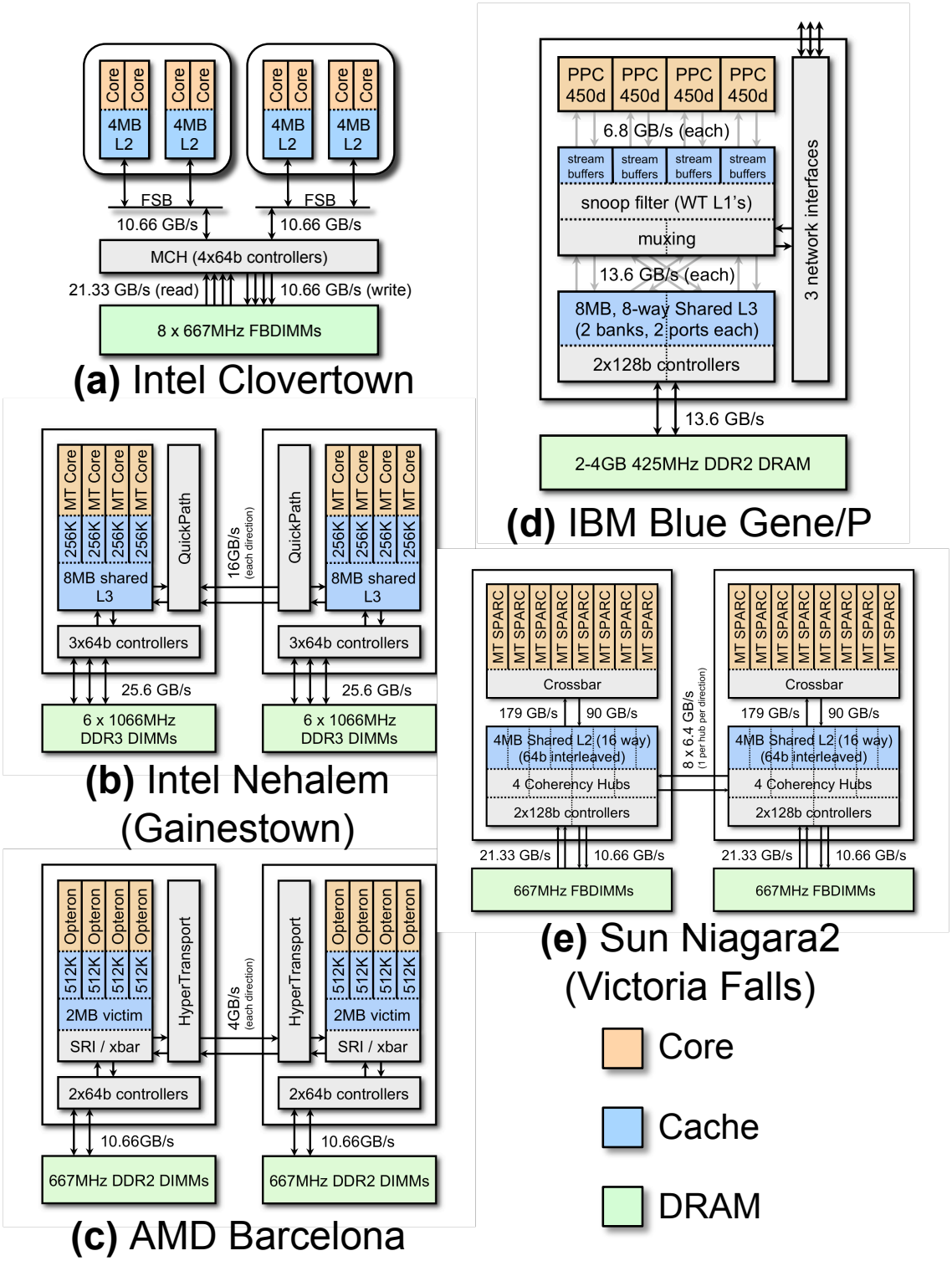


Figure 3.1: Hardware configuration diagrams for every architecture in our study. While the Blue Gene/P is single socket, all the other platforms are dual socket. These diagrams are courtesy of Samuel Webb Williams.

northbridge chip, also known as a memory controller hub (MCH), via a 10.66 GB/s frontside bus (FSB). This northbridge provides access to the caches of other sockets, DRAM memory, and other peripherals.

This type of older frontside bus architecture severely limits both system scalability and memory performance. All memory and cache coherency traffic must travel through the northbridge, which quickly becomes a bottleneck as core counts are increased. The one positive aspect of the architecture is that it provides *uniform memory access* (UMA), which makes threaded programming easier. Programmers do not have to concern themselves with varying memory access times since every core will require the same time to access any memory location. However, this is meager compensation for the significant performance penalty resulting from the FSB.

Our study evaluates a Sun Fire X4150 dual-socket platform, which contains two MCMs with dual independent busses. The chipset provides the interface to four fully buffered DDR2-667 DRAM (FBDIMM) channels, each with two DIMMs. Combined, they can deliver an aggregate read memory bandwidth of 21.33 GB/s. Each core may activate all four channels, but rarely attains peak memory performance due to the limited FSB bandwidth and the coherency protocol that consumes the FSB.

3.1.2 Intel Xeon X5550 (Nehalem)

The recently released Nehalem, shown in Figure 3.1(b), is the successor to the Intel "Core" architecture and represents a dramatic shift from Intel's previous multiprocessor designs. Unlike the frontside bus architecture employed on the Clovertown, the Nehalem adopts a modern multisocket architecture. Specifically, memory controllers have been integrated on-chip, thus requiring an inter-chip network. The resultant QuickPath Interconnect (QPI) handles access to remote memory controllers, coherency, and access to I/O.

For the programmer, this change comes at a price. With the advent of on-chip memory controllers, a discrete northbridge is no longer needed, and DRAM can be attached directly to a socket. Consequently, accessing memory on a remote socket is now significantly more expensive than retrieving data from local DRAM. This type of newer architecture design is therefore *non-uniform memory access* (NUMA). These

machines require the programmer to properly map memory pages to the requesting socket's DRAM while also minimizing remote memory accesses. The details of how this is done is discussed in Section 4.2.1. Nevertheless, the dramatic improvement in the memory performance of NUMA machines over the older FSB approach more than justifies any required code modifications.

Two other architectural innovations were incorporated into Nehalem: two-way simultaneous multithreading (SMT) and TurboMode. Two-way SMT allows two hardware threads to run concurrently on a single core. This allows for a better utilization of core resources, and, to some extent, better hiding of memory latency. The other feature, TurboMode, allows one core to operate faster than the set clock rate under certain workloads. On our machine, TurboMode was disabled due to its inconsistent timing behavior.

The system in this study is a dual-socket 2.66 GHz Xeon X5550 (Gainestown) with a total of 8 cores. Each dual-threaded core contains a 32KB L1 data cache and a 256KB unified L2 cache. An 8MB L3 cache is also shared by all the cores on the same socket. Each socket integrates three DDR3 memory controllers that can provide up to 25.6GB/s of DRAM bandwidth to each socket.

3.1.3 AMD Opteron 2356 (Barcelona)

The Opteron 2356 (Barcelona), visualized in Figure 3.1(c), was AMD's initial quad-core processor offering. It is similar to the Nehalem in that it is a NUMA architecture with integrated memory controllers and the HyperTransport (HT) inter-chip network that serves the same purposes as Intel's QuickPath. However, unlike Nehalem, Barcelona does not support SMT.

Superficially, Nehalem, Clovertown, and Barcelona all implement a very similar core microarchitecture. Each Opteron core contains a 64KB L1 cache and a 512MB L2 victim cache. In addition, each chip instantiates a 2MB L3 victim cache shared among all four cores. Each socket of the Barcelona includes two DDR2-667 memory controllers and a single cache-coherent HT link to access the other socket's cache and memory; thus delivering 10.66 GB/s per socket, for an aggregate peak NUMA memory bandwidth of 21.33 GB/s for the quad-core, dual-socket Sun X2200 M2

system examined in our study.

3.1.4 IBM Blue Gene/P

Unlike the other architectures used in this work, IBM's Blue Gene/P solution (shown in Figure 3.1(d)) was tailored and optimized for ultrascale (up to 2^{20} cores) supercomputing. To that end, it was optimized for power efficiency and cost effectiveness while maintaining a conventional programming model. Our single-socket Blue Gene/P processor is comprised of four, dual-issue PowerPC 450 embedded cores. Each core runs at a relatively slow 850MHz, includes a highly associative 32KB data cache, and has a 128-bit fused multiply add (FMA) SIMD pipeline (known as double hummer). In addition to several streaming prefetch buffers, each chip includes a shared 8MB L3 cache and two 128-bit interfaces each to 1GB of 425MHz DDR DRAM. This provides each socket with 13.6GB/s of DRAM bandwidth for its 13.6 GFlop/s of performance.

We see in Table 3.1 that Blue Gene/P sacrifices clock rate, peak computation rate, memory bandwidth, and memory capacity for an order of magnitude improvement in system power. While the rack-mounted servers in this study consume more than 300W of power, each Blue Gene/P compute node requires a mere 31W. This can be partially attributed to the fact that our Blue Gene/P is the only single socket platform in our study, but it is still a substantial power savings. For this work, we only examine the performance of one Blue Gene/P compute node configured in shared memory parallel (SMP) mode.

3.1.5 Sun UltraSparc T2+ (Victoria Falls)

The Sun "UltraSparc T2 Plus" presents an interesting departure from mainstream multicore chip design. As seen in Figure 3.1(e), the system is a dual-socket × 8-core SMP that is often referred to as *Niagara2* or *Victoria Falls*. Rather than depending on four-way superscalar execution, each of the eight strictly in-order cores supports two groups of four hardware thread contexts (referred to as Chip MultiThreading or CMT) — providing a total of 64 simultaneous hardware threads per socket. Assuming there are no floating point unit or cache conflicts, each core may issue up to one

instruction per thread group. The CMT approach is designed to tolerate instruction, cache, and DRAM latency through fine-grained multithreading. In actuality, the multithreading may hide instruction and cache latency, but may not fully hide DRAM latency.

Our study examines the Sun UltraSparc T5140 with two T2 processors operating at 1.16 GHz. Each core only has a single floating-point unit (FPU) that is shared among 8 threads. The UltraSparc was primarily designed for transaction processing, so the FPU does not support fused-multiply add (FMA) nor double precision SIMD functionality.

As for the memory hierarchy, each socket's 8 cores have access to a private 8KB write-through L1 cache and a shared 4MB L2 cache via an on-chip crossbar switch. Victoria Falls has no hardware prefetching, and software prefetching only places data into the L2. Each L2 cache is also fed by two dual-channel 667 MHz FB-DIMM memory controllers that deliver an aggregate bandwidth of 32 GB/s per socket (21.33 GB/s for reads and 10.66 GB/s for writes). Although the peak pin bandwidth of Victoria Falls is higher than any of the x86 architectures, its peak flop rate per socket is lower than even BGP.

3.2 Consistent Scaling Studies

To ensure our multicore scaling experiments are consistent and comparable across the wide range of architectures in Section 3.1, we exploit affinity routines to first utilize all the hardware thread contexts on a single core, then scale to all the cores on a socket, and finally use all the cores across all sockets. Some architectures, such as the Intel Clovertown, do not do this by default because they wish to exploit the second socket's bandwidth early. These platforms were remapped to adhere to the thread affinity policy just described.

This approach to thread affinity compels us to exhaust all the available resources on a given part of the chip before utilizing any new hardware. For instance, for multi-threaded architectures like Nehalem and Victoria Falls, we always fully utilize all the hardware threads on a single core before scaling to multiple cores. Similarly, on the multi-socket architectures, we always populate all the cores on a single socket

before including the second socket; this prevents us from exploiting a second socket's memory bandwidth until the first socket's bandwidth is maximized.

This approach to the problem of thread affinity has two main benefits. First, it is a simple policy that can be implemented across any multicore chip. However, the real beauty of this approach is that, outside of cache effects, we do not expect to see superlinear scaling results. This makes our performance results much more comprehensible. In contrast, if we scaled from one core to two cores by employing the second socket, performance could more than double initially, but then flatten as the entire system's cores are utilized.

Another problem that we had to address was the wide deviation in hardware thread counts across the architectures introduced in Section 3.1. Given that Victoria Falls supports 128 threads, it was obviously not realistic to show data for every thread count. Instead, we chose to scale the threads in powers of two. This is effective because all our machines have a power of two number of threads on a core, cores on a socket, and sockets. However, some newer processors do not always conform to this design. For instance, the recently released six-core AMD Opteron [1] forces us to modify our power of two thread scaling strategy. One way to deal with this problem is to scale in powers of two for as long as possible, and then have the final data point utilize all the cores on the chip. In the case of the six-core Opteron, this would mean collecting data for 1, 2, 4, and 6 cores.

We note that in our study, we never oversubscribed threads to cores. Specifically, the number of software threads was always less than or equal to the number of available hardware threads on a core, socket, or system– but never more than this number. While oversubscribing threads is an interesting direction in future research, we believe that this would likely cause performance degradation because of the time sharing between software threads.

3.3 Parallel Programming Models

Having explained both the hardware and the thread mapping policy employed in our evaluation, we now turn our attention to software, and specifically parallel programming models. We wish to select an appropriate programming model that

maximizes the performance of our cache-coherent shared memory multicore machines.

However, more than that, the chosen programming model should be able to best exploit *future* multicore machines as well. We anticipate that on these future multicore platforms, several changes will occur. It is likely that both the number of sockets and the number of cores on a socket will continue to scale. The total memory capacity per socket will grow relatively slowly, though– at least until advances like chip stacking take place [37, 44]. As a result, stencil grids will likely be weakly scaled (*i.e.* the grid size grows proportionally to the number of sockets or cores) at the node level, but effectively strongly scaled within a node (*i.e.* the grid size remains constant). Instead of focusing on node level scaling, this thesis will focus on good performance within a node. Thus, we need to ensure that our chosen programming model will help us perform effective intra-node strong scaling. The drawback of strong scaling is that different architectures support varying numbers of hardware threads, and thus will receive different amounts of work per thread. Unfortunately, insufficient work per thread can cause performance degradation, so we will set our workload to be large enough to avoid this issue.

In this section, we discuss three common programming models and their applicability to structured grids. For shared memory programming, the two most common models are POSIX threads (Pthreads) [51] and Open Multi-Processing (OpenMP) [39]. If we chose to use a distributed memory programming model, by far the dominant paradigm would be the Message Passing Interface (MPI) [46]. There are also hybrid programming models that combine Pthreads or OpenMP with MPI, but these incorrectly assume that adding MPI to a threaded stencil kernel is relatively straightforward. In the end, we selected Pthreads for reasons explained below.

3.3.1 Pthreads

Pthreads is a set of C language programming types and procedure calls to support parallelism on shared memory machines. The created threads run within a UNIX process, and are fairly lightweight since they only replicate enough resources so as to run as independent streams of instructions [51].

There are several advantages to using Pthreads over other programming models.

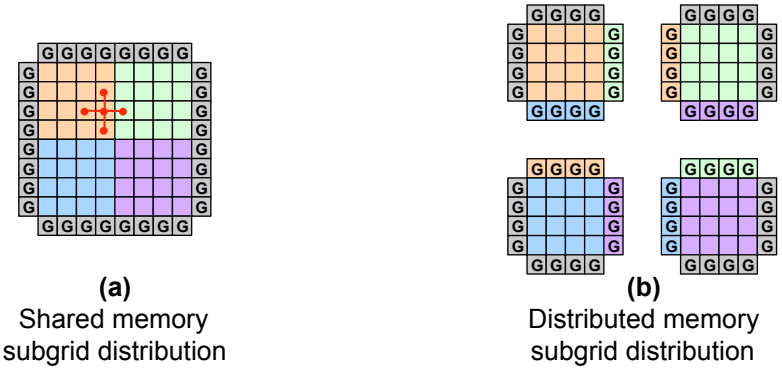


Figure 3.2: The figures above display the shared and distributed memory subgrid distributions for a 2D grid that will have a 5-point stencil (in red) sweep over it by four cores. The non-grey points reflect the data processed by each of the four cores, while the grey points along the exterior represent the boundary values of the problem. In addition, *ghost cells*, which store either boundary conditions or duplicate needed data from other subgrids, are marked with a "G". In (a), the shared memory layout has all the processor subgrids stored contiguously in memory, without any replication. In contrast, (b) shows the distributed memory layout, which not only has disjoint processor subgrids, but also redundant data in each subgrid.

The major advantage that Pthreads holds over OpenMP is explicit control of the number of threads created and their affinity. This is important, as the default thread mapping on a system may not result in the desired allocation of threads to sockets, as detailed in Section 3.2.

For multicore chips, Pthreads is also usually preferable to MPI due to its lower memory overhead. There are two main reasons for this reduced memory footprint. First, as we see in Figure 3.2, the shared memory model of the 2D grid in (a) does not require data replication, while the distributed memory model in (b) requires additional ghost cells to store needed data from other subgrids. While the 2D grid shown in (b) already has a non-trivial proportion of ghost cells, the problem becomes even worse with 3D grids due to the increased surface-to-volume ratio. Thus, the need for ghost cells certainly increases the memory requirements for MPI over Pthreads. Moreover, managing Pthreads requires fewer system resources and less state information than heavyweight MPI processes. Consequently, Pthreads typically performs better than MPI in these scenarios.

3.3.2 OpenMP

OpenMP is an Application Programming Interface (API) that can be inserted into C or Fortran code to create shared memory parallelism. From the programmer's standpoint, OpenMP is relatively easy to use because the compiler handles much of the parallelism. The most fundamental OpenMP construct is the *parallel region*. By using compiler directives to declare that a certain region of code should be parallelized, the compiler will create a certain number of OpenMP threads to execute the code.

For our purposes, OpenMP has two major drawbacks, neither of which we were able to fully address. The first is lack of complete control over the number of threads run in a parallel region. Usually, the number of threads created can be controlled by the programmer, but it is possible that the compiler sets the thread count, and this default thread count can vary depending on the OpenMP implementation [39]. The OpenMP model believes that the compiler should make the final decision on thread count, to the dismay of experienced programmers.

The second, and possibly larger problem with OpenMP is the lack of locality control. When OpenMP threads are spawned for a parallel region, the programmer has no knowledge or control over where in the multicore chip each of these threads is actually running. Ideally, the mapping of subgrids to threads should be such that the number of cells requiring data from remote sockets is minimized. However, depending on how the OpenMP compiler maps subgrids to threads, a significant amount of needless inter-socket communication may occur, creating large performance penalties.

While OpenMP is simpler to use than either Pthreads or MPI, we decided not to use it because it ceded too much control to the compiler. The programmer has no control over thread affinity and not enough control over thread count. These are critical parameters if we wish to get the best possible performance across a wide variety of multicore platforms.

3.3.3 MPI

MPI is a message passing library that, combined with C or Fortran, has become the de facto standard for distributed memory programming [46]. It is certainly

portable and relatively efficient, but it does have significant drawbacks when used on multicore machines. As seen in Figure 3.2, distributed memory models like MPI require extra ghost cells in order to perform local calculations. This adds extra memory overhead. Moreover, MPI makes it very difficult to perform optimizations like *thread blocking*, which we describe in Section 4.1.2. Essentially, thread blocking performs a block-cyclic decomposition of the grid, so that each thread or process will receive multiple subgrids from different locations in the overall grid. This optimization allows architectures like Victoria Falls, which supports 8 hardware threads per core, to gain a performance benefit using Pthreads. However, due to the disjoint memory spaces of MPI processes, we would again require extra ghost cells around each subgrid, making this optimization all but impossible.

Another issue with MPI is that its processes are fairly heavyweight, requiring non-trivial amounts of information about both program resources and execution state. In addition, it often creates system buffers to store messages that are about to be sent or have just been received. An MPI process running on every core of a multicore machine would consume a non-trivial amount of system resources. This problem is further exacerbated on manycore architectures or multicore architectures that support multiple hardware threads on a core.

Finally, for most people, MPI is harder to program in than either Pthreads or OpenMP due to the explicit communication commands that are required. This in itself is not the reason we avoided MPI– due to the reasons cited above, we felt that it was unlikely to achieve the performance of Pthreads on multicore architectures. However, if everything else were equal, programmer productivity would be a factor in our decision making.

3.4 Programming Languages

The advantage of Pthreads over other programming models is substantial enough that our choice of programming language was dictated by it. The two programming languages that we considered were C and Fortran. From a programmability standpoint, Fortran is a better fit for stencil codes because of its support for true multidimensional array indexing. However, there are only a few Pthreads implemen-

tations for Fortran [23, 25], and they are not widely available, so we chose C instead. At a secondary level, C also allows for pointer manipulation, which we used when executing multiple sweeps over a grid.

3.5 Compilers

The choice of compiler plays a significant role in the effectiveness of certain optimizations as well as overall performance. Our goal was to choose the compiler that, given appropriate compiler flags, would generate the fastest code. The chosen compilers are listed in the last row of Table 3.1. For the x86 platforms, we found that `icc` generates superior code to `gcc`, since it can often automatically perform SIMDization (discussed in Section 4.4.2) and register blocking (detailed in Section 4.4.1). In fact, for the compute-intensive 3D 27-point stencil, we observed up to a 23% improvement from using `icc` instead of `gcc` [13]. As a result, we have used the `icc` compiler across all our x86 machines, including the AMD Barcelona. However, the more bandwidth-limited the kernel, the less `icc`'s advantage. For example, the memory-intensive 3D 7-point kernel showed minimal, if any, performance difference between the two compilers.

Now we shift our attention to the non-x86 architectures. On the Blue Gene/P, IBM's `xlc` compiler was really the only choice available, so we utilized it. Finally, we did have a choice between `gcc` and `suncc` on the Sun Victoria Falls architecture, but we found that `gcc` usually compiled faster and generated better code.

For each platform, we read through the list of available optimization flags for our chosen compiler and selected the ones that were best suited to stencil codes. There was minimal exploration of these flags, however. This is an area of future work.

3.6 Performance Measurement

For all our collected results, we report the median performance over five trials to ensure consistency. Between each trial, we attempt to flush the cache by performing a simple vector operation over an array that is larger than the size of the cache. This prevents later trials from speeding up due to a warm cache. We have observed that

Unit	Respects time to solution	Problem size-independent	Kernel-independent
Seconds	✓	—	—
GFlop/s	Depends on kernel	✓	✓
GStencil/s	✓	✓	—

Table 3.2: A comparison of different possible units for reporting stencil performance results. Ideally, we would like a unit that is both problem size and kernel independent, while respecting the time to solution.

the Blue Gene/P are very reproducible, with all five trials typically falling within 1% of the median value. The x86 architectures also show little variation, as most of the trials fall within 2% of the median. The Victoria Falls results did result in more outliers, but three of the five trials were usually within a few percent of the median. Generally speaking, the Blue Gene/P and the x86 architectures were very consistent. Victoria Falls results were less reproducible, but by reporting the median value, this effect was somewhat mitigated.

When choosing the best metric for presenting our performance results, we sought a unit that would be independent of architecture, problem size, and kernel, while also respecting the time to solution. The units that we considered are shown in Table 3.2. Given that time is our ultimate metric of performance, "seconds" could be an appropriate unit. Upon closer examination, though, we see that it is dependent on both problem size and kernel; this precludes us from making broader statements about stencil code performance.

Another potential metric is "GFlop/s" (billions of floating point operations per second). This seems like a good metric, especially given that most high performance computing (HPC) results are presented this way. There are many benefits of this unit– it is problem size independent, kernel independent, and we can compare the results directly to the peak computational rate of the machine. This is, in fact, likely the best unit for presenting the 7-point stencil and Helmholtz kernel results. However, for the 27-point stencil results, the common subexpression elimination (CSE) optimization (from Section 4.4.3) causes a major problem. CSE alters the number of flops performed per grid point, and thus it makes it possible to hasten the time to

solution while *decreasing* the GFlop rate. Thus, the time to solution is not respected.

The final unit from Table 3.2, "GStencil/s" (billions of stencils per second), does respect the time to solution in all cases. Moreover, it is also independent of problem size. However, unlike the GFlop rate, it does not allow for direct comparisons between different kernels (like the 7-point and 27-point stencils).

Ultimately, we decided that for the Helmholtz kernel results in Chapter 9, the best unit would be GFlop/s, since we do not perform CSE for this kernel. For the 3D 27-point stencil results in in Chapter 8, we instead use GStencil/s, since it is vital that our metric respect the time to solution. Similarly, the 3D 7-point stencil results in Chapter 7 are also presented in GStencil/s, given that the chapter is laid out analogously to the 27-point stencil chapter. This is despite the fact that we do not perform the CSE optimization for the 7-point stencil.

We will also compare our performance results to machine performance limits, including the percentage of peak attainable bandwidth or in-cache performance. These metrics allow us to understand how much of the platform's resources we are utilizing. However, these units are not designed to make raw performance comparisons between machines.

3.7 Summary

Thus far, we have established our experimental setup, which has included hardware, software, and data collection. Still, this is only a prelude to the actual stencil experiments, results, and analysis, which will be presented in the following chapters.

Chapter 4

Stencil Code and Data Structure Transformations

In this chapter, we introduce source-level optimizations that are designed specifically for stencil codes. These optimizations can roughly be divided into four categories:

1. In Section 4.1, we discuss how we perform a hierarchical decomposition of the stencil problem. This serves multiple purposes. First and foremost, by breaking down the problem into subgrids that can be mapped onto threads, the problem is then parallelized for running on our multicore platforms. Moreover, by properly choosing the size of these subgrids, we can also achieve better cache locality. Finally, by performing unroll-and-jam within the inner loop, we can make the best use of our available registers and functional units.

2. Section 4.2 considers the question of where to place the grid data for best performance. The largest concern in this regard is ensuring that the needed grid data is on local DRAM, and we specify how we do this. We also discuss how we alter the data layout in cache so that conflict misses are minimized.

3. Stencil codes are often bound by memory bandwidth, so in Section 4.3 we introduce two methods to deal with this problem. One optimization increases our effective memory bandwidth, while the other drastically reduces the amount of memory traffic needed for out-of-place stencil calculations.

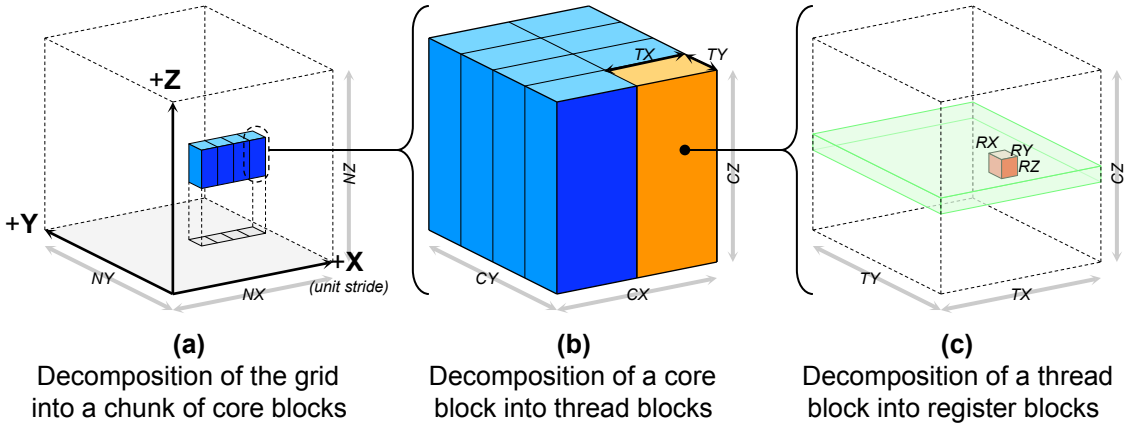


Figure 4.1: A hierarchical decomposition of the grid in order to preserve locality at each level of the memory hierarchy. This diagram is courtesy of Samuel Webb Williams.

4. Finally, for compute-intensive stencils or platforms with sufficient bandwidth, we address the problem of maximizing in-core performance in Section 4.4. The optimizations in this section include loop unrolling in different dimensions, exploiting short vector instructions, and reducing the flop count by eliminating redundant computations.

While the *register blocking* optimization is mentioned in two of these categories, the rest are placed into the one that we thought was most applicable. These optimizations are also discussed in [12] and [13].

4.1 Problem Decomposition

In Sections 3.3 and 3.4, we have already discussed why we have chosen C with Pthreads as our programming model. However, we have not discussed what optimizations would be applied nor how the stencil problem would be partitioned so as to run on multicore architectures. To this end, we have written a variety of stencil code generators that feature a three-level geometric decomposition strategy, as visualized in Figure 4.1. This multi-level decomposition simultaneously implements parallelization, NUMA-aware allocation, cache blocking, and loop unrolling, yet does

not change the original grid data structure. Note that the nature of out-of-place stencil iterations implies that all blocks are independent and can be computed in any order. This greatly facilitates parallelization, but the size of the resultant search space does complicate tuning.

4.1.1 Core Blocking

As pictured in Figure 4.1(a), the coarsest decomposition that we perform is *core blocking*. The entire grid of size $NX \times NY \times NZ$ is partitioned in all three dimensions into smaller *core blocks* of size $CX \times CY \times CZ$, where X is the unit stride dimension, and the strides increase monotonically in the Y and Z directions. The dimensions of the core blocks can then be tuned for best performance.

The introduction of core blocks serves two main purposes. First, it parallelizes the problem, so that each core (or hardware thread when multiple threads are supported per core) can process its share of core blocks. In order to get the best performance, the number of core blocks is set to be a multiple of the number of cores. This is possible because the grid dimensions and the core counts are always constrained to be a power of 2; this allows the grid to be divided equally so that none of the cores are left idle. Second, core blocks also facilitate cache blocking. If we reduce the size of a core block, we can change our traversal through the grid so that several adjacent planes are kept in cache concurrently. This will allow us to get maximal data reuse before the data is evicted from cache. In terms of the three C's model of classifying cache misses [24], cache blocking helps in reducing *capacity* misses.

While smaller core blocks may result in fewer cache capacity misses, they may also cause NUMA issues that will be discussed shortly. Therefore, we group adjacent core blocks together into *chunks* of size *ChunkSize*, where all the core blocks within a chunk are processed by the same subset of threads. In Figure 4.1(a), for instance, *ChunkSize* equals 4. Each of these chunks is then assigned to the cores in a round-robin fashion.

There are competing forces involved in the best choice of *ChunkSize*. When *ChunkSize* is large, the chunk will occupy a longer contiguous set of memory addresses. Therefore, there is a good chance that different data needed by multiple

threads will get mapped to the same set in cache, causing a *conflict* miss (in the three C's model of cache misses [24]). The positive aspect of having a large contiguous set of memory is that there are fewer memory pages that will have data from multiple sockets. This should reduce inter-socket communication, and therefore diminish NUMA effects.

In contrast, when *ChunkSize* is small, there is a lower chance that memory addresses will be mapped to the same set in cache, and therefore fewer conflict misses. The drawback of having fewer core blocks in a chunk is that the likelihood of a memory page having data from two different sockets is increased. Consequently, NUMA effects are magnified. We therefore tune *ChunkSize* to find the best tradeoff of these two competing effects.

4.1.2 Thread Blocking

For architectures where multiple hardware threads per core are supported, namely the Nehalem and Niagara2 machines in our study, a middle level of decomposition may provide additional benefit. This second decomposition level further partitions each core block into a series of *thread blocks* of size $TX \times TY \times CZ$, as shown in Figure 4.1(b). Core blocks and thread blocks are the same size in the Z (least unit stride) dimension– this is the dimension that is least important for preserving locality, so it was left unchanged. As a result, when $TX = CX$ and $TY = CY$, there is only one thread per core block.

The main benefit of thread blocking stems from the fact that stencil calculations require data not just from the point that is being written to, but from adjacent points as well. We can localize a given core's computation by having all of its hardware threads process the same core block. Consequently, when one thread needs neighboring data from outside its own thread block, there is a good chance that the adjacent thread block is being computed by a thread on the same core. This data sharing between threads helps reduce capacity misses in the core's shared caches.

This optimization could potentially benefit both the Nehalem and Niagara2 machines, but for this study it was only applied to the latter. This is because the Niagara2 has many more threads per core (8 versus Nehalem's 2) as well as a much

smaller L1 cache (8 KB versus Nehalem's 32 KB). This combination makes it likely that thread blocking can provide a performance boost on the Niagara2, but it is much less clear whether it will help on Nehalem.

There are two concerns about thread blocking that the programmer needs to address. First, since the data sharing among a core's threads is critical, it is important to keep the threads in-sync. If the work among threads is not properly load balanced, or if the threads run for a long time without any synchronization, then some of the threads may get ahead of the others, causing a loss of data sharing among threads. In our case, we made sure that each hardware thread on a machine received the same number of grid points to process by setting dimensions and thread counts to be powers of 2. However, we only performed synchronization at the end of the entire computation; there was no additional synchronization after a certain number of core blocks had been completed. Despite the cost of synchronization, this might provide better performance and is a topic of future work.

The second concern resulting from thread blocking is the additional parameters involved and the resultant increase in the search space size. Specifically, on Niagara2, the number of thread blocks per core block was fixed to be 8, since the architecture supports 8 threads per core. Accordingly, the size of the parameter space without thread blocking will be multiplied by the number of (TX, TY) pairs that fall under this constraint. In order to keep the search space tractable, both TX and TY are powers of 2. For our study, we had the luxury of searching through all valid combinations of TX and TY, although the search time was significantly longer. The searched parameter spaces will be discussed further in Section 7.2.

4.1.3 Register Blocking

Figure 4.1(c) shows the final decomposition, which partitions each thread block into *register blocks* of size $RX \times RY \times RZ$. The dimensions of the register block indicate how many times the inner loop has been unrolled and jammed (see Figure 4.2) in each of the three dimensions. By tuning for the best register block size, we can make the best use of the system's available registers and functional units. This optimization will be discussed in further detail in Section 4.4.

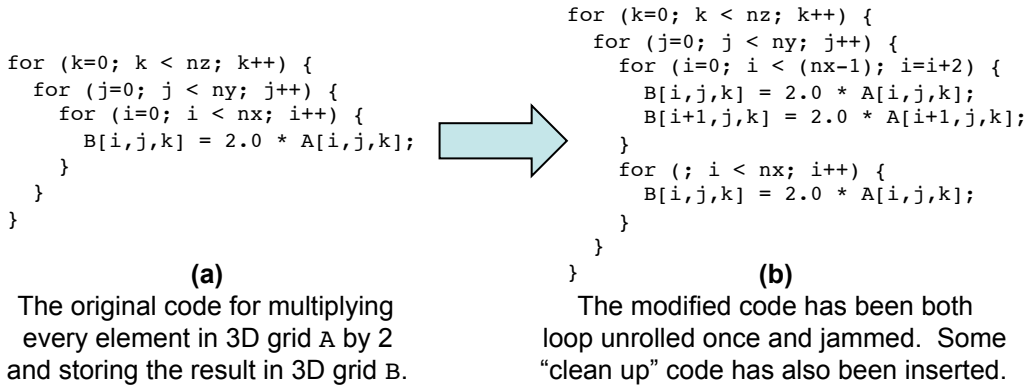


Figure 4.2: Code showing the loop unroll and jam transformation. Both versions of the code are equivalent, but (b) may make better use of registers and functional units. While the code shown is only loop unrolled in the X-dimension, it is possible to unroll it in any or all of the three dimensions. Unroll and jam is used in the *register blocking* optimization.

4.2 Data Allocation

The previous section, and specifically Figure 4.1, have detailed which threads will be processing which parts of the stencil grid. However, the layout of the grid array, both in DRAM and in cache, can greatly affect performance. For instance, on NUMA architectures, the data needed by a given socket may be located on a different socket's DRAM. However, accessing remote DRAM is typically a very expensive operation. To address this problem, we implemented a *NUMA-aware allocation* that minimizes inter-socket communication and maximizes memory controller utilization.

A second data layout issue arises when a given thread needs two data points that both map to the same set in cache. Depending on the cache associativity and replacement policy, one of these points may be evicted, causing a conflict miss. Unfortunately, the evicted point will need to be retrieved from DRAM again, requiring a much longer access time and causing more memory traffic. This motivates our use of *array padding*.

4.2.1 NUMA-Aware Allocation

Our stencil code allocates the source and destination grids as separate large arrays. On NUMA systems that implement a "first touch" page mapping policy, a memory

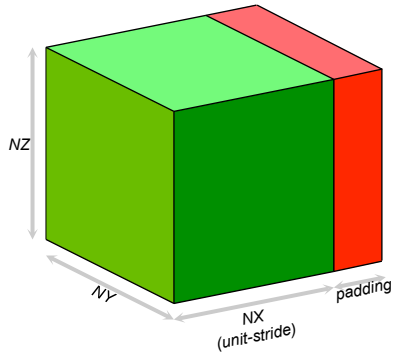


Figure 4.3: A visual representation of the array padding data structure transformation. By adding a tuned *padding* amount to the unit-stride dimension of the grid, the memory layout of the grid can be altered so as to minimize cache conflict misses.

page will be mapped to the socket where it is initialized. Naïvely, if we let a single thread fully initialize both arrays, then all the memory pages containing those arrays will be mapped to the socket that particular thread is running on. Subsequently, if we used threads across multiple sockets to perform array computations, the threads not on the same socket as the initializing thread would perform expensive inter-socket communication to retrieve their needed data.

Fortunately, these NUMA issues can be sidestepped. Since our decomposition strategy has deterministically specified which thread will update each array point, a better alternative is to let each thread initialize the points that it will later be processing. This *NUMA-aware allocation* correctly pins data to the socket tasked to update it. This optimization is only expected to help when we scale from one socket to multiple sockets, but without it, performance on memory-bound architectures could easily be cut in half.

4.2.2 Array Padding

The second data allocation optimization that we utilized is *array padding*. Some architectures have relatively low associativity shared caches, at least when compared to the product of threads and cache lines required by the stencil. On such computers, conflict misses can cause the eviction of needed data, thereby adding to the overall memory traffic and causing significant memory latency. To avoid these pitfalls, we

pad the unit-stride dimension of our arrays ($NX \leftarrow NX + padding$), as shown in Figure 4.3. The padded portion of the array is not computed over – it is merely used to alter the memory layout and avoid conflict misses. This array padding optimization is the only data structure change that we consider in our study.

4.3 Bandwidth Optimizations

For stencils with low arithmetic intensities, the 7-point stencil being the most obvious, memory bandwidth is a valuable resource that needs to be managed effectively. We therefore introduce *software prefetching* to hide memory latency (and thereby increase effective memory bandwidth) and the *cache bypass* instruction to dramatically reduce overall memory traffic.

4.3.1 Software Prefetching

The x86 architectures have hardware stream prefetchers that can recognize both unit-stride and strided memory access patterns. When these patterns are detected, successive cache lines are fetched without first being demanded. However, hardware prefetchers have two major drawbacks. First, they will not cross memory page boundaries. For machines with 4 KB pages, this can be as little as 512 consecutive doubles. Moreover, hardware prefetchers may produce extraneous cache line requests when there are discontinuities in the memory access pattern, such as if the core blocks divide the grid in the unit-stride dimension.

In contrast, *software prefetching*, which is available on all cache-based processors, is not precluded by either limitation. However, it only requests a single cache line at a time. In our case, we tune for two software prefetching parameters. First, by changing the number of software prefetch requests, we vary the number of cache lines to be retrieved. Additionally, we also adjust the proper look-ahead distance in order to effectively hide memory latency. If both parameters are properly tuned, the effective memory bandwidth of the machine can be increased.

4.3.2 Cache Bypass

On write-allocate architectures, a write miss will result in the allocation of a cache line and a read from main memory to populate its contents. However, for stencil codes performing Jacobi (out-of-place) iterations, we are only concerned with the write value being written back to DRAM; the read itself is unnecessary since that data is left unused. Therefore, in SSE (discussed further in Section 4.4.2) we can use the *movntpd* instruction to get rid of these extraneous reads. For Jacobi iterations using two grids, we can eliminate $\frac{1}{3}$ of the overall memory traffic by removing this cache line allocation. This corresponds to a 50% improvement in arithmetic intensity, which, if we are bandwidth bound, can also increase performance by up to 50%! Unfortunately, this optimization is not supported on either Blue Gene/P or Niagara2.

4.4 In-core Optimizations

For stencils with higher arithmetic intensities, of which the 27-point stencil is a good example, computation can often become a bottleneck. To address this issue, we perform *register blocking and reordering* to effectively utilize a given platform's registers and functional units. On the x86 architectures and Blue Gene/P, we also perform *explicit SIMDization* to achieve better computational performance. Finally, across all architectures, we can perform *common subexpression elimination* to reduce the total flop count.

4.4.1 Register Blocking and Instruction Reordering

After tuning for bandwidth and memory traffic, it often helps to explore the space of inner loop transformations to find the fastest possible code. One such transformation is unroll and jam, which is explained in Figure 4.2. Essentially, unroll and jam allows multiple points to be concurrently executed in the inner loop, possibly allowing for better utilization of registers and functional units. *Register blocking* takes unroll and jam one step further by applying it in each of the three spatial dimensions. Therefore, the dimensions of a register block (RX, RY, and RZ, shown in Figure 4.1(c)) are precisely the loop unroll factors in the X, Y, and Z dimen-

sions. Compilers might make this transformation automatically, but if the heuristic to decide the best register block size is poor, then the quality of the generated code will also suffer. Our code generator needs to be portable, so we do not rely on any compiler to do register blocking well.

For stencil codes, having larger register blocks will typically reduce the total number of loads performed. This is because stencil computations require data from adjacent points in addition to the point being written to. Hence, the points surrounding the register block are also needed in order to properly compute each point inside the block. As the size of the register block grows, the ratio of points around the surface of the register block to the points in the register block (the surface-to-volume ratio) drops. As a result, larger blocks will, on average, need to load a given point less often than smaller blocks.

However, having a register block the size of the entire grid is non-sensical due to register pressure. Specifically, when there are more live variables in the inner loop than available registers, some of the data is "spilled" to cache. Accessing cache memory is significantly more expensive than accessing registers, so this will cause a serious performance penalty.

Our code generator can create code for arbitrary size register blocks, as well as perform limited instruction reordering. These register blocks then sweep through each thread block. As will be discussed in Chapter 5, register blocking is an optimization that will be incorporated into our stencil auto-tuner. However, since register blocking requires actual code generation, rather than a simple runtime parameter, it will increase the size of the code and the resultant executable. Still, this should be a worthwhile tradeoff, since we can make the best use of the available registers and functional units while avoiding register spills.

4.4.2 SIMDization

The x86 platforms and Blue Gene/P both support Single Instruction Multiple Data (SIMD) instructions, which allows a single instruction to process multiple double precision words simultaneously. These double precision words are usually stored in short, typically 128-bit vector registers. For compute-intensive codes, this can

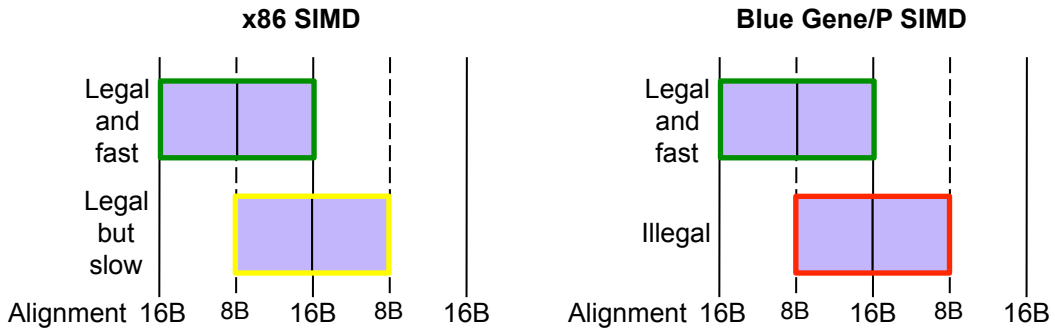


Figure 4.4: A visualization of the legality and performance of SIMD loads on x86 and Blue Gene/P. Both architectures use 128-bit registers, which can store two double precision words (shown as lavender squares). On both architectures, 16 Byte-aligned loads are legal and fast. However, for unaligned (8 Byte-aligned) loads, x86 can perform them, albeit slowly, while Blue Gene/P doesn't allow them at all.

often achieve better computational performance.

Some compilers, but not all, can take the portable C code produced by our code generator and automatically SIMDize it. In order to avoid depending on the compiler, we created several instruction set architecture (ISA) specific variants that produce *explicitly SIMDized* code. This was done through the use of *intrinsics*, which are C language commands that get expanded by the compiler into inlined assembly instructions. On x86, we used Streaming SIMD Extensions 2 (SSE2) intrinsics [50], while for Blue Gene/P, double hummer intrinsics [48] were employed. If the compiler fails to SIMDize the code, these versions can deliver significantly better in-core performance. As one might expect, though, this transformation will have a limited benefit on memory-bound stencils like the 7-point.

In order to use these SIMD intrinsics properly, we needed to ensure that our double-precision data was properly aligned. For both x86 and Blue Gene/P, the best way to populate the 128-bit registers is by loading data that is aligned to a 16-Byte boundary in memory. These loads will always be legal and relatively fast. On x86 architectures, it is also legal to load from data aligned to an 8-Byte boundary, but these unaligned loads will be slower. On Blue Gene/P, unaligned loads are not even legal. These different cases are diagrammed in Figure 4.4. Due to alignment issues associated with SIMD, the array padding optimization discussed in Section 4.2.2 is

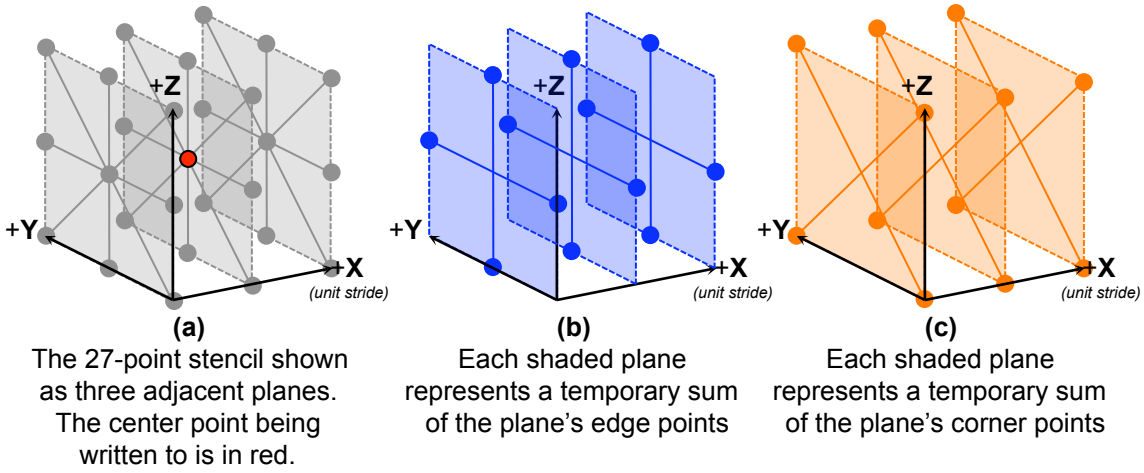


Figure 4.5: A graphic representation of common subexpression elimination for the 27-point stencil. The `sum_edges_*` and `sum_corners_*` temporary variables from the code in Figure 4.6 are visualized in (b) and (c), respectively.

not utilized for SIMD code; the padding we perform is for proper alignment only.

From a programmer productivity standpoint, SIMDization is highly unproductive, as it requires a complete rewrite of the code. Moreover, SIMD code is not portable; SSE code will only run on x86 architectures, while double hummer code will run exclusively on Blue Gene/P. As a result, we wrote separate code generators for both x86 and Blue Gene/P.

4.4.3 Common Subexpression Elimination

Our final optimization involves identifying and removing common expressions within a register block. This *common subexpression elimination* (CSE) reduces or eliminates redundant flops, but may produce results that are not bit-wise equivalent to the original implementation due to the non-associativity of floating point operations.

For the 3D 7-point stencil, there is very little opportunity to identify and eliminate common subexpressions. Moreover, the 7-point stencil is usually bandwidth-limited, so reducing the flop count may be of little value anyway. Hence, this optimization was not performed, and 8 flops are always performed for every point.

The 3D 27-point stencil, however, presents such an opportunity. Consider Fig-

ure 4.5(a). The naïve 27-point stencil implementation will perform 30 flops per stencil. However, as we iterate through the X-dimension within a register block, we can create several temporary variables of sums that will be reused, as seen in Figures 4.5(b) and (c). Actual code for this transformation is shown in Figure 4.6 for the case where the register block's X-dimension (parameter RX in Figure 4.1(c)) equals 2. The left panel, displaying naïve code, performs 30 flops per point. However, it contains redundant computation that can be removed. For instance, the value of the variable `sum_corners_1` is computed twice in the left panel. By computing it only once in the right panel, and then using that value in multiple places, the right panel performs 29 flops per point. As RX becomes larger, more redundant computation is eliminated, and thus the flops per point drops, with an asymptotic lower limit of 18 flops/point. However, as stated in Section 4.4.1, if the register block becomes too large, then register spills become an issue.

Disappointingly, neither the `gcc` nor `icc` compilers were able to apply CSE automatically for the 27-point stencil. This was confirmed by examining the assembly code generated by both compilers; neither one reduced the flop count from 30 flops/point. As seen in Figure 4.6, the common subexpressions only arise when the loop unroll factor is at least two. Thus, a compiler must loop unroll before trying to identify common subexpressions. However, even when we *explicitly* loop unrolled the stencil code, neither compiler was able to exploit CSE. It is unclear whether the compilers lacked sufficient information to perform this algorithmic transformation or are simply not capable of doing so.

4.5 Summary

Thus far, we have laid out the stencil-specific optimizations that will be used to tune stencil performance in later chapters. However, we have not yet mentioned how to combine these optimizations into a stencil auto-tuner, nor what parameter values these optimizations can take. These will be discussed in the following chapter.

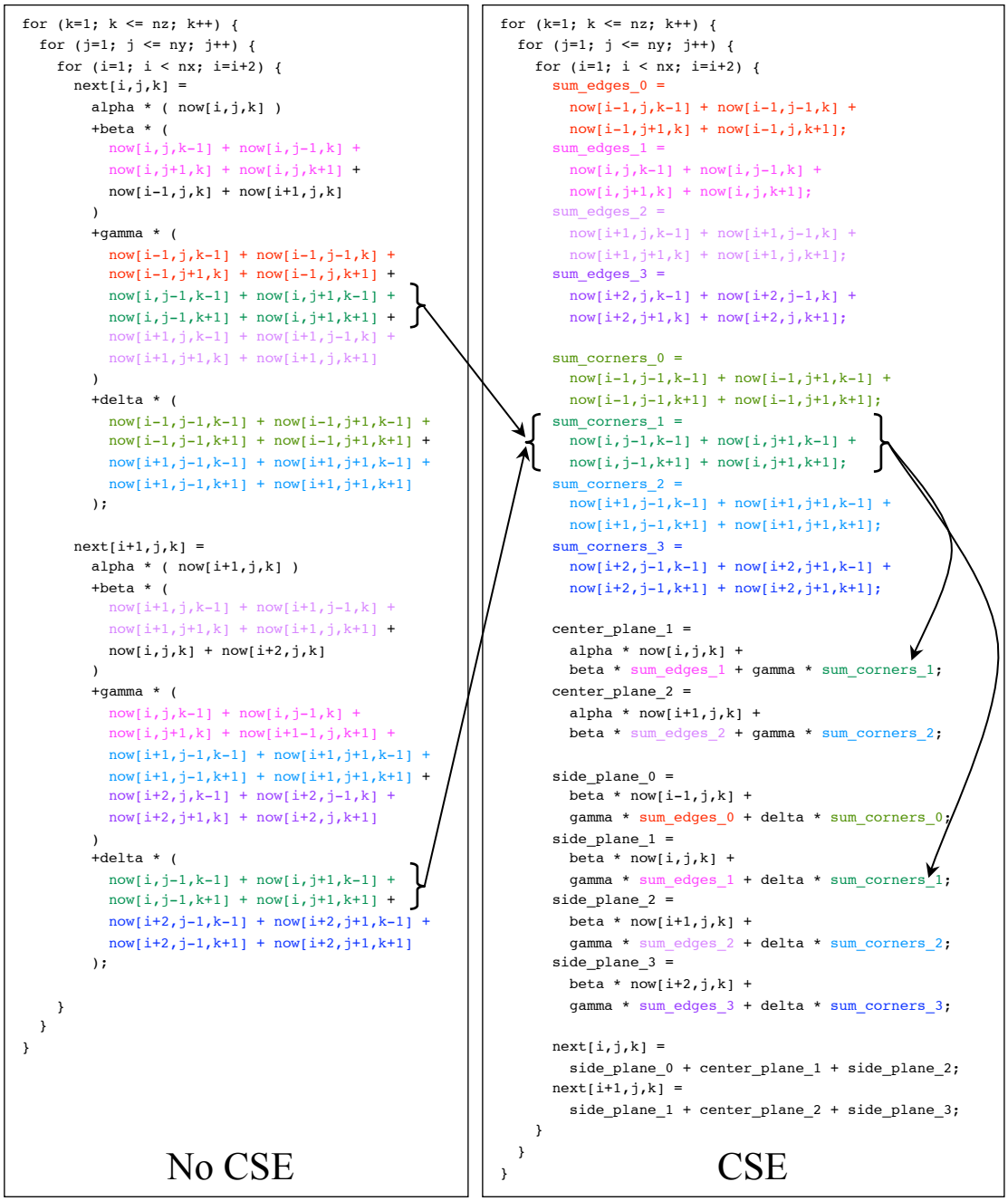


Figure 4.6: Code for a single grid sweep using a 27-point stencil. Both panels have code that has been loop unrolled once in the unit-stride (X) dimension. However, the left panel does not exploit common subexpression elimination (CSE), while the right panel does. For instance, the value of the variable `sum_corners_1` is computed twice in the left panel, but only once in the right panel. The variables `sum_edges_*` in the right panel are graphically displayed in Figure 4.5(b), while the variables `sum_corners_*` are shown in Figure 4.5(c).

Chapter 5

Stencil Auto-Tuning

In Chapter 4, we introduced many transformations for improving the performance of stencil codes. In order to take full advantage of these optimizations, we developed an automatic tuning (or *auto-tuning*) environment [12] that combines them so as to achieve good performance.

This chapter delves into more detail about our automatic tuner. We begin by introducing the concept of auto-tuning and explaining its usefulness in Section 5.1. We then talk about why domain-specific auto-tuners are preferable to general-purpose compilers in Section 5.2.

However, there are challenges to creating an auto-tuner. In Section 5.3, for instance, we discuss the problem of taking domain-specific optimizations (like the ones introduced in Chapter 4) and creating actual code. Our solution was to develop PERL code generators that produce C code variants encompassing our stencil optimizations. A second challenge is dealing with parameter space issues– how to select the specific parameters for each optimization, whether to tune offline or online, and how to find a high-performing parameter configuration from a vast configuration space. Section 5.4 details each of these topics. This approach allows us to achieve high performance across significantly varying architectures.

5.1 Auto-tuning Overview

Now that we have introduced stencil-specific optimizations in Chapter 4, we need to ask ourselves two questions. First, do we even need to tune our stencil codes for performance on multicore architectures? We will show in Chapters 7–9 that the answer is almost always yes. Then, as this is the case, how do we effectively utilize these optimizations? This is not an easy task. The specific stencil kernel, the problem size, and the choice of platform can all affect which optimizations, or specific optimization parameters, will be effective. From a productivity standpoint, it is non-sensical to create a distinct hand-tuned stencil code given the breadth of architectures, stencil kernels, and problem sizes. A better solution is to build an automatic tuner, or *auto-tuner*, so as to achieve performance portability.

The programmer does need to expend a non-trivial one-time cost to build an auto-tuner, but this cost is easily recovered in the portability the auto-tuner offers. Not only does automatic tuning allow us to run well on today's multicore platforms, it should also allow us to execute our stencil code efficiently on future architectures that support similar programming models. Moreover, even if we change the stencil kernel or the grid size, we should still be able to find a good parameter configuration for the new problem.

The concept of auto-tuning is not new. The current production-quality auto-tuners include FFTW [20] and SPIRAL [41] for spectral methods and PHiPAC [**?**] and OSKI [54] for linear algebra. More recently, the FLAME project [18] and the thesis of S. Williams [59] have designed auto-tuners with multicore platforms in mind– the former in the area of dense linear algebra, the latter for several different kernels.

These auto-tuners, and auto-tuners in general, are created through three basic steps. First, the programmer needs to determine which domain-specific code transformations are legal and potentially useful. This may be performed in conjunction with an application scientist. Once these optimizations are enumerated, the second question is how to generate the code corresponding to these optimizations. This can be difficult, especially given the difficulty of combining certain optimizations, and the possible need for correctness checking. Finally, we need to explore the resulting

parameter space efficiently so as to find a high-performing configuration.

We will be discussing each of these steps in turn.

5.2 Auto-tuners vs. General-Purpose Compilers

The first step in building an auto-tuner is the enumeration of the optimization space, which, for the case of stencils, we already discussed in Chapter 4. Currently, almost all auto-tuners are specific to one "motif" (in the parlance of the Berkeley View [2]) in order to keep the potential transformations tractable. In our case, we were able to identify relevant stencil transformations through a combination of architectural and domain-specific knowledge. This knowledge was then incorporated into our auto-tuner, thereby creating a type of *domain-specific compiler*.

In contrast to our auto-tuner, a general-purpose compiler needs to be able to accept and optimize any valid code. From the list of transformations that it can verify as legal, a general-purpose compiler then has to select the ones that will be useful in stencil codes. This is a difficult problem, and one that is made harder by the fact that the compiler may not have all the information it needs. For instance, the size of the problem, which might only be known at runtime, may dictate which transformations are best. Unfortunately, the general-purpose compiler will not have this information, and thus will resort to the use of simple, but often unreliable, heuristics.

The situation is even worse when the compiler cannot check the legality of a given optimization. In all likelihood, this eliminates the more complex ISA-specific, data structure, and algorithmic transformations. All of them require broader code modifications that currently cannot be verified through program analysis. Unfortunately, these optimizations often also provide the best speedups.

5.3 Code Generation

The first major decision in creating our stencil auto-tuner is deciding how to expose the functionality to the user. Previous auto-tuning efforts like ATLAS (Automatically Tuned Linear Algebra Software, designed for dense linear algebra) [55]

Optimization	PERL Code Generator #							
	1	2	3	4	5	6	7	8
x86 SIMD		✓				✓		
BGP SIMD			✓				✓	
Thread Blocking				✓				✓
CSE					✓	✓	✓	✓

Table 5.1: This table details the optimizations covered by each of the eight PERL code generators used for tuning the 7-point and 27-point stencil kernels.

and OSKI (Optimized Sparse Kernel Interface, for sparse linear algebra) [54] have relied on libraries. In general, linear algebra is amenable to a library interface because there is a fairly limited set of common operations that are performed. These include multiplying a pair of matrices, solving a system of equations, or finding the eigenvalues of a matrix. In addition, it is fairly simple for these library routines to handle matrices of different dimensions.

In contrast, stencil codes are generally not amenable to a library interface due to the wide variety of stencil shapes, sizes, dimensionalities, and array counts. Furthermore, the numerical operations performed in the stencil can vary from application to application. Writing a separate stencil subroutine for every common stencil would be both unrealistic and impractical, since it would require a unique auto-tuner for each stencil instantiation. Moreover, specifying the points of a stencil to a library is non-trivial, especially when multiple arrays are involved. As a result, we abandoned the library interface that is common to linear algebra and opted for a more general code generator approach instead.

We now need to determine how best to generate the actual stencil code that corresponds to our stencil optimizations. To make matters simpler, some of the optimizations from Chapter 4 can be performed without code generation; they only require that a runtime parameter is passed into the auto-tuner. However, all the bandwidth and in-core optimizations, as well as the NUMA-aware optimization, do require actual code creation. In order to avoid doing painful and error-prone manual C code development, we instead developed a PERL script that creates ISA-independent C code. This code generator can create many code variants that vary parameters like

register block dimensions or prefetching types. We also wrote several specialized code generators for the drastic code changes resulting from SIMDization, thread blocking, and CSE. More specifically, x86 SIMD and Blue Gene/P SIMD intrinsics are different, so a distinct code generator was created for each one. In addition, the CSE optimization can be applied to either SIMD or thread blocked code in order to potentially increase performance. As a result, we also created code generators for SIMD or thread blocked code either with or without the CSE optimization. Eight different code generators were created in all– the optimizations encompassed by each one are listed in Figure 5.1. While developing all these code generators was likely still more productive than hand-tuning for each architecture, trying to accommodate non-portable optimizations and the CSE optimization certainly reduced programmer productivity.

It should be noted that the NUMA-aware optimization, discussed in Section 4.2.1, is unique in that it does not change the actual timed stencil functions. Rather, it alters how the grid data is initialized. In our code, this is performed through a simple C preprocessor flag. In addition, while all of the PERL code generators that we utilized can create many equivalent stencil functions, we still need the ability to access the desired code variant in order to measure its performance. To this end, we created a table of function pointers in the C code. We can choose the function we wish to run by properly indexing into this table.

Despite these code generators, the question of when different transformations can be applied, while still preserving correctness, lingers. Our stencil auto-tuner does not have any semantic knowledge of the stencil, and thus relies on the programmer to verify all code correctness via a separate DEBUG macro. However, this is neither practical nor scalable. Any new transformations will again need to be checked by the programmer before inclusion into the auto-tuning system.

On more developed auto-tuning systems like FLAME and SPIRAL, correctness checking is built in. In the case of SPIRAL, a domain-specific language called SPL represents the specific DSP (Digital Signal Processing) transform to be executed. A compiler then manipulates the abstract syntax tree (AST) of the SPL expression so as to explore different algorithms and, ultimately, generate target code. This type of system automatically performs correctness checking, since the relatively simple set

of rules followed by the compiler have all been previously verified.

Ultimately, we would also like our stencil auto-tuner to have a formal framework that similarly allows for algorithm exploration and program verification. The preliminary work of Kamil et al [26] has shown promising results in fulfilling this mission. Their work uses a domain-specific transformation and code generation framework, combined with an automated search, to replace stencil kernels with their optimized versions. The framework's front-end can parse a straightforward Fortran 95 stencil expression and generate the corresponding abstract syntax tree. A transformation engine can then take this AST and manipulate it by applying many of the auto-tuning optimizations that we introduced in Chapter 4. Finally, the backend code generation engine can again convert the auto-tuned AST back into actual code. This is akin to the work done in FLAME and SPIRAL, where auto-tuning and verification are performed hand-in-hand. However, this framework is initial work and is still considered to be proof-of-concept at this point.

Similar to how our PERL code generators are one level higher than basic hand-tuning, this type of formal stencil auto-tuning framework is one level higher than our PERL code generators. Ideally, the framework should be able to capture any new optimizations through proper manipulation of the AST and the backend code generaton engine. This would allow for all of the optimizations to be captured and verified in the same framework. As mentioned earlier, we needed to create new PERL code generators whenever drastic code changes were introduced, so the formal framework should be more productive as well. The work of Kamil et al is a promising proof of concept, but much of it still falls into future work.

5.4 Parameter Space

Now that we have identified effective stencil-specific optimizations and generated the appropriate code for each of them, we still need to identify appropriate optimization parameters, decide which optimizations will be searched online and which ones offline, and then determine how to search quickly to find a high-performing configuration.

5.4.1 Selection of Parameter Ranges

Our first decision was choosing the parameter values associated with each optimization. Optimizations such as NUMA-aware, explicit SIMDization, cache bypass, and CSE are mere booleans, indicating whether the optimization is employed or not. These were simple. However, for the other optimizations, our goal was to ensure that the optimal configuration for every benchmarked architecture was captured within the chosen range of parameter values. Given the great diversity in our set of platforms, we therefore allowed ourselves a very broad range of parameter values, which will be detailed in Tables 7.1 and 9.2.

Of course, a large range of parameters for each optimization creates a large number of total parameter configurations. We attempted to limit the amount of time spent tuning through the use of a clever search algorithm that we introduce in Section 5.4.3. Despite this, the search time required on a given architecture varied from a few minutes to several hours. For this study, we afforded ourselves the luxury of spending up to a day tuning on a single node, since large scale stencil applications may be scaled to thousands of nodes and run many times. At this level of parallelism, it is vital to ensure that the software is as efficient as possible. In addition, much of this thesis is a proof-of-concept for designing future multicore stencil auto-tuners, so it is just as important to understand how to find a good parameter configuration as it is to actually find it. Consequently, the extra tuning time was justified.

5.4.2 Online vs. Offline Tuning

For our study, all of our optimizations are applied offline, but with a complete problem specification available to the auto-tuner. For production stencil auto-tuners, a full specification of the problem may not always be needed; this really depends on how much tuning will need to be re-done each time the problem changes. However, it does make sense to determine which optimizations could be shifted offline so as to minimize the time required for runtime search. This is the approach taken by many numerical libraries, including ATLAS [55]. ATLAS attempts to provide portably optimal linear algebra software by tuning itself during installation on a given machine. It then builds the libraries based on these tuning results. Essentially, extra time is

spent benchmarking for portability during installation so that parameter searching can be avoided during runtime. This is a tradeoff that most software users are happy to make.

For stencil codes, it is logical to adopt a similar philosophy of trying to perform as much tuning offline as possible. However, this depends on how long we wish to spend performing install time tuning and how large a performance database we wish to maintain. At the most basic level, we could measure the performance of an "Optimized Stream" benchmark that could serve as one upper bound on performance. Many stencil kernels are bandwidth-bound, so if we find that we are achieving some pre-determined fraction of the Optimized Stream performance, we can then halt tuning. In order to set upper bounds on performance. we will implement an actual Optimized Stream benchmark in Section 6.3.

If we wished to do further offline benchmarking, at the expense of keeping a larger performance database, then we could also set a computational limit for certain common stencils (*e.g.* the 3D 7-point and 27-point stencils). Many sweeps of each common stencil can be benchmarked over a small problem size that fits into some pre-determined level of a core's cache. In this scenario, all the data should be read in only once, and then should stay resident in cache, thereby avoiding additional DRAM or inter-socket bandwidth. Thus, the code will likely be bound by computation. By varying the register block dimensions to determine the one that best utilizes the core's registers and functional units, we will have not only discovered the best register block dimensions for each stencil, but also another upper bound on performance. Moreover, the optimal register block dimensions for a stencil from this experiment should be the best dimensions for a grid of any size, although this advantage will only be observed when the code is computation-bound. Again, we actually do benchmark the in-cache performance of the 7-point and 27-point stencils in Section 6.4 so as to bound performance.

5.4.3 Parameter Space Search

The parameter ranges chosen for each of the stencil optimizations are shown in Tables 7.1 and 9.2. They are individually tractable, but the parameter space

generated by combining these optimizations results in a combinatorial explosion. Even if some (or many) of these parameter searches are performed offline, we would still like a clever algorithm to minimize the total time required. This section discusses various methods for dealing with this issue, including the one used in Chapters 7–9.

Exhaustive Search

The naïve approach to handling the number of generated parameter configurations is to search everything– thus, an *exhaustive search*. In the case of the GSRB Helmholtz auto-tuner, where we don't perform any SIMD, cache bypass, thread blocking, or CSE optimizations, this may be a possible, if not practical, approach. For the 7-point and 27-point stencil auto-tuners, the number of optimization configurations can quickly escalate into the millions without thread blocking, and into the billions with it. Clearly, an exhaustive search is no longer feasible– tuning on a single node would take far too long, while tuning on a supercomputer with many identical nodes would consume too many resources.

For all three kernels, a major difficulty lies in the fact that the optimizations are not independent of one another; they can often interact in subtle ways that vary from platform to platform. Hence, our auto-tuners require a clever search strategy in order to find high-performing parameter configurations quickly.

Heuristics

One way to prune the search space is through the use of *heuristics*, or simple rules based on architectural or compiler models. Many compilers use heuristics to quickly, if not precisely, tackle NP-hard problems such as register allocation and instruction scheduling [5, 8].

However, there are some drawbacks to using heuristics. While heuristics can be used as a blunt tool for shedding obviously poor optimization parameters, they are often too imprecise for predicting the best parameters. Moreover, they can typically be applied to only a subset of all the applied optimizations. Thus, further search will still be needed. In addition, if the underlying model that the heuristic is based on is flawed, then there is a good chance that the heuristic will be too.

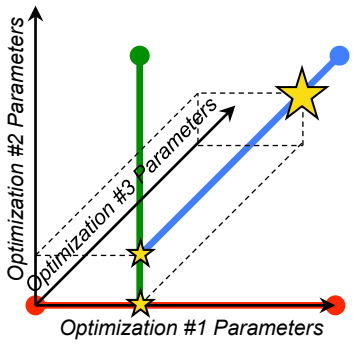


Figure 5.1: A visual representation of the iterative greedy search that we used to find the best parameter configuration. First, the optimizations were ordered. Then, we searched over all the parameter values for Optimization #1, while keeping the parameters for Optimizations #2 and #3 fixed. The best-performing parameter was set as the value for Optimization #1 (represented as a small gold star). The same is done for Optimization #2, resulting in a second small gold star. Finally, after completing the search with Optimization #3, the selected parameter configuration is shown as a large gold star.

We incorporated limited heuristics into the design of our search space. For instance, on machines with hardware prefetchers, the fact that we keep the unit-stride dimension of our core blocks undivided helps keep the search space size tractable. However, to avoid pruning any fast-performing configurations, we kept heuristics to a minimum.

Iterative Greedy Search

Our main method in finding the best configuration parameters was through an *iterative greedy search*, also commonly referred to as *hill climbing* [59]. For this search, we first fixed the order of optimizations. While there was some expert knowledge involved in the ordering of optimizations, they were generally ordered by their level of complexity. If n represents the number of optimizations in our search space, and consequently the dimension of the search space, then there are also n steps to the iterative greedy algorithm. For each optimization in our ordered list, we search over the entire parameter range of that particular optimization *while keeping all other optimization parameters fixed*. After we find the best parameter for the current optimization, we fix that value and proceed to the next optimization. After all

n steps are complete, the final configuration is the one that we select. A visual representation of this method is shown in Figure 5.1.

This iterative greedy search is essentially performing n line searches through an n-dimensional hypercube. We deem it to be "greedy" because at each step, it performs the best action locally. If all n optimizations were independent of one another, this search method would find the global performance maxima. However, due to subtle interactions between certain optimizations, this usually won't be the case. Nonetheless, we expect that it will find a high-performing set of parameters after doing a full sweep through all applicable optimizations.

In subsequent chapters, the auto-tuning will mostly be conducted using this iterative greedy search. However, as will be explained in Tables 7.2 and 9.3, applying the iterative greedy search exactly as explained above is not always feasible. For cases where different code generators are used (e.g. for SIMD, thread blocking, or CSE), slightly different rules apply which will be explained later.

Machine Learning

Machine learning is a relatively new method for doing parameter space searches quickly, and for the most part, effectively. The power of machine learning lies in the fact that, unlike the iterative greedy search, little architectural or domain knowledge is required. Typically, these algorithms collect data from a random set of configurations, which is then used to identify and exploit any relationships between the input parameters and the output metrics. Machine learning is one of the methods used in the SPIRAL project for finding the best DSP algorithms. We used a similar machine learning technique for stencil codes [22], which is discussed in more depth in Section 10.3. However, actual results will not be included in this thesis.

5.5 Summary

We have now discussed the main components of our auto-tuners, from setting individual parameter ranges, to then combining optimizations, and finally exploring the vast resultant parameter spaces. The performance results that we present in subsequent chapters will reveal how effective these auto-tuners actually are.

Chapter 6

Stencil Performance Bounds Based on the Roofline Model

Before auto-tuning the 7-point and 27-point stencils, we would like to determine approximate upper bounds on performance for each architecture. These bounds would not only serve to judge the quality of our tuning, they would also help us decide when to stop tuning. The Roofline model, described in this chapter, provides such bounds. In order to properly develop the Roofline model, we will also introduce an Optimized Stream benchmark that sets an upper bound on memory bandwidth.

6.1 Roofline Model Overview

The Roofline model [59, 58, 57] provides a visual assessment of potential performance and impediments to performance constructed using bound and bottleneck analysis. Each model is constructed using a communication-computation abstraction where data is moved from a memory to computational units. This "memory" could be registers or different levels of cache, but is typically DRAM. Computation, for our purposes, will be the floating-point datapaths. We use arithmetic intensity as a means of expressing the balance between computation and communication. Often a first order model (*e.g.* DRAM–FP) is sufficient for a given architecture for a range of similar kernels. However, for certain kernels, depending on the degree of optimization, either bandwidth from DRAM or bandwidth from the L3 cache could be the

bottleneck. For the purposes of this thesis, we will only use a DRAM–FP Roofline model.

The Roofline model defines three types of potential bottlenecks: computation, communication, and locality (arithmetic intensity). Evocative of the roofline analogy, these are labeled as ceilings and walls. The in-core ceilings (or computation bounds) are perhaps the easiest to understand. To achieve peak performance, a number of architectural features must be exploited — thread-level parallelism (*e.g.* multicore), instruction-level parallelism (*e.g.* keeping functional units busy by unrolling and jamming loops), data-level parallelism (*e.g.* SIMD), and proper instruction mix (*e.g.* balance between multiplies and adds or total use of fused multiply add). If one fails to exploit any of these (through either a failing of the compiler or programmer), the performance is diminished. We define ceilings as impenetrable impediments to improved performance without the corresponding optimization. Bandwidth ceilings are similar but are derived from incomplete expression and exploitation of memory-level parallelism. As such we often define ceilings such as no NUMA or no prefetching. Finally, locality walls represent the balance between computation and communication. For many kernels the numerator of this ratio is fixed (*i.e.* the number of floating-point operations is fixed), but the denominator varies as compulsory misses are augmented with capacity or conflict misses, as well as speculative or write allocation traffic. As these terms are progressively added they define a new arithmetic intensity and thus a new locality wall. Moreover, for each of these terms there is a corresponding optimization which must be applied (*e.g.* cache blocking for capacity misses, array padding for conflict misses, or cache bypass for write allocations) to remove this potential impediment to performance. It should be noted that the ordering of ceilings is based on the perceived abilities of compilers. Those ceilings most likely not to be addressed by a compiler are placed at the top.

6.2 Locality Bounds

As previously mentioned, the locality bounds represent the balance between computation and communication. The ideal arithmetic intensities (*i.e.* the ratio of flops performed per stencil to DRAM Bytes transferred per stencil) for the three kernels

Kernel	Ideal Arithmetic Intensity w/o Cache Bypass (in Flops/Byte)	Ideal Arithmetic Intensity with Cache Bypass (in Flops/Byte)
3D 7-Point Stencil	8/24 = 0.33	8/16 = 0.50
3D 27-Point Stencil (no CSE)	30/24 = 1.25	30/16 = 1.88
3D 27-Point Stencil (with CSE)	Varies from 0.75–1.25	Varies from 1.13–1.88
3D GSRB Helmholtz Kernel	12.5/64 = 0.20	Unnecessary

Table 6.1: This table presents the ideal arithmetic intensities for the three kernels studied in this thesis. The numerator in each fraction represents the actual number of flops performed per stencil, while the denominator is the idealized number of Bytes transferred from DRAM for each stencil. This assumes that we only incur compulsory cache misses when retrieving data from DRAM, because capacity and conflict misses have been eliminated through our other optimizations. Of course, this is very difficult to do in practice, and thus these arithmetic intensities serve as theoretical upper bounds.

studied in this thesis are displayed in Table 6.1. We deem these ratios to be "ideal" because they assume that only compulsory cache misses are occurring when we retrieve data from DRAM. While it is true that we perform optimizations like array padding and core blocking to eliminate conflict and capacity misses, respectively, in practice it is very difficult to eliminate them completely. Furthermore, without available performance counters, it is hard to know exactly how much memory traffic is being transferred. As a result, the true arithmetic intensity is also somewhat fuzzy, but we do have some intuition as to the expected range that it falls in. We will show this range when we present the Roofline models for each architecture, but the actual performance bounds will be based on the ideal arithmetic intensities shown in Table 6.1.

For the 27-point stencil with the CSE optimization, recall that the flop count per stencil can vary from 18 to 30 depending on the dimensions of the best register block. This was detailed in Section 4.4.3. Due to the wide variance in arithmetic intensity when using CSE, we will not be using the Roofline model to bound its performance. Instead, we will use the in-cache performance of the 27-point stencil kernel, which we will discuss in Section 6.4.

Finally, the Helmholtz kernel is significantly more complex than either the 3D

7-point or 27-point stencils. First, it no longer employs a simple Jacobi iteration. Instead, it uses a Gauss-Seidel Red Black (GSRB) ordering, meaning that it only writes to every other grid point during a single sweep. Thus, only half the grid points are updated during a single sweep. The Helmholtz kernel performs 25 flops for every updated grid point, so we can approximate that 12.5 flops are performed per point. Second, the memory traffic is more complex as well; the Helmholtz kernel employs six read grids, as well as one grid that is both read and written. It is true that we are still performing GSRB ordering, so only half the points are being updated. However, memory traffic is transferred in units of cache lines, so modifying half of a cache line still requires that the entire line be written back to DRAM. Consequently, there are ideally 64 Bytes of memory traffic per point. The resulting arithmetic intensity is approximately 0.20. Interestingly, the cache bypass optimization does not improve this kernel's performance since the same array is both being read and written.

The rest of this chapter will only cover the 3D 7-point and 27-point stencils. The GSRB Helmholtz kernel will be discussed further in Chapter 9.

6.3 Communication Bounds

In order to set the communication bound for the Roofline models, we created an "Optimized Stream" benchmark that addresses many of the ceilings in the Roofline model. This Optimized Stream benchmark is completely analogous to our stencil auto-tuner in that it is auto-tuned using many of the same optimizations we introduced in Chapter 4 (*e.g.* array padding, loop unrolling, software prefetching, SIMDization, and cache bypass). These optimizations are again applied using the same iterative greedy search that we discussed earlier. The main difference between the two is that instead of performing 3D stencil computations, the Optimized Stream benchmark performs floating point multiplies over varying numbers of 1D read arrays, and writes the result to a varying number of 1D write arrays. However, in order to avoid having in-core computation become a bottleneck, the number of floating point operations is kept to a minimum. In addition, the Optimized Stream benchmark never accesses the same array element twice, so all cache misses are compulsory. As such, there is no reason to avoid capacity cache misses through core blocking, so this

optimization was not utilized in the benchmark.

The best results from the Optimized Stream benchmark for a range of read and write arrays (per thread) is shown in Figure 6.1. For each platform, we used the maximum number of hardware threads available. Please note that the plots in Figure 6.1 are the maximum bandwidths over several different optimizations. The best bandwidths that we achieved when using only portable C code, SIMD code, or the cache bypass optimization can be found in Appendix A.

As expected, the plots in Figure 6.1 generally show bandwidths that are below the DRAM pin bandwidths from Table 3.1. The one exception is Blue Gene/P, whose pin bandwidth is 13.6 GB/s, even though we show bandwidths up to 21.1 GB/s. However, these impossibly high bandwidths appear only when there are no read streams present. While cache bypass is not supported on this architecture, it is possible that the compiler and/or hardware has changed the program execution so that data is not actually being written back to DRAM. This needs to be verified with performance counter data, but, of course, it is impossible to surpass a machine's hardware pin bandwidth. Fortunately, we will not being using Optimized Stream data with zero read streams to bound the performance of any of the kernels in this thesis.

The Roofline model bandwidth bounds are taken from Appendix A– specifically, the data points where a single read stream and a single write stream are present per thread. This essentially mimics the memory access pattern that we perform for the out-of-place sweeps of the 7-point and 27-point stencils. In order to convert from the units of GB/s in the Optimized Stream plots to the GFlops/s used in the Roofline model, we multiplied the bandwidth by the ideal arithmetic intensity of the kernel, which we calculated in Table 6.1. Again, for the 27-point stencil with CSE, we will not use the Roofline model to bound performance because of the resulting variability in arithmetic intensity.

We note that there is a widely distributed "Stream" benchmark that also attempts to measure the peak sustained memory bandwidth on a given system [35]. Unlike our Optimized Stream benchmark, it is not auto-tuned, so the final bandwidth numbers are gathered more quickly, but we do not expect the attained bandwidths to be as high. Table 6.2 compares the results of the Stream benchmark against our Optimized

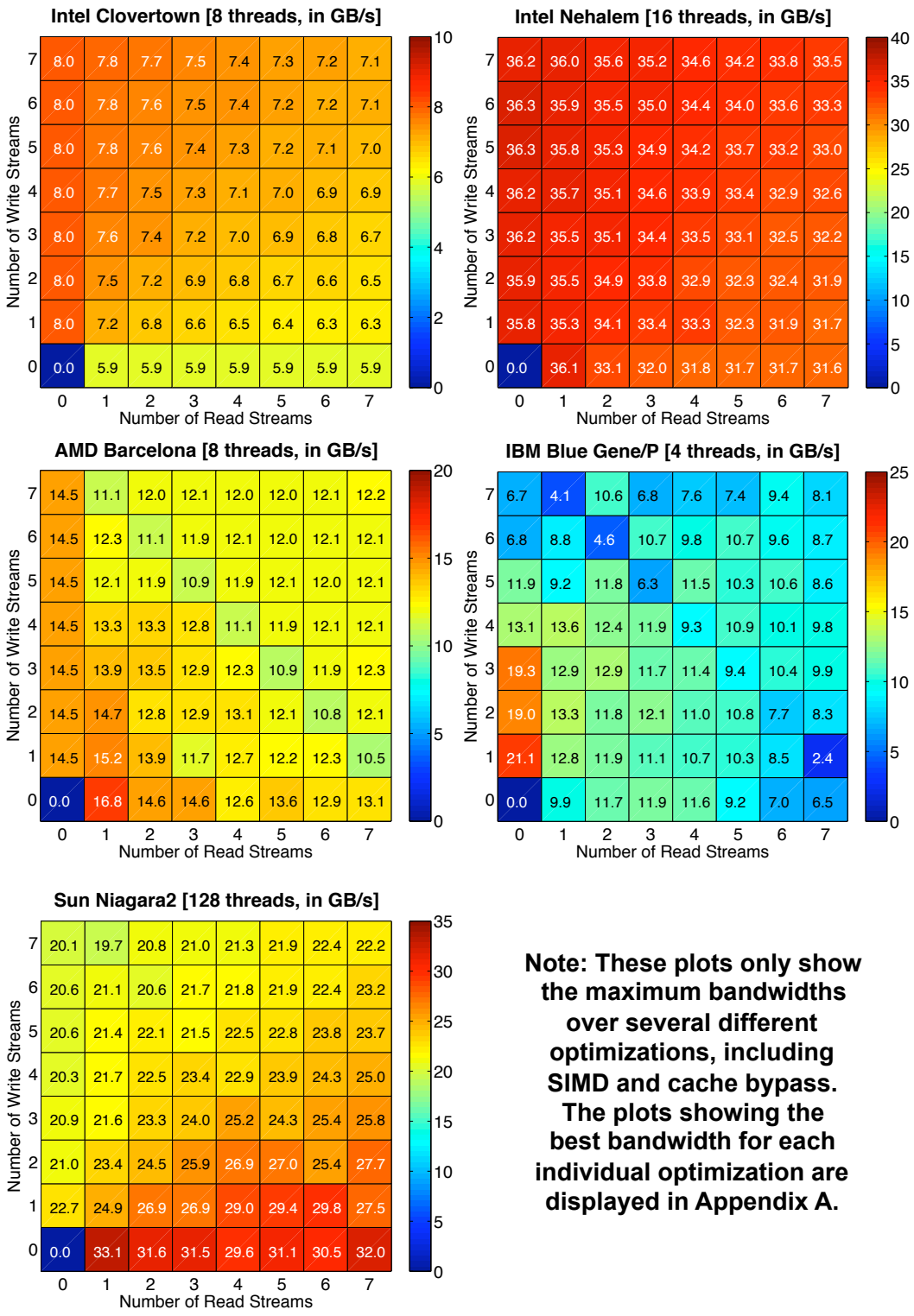


Figure 6.1: The results from the Optimized Stream benchmark for varying numbers of read and write streams per thread.

Platform	Best Stream Copy Performance (in GB/s)	Best Optimized Stream Copy Performance (in GB/s)	Percentage Improvement
Intel Clovertown	7.14	7.16	0.3%
Intel Nehalem	34.5	35.3	2.1%
AMD Barcelona	14.7	15.2	3.4%
IBM Blue Gene/P	12.2	12.8	4.9%
Sun Niagara2	22.0	24.9	13.3%

Table 6.2: This table presents the performance improvement that we observe when using the Optimized Stream benchmark over the normal Stream benchmark. Both benchmarks are measured while performing a simple array copy operation.

Stream benchmark. Both benchmarks perform a simple array copy operation, where the contents of a single 1D array are copied into a second 1D array of equal length. This again corresponds to having a single read stream and a single write stream per thread. We find that our Optimized Stream benchmark consistently outperforms the Stream benchmark across all the platforms in our study, but the median advantage is a modest 3.4%. However, on the Sun Niagara2, our auto-tuning does pay significant dividends, as performance improved by 13.3% over the normal Stream benchmark.

6.4 Computation Bounds

Now that we have placed a communication upper bound on our performance, we would also like to set a computation bound as well. This will likely be needed for several of our platforms when running the 27-point stencil, and may be needed for the 7-point stencil as well.

The Roofline model places several ceilings on computation, including exploiting full thread-level parallelism, instruction-level parallelism, SIMD, and optimizing the floating point multiply-add balance. In order to be independent of the compiler, many of these ceilings are established using information from hardware manuals about floating point datapaths. This sets a true upper bound on the rate of computation that a chip can deliver.

However, for this thesis, we wished to place a second, tighter computation bound

Platform	Best 7-point Stencil In-cache Performance (in GStencil/s)	Best 27-Point Stencil In-cache Performance (in GStencil/s)
Intel Clovertown	2.61	1.11
Intel Nehalem	3.36	1.19
AMD Barcelona	3.02	0.93
IBM Blue Gene/P	0.29	0.10
Sun Niagara2	1.26	0.60

Table 6.3: This table displays the in-cache performance of both the 7-point and 27-point stencils. These can serve as computation bounds for our stencil performance.

that takes the quality of the code generated by the compiler into account. As such, we first chose a small stencil problem that fit into the last-level (largest) cache of a single socket on each of the systems in our study. The grid dimensions of this stencil problem were chosen with a long unit-stride dimension and a relatively short least contiguous dimension so as to avoid having many discontinuities in the memory access pattern. We then performed at least 100 stencil sweeps over this grid in order to amortize the initial time needed to retrieve data from DRAM into cache. This experiment was again auto-tuned, as we performed array padding, core blocking, register blocking, prefetching, SIMDization, and cache bypass through our usual iterative greedy search. The computational rate achieved from this experiment should serve as the "speed-of-light" for the given kernel.

Table 6.3 shows the best in-cache computational rates achieved for both the 7-point and 27-point stencils. As expected, the 7-point stencil always attains a higher GStencil rate than the 27-point stencil because it performs far fewer flops per stencil. In order to convert the 7-point stencil results from GStencil/s to GFlop/s, we merely need to multiply by 8 flops/stencil. However, performing the same conversion for the 27-point stencil is again complicated by the CSE optimization. The 27-point stencil results in Table 6.3 include the CSE optimization, so the number of flops performed per point varies. Consequently, we will only be presenting these results as GStencil/s.

6.5 Roofline Models and Performance Expectations

Using a combination of the Optimized Stream benchmark and a knowledge of the floating point datapaths for each platform in our study, S. Williams created the Roofline models shown in Figure 6.2. These models may be used to identify potential bottlenecks for each architecture. Given an arithmetic intensity, we can simply scan upward from the x-axis. Performance may not exceed a ceiling until the corresponding optimization has been implemented. For example, with cache bypass, the 27-point stencil on Nehalem requires full instruction-level parallelism (ILP) including unroll and jam, full data-level parallelism using SSE instructions (SIMDization), and NUMA-aware allocation to have any hope of achieving peak performance. Conversely, without cache bypass, the 7-point won't even require full thread-level parallelism (TLP), that is using all cores, to achieve peak performance.

By combining these Roofline models with a knowledge of each kernel's arithmetic intensity and instruction mix, we may not only bound the ultimate performance for each architecture, but also broadly enumerate the optimizations required to achieve it. However, please note that the CSE optimization for the 27-point stencil is outside the scope of our analysis since the arithmetic intensity varies with the register block size.

6.5.1 Intel Clovertown

The older front side bus (FSB) based Clovertown architecture has similar computational capabilites to Nehalem and Barcelona, but with signficantly lower memory bandwidth. As such, both the 7-point and 27-point kernels will be memory-bound. Interestingly, although the Optimized Stream benchmark time-to-solution is superior using the cache bypass optimization (implemented with the *movntpd* instruction), the observed Optimized Stream bandwidth (based on total bytes including those from write allocations) is substantially lower than the bandwidth without cache bypass. This can be clearly observed in Figure A.1 for the data point where a single read stream and a single write stream are present per thread. Consequently, there are two

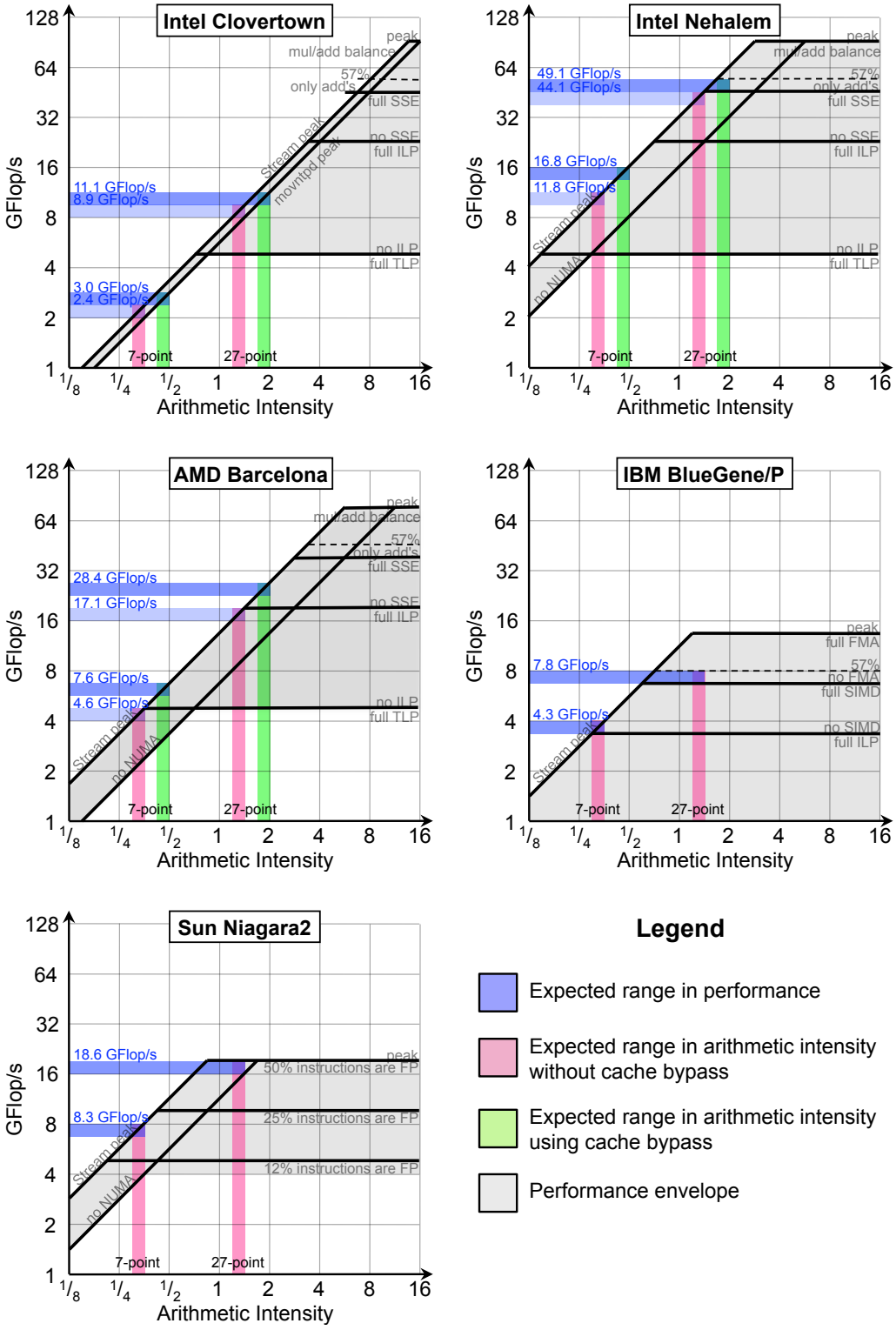


Figure 6.2: The Roofline model for each of the architectures in our study. These diagrams were designed by Samuel Webb Williams [59, 58, 57].

bandwidth ceilings shown in the Clovertown Roofline model. The performance upshot is that the benefit from exploiting cache bypass on stencil operations is muted to about 25% instead of the ideal 50%. We expect Clovertown to be so heavily memory-bound that simple parallelization should be enough to achieve peak performance on the 7-point stencil, and only moderate unrolling is sufficient for the 27-point stencil. Clovertown performance should be limited to about 3.0 GFlop/s (0.38 GStencil/s) and 11.1 GFlop/s (0.37 GStencil/s) for the 7-point and 27-point stencils respectively.

6.5.2 Intel Nehalem

For the 7-point stencil, we expect that Nehalem will ultimately be memory-bound with or without cache bypass. Given the Optimized Stream bandwidth and ideal arithmetic intensity, 16.8 GFlop/s (2.1 GStencil/s) is a reasonable performance bound. To achieve it, some instruction-level parallelism or data-level parallelism coupled with correct NUMA allocation is required. However, as we move to the 27-point stencil, we observe that Nehalem will likely become compute limited, likely achieving between 44 and 49 GFlop/s (1.5–1.6 GStencil/s). There is some uncertainty here due to the confluence of a broad range in arithmetic intensity at a point on the roofline where computation is nearly balanced with communication. The transition from being memory-bound to being compute-bound implies that the benefits of cache bypass will be significantly diminished as one moves from the 7-point to the 27-point stencil.

6.5.3 AMD Barcelona

The Barcelona processor is architecturally similar to the Nehalem but built on a previous generation's technology. As a result, we see that although it exhibits similar computational capability, its substantially lower memory bandwidth mandates that both stencil kernels will be memory-bound. Barcelona performance should be limited to 7.6 GFlop/s (0.95 GStencil/s) and 28.4 GFlop/s (0.95 GStencil/s) for the 7-point and 27-point stencils respectively. However, to achieve high performance on the latter, SIMD (DLP), substantial unrolling (ILP), and proper NUMA allocation will be required.

6.5.4 IBM Blue Gene/P

Architectures like the chip used in Blue Gene/P are much more balanced, dedicating a larger fraction of their power and design budget to DRAM performance. This is not to say they have higher absolute memory bandwidth, but rather the design is more balanced given the low frequency quad core processors. As we were not able to exploit the cache bypass optimization on BG/P, we achieve substantially lower arithmetic intensities than for the x86 architectures. The Roofline model suggests that if we were able to perfectly SIMDize and unroll the code, we would expect the 7-point stencil to be memory bound, yielding 4.3 GFlop/s (0.54 GStencil/s). On the other hand, the 27-point stencil should be compute-limited due to the relatively small fraction of multiplies in the code. We should expect the performance to be about 7.8 GFlop/s (0.26 GStencil/s), but this may be difficult to achieve given the limited issue-width and in-order nature of the PPC450 architecture.

6.5.5 Sun Niagara2

Unlike the other architectures in our study, Victoria Falls uses massive thread-level parallelism (TLP) to express memory-level parallelism. Nonetheless, we expect that its performance characteristics should be similar to Blue Gene/P in that it will be memory bound for the 7-point stencil and compute bound for the 27-point stencil, with performances of 8.3 GFlop/s (1.04 GStencil/s) and 18.6 GFlop/s (0.62 GStencil/s) respectively. Victoria Falls is also similar to Blue Gene/P in that we were unable to exploit the cache bypass optimization. As multithreading provides an attractive solution to avoiding the ILP pitfall, our primary concern after proper NUMA allocation is that floating-point instructions dominate the instruction mix on the 27-point stencil. As such, the Niagara2 Roofline model is shown with computational ceilings corresponding to various ratios of floating-point instructions. Note that the SPARC ISA does not support either FMA instructions nor double precision SIMD, so this simplifies the computational ceilings for Victoria Falls.

6.6 Summary

Thus far, we have discussed performance expectations and bounds for the 7-point and 27-point stencils. The actual results in the following chapters will bear out the accuracy of these predictions.

Chapter 7

3D 7-Point Stencil Tuning

7.1 Description

As discussed in Chapter 2, the 3D 7-point stencil performs 8 flops per point and ideally transfers either 24 Bytes of DRAM traffic (without cache bypass) or 16 Bytes (with cache bypass) per point. In the previous chapter, we laid out our expectations and upper bounds for how this kernel would perform on each of the platforms in our study. In this chapter, we present the actual performance results that we achieved after full auto-tuning. We then compare these results against the Roofline models and the in-cache performance results from Chapter 6.

7.2 Optimization Parameter Ranges

We developed auto-tuners for all three 3D stencil kernels presented in this thesis– the 7-point stencil, the 27-point stencil, and the GSRB Helmholtz kernel. For each, we also performed "data-aware" tuning. This means that we selected appropriate parameter ranges based on a priori knowledge of the problem size. We did not tune for a generic problem size and then hope that the tuning would be effective for our given problem. Consequently, for a given optimization, there is a reasonably high expectation of the optimal parameter value falling within the chosen parameter range. Moreover, because several of the parameter ranges are large, we explore them in powers of two.

Category	Optimization Parameter	Name	parameter tuning range by architecture x86 Machines	Blue Gene/P	Niagara2
Domain Decomp	Core Block Size	CX	NX	NX	{8...NX}
		CY	{4...NY}	{4...NY}	{4...NY}
		CZ	{4...NZ}	{4...NZ}	{4...NZ}
	Thread Block Size	TX	CX	CX	{8...CX}
		TY	CY	CY	{8...CY}
	Chunk Size		$\{1...\frac{NX \times NY \times NZ}{TX \times TY \times CZ \times NThreads}\}$		
Data Allocation	NUMA Aware		✓	N/A	✓
	Pad by a maximum of:		31	31	31
	Pad by a multiple of:		1	1	1
Bandwidth	Prefetching Type		{none, register block, plane, pencil}		
	Prefetching Distance		{0...64}	{0...64}	{0...64}
	Cache Bypass		✓	—	N/A
In-Core	Register Block Size	RX	{1...8}	{1...8}	{1...8}
		RY	{1...4}	{1...4}	{1...4}
		RZ	{1...4}	{1...4}	{1...4}
	Explicit SIMDization		✓	✓	N/A
	CSE		✓	✓	✓
Tuning	Search Strategy		Iterative Greedy		
	Data-aware		✓	✓	✓

Table 7.1: Attempted optimizations and the associated parameter spaces explored by the auto-tuner for the 3D 7-point and 27-point stencils sweeping over a 256^3 problem ($NX, NY, NZ = 256$). The parameter names are visually represented in Figure 4.1. For simplicity, all of the block dimensions and chunk sizes are set to be powers of two, while the prefetching distances are either zero or a power of two. In addition, all numbers are in terms of doubles.

For both the 3D 7-point and 27-point stencils, we kept a fixed problem size of 256^3 (*i.e.* $NX, NY, NZ = 256$). Based on this knowledge, we chose the parameter values in Table 7.1. Several of the parameters, including the ones for the NUMA-aware, cache bypass, explicit SIMD, and CSE optimizations, are booleans. The other non-boolean parameters intentionally search over a very wide set of legal values so that we find the optimal value for every machine. Some of these parameter ranges do require further explanation, though.

First, we know from Section 3.1.5 that, in lieu of hardware prefetching, Victoria Falls has adopted a unique multithreaded approach to hiding memory latency. Therefore, we have made several unique tuning range decisions specifically for it. We know from previous work [28] that hardware prefetchers handle unit-stride accesses well, but are also severely disrupted when there are discontinuities in the memory access pattern. By leaving core blocks undivided in the unit-stride (X) dimension (i.e. $CX = NX$) for our other architectures, we expect the hardware stream prefetchers to remain engaged and effective. However, for Victoria Falls, the lack of a hardware prefetcher allows us to set $CX \leq NX$; the resulting short stanza lengths may actually result in better performance. Moreover, since Victoria Falls supports 8 threads per core and has relatively small, low associativity caches, we allow each core block to contain either 1 or 8 thread blocks (explained in Section 4.1.2). In essence, this allows us to conceptualize Victoria Falls as either a 128 core machine or a 16 core machine with 8 threads per core. On the other architectures, in contrast, we always set the thread block size equal to the core block size. Finally, there are no supported SIMD or cache bypass instrinsics on Victoria Falls, so only the portable Pthreads C code was run.

The parameter range of *ChunkSize* in Table 7.1 also deserves more elaboration. Recall from Section 4.1 that *ChunkSize* represents the number of adjacent core blocks that have been grouped together into a *chunk*. Each of the core blocks within this chunk are then processed by the same subset of threads. If *ChunkSize* becomes too large, then some threads may not receive any core blocks to process. Consequently, we limit *ChunkSize* to values where all threads are still properly load balanced.

Finally, all of the machines in our study support software prefetching. Table 7.1 shows that we vary both the *prefetching type* and *prefetching distance* at the source

code level. The prefetching type indicates how many software prefetch streams are in place. If it is set to "register block", then there is a single software prefetch stream for the entire register block, regardless of its dimensions. If the prefetching type is set to "plane" or "pencil", then every plane or pencil of the register block will have a software prefetch stream. In these cases, the size of the register block does determine how many software prefetch streams are in place. It is possible to have too many software prefetch streams, or to prefetch a distance too far ahead, so both of these parameters are tunable.

7.3 Parameter Space Search

There are some exceptions to how the iterative greedy search is conducted, and they are indicated in Table 7.2. As we add optimizations, the columns of Table 7.2 indicate which optimization parameters are already fixed ("F") and which ones need to be searched over ("S"). The exceptions occur where "F"'s are absent below the main "S" diagonal. The first such exception occurs when we SIMDize the code. Since this optimization requires a complete code rewrite, it causes us to re-adjust some of the previously fixed parameters. The register blocking and prefetching optimizations are both involved in the modified inner loop, so these parameters require another search. In addition, since SIMD code requires proper data alignment, the purpose of the padding optimization was altered from reducing conflict misses (as described in Section 4.2.2) to performing proper data alignment. The other parameters were left unchanged.

The cache bypass optimization, which is only applicable for our x86 machines, builds upon the SIMDized code. It again searches over the best register block size and prefetching in order to find the fastest code.

On Victoria Falls, neither the SIMD nor the cache bypass optimization was available, so instead we performed thread blocking. However, since the thread block dimensions are directly affected by the size of the core blocks, we searched over the legal values for both simultaneously. This search took significantly longer than for core blocking only, but search time was not a constraint in our experiment.

The final optimization, common subexpression elimination (CSE), only applies

Optimization Order	NUMA-Aware	Padding	Core Blocking	Register Blocking	Prefetching	SIMD	Cache Bypass	Thread Blocking	CSE
Naïve									
+NUMA-Aware	S								
+Padding	F	S							
+Core Blocking	F	F	S						
+Register Blocking	F	F	F	S					
+Prefetching	F	F	F	F	S				
+SIMD (BGP and x86 only)	F	S	F	S	S	S			
+Cache Bypass (x86 only)	F	F	F	S	S	F	S		
+Thread Blocking (VF only)	F	F	S	F	F			S	
+Register Blocking, CSE	F	F	F	S					
+Prefetching, CSE	F	F	F	F	S				
+SIMD, CSE (BGP and x86 only)	F	S	F	S	S	S			
+Cache Bypass, CSE (x86 only)	F	F	F	S	S	F	S		
+Thread Blocking, CSE (VF only)	F	F	S	F	F			S	

Table 7.2: The specifics of the iterative greedy search we performed with the 7-point and 27-point stencil auto-tuners. The optimizations shown in the leftmost column are applied from top to bottom. As each new optimization is applied, the optimizations labeled "F" have already had their parameter values previously fixed, while the ones labeled "S" are currently being searched over. In addition, the row breaks indicate that a new code generator is being employed due to substantial changes in the stencil code.

to the 27-point stencil, and thus will be described in the following chapter. However, the CSE optimization essentially caused us to perform a second, smaller iterative greedy search due to the drastic nature of the optimization.

7.4 Performance

The results from auto-tuning the 7-point stencil across the five architectures we detailed in Section 3.1 are shown in Figure 7.1. In addition, we placed performance upper bounds on each platform by taking the minimum of the in-cache performance

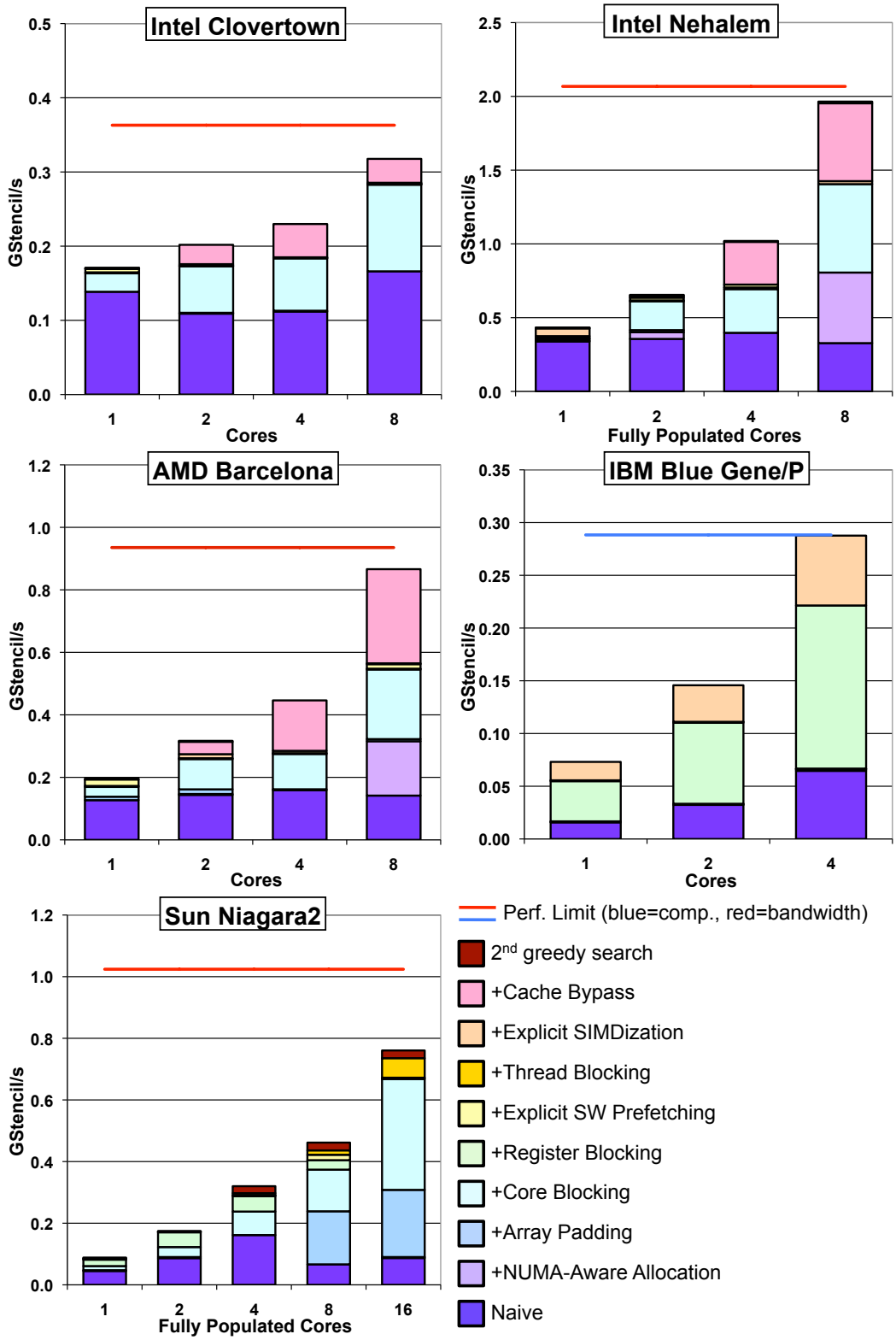


Figure 7.1: Individual performance graphs for the 7-point stencil. The performance limits are set by the minimum of the in-cache performance and Optimized Stream performance.

and the Optimized Stream performance, both of which were discussed in Chapter 6.

The naïve code from which we begin our tuning represents a programmer's likely first attempt at coding the 3D 7-point stencil. There is no NUMA-aware data allocation, array padding, loop unrolling, software prefetching, SIMDization, or cache bypass optimization. However, to make the comparison fair, this naïve code is threaded, not serial. The domain decomposition for this code is performed by dividing the least contiguous (Z) dimension of the grid equally so that every hardware thread is load balanced. Thus, the resulting core blocks are left undivided in the contiguous and middle (X and Y) dimensions.

Unfortunately, we see that the naïve code alone exhibits no parallel scalability on most of these architectures. This is certainly true for the three x86 machines, as additional cores do not result in any additional speedup. On the Sun Niagara2, there is some scalability up to 4 cores, but then the performance drops again at 8 and 16 cores. This is a disturbing trend, as the machines are not taking advantage of the extra cores available to them. However, this is precisely the reason why auto-tuning was necessary. The one exception to the scalability trend is the Blue Gene/P, which does show monotonically improving performance with increasing core count. In this case, we can still show that auto-tuning allows us to achieve substantially better performance than the naïve code alone.

7.4.1 Intel Clovertown

The performance of the 7-point stencil on the Intel Clovertown is shown in the upper left graph of Figure 7.1. As we mentioned previously, the Clovertown is a Uniform Memory Access (UMA) machine with an older front side bus architecture. This implies that the NUMA-aware optimization will not be useful and that the 7-point stencil kernel will likely be bandwidth-constrained. Indeed, we see that both of these predictions are true. There is no speedup from the NUMA-aware optimization as we scale from one socket (four cores) to two (eight cores), and the Optimized Stream performance is less than the in-cache stencil performance. Not only are we bandwidth-limited, but only bandwidth optimizations produce any speedups on this architecture. At maximum concurrency, core blocking produces a 1.7× speedup, while the cache

bypass optimization improved performance by another 11%. Presumably, in-core optimizations are ineffective because the machine is already bandwidth-starved.

Clovertown's poor multicore scaling indicates that the system has rapidly become memory-bound. Given the snoopy coherency protocol overhead, it is not too surprising that the performance only improves by 38% when scaling from one socket to two (when both FSBs are engaged), despite the doubling of the peak aggregate FSB bandwidth.

The Clovertown Roofline model in Figure 6.2 accurately predicts that we will be heavily bandwidth-bound for the 7-point stencil. Moreover, its predicted upper bounds of 0.30 GStencil/s (without cache bypass) and 0.38 GStencil/s (with it) correspond well with the actual performance of 0.29 GStencil/s and 0.32 GStencil/s, respectively. The discrepancy between the actual results and the predicted upper bounds is most likely due to the non-compulsory memory traffic transferred during the stencil computation that we avoided in the Optimized Stream benchmark.

Overall, auto-tuning produced a 1.9× improvement over the performance of the best naïve code, and it created a similar 1.9× speedup when examining the tuned performance from one core to all eight cores.

7.4.2 Intel Nehalem

The upper right graph in Figure 7.1 shows 7-point stencil performance on the Intel Nehalem. If we only examine the naïve implementation, the performance is fairly constant *regardless of core count*. This is discouraging news for programmers; the compiler, even with all optimization flags set, cannot take advantage of the extra resources provided by more cores.

In order to address this problem, the first optimization we applied was using a NUMA-aware data allocation. By correctly mapping memory pages to the socket where the data will be processed, this optimization provides a speedup of 2.5× when using all 8 cores of the SMP. Subsequently, core blocking and cache bypass also produced performance improvements of 74% and 37%, respectively. Both of these optimizations attempt to reduce memory traffic (capacity misses), suggesting that performance is bandwidth-bound at high core counts. By looking at the Roofline

model for the Nehalem (Figure 6.2), we see that this is indeed the case. The model predicts that if the stencil calculation can achieve the Optimized Stream bandwidth, while minimizing non-compulsory cache misses, then the 7-point stencil will attain a maximum of 2.1 GStencil/s (16.8 GFlop/s). In actuality, we achieve 2.0 GStencil/s (15.8 GFlop/s). As this is acceptably good performance, we can stop tuning. Overall on Nehalem, auto-tuning produced a speedup of 4.9× over the naïve code at full concurrency, as well as an improvement of 4.5× when comparing the tuned performance of one core to all eight cores.

Observe that register blocking and software prefetching ostensibly had little performance benefit– a testament to the `icc` compiler and hardware prefetchers. Remember, the auto-tuning methodology explores a large number of optimizations in the hope that they may be useful on a given architecture–compiler combination. As it is difficult to predict this beforehand, it is still important to try each relevant optimization.

7.4.3 AMD Barcelona

In many ways, the performance of the 7-point stencil on Barcelona is very similar to that of Nehalem. This is not surprising, given the similar architectures. The Barcelona performance is shown in the middle left graph of Figure 7.1, and we again see that the naïve implementation shows no parallel scaling at all. However, the NUMA-aware code increased performance by 115% when both sockets are engaged.

Like on the Nehalem, the core blocking and cache bypass optimizations again made a large impact. First, core blocking improved performance by 69% at maximum concurrency. Then, the cache bypass (streaming store) intrinsic reduced memory traffic by 33%, thus changing the 7-point stencil kernel's flop:byte ratio (*i.e.* arithmetic intensity) from $\frac{1}{3}$ to $\frac{1}{2}$. This potential 50% improvement corresponds closely to the 53% observed performance improvement, thereby confirming the memory bound nature of the 7-point stencil kernel on this machine.

The Barcelona Roofline model in Figure 6.2 accurately predicts the heavily memory-bound nature of the 7-point stencil. In addition, its predicted upper bound of 0.95 GStencil/s (7.6 GFlop/s) correlates well with our attained performance of 0.87 GS-

tencil/s (6.9 GFlop/s). Overall, auto-tuning served us well on the Barcelona, as it produced a 5.4× speedup over the best naïve code and a 4.4× speedup when scaling from a single core to all eight cores.

7.4.4 IBM Blue Gene/P

The performance of the IBM Blue Gene/P node is an interesting departure from our previous x86 architectures since it is compute-bound, not bandwidth-limited, for the 7-point stencil. As seen in the middle right graph in Figure 7.1, memory optimizations like padding, core blocking, or software prefetching make no difference in performance. Instead, the only optimizations that help performance are computation-related, like register blocking (3.3× speedup) and SIMDization (1.3× speedup).

After full tuning of the 7-point stencil, we see a performance improvement of 4.4× at full concurrency, as well as nearly perfect parallel scaling of 3.9× going from 1 to 4 cores. Moreover, the computation rate now matches the best in-cache computation rate (shown as a blue line). Thus, we are definitively compute-bound.

Interestingly, the Blue Gene/P Roofline model in Figure 6.2 incorrectly predicts that the 7-point stencil will be bandwidth-bound, with a performance upper bound of 0.53 GStencil/s (4.3 GFlop/s). This is an 85% overestimate over our attained performance of 0.29 GStencil/s (2.30 GFlop/s). The reason for this requires further exploration, but may likely be explained by the `xlc` compiler's inability to generate quality code and fully utilize the computational capabilities of the Blue Gene/P chip. This, in turn, causes the 7-point stencil to still be compute-bound instead of bandwidth-limited. Unlike the Roofline model, the in-cache stencil performance *is* dependent on the compiler, which is why its performance matches that of the 7-point stencil.

We note that unlike the three previous architectures, the IBM Blue Gene/P's `xlc` compiler does not generate or support cache bypass at this time. As a result, the best arithmetic intensity we can achieve is 0.33 for the 7-point stencil.

7.4.5 Sun Niagara2

Like the Blue Gene/P, the Niagara2 does not exploit cache bypass. Moreover, it is a highly multi-threaded architecture with low-associativity caches. We will observe both of these architectural features as we tune.

Initially, if we look at the performance of our naïve 7-point stencil implementation in the lower left corner of Figure 7.1, we see that the we attain 0.16 GStencil/s (1.29 GFlop/s) at 4 cores, but only 0.09 GStencil/s (0.70 GFlop/s) using all 16 cores! Clearly the machine's resources are not being utilized properly. Now, as we begin to optimize, we find that properly-tuned padding improves performance by 3.6× using 8 cores and 3.4× when employing all 16 cores. The padding optimization produces much larger speedups on Victoria Falls than for all previous architectures, primarily due to the increased conflict misses resulting from its low associativity caches.The highly multithreaded nature of the architecture results in each thread receiving only 64 KB of L2 cache. Consequently, core blocking also becomes vital, and, as expected, produces large gains across all core counts.

A new optimization that we introduced specifically for the Sun Niagara2 was thread blocking. In the original implementation of the stencil code, each core block is processed by only one thread. When the code is thread blocked, threads are clustered into groups of 8; these groups work collectively on one core block at a time. The Niagara2 supports 8 hardware threads per core, so we suspected that this approach would better exploit cache locality than our original approach. When thread blocked, we see a 3% performance improvement with 8 cores and a 9% improvement when using all 16 cores. However, the automated search to identify the best parameters for thread blocking was relatively lengthy, since we needed to determine both the optimal core block and thread block dimensions simultaneously.

Finally, we also saw a small improvement when we performed a second pass through our greedy algorithm. Specifically, we started with the best parameter configuration from our first iterative greedy search, and then performed another iterative greedy search. For the higher core counts, this improved performance by about 0.025 GStencil/s (0.20 GFlop/s). Overall, the tuning for the 7-point stencil resulted in a 4.7× speedup at maximum concurrency and a 8.6× parallel speedup as we scale

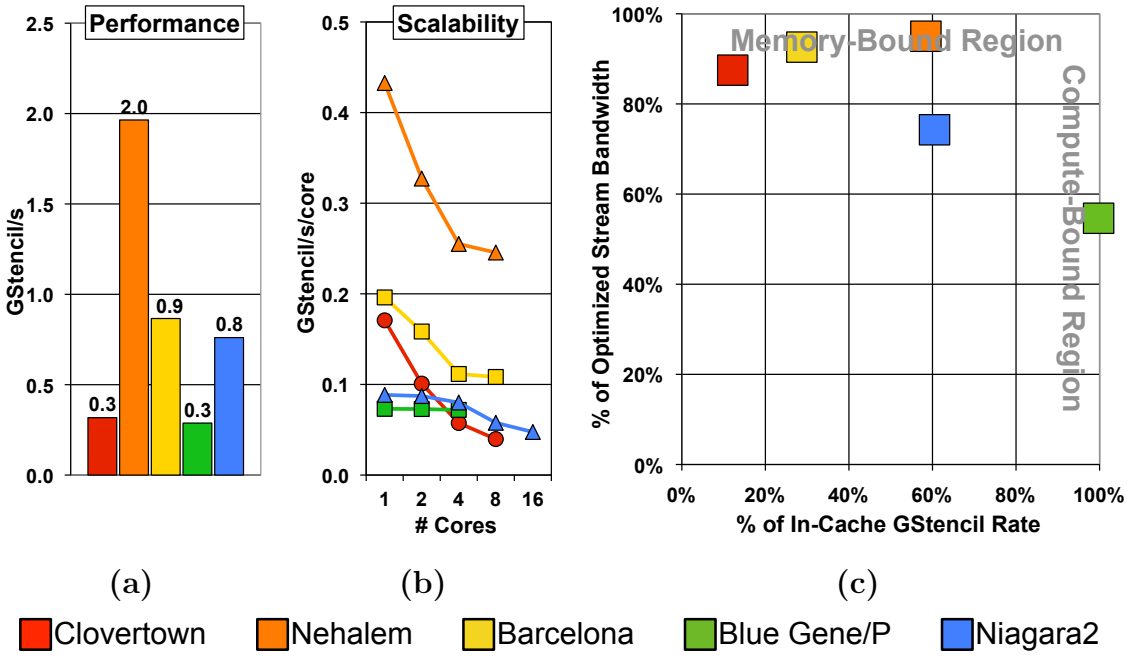


Figure 7.2: Performance summary graphs for the 7-point stencil.

to 16 cores.

The Roofline prediction for the Niagara2 (shown in Figure 6.2) correctly predicts that the 7-point stencil will be bandwidth-bound, but its predicted upper bound of 1.04 GStencil/s (8.3 GFlop/s) is 37% higher than our actual attained performance of 0.76 GStencil/s (6.1 GFlop/s). It is likely that the small working set size and the low associativity caches of the Niagara2 have caused a significant number of capacity and conflict misses, many of which we were unable to eliminate through auto-tuning. This would explain the relatively large gap between our attained bandwidth and the bandwidth achieved by the Optimized Stream benchmark.

7.4.6 Performance Summary

Figure 7.2 captures some of the major trends among the five architectures. In terms of overall performance, Figure 7.2(a) indicates that there are three tiers of raw performance. In the first tier, the Nehalem system more than doubles the performance of either the Barcelona or the Victoria Falls systems. These two machines can then be considered to be the second tier of performance, as they again almost triple

the performance of either the Clovertown or the Blue Gene/P. Ironically, both the Clovertown and the Blue Gene/P achieve about the same low overall performance despite over a 6× advantage in peak computational rate for the Clovertown. However, we know that the Clovertown is severely bandwidth-constrained by its front-side bus, while the Blue Gene/P has already become compute-bound.

In order to better understand scalability, we also plotted the GStencil rates per core in Figure 7.2(b). Only the Blue Gene/P exhibits perfect linear scaling, and that, again, is because it is compute-bound at all core counts. All the other architectures display some sub-linear scaling, although the Clovertown performance decays more quickly than the other architectures.

Finally, in order to identify the bottlenecks to performance, we plotted the percentage of the Optimized Stream bandwidth versus the percentage of in-cache GStencil rate in Figure 7.2(c). As expected, the Blue Gene/P is completely compute-bound. In contrast, the x86 architectures are clearly bandwidth-bound, since all three platforms achieve at least 87% of the Optimized Stream bandwidth while attaining no more than 60% of the in-cache GStencil rate. This also reflects the fact that, after full tuning, we have eliminated almost all of the capacity and conflict misses on these machines. Finally, the Sun Niagara2 achieves 74% of optimized stream bandwidth and 61% of the in-cache computation rate. It is likely that the Niagara2 is bandwidth-bound, even though we don't achieve as high a fraction of Optimized Stream bandwidth as the x86 machines. The highly multi-threaded nature of the machine, along with its small, low-associativity caches does make it harder to tune, but we still achieve a respectable fraction of Optimized Stream bandwidth.

Across all architectures, Table 7.3 shows that we achieve between a 1.9×–5.4× speedup from performance tuning the 7-point stencil, with a median speedup of 4.7×. Thus, auto-tuning plays an essential role in achieving good performance from these multicore systems. Moreover, after full tuning, we see a parallel scaling speedup of between 1.9× and 8.6× over the single core performance, with a median speedup of 4.4×. Therefore, our tuning is also critical in achieving good scalability on these systems.

Platform	Tuning Speedup Over Best Naïve Code	Parallel Scaling Speedup Over Tuned Single Core Performance	Percent of Perfect Speedup
Intel Clovertown	1.9×	1.9×	23%
Intel Nehalem	4.9×	4.5×	57%
AMD Barcelona	5.4×	4.4×	55%
IBM Blue Gene/P	4.4×	3.9×	98%
Sun Niagara2	4.7×	8.6×	54%

Table 7.3: This table presents the performance tuning speedups and parallel scaling speedups for the 7-point stencil.

7.5 Comparison of Best Parameter Configurations

Despite the results that we have shown, it is still unclear whether the best tuning parameters for one platform will translate to good performance on one or more of the other platforms in our study. Looking solely at the architectural features of our machines, this idea does have some merit– several of our machines support similar numbers of hardware threads and have comparable amounts of last-level cache. If this is indeed the case, then we can imagine tuning on only one platform and employing that parameter configuration on several other platforms as well. This would certainly reduce the overall time required for tuning.

In order to test this proposition, we collected data on how well the best parameter configuration for one platform performs on all the other platforms in our study. However, this was an inexact study for two reasons. First, the number of available hardware threads varies from platform to platform, so the ideal number of threads on one machine may undersubscribe or oversubscribe the threads on a different machine. In order to normalize for this, we always chose to employ the maximum number of hardware threads supported on the given architecture. However, depending on the core block dimensions and the chunk size, it is still possible that not all the threads on a given architecture will be utilized. In fact, this did occur, and we will see this effect shortly.

The second reason that this was an inexact study was that not all the optimizations for one platform are available on the others. For instance, the cache bypass

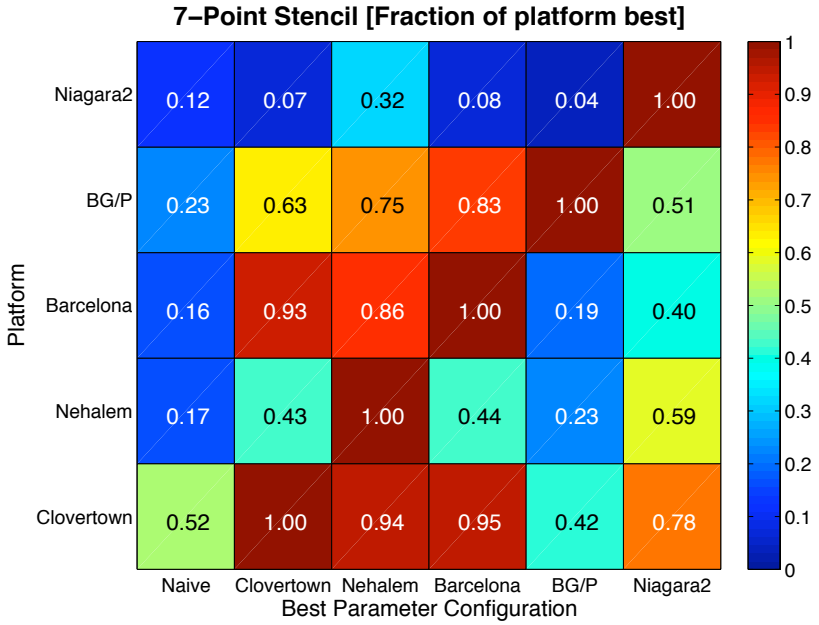


Figure 7.3: The 7-point stencil performance results from running the best parameter configuration for each platform on every other platform in our study. As a point of comparison, we also show the performance of the naïve code in the leftmost column of this plot.

optimization is unavailable on the Niagara2 and Blue Gene/P, but is available, and often quite effective, on the x86 machines. As a result, we implemented the following policy. If the best parameter configuration for one platform could be tested on the new platform, then we did so and reported the results. For instance, the thread blocking optimization produced the best performance results for the Niagara2. Fortunately, this was still portable C code, so we ran the same thread blocked code on all the other architectures to measure how well the parameter configuration performed. On the other hand, if the best parameter configuration could not be directly run on the new platform, then we modified the optimization so that we could still run a somewhat similar configuration on the new platform. For example, SIMDization achieved the best performance on the Blue Gene/P, but this optimization is unavailable on the Sun Victoria Falls machine. Consequently, we ran portable C code on Victoria Falls, but with the register block size having the same dimensions (in doubles) as the best SIMD register block size of the Blue Gene/P. The other parameters that were transferrable between the two architectures were still kept fixed. We felt

that this was the fairest and most consistent method for testing the best parameter configuration across such a wide range of platforms.

The results from this experiment are shown in Figure 7.3. The data collected on each of the five machines in our study is shown as a distinct row in this plot. The leftmost column shows how well the naïve code performs on each platform. The five columns thereafter show the performance of the best parameter configuration for each machine on every machine (including itself). The results are shown as a "fraction of platform best". Specifically, when the best parameter configuration for a given platform is actually run on that platform, we achieve 100% (or 1.00) of platform best. All other results on that platform are normalized to this result.

There are several interesting trends in Figure 7.3. First, the Clovertown architecture seems to be the easiest platform to tune for; the best parameter configurations for the other two x86 platforms perform very well on the Clovertown, achieving between 94%–95% of the platform best. Moreover, the best Niagara2 parameter configuration also performs well, achieving 78% of the platform best, which is still significantly better than the naïve code. We are able to achieve such good performance from the Clovertown because of the front-side bus bottleneck that we discussed earlier. By performing even moderate core blocking, we can sufficiently reduce the bandwidth requirements of the problem so that we achieve a relatively high fraction of the platform best, even though this still translates to poor raw performance. The only platform whose best parameter configuration suffers on the Clovertown is the Blue Gene/P, but this is because the core block dimensions and the chunk size only provide enough work for four hardware threads. The other four available threads on the Clovertown are left idle, explaining why we achieve less than half of the best Clovertown performance.

Similar to the Clovertown, the Barcelona architecture also exhibits very good performance with the best parameter configurations of the other x86 machines, attaining 86% and 93% of the platform best. However, the performance drops significantly for the non-x86 parameter configurations; we achieve only 40% of platform best for the best Niagara2 parameter configuration, and only 19% of platform best for the Blue Gene/P configuration. We can partially explain the poor performance of the Blue Gene/P configuration because it only provides enough work for four threads.

However, in general, the Barcelona does seem to be more sensitive to the parameter configuration than the Clovertown.

The Nehalem is harder to tune for than either the Clovertown or the Barcelona. We can see that the best parameter configurations of the Clovertown and the Barcelona only achieve 43% and 44% of the platform best, respectively. In the case of the Barcelona, the core block dimensions and the chunk size only allow enough work for 8 hardware threads. Thus, the Nehalem's other 8 available threads are left unused. The best Clovertown configuration, on the other hand, exposes enough parallelism for all 16 Nehalem threads, but the configuration is not well-suited to the Nehalem architecture. As for the non-x86 configurations, the Niagara2 configuration does moderately well on the Nehalem, achieving 59% of the platform best, but the Blue Gene/P again suffers, attaining only 23% of the platform best.

The Blue Gene/P is moderately easy to tune, as each of the best parameter configurations attains at least 50% of the platform best, while the Barcelona configuration achieves 83% of platform best. This is partially due to the fact that BG/P supports only four hardware threads, while all the other architectures support at least 8 threads, so we can guarantee that none of the BG/P threads will be left idle.

The Sun Niagara2 lies at the other end of the spectrum– it supports 128 hardware threads, which is 8× more than the next highest platform. As a result, there is a good chance that some of its threads will be left idle when running the best parameter configuration from different platforms. In fact, out of the other four platforms in our study, only the Nehalem's best parameter configuration provides enough parallelism for all 128 Niagara2 threads. However, despite providing enough work to keep all the threads busy, the Nehalem configuration still only achieves 32% of the best platform performance. The other platform configurations do not even have enough work for all 128 threads, and thus their performance ranges from a mere 4%–8% of the platform best. It has become painfully obvious that the highly multithreaded Niagara2 architecture, with its small, low associativity caches, is extremely sensitive to the choice of tuning parameters.

Thus far, we have analyzed Figure 7.3 by row. However, it is also instructive to briefly analyze it by column. For instance, we know that the best Blue Gene/P parameter configuration only provides enough work for four threads. If we look at

the column corresponding to the Blue Gene/P configuration, we see that for non-BG/P platforms, we only achieve up to 42% of platform best, and usually well below this fraction. This is understandable, as all the other platforms in our study support at least 8 threads. Next, if we examine the column corresponding to the best Nehalem configuration, we see that outside the Niagara2, it does well on the other platforms (ranging from 75%–94% of platform best). This is partly explained by the fact that the Nehalem configuration provides enough parallelism for up to 128 threads, so none of the architectures will have idle threads. Moreover, the Nehalem configuration employs both the SIMDization and cache bypass optimizations, both of which are usually very effective on x86 architectures. Finally, let us examine the column associated with the best Niagara2 configuration. Like the Nehalem, there is enough parallelism to support at least 128 threads. However, unlike the Nehalem configuration, the Niagara2 code does not utilize either the SIMDization or cache bypass optimizations, since neither is supported on the architecture. Instead, the code is thread blocked to take mitigate the effects of the Niagara2's relatively small caches. Unfortunately, tuning for Victoria Falls does not translate to good performance on the other architectures, as the Nehalem configuration consistently outperforms the thread blocked code.

In general, the first step to ensuring that the best parameter configuration for one platform works well on another is to verify that there is enough parallelism available to avoid leaving any hardware threads idle. However, meeting this criteria alone may still be insufficient for good performance. Some architectures, especially highly-multithreaded ones like the Sun Niagara2, may still require significantly more work to attain good performance.

7.6 Conclusions

Thus far, we have exhaustively explored the performance of the 7-point stencil on each of the systems in our study. We have also seen that, in general, the Roofline model has supplied a relatively tight upper bound on stencil performance. However, except for the Blue Gene/P, the 7-point stencil kernel taxes the memory subsystem far more than the chip's computational abilities. The 27-point stencil, examined in

the following chapter, has a significantly higher arithmetic intensity, so we should finally expose the floating point capabilities of our multicore chips.

Chapter 8

3D 27-Point Stencil Tuning

8.1 Description

In Chapter 2, we mentioned that the 3D 27-point stencil performs 30 flops per point without common subexpression elimination (CSE), and between 18–30 flops per point with it. This is many more flops than for the 7-point stencil. The DRAM traffic, however, should be similar to the 7-point stencil. Ideally, we should expect 24 Bytes of traffic per point without the cache bypass optimization, and 16 Bytes with it. This chapter will present the full auto-tuning results for the 27-point stencil kernel, and will also analyze the results using the performance upper bounds that we laid out in Chapter 6.

8.2 Optimization Parameter Ranges and Parameter Space Search

Like the 7-point stencil, for the 27-point stencil we again kept a fixed problem size of 256^3 (*i.e.* $NX, NY, NZ = 256$). As a result, the optimization parameter ranges for the 27-point stencil are also identical to that of the 7-point stencil, which we detailed in Section 7.2. Similarly, the parameter space search that we explained in Section 7.3 applies equally to the 27-point stencil, with one exception– the common subexpression elimination (CSE) optimization only applies to the 27-point stencil,

and thus it will be discussed here instead of Chapter 7.

As shown in Table 7.2, the CSE optimization is the final optimization that we apply during our iterative greedy (or hill-climbing) search. However, it can be built on top of many different optimized code versions, and thus requires further explanation. Across all architectures, the drastic nature of the optimization compels us to perform a second, smaller iterative greedy search specifically for CSE. For the x86 and BGP architectures, this required new CSE code generators that create both portable C and SIMD code. The results that we show are for the best performing CSE code, regardless of which code generator was used. Similarly, for Victoria Falls, we created a new CSE code generator for thread blocking as well. Again, the presented result for Victoria Falls is for the best performing CSE code, regardless of whether it is thread blocked.

8.3 Performance

The results from auto-tuning the 27-point stencil across the five platforms in our study are displayed in Figure 8.1. We again bounded the performance on each platform by taking the minimum of the in-cache performance and the Optimized Stream performance. In addition, our results are again presented in terms of GStencil/s. The 27-point stencil sweeps over the same amount of memory as the 7-point stencil, but performs several times more flops. As such, we should expect that for the same platform, the GStencil rate of the 27-point stencil will be the same or less than that of the 7-point stencil. By comparing the performance of each of the five platforms in Figures 7.1 and 8.1, we see that this is indeed the case.

Finally, we note that the naïve code from which we start tuning is very similar to the naïve 7-point stencil code that we described in Section 7.4. The naïve 27-point stencil code does not perform any NUMA-aware data allocation, array padding, loop unrolling, software prefetching, SIMDization, cache bypass optimization, or CSE. It is threaded, however, and the domain is decomposed by dividing the grid equally in the least contiguous (Z) dimension such that every hardware thread receives an equal amount of work.

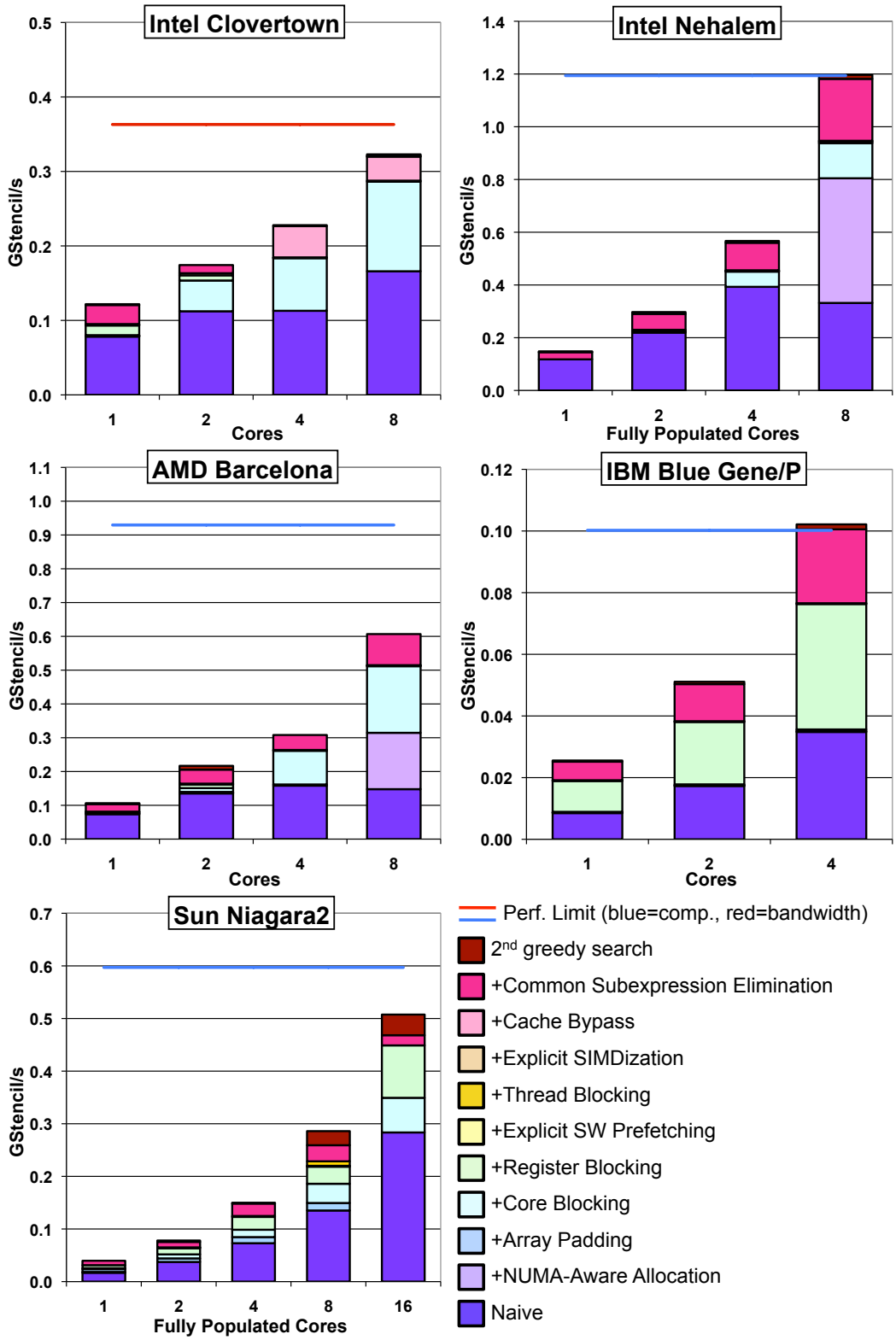


Figure 8.1: Individual performance graphs for the 27-point stencil. The performance limits are set by the minimum of the in-cache performance and Optimized Stream performance.

8.3.1 Intel Clovertown

The 27-point stencil performance on Clovertown is displayed in the upper left graph of Figure 8.1. Despite the high flop:byte ratio of the 27-point stencil, memory bandwidth is again an issue at the higher core counts. When we utilize one or two cores, cache bypass is not helpful, while the CSE optimization produces speedups of at least 30%, implying that the lower core counts are compute-bound.

However, as we scale to four and eight cores, we observe a transition to being memory-bound– in fact, since the 7-point and 27-point stencils are both memory-bound at these higher core counts, we observe that they achieve identical GStencil rates. At four and eight cores, the cache bypass instruction improves performance by at least 10% for the 27-point stencil, while the effects of CSE are negligible. The Clovertown Roofline model in Figure 6.2 predicts this behavior, as it shows that the 27-point stencil is still memory-bound, and thus cache bypass should be effective. The Roofline model also predicts a performance upper bound of 0.36 GStencil/s (11.1 GFlop/s), while we actually attained 0.32 GStencil/s (9.6 GFlop/s). Hence, further tuning will have diminishing returns. All in all, full tuning for the 27-point stencil resulted in a 1.9× improvement using all 8 cores, as well as a 2.7× speedup when scaling from one to all eight cores.

8.3.2 Intel Nehalem

The Nehalem performance results for the 27-point stencil are shown in the upper right graph of Figure 8.1. We see that the naïve code improves by 3.3× when scaling from one to four cores, but then drops slightly when we use all eight cores across both sockets. This performance quirk is eliminated when we apply the NUMA-aware optimization.

After full tuning, there are several indicators that strongly suggest that performance has become compute-bound — core blocking shows less benefit than for the 7-point stencil, cache bypass doesn't show any benefit, performance scales linearly with the number of cores, and the CSE optimization is successful across all core counts. However, perhaps the most obvious sign is that after full tuning, we match the in-cache performance of the 27-point stencil code. Ultimately, this tuning re-

sulted in a 3.6× speedup over the best naïve code, as well as a parallel scaling of 8.1× when going from one to eight cores– the ideal multicore scaling.

The Nehalem Roofline model, shown in Figure 6.2, correctly predicts that the 27-point stencil will be compute-bound, but predicts an upper bound of 1.63 GStencil/s (49.1 GFlop/s) before applying the CSE optimization. In actuality, we attain about 0.95 GStencil/s (28.4 GFlop/s) before the CSE optimization, which is 42% less than our predicted upper bound. However, as we explained in Section 6.5.2, there was some uncertainty with this upper bound– not only was there a broad range in actual arithmetic intensity, but computation and communication are also nearly balanced at this point in the Roofline model.

8.3.3 AMD Barcelona

Unlike the 27-point stencil on Nehalem, the sub-linear scaling of Barcelona in the middle left graph of Figure 8.1 seems to indicate that the kernel is constrained by memory bandwidth. However, the fact that cache bypass did not improve performance, while the CSE optimization improves performance by 18% at maximum concurrency, hints that it is close to being compute-bound. In actuality, by examining Figure 8.2(c), we see that Barcelona achieves about 65% of both Optimized Stream bandwidth and in-cache GStencil rate. Thus, it is about equally constrained by bandwidth and computation; this may explain why we achieve such a low fraction of peak for both.

The Roofline model for Barcelona, displayed in Figure 6.2, predicts that the 27-point stencil will be bandwidth-bound. Similar to the Nehalem predictions, the 27-point stencil (without CSE) predictions for Barcelona are looser than for the 7-point stencil. The Roofline model predicts an upper bound of about 0.95 GStencil/s (28.4 GFlop/s), but our attained performance is 0.51 GStencil/s (15.4 GFlop/s). This discrepancy is likely due to the transition from being limited by bandwidth to being limited by computation. This suggests that as we approach a compute-bound state, the DRAM–FP Roofline model alone is insufficient. Instead, it may be useful to employ multiple Roofline models per architecture.

Overall, auto-tuning the 27-point stencil on Barcelona was able to produce a 3.8×

speedup using all eight cores. In addition, parallel scaling from a single core to all eight cores produces a speedup of 5.7×, which is 71% of the perfect parallel speedup.

8.3.4 IBM Blue Gene/P

As we observed in Section 7.4.4, the 7-point stencil was already compute-bound on the Blue Gene/P. While the 27-point stencil may incur slightly more cache capacity misses, the 3.8× increase in the number of flops per point should ensure that it remains compute-bound as well. As seen in the middle right graph of Figure 8.1, we again observe that memory optimizations like padding, core blocking, and software prefetching are futile. Similar to the 7-point stencil, only computation-related optimizations like register blocking (2.1× speedup) and CSE (1.4× speedup) show any benefit. After full tuning, the 27-point stencil kernel exhibits the perfect multicore scaling of 4.0× when scaling from one to four cores. Moreover, the GStencil rate slightly *exceeds* the in-cache performance of this kernel, most likely because of the second iterative greedy search that we performed. All these observations confirm the compute-bound nature on this kernel on Blue Gene/P.

If we examine the Blue Gene/P Roofline model in Figure 6.2, we see that the 27-point stencil is clearly compute-limited, with a predicted upper bound of 0.26 GStencil/s (7.8 GFlop/s) before CSE. In actuality, we are compute-bound, but we only attain a performance of 0.08 GStencil/s (2.3 GFlop/s) without CSE, which is less than a third of the Roofline predicted upper limit. Again, there are several potential reasons for this difference. First, we observed a similar overestimate of the Roofline upper bound for the 7-point stencil on BG/P, so it is possible that the `xlc` compiler is not generating sufficiently high-quality code. Moreover, as we noted in Section 6.5.4, the limited issue-width and the in-order nature of the PPC450 architecture could also be hindering the computational abilities of the chip.

8.3.5 Sun Niagara2

The naïve implementation of the 27-point stencil on Victoria Falls, shown in the lower left graph of Figure 8.1, seems to scale well. Nonetheless, auto-tuning was still able to achieve significantly better results than the naïve implementation

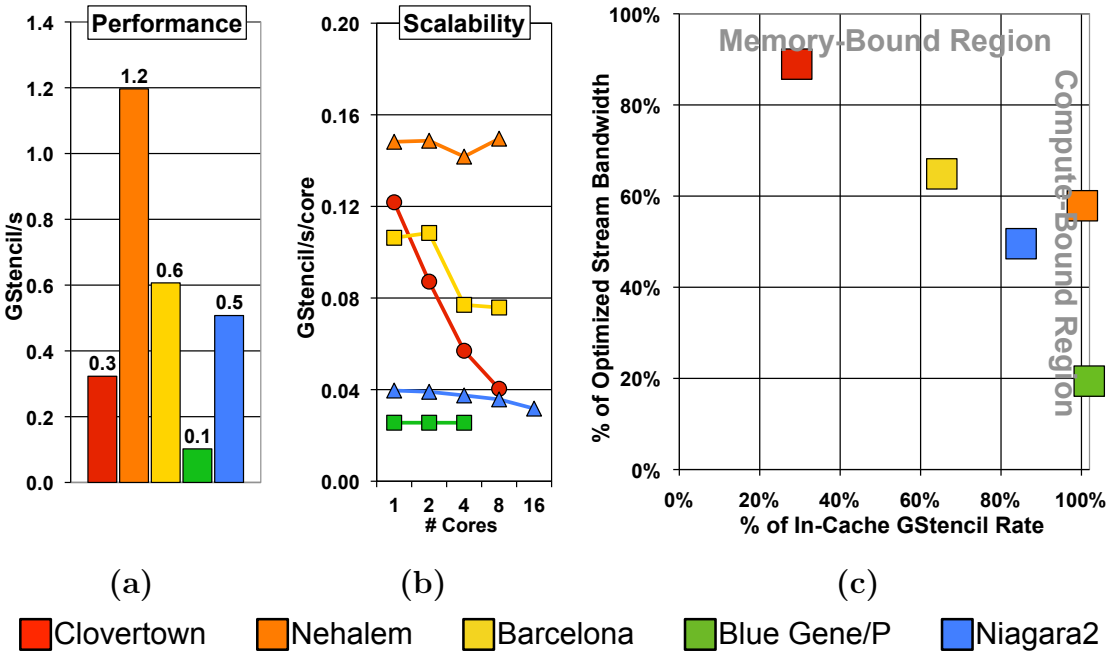


Figure 8.2: Summary performance graphs for the 27-point stencil.

alone. Many optimizations combined together to improve performance, including array padding, core blocking, common subexpression elimination, and a second sweep of the iterative greedy algorithm. After full tuning, performance improved by 1.8× over the naïve code, and we also observed good parallel scaling of 12.8× when scaling from one to 16 cores. The fact that we achieve this nearly linear scaling suggests that we are compute-bound on Victoria Falls. Figure 8.2(c) confirms this idea, showing that while we only achieve 50% of our bandwidth peak, we attain 85% of our in-cache performance.

The Roofline model for the Niagara2, shown in Figure 6.2, predicts that the 27-point stencil is compute-bound, with an upper limit of 0.62 GStencil/s (18.6 GFlop/s) before CSE. This is a reasonable, if not tight, upper bound, as we actually achieve 0.45 GStencil/s (13.5 GFlop/s). Thus, the Roofline-predicted upper bound is 38% higher than our actual performance; this discrepancy may be partially due to the fact that this Roofline upper bound is compiler independent.

8.3.6 Performance Summary

Figure 8.2 displays performance summary graphs for the 27-point stencil. Figure 8.2(a) shows the best raw performance attained by each platform. Similar to the corresponding figure for the 7-point stencil (Figure 7.2(b)), we again see that Nehalem performs twice as fast as Barcelona, which again performs at a similar rate to Victoria Falls. However, unlike Figure 7.2(b), we now see that Clovertown now triples the performance of Blue Gene/P. Apparently, the compute-intensive 27-point stencil is finally able to expose the overwhelming advantage in peak GFlop rate that Clovertown holds over Blue Gene/P.

Interestingly, even though the x86 systems have nearly the same peak GFlop rates (as seen in Table 3.1), they still attain very different actual GFlop rates for a kernel with a relatively high arithmetic intensity. However, we can explain this phenomenon. As we observed before, the Nehalem platform achieves the highest GFlop rate primarily because it is compute-bound across all core counts. Its nearly perfect parallel scaling from one to eight cores can be clearly seen in Figure 8.2(b). The Barcelona performance falls in the middle since it is compute-bound only up to two cores, at which point parallel scaling drops off and it becomes bandwidth-bound. This is also visualized in Figure 8.2(b), since the performance per core drops by about one-third when scaling up to four and eight cores. Finally, Clovertown seems to be bandwidth-bound beyond a single core, primarily due to its older front-side bus architecture. Figure 8.2(b) shows that the single core performance on Clovertown is better than that of Barcelona, but performance decays quickly as we scale up to eight cores. Thus, when using all eight cores, only Nehalem has a chance of exploiting its full computational resources for executing the 27-point stencil.

The Blue Gene/P and Niagara2 also exhibit very good parallel scaling for the 27-point stencil kernel, primarily because both machines are already compute-bound. This is not surprising, given that both machines have such low computational peaks (about one-fifth that of the x86 machines).

Finally, Figure 8.2(c) plots the percent of Optimized Stream bandwidth against the percent of in-cache GStencil rate. This comparison of communication against computation helps us identify the limiting factor for each of our five architectures.

Platform	Tuning Speedup Over Best Naïve Code	Parallel Scaling Speedup Over Tuned Single Core Performance	Percent of Perfect Speedup
Intel Clovertown	1.9×	2.7×	33%
Intel Nehalem	3.0×	8.1×	101%
AMD Barcelona	3.8×	5.7×	71%
IBM Blue Gene/P	2.9×	4.0×	100%
Sun Niagara2	1.8×	12.8×	80%

Table 8.1: This table presents the performance tuning speedups and parallel scaling speedups for the 27-point stencil.

Clearly, both Nehalem and Blue Gene/P are computation-limited, as they both achieve about 100% of the in-cache computation rate. Victoria Falls also seems to be compute-bound, although it achieves a slightly lower percentage of in-cache performance. The Clovertown architecture, as we noted before, is heavily memory-bound, as it achieves 89% of the Optimized Stream bandwidth, but only 29% of the in-cache GStencil rate. Finally, Barcelona seems to be bound by both bandwidth and computation, as it achieves 65% of both peak bandwidth and peak in-cache performance. The reason it achieves such a low fraction of both metrics may be because it is limited by two resources.

Table 8.1 summarizes the benefits we see from tuning this kernel. While the improvements from tuning are not as dramatic as for the 7-point stencil, we still attain a speedup between 1.8×–3.8×, with a median of 2.9×. This is because, in general, memory optimizations often provide larger speedups than computation optimizations. The 7-point stencil is usually memory-bound, so it shows better performance improvements over the naïve code than the 27-point stencil.

However, in terms of parallel scaling, the two kernels are reversed. The 27-point stencil now exhibits significantly larger improvements than the 7-point stencil, as we achieve between a 2.7×–12.8× performance improvement over the single core performance, with a median speedup of 5.7×. This is because a compute-intensive kernel like the 27-point stencil can easily exploit more cores by offloading computation onto them. Unfortunately, the 7-point stencil is often bandwidth-bound– since bandwidth usually does not increase with more cores, scalability is also suppressed.

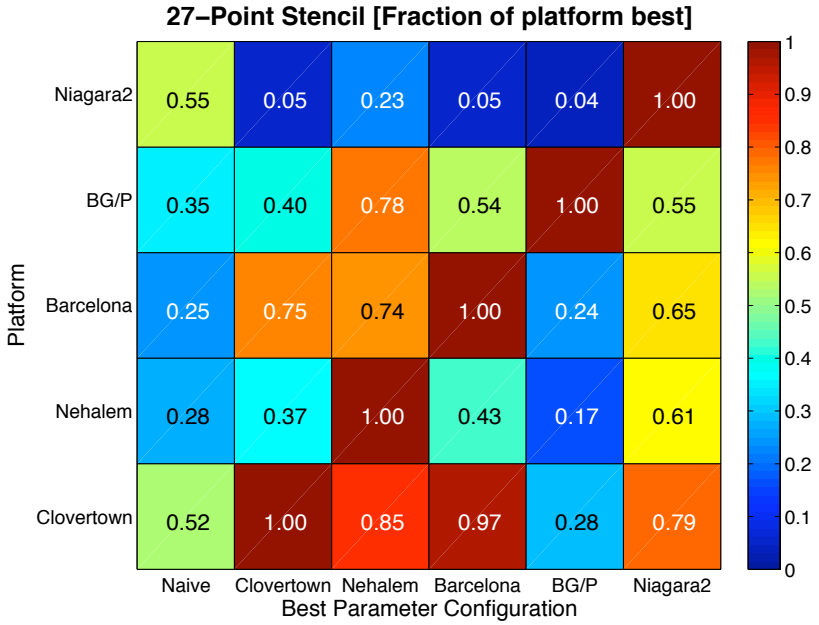


Figure 8.3: The 27-point stencil performance results from running the best parameter configuration for each platform on every other platform in our study. As a point of comparison, we also show the performance of the naïve code in the leftmost column of this plot.

8.4 Comparison of Best Parameter Configurations

Finally, we again examine the topic of whether the best parameter configuration for one system can produce good performance on other systems. This has the potential to greatly trim tuning time, as we can possibly tune for only a single architecture and employ the chosen configuration on all other platforms.

However, as we explained in Section 7.5, translating the best parameter configuration on one machine to other multicore architectures is not an exact science. In short, we always utilize the maximum number of hardware threads on a given machine, and we always attempt to run the same code with the same parameters on every other platform. If the code is not valid on a given platform (*e.g.* SIMD code on Victoria Falls), then we still transfer all the valid parameters (*e.g.* core blocking dimensions and array padding amounts), and try to modify the offending optimization so that it can still be translated onto the new platform. This seemed to be the fairest and most consistent way of carrying out this comparison.

Figure 8.3 displays the results of taking the best parameter configuration for a given architecture and running it on every other platform in our study. All results are shown as a fraction of the platform's best performance, which is set to 1. In addition, as a comparison, we also show the performance of the naïve code on each architecture in the first column.

Many intriguing trends arise from this plot. First, it again looks like the Clovertown architecture is the easiest to tune for. For instance, the best parameter configurations for Nehalem and Barcelona achieve 85% and 97% of the platform best, respectively. The best Niagara2 configuration also does very well, attaining 79% of the platform best. The only platform configuration that suffers is for the Blue Gene/P, and this is because it only has enough work for four threads; all the other available threads will sit idle. If we look at the column representing the best Blue Gene/P configuration, we see that, outside the Blue Gene/P itself, this problem plagues all the platforms in our study.

It is not surprising that most parameter configurations also execute well on Clovertown. As we explained previously, this platform is heavily memory-bound even for the 27-point stencil. However, moderate core blocking can sufficiently reduce memory bandwidth so that we still achieve a high fraction of the platform best, although the actual raw performance is still quite low.

The Barcelona platform also does an adequate job of running other tuned parameter configurations well. The parameter configurations for the two other x86 platforms achieve about three-quarters of the platform best, and the Niagara2 achieves 65% of the best. Both of these results are still at least 2.6× better than the naïve code performance. Only the Blue Gene/P code performs poorly, and that is again due to a lack of parallelism.

The Nehalem is the only x86 platform where the other tuned parameter configurations generally don't attain a high fraction of the platform best. However, for three of the other four architectures, this is due to a lack of parallelism. The Nehalem supports a total of 16 hardware threads, but the best parameter configurations for Blue Gene/P, Clovertown, and Barcelona only provide enough work for 4, 8, and 8 threads, respectively. Specifically, the best parameter configurations on these machines only generates either four or eight core blocks, so machines like the Nehalem

with more than eight hardware threads will suffer.

Given that the 27-point stencil is compute-bound on Nehalem, we expect that the Blue Gene/P configuration will only achieve a maximum of one-quarter of the platform best, while Clovertown and Barcelona configurations should attain at most half of platform best. This holds true in actuality, as Blue Gene/P, Clovertown, and Barcelona attain 17%, 37%, and 43% of the platform best, respectively. The Niagara2, which supports 128 hardware threads, is the only platform where the best parameter configuration provides sufficient parallelism to keep all 16 Nehalem threads busy. As a result, it is able to achieve 61% of the platform best.

The Sun Niagara2, which supports 128 threads, suffers the same lack of parallelism issues as the Nehalem, only to a much greater extent. In this case, the Blue Gene/P, Clovertown, and Barcelona configurations will only be exploiting between 3%–6% of the available hardware threads. Unfortunately, we see that this translates to performances ranging from 4%–5% of the platform best for these architectures. The Nehalem configuration does better, attaining 23% of the platform best, but it only provides enough work for 32 hardware threads. Clearly, the lack of sufficient parallelism is the major reason why the other platform configurations suffer on the Niagara2; none of the configurations even achieve half of the naïve code performance.

Finally, the Blue Gene/P architecture seems to show mixed results when running the tuned parameter configurations from our other platforms. However, given that it supports the fewest number of hardware threads of any of our architectures (four), we can be assured that the other parameter configurations provide sufficient parallelism. Moreover, we have already seen that the Blue Gene/P is heavily compute-bound when executing the 27-point stencil, so it is likely that the parameters for our in-core optimizations (*e.g.*register blocking) will dictate the performance on this platform. We observe that while the Nehalem configuration achieves 78% of the Blue Gene/P best, the other platform configurations only achieve between 40%–55%. In all likelihood, this is because the Nehalem in-core optimization parameters were most amenable to the Blue Gene/P.

For this experiment, we can draw similar conclusions for the 27-point stencil as for the 7-point stencil. The first priority in achieving good performance for a tuned

parameter configuration from a different platform is to ensure that there is sufficient parallelism to keep all the hardware threads on the system busy and load balanced. This was certainly an issue on Nehalem and Victoria Falls. However, this alone does not always lead to good performance. For instance, if we look at the column in Figure 8.3 representing the best Niagara2 parameter configuration, we know that this configuration has enough parallelism for 128 threads. However, it only achieves between 55%–79% of the platform best on the other machines in our study. Clearly, many of these platforms could still benefit from individualized tuning.

8.5 Conclusions

This chapter has examined the performance of the compute-intensive 27-point stencil kernel on the systems in our study. Unlike the 7-point stencil, the 27-point stencil is much more likely to be computation-bound, not memory-limited, thereby exposing a different dimension to these architectures. In general, the 27-point stencil exhibited much better parallel scaling than the 7-point stencil, since computation readily scales with more cores, but bandwidth generally does not.

However, we noted that the Roofline model typically provided a much looser upper bound than for the 7-point stencil. Generally speaking, placing upper limits on compute-bound kernels is more difficult than memory-bound kernels because the compiler plays a larger role in attained performance. The Roofline model is compiler independent, as it only looks at the hardware features of a platform in order to determine the performance upper bound. Moreover, the CSE optimization made the Roofline model much more difficult to employ, since the arithmetic intensity became variable. Both these details made the actual achieved performance much harder to bound, since the performance is compiler dependent, and the CSE optimization was utilized. Consequently, we also used the in-cache performance of the 27-point stencil as a secondary, tighter upper bound. This served us well in capping the actual attained performance.

Chapter 9

3D Helmholtz Kernel Tuning

As we will explain shortly, the Helmholtz kernel is signficantly more complex than either the 7-point or 27-point stencil problems that we have presented thus far. Therefore, it presents its own challenges to tuning and performance analysis. We will explore these topics in this chapter.

9.1 Description

We will be tackling issues with the 3D GSRB Helmholtz kernel that we did not encounter with either of the previous stencil kernels. This is because there are three major differences between the tuning of the Helmholtz kernel and our prior 7-point and 27-point stencil tuning.

First, we will no longer be solving a single large stencil problem. This GSRB Helmholtz kernel is actually taken from Chombo [9], a software framework for performing Adaptive Mesh Refinement (AMR) [3]. AMR codes often produce numerous small problems, each of which may be executing the Helmholtz kernel to iteratively "relax" the results from a larger solver (like multigrid [6, 52]). In order to mimic this type of behavior, we also perform the kernel over many small problems, which we call "subproblems". Thus, we will be taking some of the lessons learned from executing the 7-point and 27-point stencils on a single large grid and apply them to a more realistic scenario of processing many small grids.

For standardization purposes, we fixed the total memory footprint of all the

Helmholtz problems to be 0.5, 1, 2, or 4 GB. Most current multicore architectures support at least 2–4 GB of memory, so these memory restrictions should be reasonable. In addition, for simplicity, we only deal with cubic problem sizes, even though the actual Chombo code produces grids with many different aspect ratios. Table 9.1 displays the number of Helmholtz problems of a given size that fit into the specified memory footprint. This table will be useful in understanding the performance results presented later in the chapter.

Second, as we mentioned in Section 2.3.2, the Helmholtz kernel involves a variable coefficient stencil. As such, the stencil coefficients are no longer constant scalars; the coefficients themselves vary, and therefore need to be stored as separate arrays. The upshot is that the Helmholtz kernel involves seven grids; six of these grids are exclusively read from, while one grid is both read and written. These grids, as well as their dimensions (relative to some NX, NY, and NZ), are shown in Table 2.4. Unfortunately, the presence of seven grids translates to 64 Bytes of memory traffic per point, as opposed to 24 Bytes (or 16 Bytes with cache bypass) for the 7-point and 27-point stencils.

Finally, we are no longer performing a Jacobi sweep, where we simply read the data from one grid and write the stencil computation result to another. Now, we are executing Gauss-Seidel Red Black (GSRB) updates (described in Section 2.1.2). The GSRB ordering allows for parallelizable in-place updates by only modifying every other point in the write grid. Unlike the GSRB sweep definition given in Section 2.1.2, in this chapter each sweep will consist of updating a single color, not both colors. The Helmholtz kernel performs 25 flops for every updated point, but since only half the total points are updated during a single sweep, there are approximately 12.5 flops per point. As we discussed in Section 6.2, the resulting ideal arithmetic intensity is about $12.5/64 = 0.20$. This is less than the arithmetic intensity for the 7-point stencil, so we expect that this kernel will be bandwidth-bound on most architectures.

9.2 Optimization Parameter Ranges

Based on the description of the Helmholtz kernel given in the previous section, we selected the optimizations and associated parameter values shown in Table 9.2. It

Helmholtz Subproblem Size	Memory Footprint			
	0.5 GB	1 GB	2 GB	4 GB
$NX = NY = NZ = 16$	2100	4201	8403	16806
$NX = NY = NZ = 32$	277	554	1108	2217
$NX = NY = NZ = 64$	35	71	142	284
$NX = NY = NZ = 128$	4	9	18	36

Table 9.1: For a given Helmholtz problem size, the number of such problems that fit into the listed memory footprint.

is similar to the corresponding table for the 7-point and 27-point stencils (Table 7.1), but there are some differences. Most obvious, perhaps, is that we no longer perform the SIMDization, or cache bypass optimizations. We did not SIMDize the Helmholtz kernel because, in accordance with the GSRB ordering, we perform updates on every other point. On our architectures, the SIMD optimization only allows for data parallelism across two doubles– if every other point is being written to, SIMDization does not make much sense without a change of data structure or algorithm. These are larger transformations that are left as future work. We also did not utilize the cache bypass operation, since we are no longer performing out-of-place Jacobi sweeps. For the Helmholtz kernel, the `phi` array is both being read and written. In this case, we should not suffer write misses, since the needed cache lines should already be read into the on-chip cache.

One parameter that is not listed in Table 9.2 but comes into play when processing so many small subproblems is the number of *threads per subproblem*. When presenting our performance results, this parameter will be varied to test whether coarse-grained parallelism (few threads per subproblem) or fine-grained parallelism (many threads per subproblem) performs best for this kernel.

Finally, we note that the Helmholtz kernel was only evaluated on two of the platforms in this thesis, the Intel Nehalem and AMD Barcelona. However, executing this code on other multicore architectures is a subject of future work.

Category	Optimization Parameter	Name	parameter tuning range Nehalem and Barcelona
Domain Decomp	Core Block Size	CX	NX
		CY	$\{4...NY\}$
		CZ	$\{4...NZ\}$
	Chunk Size		$\{1...\frac{NY \times NZ}{CY \times CZ \times NThreads}\}$
Data Allocation	NUMA Aware		✓
	Pad by a maximum of:		7
	Pad by a multiple of:		1
Bandwidth	Prefetching Type		{none, register block, plane, pencil}
	Prefetching Distance		$\{0...64\}$
In-Core	Register Block Size	RX	$\{2...8\}$
		RY	$\{2...4\}$
		RZ	$\{2...4\}$
Tuning	Search Strategy		Iterative Greedy
	Data-aware		✓

Table 9.2: Attempted optimizations and the associated parameter spaces explored by the auto-tuner for the GSRB Helmholtz kernel. The parameter names are visually represented in Figure 4.1. For simplicity, all of the block dimensions and chunk sizes are set to be powers of two, while the prefetching distances are either zero or a power of two. In addition, all numbers are in terms of doubles.

9.3 Parameter Space Search

After deciding on the appropriate optimizations and parameter values, the subsequent iterative greedy search for the Helmholtz kernel is relatively straightforward. The details of the search are shown in Table 9.3. We see that now, unlike the 7-point and 27-point stencil tuning, there are no exceptions to the iterative greedy search; each time we add a new optimization, all the previous optimization parameters are still kept fixed. Furthermore, we no longer have multiple code generators for implementing the more drastic optimizations (*e.g.* SIMDization or CSE). While we implemented fewer optimizations for tuning the Helmholtz kernel, we we will still be comparing our performance against the same performance limits used in the previous two chapters.

Optimization Order	NUMA-Aware	Padding	Core Blocking	Register Blocking	Prefetching
Naïve					
+NUMA-Aware	S				
+Padding	F	S			
+Core Blocking	F	F	S		
+Register Blocking	F	F	F	S	
+Prefetching	F	F	F	F	S

Table 9.3: The specifics of the iterative greedy search we performed for the GSRB Helmholtz kernel. The optimizations shown in the leftmost column are applied from top to bottom. As each new optimization is applied, the optimizations labeled "F" have already had their parameter values previously fixed, while the ones labeled "S" are currently being searched over.

9.4 Single Iteration Performance

We begin our analysis of the Helmholtz kernel by examining the performance for a single iteration. Then, we will extend our performance analysis to multiple iterations, since it may provide better numerical convergence.

9.4.1 Fixed Memory Footprint

In order to understand the performance of the Helmholtz kernel, we initially fixed the memory footprint at 2 GB. Then, we began auto-tuning a single iteration of the Helmholtz kernel for 16^3, 32^3, 64^3 and 128^3 problems. The number of such subproblems that fit into our 2 GB footprint is shown in Table 9.1.

The performance results from this experiment are shown in Figure 9.1. The x-axis of these plots displays two quantities– the major x-axis represents the cubic grid dimension, while the minor x-axis displays the number of threads per problem. In addition, the bandwidth of the Optimized Stream benchmark is used to place an upper bound on performance. Unlike previous chapters, this bound is no longer

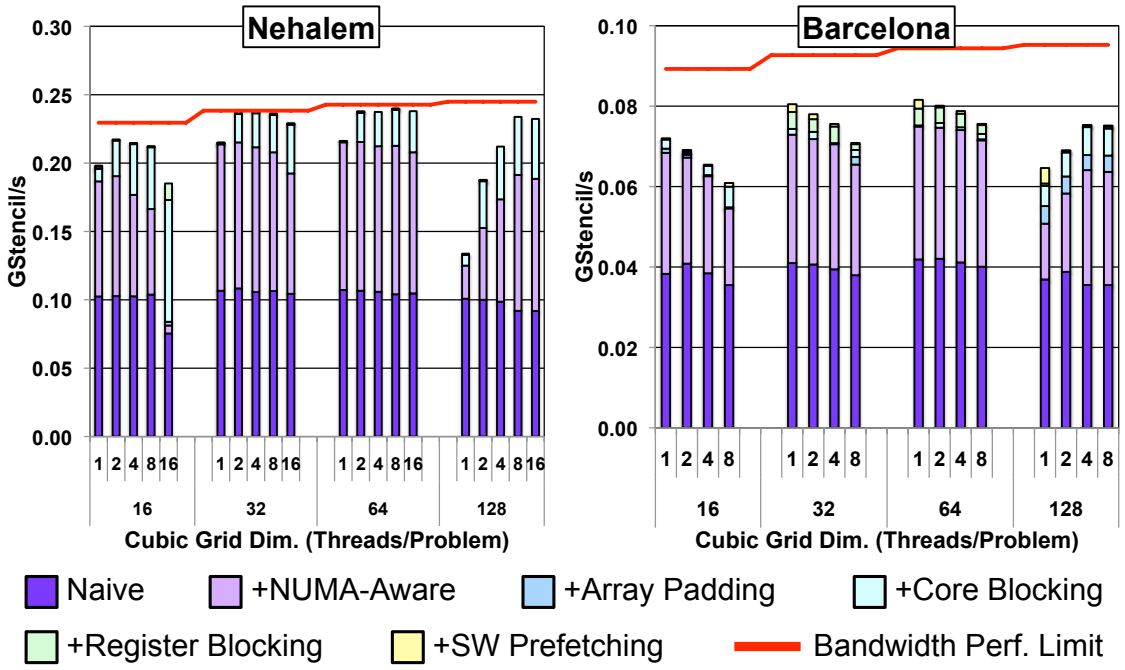


Figure 9.1: Individual performance graphs for a single iteration of the Helmholtz kernel with a 2 GB total memory footprint. The major x-axis shows the cubic grid dimensions of a single Helmholtz problem, while the minor x-axis displays the number of threads per problem. The performance limits are set by the Optimized Stream benchmark.

a straight line because we are dealing with several different problem sizes. The bandwidth bound translates to higher GStencil rates for larger problem sizes because the surface to volume ratio decreases. As such, the ghost cells from the grids in Table 2.4 constitute a smaller fraction of the overall memory traffic.

There are several intriguing trends in these plots. For one, it seems that having more threads per subproblem is beneficial for larger grid sizes. However, closer examination reveals that grid size is not the real issue, but rather the number of Helmholtz subproblems that need to be processed. We see from Table 9.1 that there are only 18 128^3 Helmholtz subproblems contained in a 2 GB footprint. If we utilize a single thread per subproblem, then on Nehalem, this means that two threads will need to process two 128^3 subproblems, while the other 14 threads will only process a single 128^3 subproblem. On Barcelona, the load imbalance is less severe, but still present; two threads will process three subproblems, while the other six threads will process only two subproblems. Clearly, when only a small number

of Helmholtz subproblems need to be processed, load balancing becomes a major performance issue. However, by employing more threads per subproblem, this effect can be mitigated. For instance, the Nehalem performance for the 128^3 problem improves by 77% when we use all 16 threads per subproblem instead of a single thread per subproblem, while for Barcelona, there is a 16% improvement from employing all eight threads per subproblem.

Another interesting trend is that for smaller subproblem sizes, utilizing fewer threads per subproblem seems to be advantageous. The one major exception to this trend is on Nehalem, where employing two threads per subproblem always performs better than using one thread per subproblem. However, this is likely due to the fact that Nehalem supports two threads per core; by having each of these threads work on a different subproblem, any core-level locality is destroyed. Outside of this artifact, we observe that having many threads processing the same small Helmholtz problem produces poor memory access patterns. Specifically, since each thread receives a relatively small amount of work per problem, the resulting memory access pattern has more short unit-stride stanzas. In previous work, we have already shown that this type of traversal causes noticeable performance degradation for machines with hardware prefetchers [28], which include both the Nehalem and Barcelona. Victoria Falls, a highly multi-threaded architecture without any hardware prefetchers, may benefit from such a memory access pattern, but this is left for future work.

Overall on Nehalem, our median auto-tuning speedup is 2.2×, much of it coming from the NUMA-aware and core blocking optimizations. However, the reason we do not achieve even larger speedups is because we are approaching the machine's bandwidth limit. We attain a median of 95% of the Optimized Stream bandwidth on Nehalem, but in one case, we are actually able to match this bandwidth. This is not wholly surprising, given that for the 7-point stencil (another memory-bound kernel on Nehalem), we were able to attain 95% of the Optimized Stream bandwidth.

On Barcelona, the performance numbers are slightly less impressive, but still good. Our median speedup from auto-tuning is 1.9×, with the great majority of this speedup coming from the NUMA-aware optimization. In addition, after full tuning we achieve between 68%–87% of the Optimized Stream bandwidth, with a median of 80%. However, the Helmholtz kernel still falls short of the 93% of Optimized Stream

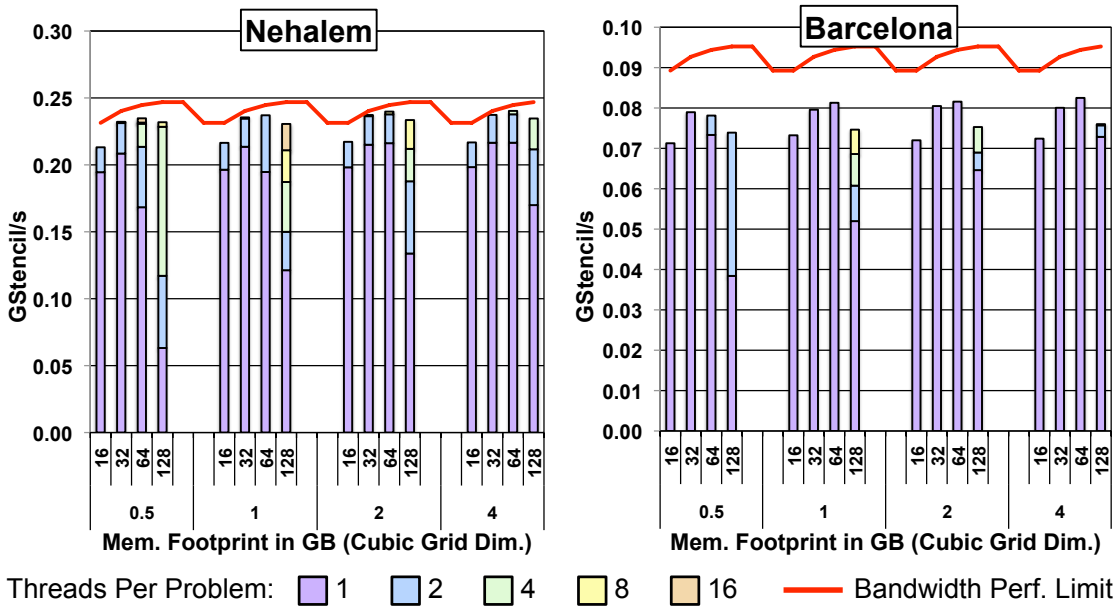


Figure 9.2: Individual performance graphs for a single iteration of the Helmholtz kernel. The major x-axis shows the total memory footprint in Gigabytes, while the minor x-axis displays the cubic grid dimensions of a single Helmholtz problem. The performance limits are set by the Optimized Stream benchmark.

bandwidth that we attained for the 7-point stencil.

9.4.2 Varying Memory Footprints

Now that we have explored the basic performance trends while having a fixed 2 GB memory footprint, we will also explore the performance from memory footprints of 0.5 GB, 1 GB, and 4 GB. Figure 9.2 presents these results, but the plots do require some explanation. First, we again show two quantities on the x-axis. The major x-axis represents the memory footprint size, while the minor x-axis shows the Helmholtz problem size. In addition, these plots no longer show the benefits of each auto-tuning optimization; we now show the fully auto-tuned performance for a variety of threads per grid. This is done by first showing the performance of a single thread per grid, and then showing the performance results from having more threads per grid *if they perform better*. Again, the bandwidth bound is no longer a straight line because we are now dealing with different problem sizes with varying surface to volume ratios. The surface to volume ratio decreases with increasing problem size, resulting in a

smaller fraction of memory traffic being used to transfer ghost cells.

We observe that on Nehalem, executing a single thread per subproblem is never optimal; at least two threads per subproblem is always required for best performance. As mentioned previously, this is likely due to the fact that Nehalem supports two threads per core. In contrast, Barcelona supports only a single thread per core. For situations where load balancing issues are unlikely– either when we have small subproblem sizes or large memory footprints– two threads per subproblem is optimal on Nehalem, while a single thread per subproblem performs best on Barcelona. However, whenever we deal with larger grid sizes or smaller memory footprints, the resulting decrease in the number of Helmholtz subproblems (as listed in Table 9.1) causes load balancing issues.

The worst case is when we attempt to process the 128^3 problem on a 0.5 GB memory footprint. This results in only four Helmholtz subproblems. If we utilize a single thread per subproblem, then only four threads will be busy in this scenario. On Nehalem, this means that 12 threads will still be left idle, while on Barcelona, four threads will not receive any work. The plots in Figure 9.2 show that we attain speedups of 3.7× and 1.9× on Nehalem and Barcelona, respectively, when we utilize more threads per problem.

Analyzing Figure 9.2 further, it seems that when load balancing is not a concern, then two threads per subproblem on Nehalem and one thread per subproblem on Barcelona is always optimal. This is again likely due to the poor memory access patterns that results from having many threads per problem. However, this is a concern as core counts continue to escalate. If only 1–2 threads per problem continues to be optimal as we approach the manycore era, then load balancing will be an even larger impediment to good scalability.

9.5 Multiple Iteration Performance

From our single iteration performance analysis of the Helmholtz kernel, we know that this kernel is memory-bound on both the Nehalem and Barcelona architectures. If this is the case, then we may be able to take advantage of this fact by performing a second (or possibly even a third) iteration “for free”. By this, we mean that we

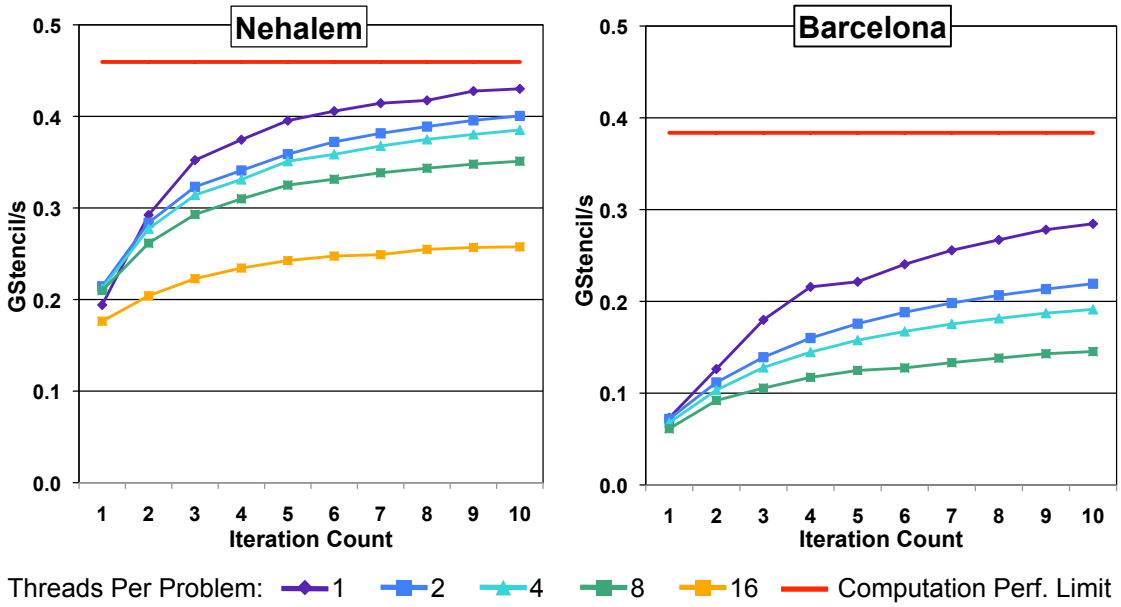


Figure 9.3: Individual performance graphs for many iterations of the Helmholtz kernel when the problem size is 16^3 and the memory footprint is 0.5 GB. The computation limit was set by the in-cache performance of a single 16^3 Helmholtz problem.

may be able to perform more than a single iteration without increasing the running time substantially because bandwidth, not computation, may still be our limiting factor. This argument can only be taken so far, though– after a certain iteration count, computation will become the bottleneck.

There were some design decisions that we made prior to exploring the performance of multiple iterations. First, we selected a Helmholtz problem size of 16^3. This was because each 16^3 problem occupies approximately 280 KB of memory (without any array padding), so several of these problems will fit into the last-level cache of either Nehalem (8 MB) or Barcelona (2 MB). In most cases, this means that the problems will stay resident in cache as we perform multiple iterations. The one exception is when we execute a single thread per problem on Barcelona. In this case, eight different Helmholtz problems may need to be in cache simultaneously, but the combined memory footprint of these problems exceeds Barcelona's last-level cache. This may result in performance degradation, but in all other cases, this should not be an issue.

The second design choice was to fix the memory footprint size at 0.5 GB. This

allows us to process 2100 16^3 Helmholtz problems, which is more than enough to avoid any load balancing issues. Larger memory footprints will result in longer running times but will not produce any new information.

Figure 9.3 shows the performance results from this experiment. Similar to the single iteration performance, we again varied the number of threads per grid from one up to the number of hardware threads available on the system. We note that when we used a single thread per problem, no barrier was present between iterations, since none was needed to preserve correctness. However, whenever we utilized multiple threads per problem, a barrier was present between iterations. This certainly gives the one thread per problem case an advantage.

In addition, in order to place an upper bound on performance, we executed 10,000 iterations over a single 16^3 Helmholtz problem. This problem size will easily fit into the last-level cache of both the Nehalem and Barcelona. The thread count for this experiment was varied so that at a minimum, one core was employed, and at a maximum, all the cores on the system were utilized. We displayed the best GStencil rate from these different thread counts.

There are two obvious trends in Figure 9.3. First, on both architectures, we observe that performance monotonically decreases as we increase the number of threads per problem. We know that the case of a single thread per problem is already at an advantage due to the lack of barriers between iterations. The fact that adding more threads per problem continues to degrade performance may again be due to the poor memory access patterns that result. Specifically, each 16^3 problem is being divided among more threads, creating shorter unit-stride stanzas for each one. However, since the Helmholtz problems are resident in cache, the issue is no longer DRAM latency, but rather the cache to register latency.

Second, as expected, as the iteration count increases, the performance for each thread count per problem monotonically increases, approaching some asymptote; this asymptote is sometimes the best in-cache performance, but not always. This behavior is due to the fact that the first iteration is by far the slowest, since the grids are read from DRAM during that time. Subsequent iterations should be limited by computation, not bandwidth, and thus are significantly faster. As a result, these later iterations help to amortize the time for the slow first iteration.

This theory is confirmed by examining the running times of the first few iterations. In the one thread per problem case on Nehalem, the second and third iterations both require about 33% of the running time of the first iteration. On Barcelona, the second iteration takes about 16% of the first iteration running time. Thus, while later iterations are not "free", they are significantly faster than the first iteration.

We note that when we performed multiple iterations, we never performed any ghost cell exchanges between iterations. To be mathematically correct, an exchange should occur after every iteration. While it is possible to perform two iterations without any ghost cell exchanges, more than two may cause stability issues with the solver. Our experiment executed up to 10 iterations without an exchange so that we could better understand the resulting performance behavior, even though it is not mathematically sound.

9.6 Conclusions

The Helmholtz kernel in this chapter was designed to imitate the behavior of AMR codes, which have very different characteristics from the 7-point and 27-point stencil problems that we had solved earlier. Since we are now dealing with many small problems, the number of threads per problem becomes another tunable parameter. In both the single iteration and multiple iteration cases, we noticed that fewer (1–2) threads per problem was usually optimal. However, the resulting coarse-grained parallelism caused serious load balancing issues when there was insufficient work for all the available threads. If fewer threads per problem continues to be optimal in the manycore era, load imbalance will be an even larger problem than it is now.

Chapter 10

Related and Future Work

This chapter covers relatively disparate topics that are related to tuning stencil codes, but were not appropriate to mention in previous chapters. Specifically, the three areas of focus are: grid traversal algorithms for performing multiple sweeps, more productive domain-specific stencil compilers, and the use of statistical machine learning in searching the auto-tuning parameter space. We also discuss directions for future research whenever they arise.

10.1 Multiple Iteration Grid Traversal Algorithms

This thesis has not discussed the topic of performing multiple stencil iterations in great detail. While we did perform multiple Helmholtz kernel sweeps in Chapter 9, we usually performed a barrier between iterations (unless a single thread was processing the entire grid). Barriers, however, are expensive operations that require communication between all threads that are processing the grid; this will likely become a larger issue as we head towards the manycore era. Moreover, barriers are typically placed after each iteration, potentially resulting in significant amounts of time spent in communication.

This section will explore other algorithms for performing multiple stencil sweeps that (mostly) avoid barriers. These algorithms attempt to take advantage of spatial and temporal locality so that we minimize memory traffic and effectively utilize any hardware prefetchers.

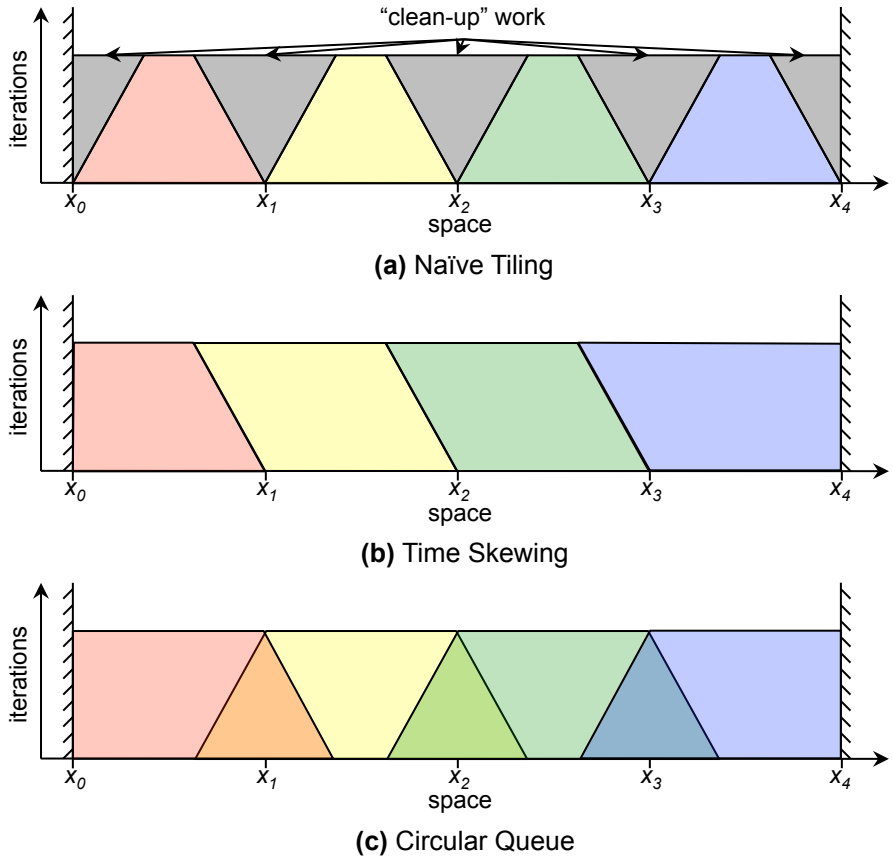


Figure 10.1: A visual depiction of three grid traversal algorithms performing multiple iterations of a 1D three-point stencil (with constant boundaries at each edge of the grid). Each colored tile is performed consecutively going from left to right, but in the case of (a) and (c), the colored tiles can be processed in parallel safely. In addition, the points within each colored tile are performed a single iteration at a time.

We note that the following algorithms can be used in conjunction with barriers. For example, instead of having a barrier after each iteration, we can perform n iterations with one of the following algorithms, execute a barrier, and then perform another n iterations. In this way, a barrier becomes a part of the overall tuning space.

10.1.1 Naïve Tiling

Naïve tiling, where we decompose the grid into blocks spatially (but not temporally), is the simplest approach to improving the memory access pattern when performing multiple iterations. An example of this algorithm is shown for a simple

1D 3-point stencil in Figure 10.1(a), where we assume that no barrier exists between iterations. This type of tiling can be performed in serial or parallel. In the serial case, we can tune the colored tiles such that each one fits in cache. This algorithm is also easily parallelizable by having each thread process one or more colored tiles.

The major problem with naïve tiling is that by not performing any type of temporal skewing, the tile boundaries retreat from each other after each iteration due to stencil dependencies. We can rectify this in two ways. First, if this is implemented as a parallel algorithm, then we can place a barrier between each iteration, which would allow the tile boundaries to remain stationary. However, as mentioned, a barrier is an expensive operation, especially as the processor count grows. The other option is to perform "clean up" work (shown as black triangles) after the initial colored tiles are complete. This, too, is undesirable, since a second, cache-unfriendly grid traversal needs to be executed.

10.1.2 Time Skewing

In order to avoid this clean up work, more contemporary approaches to stencil optimization are geared towards techniques that leverage tiling in both the spatial and temporal dimensions. This can be performed using loop skewing in order to increase data reuse within the cache hierarchy. Initial work by Wolf [60] showed loop skewing generally did not improve performance, but subsequent studies by McCalpin [34] et al and others [47, 61, 27] have shown a modified form of loop skewing called *time skewing* can improve performance for many stencil kernels.

Figure 10.1(b) shows a simplified diagram of time skewing for the same 1D three-point stencil. The grid is divided into tiles by several skewed cuts that preserve the data dependencies of the stencil. In order for the time skewing algorithm to execute correctly, each of the colored tiles needs to be processed consecutively from left to right. For example, the red tile needs to be fully calculated before beginning on the yellow tile. In general, this holds true between the n^{th} and $(n+1)^{th}$ tiles.

As a result, time skewing becomes an inherently sequential algorithm; if the tiles are executed out of order, the final result will be incorrect. Thus, the colored tiles in Figure 10.1(b) can be considered to be cache blocks. There has been some work

in making the time skewing algorithm parallel [62], but it suffers from some of the same issues with "clean up" work that we observed with naïve tiling.

10.1.3 Circular Queue

In order to execute in parallel and avoid clean up work, we can utilize the *circular queue* algorithm, which was implemented by S. Williams to parallelize a stencil code for the STI Cell processor [27, 11]. Figure 10.1(c) shows that this algorithm performs redundant work per colored tile (represented as overlapping tiles) so as to avoid synchronizing with other tiles. Since every tile is now completely self-contained, the tiles can be processed either serially or in parallel. This algorithm is similar to the naïve tiling approach, but with initial tiles that are large enough for us to avoid any subsequent clean up work.

The circular queue algorithm also includes an additional data structure that was not present in the previous two algorithms. Separate from the other grids, this algorithm maintains an additional set of planes for each iteration that is performed. The number of planes corresponds to the height of the stencil– in the case of the 3D 7-point or 27-point stencil, three planes are required. For a simple Jacobi iteration, initially the first three planes of the read grid are copied into the three planes of this external data structure. The calculation is performed in this set of planes and the result is updated in the write grid. Then, the bottom plane of this data structure is updated with the next plane from the read grid, and the plane pointers are updated properly, so that the next higher plane in our calculation can be computed. This then continues for the rest of the grid. The power of this extra "circular queue" data structure is that it exposes spatial and temporal locality explicitly by storing the working set of the stencil. This helps ensure that the hardware does not evict needed cache lines prematurely.

The major drawback of the circular queue algorithm is that redundant computation is performed in each tile. This effect is exacerbated if we are executing higher-dimensional stencils (due to the increased surface-to-volume ratio) or if we are performing many iterations. However, we can ameliorate this effect through the use of additional barriers.

10.1.4 Cache Oblivious Traversal/Recursive Data Structures

Cache oblivious optimizations optimize algorithms without using cache sizes as a tuning parameter. Such optimizations have been shown to improve performance for some classes of matrix operations [40] including matrix transpose, Fast Fourier Transform (FFT), and sorting [21]. More recently, Frigo et al [19] showed the potential of cache oblivious optimizations for improving stencil kernel performance.

In our previous work [27], we examined the *implicit* cache oblivious tiling methodology on serial stencil codes. This algorithm [19] further leverages the idea of combining temporal and spatial blocking by organizing the computation in a manner that doesn't require any explicit information about the cache hierarchy. More precisely, it considers an $(n+1)$-dimensional *spacetime trapezoid* consisting of the n-dimensional spatial grid together with an additional dimension in the time (or sweep) direction. In order to recursively operate on smaller spacetime trapezoids, we cut an existing trapezoid either in time or space and then recursively called the cache oblivious stencil function to operate on the two smaller trapezoids. This recursion continued until there was only one time step in the calculation. At this point, we executed in the usual explicit manner.

Unfortunately, the implicit cache oblivious stencil never performed as well as the explicit time skewed stencil. The poor results are partly due to the compiler's inability to generate optimized code for the complex loop structures required by the cache oblivious implementation. The performance problems remained despite several layers of optimization, which included reducing the function call overhead, eliminating modulo operations for periodic boundaries, taking advantage of prefetching, and terminating the recursion early.

As future work, it may be beneficial to use recursion not to traverse the grid, but rather to alter the grid data structure itself. Specifically, by ordering the grid in a space-filling curve layout [49], we may be able to achieve better memory locality than the normal layout (diagrammed in Figure 2.2). On architectures where we have already attained over 90% of our performance limit, we have already shown that we are maximizing the available resources. However, if we are bandwidth-bound and attaining less than this percentage, having a recursive data structure may be

useful. This is a relatively complex data structure, though. Before attempting this optimization, it may be worth executing a stencil with better locality properties to confirm that a lack of locality is the actual performance bottleneck.

10.2 Stencil Compilers

As shown in subsequent chapters, the PERL scripts we created for generating C+Pthreads code are effective in getting very good performance across a diverse set of architectures. As we know, they do suffer from a lack of program analysis and verification, which we addressed in Section 5.3 by introducing a formal stencil framework. From a programmer's standpoint, creating these PERL scripts is also not very productive. A recently introduced technique called SEJITS (Selective Embedded Just-In-Time Specialization) is an alternative approach that achieves both productivity and good performance [7].

In essence, SEJITS allows programmers to prototype code quickly in a productivity-level language (PLL) like Python or MATLAB, while achieving close to the performance of an efficiency-layer language (ELL) like CUDA or C with OpenMP. The secret is the use of provided class libraries written in a modern scripting language like Ruby or Python. These class libraries represent domain-appropriate abstractions in the productivity language. However, the library functions generate source code in an efficiency language, which is then JIT-compiled, cached, dynamically linked, and executed. Note that the JIT specialization is *selective*, meaning that the overhead of runtime specialization is paid only when performance can be significantly improved. Furthermore, the JIT is *embedded* in the PLL itself, allowing for the addition of new extensions. In our case, we could imagine writing a new specializer for each of the different PERL code generators in our study; this would likely be more productive for the programmer.

For the 7-point stencil, the slowdown from hand-coding to SEJITS was about 1.3× on Barcelona and about 2.8× on Nehalem. This is largely attributable to the specialization overhead, but this tradeoff of productivity for performance may be appropriate in many realms where the utmost performance is unnecessary.

10.3 Statistical Machine Learning

Statistical machine learning (SML) is a relatively new method for doing parameter space searches quickly yet effectively. SML has previously been used to address performance optimization for simpler High Performance Computing (HPC) problems. For instance, Brewer [4] used linear regression over three parameters (width, height, and iterations) to select the best data partitioning scheme for 2D stencil code parallelization; Vuduc [53] later used support vector machines (SVMs) to select between three optimization algorithms for dense matrix multiply of varying matrix dimensions; and Cavazos et al [17] have used a logistic regression model to predict the optimal set of compiler flags.

All three previous SML examples showed very promising results. In our case, we want SML to identify and exploit any relationships between our optimization parameters and the resultant performance metrics. *Kernel Canonical Correlation Analysis* (KCCA) [16] is a recent SML algorithm that does this effectively. Specifically, KCCA finds multivariate correlations between optimization parameters and performance metrics on a training set of data. We can then leverage these statistical relationships to optimize for performance.

In joint work with Archana Ganapathi, we utilized KCCA for optimizing the performance of the 7-point and 27-point stencil auto-tuners [22]. We first trained the algorithm by randomly sampling 1500 data points from the configuration space. Then, we evaluated these configurations on the Intel Clovertown and AMD Barcelona. We not only collected timing data during this experiment, but also relevant performance counter information. This includes data about cache line misses, TLB misses, and cache coherency traffic. However, we did not collect data on flop counts since the grid size was fixed, and thus the number of floating point operations for our stencil kernel was also constant.

KCCA then processed this data to find hidden relationships between the optimization configurations and performance. This was done by transforming the configuration vectors and performance vectors into their respective similarity matrices. These similarity matrices were then used as inputs to a generalized eigenvalue problem, the solution of which was used to create a new *KCCA data space*. In this

transformed KCCA data space, the input space and output space are maximally correlated. This is because the hidden relationships between optimization configurations and performance in the raw data space are made explicit in the transformed KCCA data space.

The power of the KCCA output space lies in the fact that it clusters the best performing points together. By finding the best performing point from our 1500 training data points, we can find this point's nearest neighbors in the KCCA output space, and they should all be high-performing data points as well. We can then map these points back to the input space to determine their optimization configurations. We can then create new test data points by taking the configuration parameters of these points and permuting them. We expect that some of these new points will also perform well.

Our results confirm that these newly generated data points do indeed perform well. Using a subset of the optimizations performed in the iterative greedy search, we found that KCCA was able to do about as well– and occasionally even better– than the iterative greedy search.

However, there are two hurdles to KCCA that need to be addressed before it can be a viable alternative to more naïve methods. First, the time to create the KCCA model is prohibitive, and this time grows superlinearly with the number of training points. We need to investigate whether there are better eigenvalue solvers for keeping the model creation time tractable. Moreover, KCCA typically requires that performance counter data be collected in conjunction with timing data. For many architectures, multiplexing certain performance counter events is impossible, so multiple runs need to be executed instead. This also adds to the auto-tuning time. We need to examine whether using *only* timing data in the KCCA model degrades the quality of the final solution, and if it does, then if the tradeoff is worthwhile.

Assuming these problems can be rectified, the future of machine learning in auto-tuning appears bright. Unlike our iterative greedy search, KCCA is a general algorithm that requires very little architectural or application knowledge. As more software developers create auto-tuners for their multicore application, fewer of them will be expert programmers; SML offers a way to search their parameter space quickly to find a high-quality solution. Moreover, if we consider alternative compilers, dif-

ferent compiler flags, a composition of several kernels, multichip NUMA systems, or heterogeneous hardware, then the size of the parameter space will continue to grow exponentially larger. SML is designed to handle these extra configuration parameters.

10.4 Summary

All of the research areas that we discussed in this chapter aim to improve key elements of our stencil auto-tuner, including faster multiple iteration performance, more programmer productivity, and better parameter searching. Several of the ideas that were introduced are applicable outside the realm of auto-tuning as well.

Chapter 11

Conclusion

The rise of multicore architectures has certainly made a great impact on the software industry. Not only are programmers now expected to write parallel code, but this code will often need to be tuned in order to fully utilize a multicore machine's resources. Given the rate at which machines are doubling their core counts, as well as the diversity of multicore architectures in today's market, we knew that hand-tuning our stencil kernels would be unproductive. Instead, we found that building a stencil auto-tuner was a better option, despite the relatively high one-time cost. Across the five heterogenous architectures of this study, all of the performance gains that we discussed were achieved with the same set of stencil auto-tuners.

Auto-tuning has enabled us to fully exploit either the bandwidth or the computational abilities in almost all cases. For instance, when executing the 7-point and 27-point stencils, four of the five architectures achieved at least 85% of either the in-cache GStencil rate or the Optimized Stream bandwidth. In addition, for the Helmholtz kernel, we achieved between 89%–100% of the peak attainable bandwidth. Across all kernels, in the two cases where we failed to attain at least 85% of the peak attainable bandwidth or computation rate, further examination is required. However, part of the problem may be that our current model assumes that either computation or bandwidth will bound the performance of the hardware. Such a simplistic model may not be accurate in this case. For instance, cache or instruction bandwidth may present other impediments to good performance. This needs to be studied further, but if we can uncover these other potential performance bounds, we may be able to

address them in software.

In almost all cases, though, we are nearly maximizing either bandwidth or computation. Thus, ultimately, we are limited by the hardware. We can only do as well as the peak memory bandwidths or computational rates of the machines we are executing on. However, current architectural trends indicate a growing disparity between computational rates and bandwidth rates on multicore chips– the so-called *memory wall*. With the number of cores per chip doubling every 18–24 months, not only does the computational ability of the entire processor steadily increase, but the bandwidth per core steadily decreases. For most stencil codes, this means that many of the cores on a chip will be effectively wasted because there is insufficient bandwidth to keep them busy. The hardware industry is trying to address this problem through better integration of memory and processors, including innovations like stacking memory chips atop processors [37, 44]. However, we will have to wait and see whether newer technologies can adequately address this computation–bandwidth gap.

Another disturbing trend was discovered during tuning of the Helmholtz kernel. For this kernel, we tuned for many small subproblems, which is characteristic of the Adaptive Mesh Refinement (AMR) codes that the kernel was ported from. Therefore, we introduced a new tunable parameter– the number of threads per subproblem. We discovered that for the Nehalem and Barcelona architectures, the optimal number of threads per subproblem was either one or two, assuming that load balancing was not an issue. Thus, coarse-grained parallelism seemed to outperform fine-grained parallelism. Unfortunately, coarse-grained parallelism leaves us exposed to load balancing issues whenever there are few subproblems to process. There are ways to deal with this problem, but they require extra work on the part of the programmer. For instance, one solution is to have all the threads work on the maximum number of load-balanced subproblems using coarse-grained parallelism, and then have all the threads process the remaining subproblems using fine-grained parallelism. Another more drastic, but general solution is to use a work queue model. This is better suited to situations where the subproblems also vary in size and shape. Nonetheless, going into the manycore era, load balancing will continue to be an issue, but one that can be addressed.

Bibliography

[1] Six-Core AMD Opteron Processor. `http://www.amd.com/us/products/server/processors/six-core-opteron/Pages%/six-core-opteron.aspx`, 2009.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[3] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] Eric Allen Brewer. *Portable high-performance supercomputing: high-level platform-dependent optimization*. PhD thesis, Massachusetts Institute of Technology, 1994.

[5] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. *SIGPLAN Not.*, 24(7):275–284, 1989.

[6] W.L. Briggs, V. Henson, and S.F. McCormick. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, Philadelphia, PA, Second edition, 2000.

[7] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. In *First Workshop on Programmable Models for Emerging Architecture (PMEA) at the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[8] John Cavazos. *Automatically constructing compiler optimization heuristics using supervised learning*. PhD thesis, University of Massachusetts Amherst, 2005. Director-Moss, J. Eliot.

[9] Chombo homepage. `http://seesar.lbl.gov/anag/chombo`.

[10] Lothar Collatz. *The Numerical Treatment of Differential Equations*. Springer-Verlag, 1960.

[11] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. In *SIAM Review (SIREV)*, 2008.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proc. SC2008: High performance computing, networking, and storage conference*, 2008.

[13] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point stencil for multicore. In *Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[14] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[15] Hikmet Dursun, Ken-Ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 642–653, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] F.R. Bach et al. Kernel independent component analysis. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Hong Kong, China, 2003.

[17] J. Cavazos et al. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2007.

[18] The FLAME Project. `http://z.cs.utexas.edu/wiki/flame.wiki/`.

[19] M. Frigo and V. Strumpen. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS05)*, Boston, MA, 2005.

[20] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[21] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[22] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A Case for Machine Learning to Optimize Multicore Performance. In *First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[23] Richard J. Hanson, Clay P. Breshears, and Henry A. Gabb. Algorithm 821: A fortran interface to posix threads. *ACM Trans. Math. Softw.*, 28(3):354–371, 2002.

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

[25] IBM Fortran 90 Pthreads Library Module. `http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topi%c=/com.ibm.xlf101l.doc/xlfopg/posix.htm`.

[26] S. Kamil, C. Chan, S. Williams, L. Oliker, J. Shalf, M. Howison, E.W. Bethel, and Prabhat. A Generalized Framework for Auto-tuning Stencil Computations. In *Proceedings of the Cray User Group Conference*, 2009.

[27] S. Kamil, K. Datta, S. Williams, L. Oliker. J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Memory Systems Performance and Correctness (MSPC)*, 2006.

[28] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 36–43, New York, NY, USA, 2005. ACM.

[29] Theodore Kim. Hardware-aware analysis and optimization of stable fluids. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 99–106, New York, NY, USA, 2008. ACM.

[30] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2007.

[31] T. Ligocki. private communication, 2009.

[32] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[33] D.F. Martin and K.L. Cartwright. Solving Poisson's Equation using Adaptive Mesh Refinement, 1996.

[34] J. McCalpin and D. Wonnacott. Time Skewing: A Value-Based Approach to Optimizing for Memory Locality. Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.

[35] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream`.

[36] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[37] S.K. Moore. Multicore is bad news for supercomputers. *Spectrum, IEEE*, 45(11):15–15, November 2008.

[38] A. Nakano, P. Vashishta, and R.K. Kalia. Multiresolution molecular dynamics for realistic materials modeling on parallel computers. *Computer Physics Communications*, 83:197–214, 1994.

[39] OpenMP. `http://openmp.org`, 1997.

[40] H. Prokop. Cache-oblivious algorithms, June 1999. Master's thesis, MIT Department of Electrical Engineering and Computer Science.

[41] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[42] G. Rivera and C. Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000. Supercomputing 2000.

[43] S. Sellappa and S. Chatterjee. Cache-Efficient Multigrid Algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[44] J. Shalf, J. Bashor, D. Patterson, K. Asanovic, K. Yelick, K. Keutzer, and T. Mattson. The MANYCORE Revolution: Will HPC LEAD or FOLLOW? *SciDAC Review*, 14:40–49, Fall 2009.

[45] F. Shimojo, R.K. Kalia, A. Nakano, and P. Vashishta. Divide-and-conquer density functional theory on hierarchical real-space grids: Parallel implementation and applications. *Physical Review B*, 77, 2008.

[46] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference (Vol. 1)*. The MIT Press, 1998.

[47] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[48] Carlos P. Sosa. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. International Technical Support Organization, first edition, December 2007.

[49] Space-filling curve. `http://en.wikipedia.org/wiki/Space-filling_curve`.

[50] Microsoft Developers Network SSE2 Intrinsics for Floating Point. `http://msdn.microsoft.com/en-us/library/4atda1f2(VS.80).aspx`.

[51] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, 2004.

[52] U. Trottenberg, C.W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, San Francisco, CA, 2001.

[53] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.

[54] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*. Institute of Physics Publishing, June 2005.

[55] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

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

[57] S. Williams, D. Patterson, L. Oliker, J. Shalf, and K. Yelick. The roofline model: A pedagogical tool for auto-tuning kernels on multicore architectures. *IEEE HotChips Symposium on High-Performance Chips (HotChips 2008)*, August 2008.

[58] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[59] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2008.

[60] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[61] D. Wonnacott. Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations. In *IPDPS:Interational Conference on Parallel and Distributed Computing Systems*, Cancun, Mexico, 2000.

[62] David Wonnacott. Time skewing for parallel computers. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 477–480, London, UK, 2000. Springer-Verlag.

Appendix A

Supplemental Optimized Stream Data

This appendix is an extension of the data presented in Figure 6.1, which displays the maximum attained DRAM bandwidths for the Optimized Stream benchmark. However, other than the Niagara2 architecture, all the plots shown take the maximum bandwidths over several different optimizations, including SIMD and cache bypass. In contrast, the plots presented in this appendix show the actual bandwidths achieved for each of these optimizations separately.

For instance, Figure A.1 shows the bandwidth results on Clovertown from three different code variants. The upper left plot shows the performance of the portable C code, which look very similar to the SIMDized code results shown in the upper right plot. It is likely that the `icc` compiler has automatically SIMDized the portable C code, resulting in both codes having similar executables. The bottom graph is SIMDized at the source code level, but also employs the cache bypass instruction (*movntpd*). As explained in Section 4.3.2, this optimization avoids reading a cache line from memory when the data only needs to be written back to DRAM. Thus, the amount of DRAM traffic for a write miss drops from 16 to 8 Bytes per point! Thus, even though the bottom graph shows lower bandwidths than the other two plots, it often exhibits better stencil performance because fewer Bytes of memory traffic need to be moved. Unfortunately, when more than five write streams per thread are present, this code shows a severe bandwidth drop. In these cases, it obviously makes

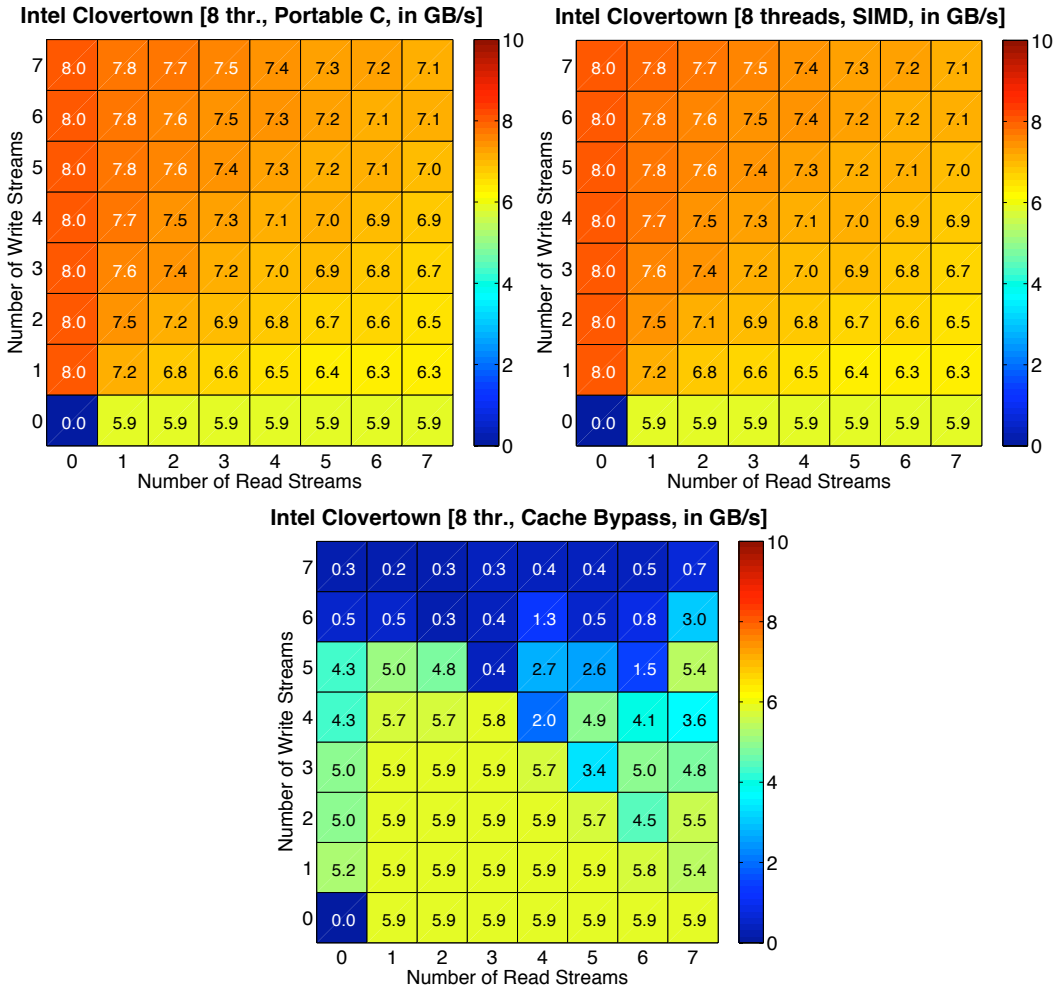


Figure A.1: The results from the Optimized Stream benchmark on the Intel Clovertown for varying numbers of read and write streams per thread.

sense to disable the cache bypass option.

The Optimized Stream performance of Nehalem, shown in Figure A.2, exhibits similarities to the Clovertown performance. First, we again see that the portable C code and the SIMD code show remarkably similar bandwidth results. Presumably, the `icc` compiler is automatically SIMDizing the portable C code. Furthermore, the cache bypass code again exhibits a severe bandwidth drop when at least six write streams are present per thread. This phenomenon is likely due to a hardware feature that requires further analysis.

The AMD Barcelona Optimized Stream results, displayed in Figure A.3, again displays the same features that were mentioned for both the Clovertown and Ne-

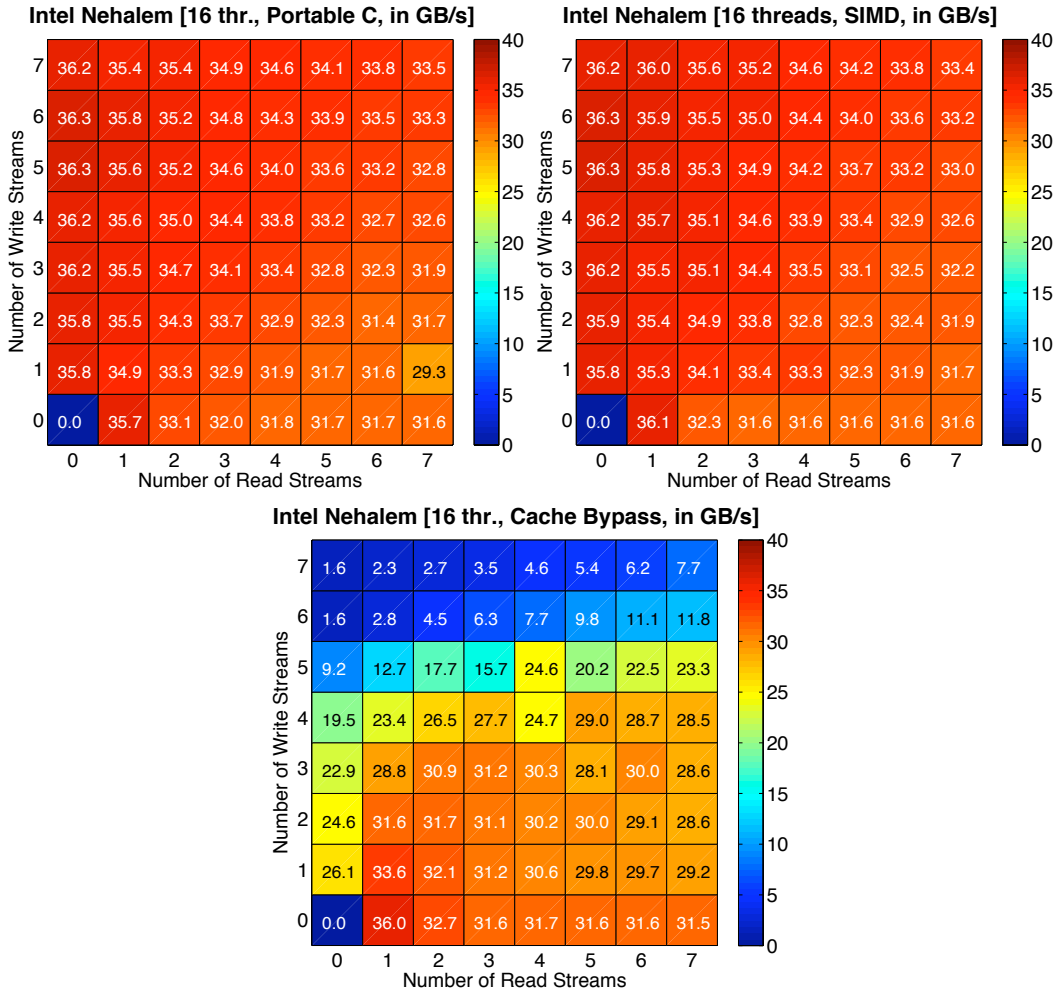


Figure A.2: The results from the Optimized Stream benchmark on the Intel Nehalem for varying numbers of read and write streams per thread.

halem. The only difference seems to be that for the cache bypass optimization, the bandwidth drop occurs with five or more write streams per thread, not six like the Intel platforms. However, the phenomenon itself is common to all three x86 architectures.

Finally, the Blue Gene/P results are shown in Figure A.4. Like the x86 architectures, the Blue Gene/P does support SIMDization (but by using different SIMD intrinsics than the x86 machines). Unlike the x86 architectures, the Blue Gene/P does not support the cache bypass optimization. Thus, the figure only shows the bandwidth performance of the portable C code and the SIMD code. Considering their similarities, it is possible that like the `icc` compiler, the `xlc` compiler is also

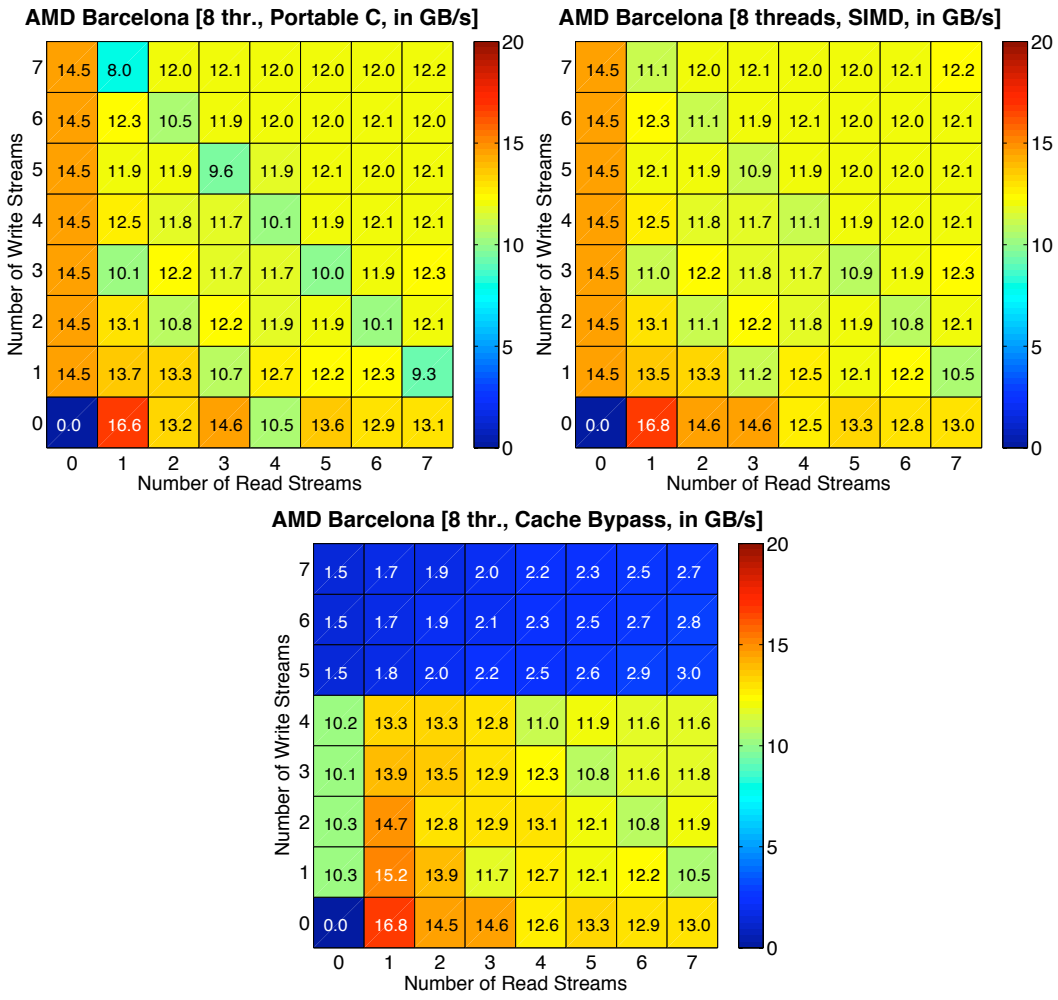


Figure A.3: The results from the Optimized Stream benchmark on the AMD Barcelona for varying numbers of read and write streams per thread.

SIMDizing the portable C code automatically.

As we explained in Section 6.3, the one caveat about Figure A.4 is that the data with zero read streams shows bandwidths higher than the platform's 13.6 GB/s DRAM pin bandwidth. Most likely, the compiler and/or hardware is changing the Optimized Stream program execution so that data is not actually being written back to DRAM in these cases. Ultimately, this will need to be confirmed with performance counter data.

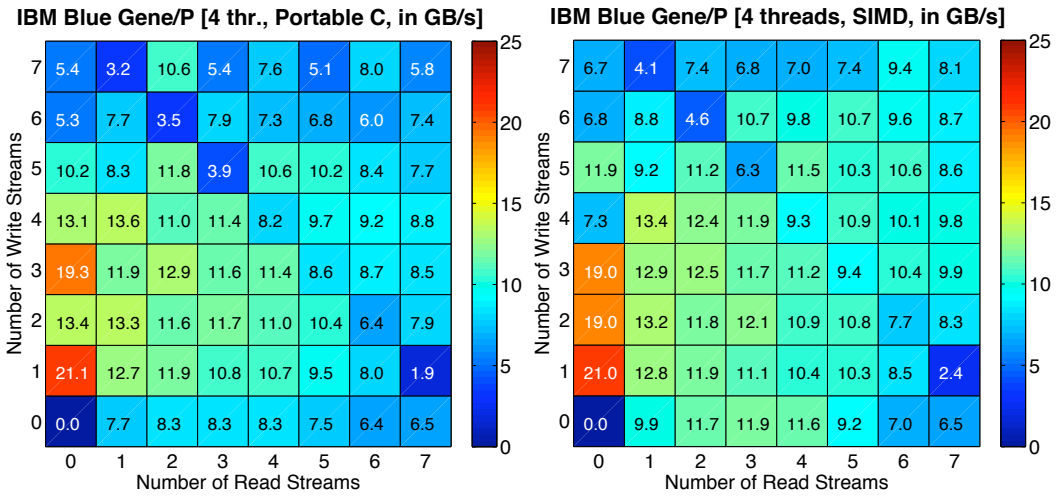


Figure A.4: The results from the Optimized Stream benchmark on the IBM Blue Gene/P for varying numbers of read and write streams per thread.