Auto-tuning Performance on Multicore Computers

Samuel Webb Williams

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-164
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html

December 17, 2008

Auto-tuning Performance on Multicore Computers

by

Samuel Webb Williams

B.S. (Southern Methodist University) 1999
B.S. (Southern Methodist University) 1999
M.S. (University of California, Berkeley) 2003

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 David A. Patterson, Chair
Professor Katherine Yelick
Professor Sara McMains

Fall 2008

Auto-tuning Performance on Multicore Computers

Abstract

Auto-tuning Performance on Multicore Computers

by

Samuel Webb Williams

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Patterson, Chair

For the last decade, the exponential potential of Moore's Law has been squandered in the effort to increase single thread performance, which is now limited by the memory, instruction, and power walls. In response, the computing industry has boldly placed its hopes on the multicore gambit. That is, abandon instruction-level parallelism and frequency-scaling in favor of the exponential scaling of the number of compute cores per microprocessor. The massive thread-level parallelism results in tremendous potential performance, but demands efficient parallel programming — a task existing software tools are ill-equipped for. We desire *performance portability* — the ability to write a program once and not only have it deliver good performance on the development computer, but on all multicore computers today and tomorrow.

This thesis accepts for fact that multicore is the basis for all future computers. Furthermore, we regiment our study by organizing it around the computational patterns and motifs as set forth in the Berkeley View. Although domain experts may be extremely knowledgeable on the mathematics and algorithms of their fields, they often lack the detailed computer architecture knowledge required to achieve high performance. Forthcoming heterogeneous architectures will exacerbate the problem for everyone. Thus, we extend the auto-tuning approach to program optimization and performance portability to the menagerie of multicore computers. In an automated fashion, an auto-tuner will explore the optimization space for a particular computational kernel of a motif on a particular computer. In doing so, it will determine the best combination of algorithm, implementation, and data structure for the combination of architecture and input data.

We implement and evaluate auto-tuners for two important kernels: Lattice Boltzmann Magnetohydrodynamics (LBMHD) and sparse matrix-vector multiplication (SpMV). They are representative of two of the computational motifs: structured grids and sparse linear algebra. To demonstrate the performance portability that our auto-tuners deliver, we selected an extremely wide range of architectures as an experimental test bed. These include conventional dual- and quad-core superscalar x86 processors both with and without integrated memory controllers. We also include the rather unconventional *chip multithreaded* (CMT) Sun Niagara2 (Victoria Falls) and the heterogeneous, local store-based IBM Cell Broadband Engine. In some experiments we sacrifice the performance portability of a common C representation, by creating ISA-specific auto-tuned versions of these kernels to gain architectural insight. To quantify our success, we created the Roofline model to perform a bound and bottleneck analysis for each kernel-architecture combination.

Despite the common wisdom that LBMHD and SpMV are memory bandwidth-bound, and thus nothing can be done to improve performance, we show that auto-tuning

consistently delivers speedups in excess of 3× across all multicore computers except the memory-bound Intel Clovertown, where the benefit was as little as 1.5×. The Cell processor, with its explicitly managed memory hierarchy, showed far more dramatic speedups of between 20× and 130×. The auto-tuners includes both architecture-independent optimizations based solely on source code transformations and high-level kernel knowledge, as well as architecture-specific optimizations like the explicit use of single instruction, multiple data (SIMD) extensions or the use Cell's DMA-based memory operations. We observe that the these ISA-specific optimizations are becoming increasingly important as architectures evolve.

Professor David A. Patterson
Dissertation Committee Chair

To those who always believed in me,
even when I didn't.

Contents

List of Figures

List of Tables

List of symbols

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
AMR	Adaptive Mesh Refinement
ASV	Alberto Sangiovanni-Vincentelli
ATLAS	Automatically Tuned Linear Algebra Software
AVX	Advanced Vector Extensions (Intel)
BCOO	Blocked Coordinate (sparse matrix format)
BCSR	Blocked Compressed Sparse Row (sparse matrix format)
BIOS	Basic Input/Output System
BLAS	Basic Linear Algebra Subroutines
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CMOS	Complementary Metal-Oxide-Semiconductor
CMT	Chip Multithreading (Multicore + Multithreading)
COO	Coordinate (sparse matrix format)
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSC	Compressed Sparse Column (sparse matrix format)
CSR	Compressed Sparse Row (sparse matrix format)
CUDA	Compute Unified Device Architecture (NVIDIA)
DAG	Directed Acyclic Graph
DDR	Double Data Rate (DRAM)
DGEMM	Double-Precision General Matrix-Matrix Multiplication
DIB	Dual Independent (front side) Bus
DIMM	Dual In-line Memory Module (DRAM)
DIVPD	Divide, Parallel, Double-Precision (an SSE instruction)
DLP	Data-Level Parallelism
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
ECC	Error-Correcting Codes
EIB	Element Interconnect Bus (Cell)
ELL	Sparse matrix format used by ELLPACK
FBDIMM	Fully Buffered DIMM
FFT	Fast Fourier Transform

FFTW	Fastest Fourier Transform in the West
FMA	Fused Multiply-Add
FMM	Fast Multipole Method
FPU	Floating-Point Unit
FSB	Front Side Bus
GCSR	General Compressed Sparse Row
GPU	Graphics Processing Unit
GTC	Gyrokinetic Toroidal Code
HPC	High Performance Computing
ILP	Instruction-Level Parallelism
IRAM	Intelligent RAM (a chip from Berkeley)
ISA	Instruction Set Architecture
KCCA	Kernel Canonical Correlation Analysis
LBM	Lattice-Boltzmann Method
LBMHD	Lattice-Boltzmann Magnetohydrodynamics
LU	LU Factorization (Lower-Upper Triangular)
MACC	Multiply Accumulate
MCH	Memory Controller Hub
MCM	Multi-Chip Module
MESI	Cache Coherency Protocol
MFC	Memory Flow Controller (Cell)
MHD	Magnetohydrodynamics
MLP	Memory-Level Parallelism
MOESI	Cache Coherency Protocol
MPI	Message Passing Interface
MPICH	A Free, Portable Implementation of MPI
MT	Multithreading
NNZ	Number of Non-Zeros in a Sparse Matrix
NUMA	Non-Uniform Memory Access
OSKI	Optimized Sparse Kernel Interface
PhiPAC	Portable High Performance Ansi C
PDE	Partial Differential Equation
PIC	Particle in Cell code
PPE	PowerPC Processing Element (Cell)
QCD	Quantum Chromodynamics
RGB	Red, Green, Blue
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Language
SBCSR	Sparse Blocked Compressed Sparse Row
SIMD	Single-Instruction, Multiple-Data
SIP	System In Package approach to integration
SKY	Skyline Sparse Matrix Format
SMP	Shared Memory Parallel (A multiprocessor computer)
SOR	Successive Over-Relaxation

SPD	Symmetric, Positive Definite
SPE	Synergistic Processing Element (Cell)
SPMD	Single-Program, Multiple-Data
SpMV	Sparse Matrix-Vector Multiplication
SpTS	Sparse Triangular Solve
SPU	Synergistic Processing Unit (Cell)
SSE	Streaming SIMD Extensions (Intel)
STI	Sony-Toshiba-IBM — the partnership that produced Cell
TLB	Translation Lookaside Buffer
TLP	Thread-Level Parallelism
UMA	Uniform Memory Access
UPC	Unified Parallel C
VF	Victoria Falls (multisocket Niagara2 SMP)
VIS	Visual Instruction Set (Sun's SIMD)
VL	Vector Length
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VMX	PowerPC SIMD unit (AltiVec)
WB	Write-Back cache
WT	Write-Through cache
XDR	RAMBUS eXtreme Data Rate DRAM

Acknowledgments

First, I want to thank my thesis advisor, Dave Patterson, for his research guidance, exuberance, patience, and long-term career advice. These were lessons that cannot be learned in any classroom, but I'll remember them forever.

Second, I'd like to thank Kathy Yelick, Sara McMains, and Jim Demmel for agreeing to sit on my dissertation committee. Their feedback after the qualification exam and throughout the last few years has been invaluable.

Lawrence Berkeley National Laboratory provided the means for a fundamental change in the direction of my research when they offered funding in January of 2005. Although the funding is hugely appreciated, the greatest benefit was the opportunity to work with researchers like Leonid Oliker and John Shalf. Working with them for the last four years has been the greatest boost to my career, and I hope this will continue in the forthcoming years.

I'd like to thank the Berkeley's BeBOP group for their support including several members in particular. I've had a number of extremely productive discussions with Kaushik Datta, Shoaib Kamil, and Rajesh Nishtala, in which we would flush out the ins and outs of half-baked or random ideas on auto-tuning and parallel computing. In addition, my telecons and discussions with Rich Vuduc were incredibly useful.

My research at Berkeley began in the IRAM project. Working in this group provided many of the fundamentals that carried me into my work on multicore processors. I wish to express my gratitude to all members, but especially Christos Kozyrakis and Joe Gebis. Over the years, our lively and engaging conversations ranged from thought experiments in computer architecture to geopolitics to quoting the Simpsons.

I am deeply indebted to the Par Lab / RADLab sysadmins — Jon Kuroda, Mike Howard, and Jeff Anderson-Lee — for their support and implementation of my seemingly unreasonable and frequent admin requests. Moreover, they performed a miraculous job keeping our diverse preproduction hardware up and running.

I'd like to thank the many researchers within the Parallel Computing Laboratory, but several deserve individual acknowledgment. Over the summer of 2008, Andrew Waterman was incredibly helpful in flushing out the details of the Roofline model and associated paper. In addition, my discussions with Heidi Pan, Jike Chong, and Bryan Catanzaro over the years provided the depth and clarity I needed to both explain auto-tuning to the laymen, inspire many of the future directions in auto-tuning, and garner a broader understanding of the applications of parallel computing.

I would like to thank Sun Microsystems for their donations of four generations of Niagara computers. The rapid increase in thread-level parallelism drove much of the work presented in this thesis. Similarly, I would like to thank IBM, the Forschungszentrum

Jülich, and AMD for remote access to their newest Cell blades and Opteron processors. Moreover, I specifically wish to thank a number of industrial contacts, including Michael Perrone, Fabrizio Petrini, Peter Hofstee, Denis Sheahan, Sumti Jairath, Ram Kunda, Brian Waldecker, Joshua Mora, Allan Knies, Anwar Ghuloum, and David Levinthal for their willingness to answer my detailed questions or put me into contact with those who can. Finally, I wish to acknowledge the Millennium Cluster group for access to the PSI cluster as well as installing and maintaining the Niagara and Opteron machines.

Finally, I'd like to thank my parents, Richard and Kay, and my brother, Joe, for their enduring love and support. They have inspired and spurred me to achieve.

This research was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231, by Microsoft and Intel funding through award #20080469, and by matching funding by U.C. Discovery through award #DIG07-10227.

Chapter 1

Introduction

Architectures developed over the last decade have squandered the promises of Moore's Law by expending transistors to increase instruction- and data-level parallelism — not to mention cache sizes — in the hope that increases in these linearly translates into increases in single-thread performance. During this period, architectural innovation was constrained by assumptions that the Instruction Set Architecture (ISA) must be backward compatible, that compiled code must run transparently on successive generations of processors, and that applications must be single-threaded. In this dissertation we diverge from tradition by combining three novel concepts in architecture, tools, and program conceptualization.

First, we accept multicore as the architectural paradigm of the future. Multicore integrates two or more discrete processing cores into a single socket. Significant innovation can still be made in how one interconnects these cores with each other, main memory, and I/O. Multicore is the only viable solution capable of translating the potential of Moore's law into exponential increasing performance by exponentially increasing the number of cores. Multicore is area efficient, power efficient, and VLSI-friendly. Its biggest limitation is the fact that software must be explicitly parallelized into multiple threads to exploit multiple cores. As compilers have failed at this, the responsibility will fall on domain and architectural expert programmers writing applications, libraries, or frameworks.

Second, we exploit the concept of computational motifs as set forth in the Berkeley View [7] to structure our work. Moreover, motifs abstract away rigid implementations and algorithms in favor of a flexible implementation constrained only by the mathematical problem to be solved. Such an approach gives us the freedom to innovate in software to keep pace with the innovations in multicore architecture. In the future, extensible motif frameworks or libraries will be created by domain experts and used by application framework programmers.

Finally, given the breadth and continuing evolution of multicore architectures, any single implementation of a kernel of a motif optimal for one architecture will be obsolete on its successors. To that end, we embrace automated tuning or auto-tuning as a means by which we can write one implementation of a kernel and achieve good performance on virtually any multicore architecture today or tomorrow.

Thus, this thesis applies the auto-tuning methodology to a pair of kernels from

two different motifs — Lattice-Boltzmann Magnetohydrodynamics (LBMHD) and Sparse Matrix-Vector Multiplication (SpMV). We evaluate the performance portability of this approach by benchmarking the resultant auto-tuned kernels on six multicore microarchitectures. Due to the diversity of architectures and kernels, we qualify our results using the Roofline model to bound performance.

Thesis Contributions

The following are the primary contributions of this thesis.

- We create the Roofline model, a visually-intuitive graphical representation of a machine's performance characteristics. Although we only define the parameters relevant for this dissertation, we outline how one could extend the Roofline model by adding additional ceilings, using other communication or computation metrics, or even how one could use performance counters to generate a runtime Roofline model.
- We expand auto-tuning to the structured grid motif. To that end, we select one of the more challenging structured grid kernels — Lattice-Boltzmann Magnetohydrodynamics (LBMHD) — to demonstrate our approach.
- As all future machines will be multicore, and existing auto-tuners optimize single-threaded performance, we introduce techniques for auto-tuning multicore architectures. Note, this is a fundamentally different approach from tuning single-thread performance and then running the resultant code on a multicore machine. Motivated by trends in computing, we believe heuristics are an effective means of tackling the search space explosion problem for kernels with limited locality. We apply this multicore auto-tuning approach to both the LBMHD and SpMV kernels on six multicore architectures.
- Fourth, we analyze the breadth of multicore architectures using these two auto-tuned kernels. We provide insights that can be exploited by architects and algorithm designers alike. For example, performance can be easily attained using local store-based architectures. Moreover, although multithreaded architectures greatly simplify many aspects of program optimization, they place significant demands on cache capacity and associativity.
- Finally, we discuss future directions in auto-tuning. This spans four axes: making auto-tuning more efficient, expanding auto-tuning to kernels in other motifs, achieving generality within each motif, and ideas on composing different motifs in an application.

Thesis Outline

The following is an outline of the thesis:

Chapter 2 provides context, motivation, and background for this thesis. After examining the trends in computing, it discusses the productivity-minded — and Berkeley View-inspired — motifs as well as the performance-oriented concept of auto-tuning. It

then coalesces these concepts into the premise for this work: motif-oriented auto-tuning of kernels on multicore architectures.

Chapter 3 discusses the experimental setup. This setup includes the computers and novel architectural details, as well as the details and motivations of the selected programming model, language, tools, and methodology.

Chapter 4 introduces the Roofline Performance Model. The Roofline Model uses bound and bottleneck analysis to produce a visually-intuitive figure representing architecture performance as a function of locality and utilized optimizations. We use the Roofline model throughout the rest of this work to predict performance, qualify our results, and quantify any further potential performance gains. Our discussion of the Roofline Model is longer than required for this work because we believe the Roofline Model will have value to performance-oriented programmers well beyond the scope of the kernels and auto-tuning approach used in this thesis.

Chapter 5 expands upon one of the computational motifs introduced in Chapter 2, namely the structured grid motif. It begins by describing the orthogonal characteristics of structured grids and computations on structured grids. It finishes with a discussion of methods by which one can accelerate operations on structured grids. By no means is this chapter comprehensive, but it encapsulates far more than the knowledge required to understand the next chapter.

Chapter 6 applies the auto-tuning technique to a structured grid-dominated application — Lattice-Boltzmann Magnetohydrodynamics (LBMHD) — using the multisocket, multicore shared-memory parallel computers (SMPs) introduced in Chapter 3. It begins by modeling LBMHD using the Roofline Model introduced in Chapter 4. This provides reasonable performance bounds and implies the requisite optimizations. The chapter then proceeds with an incremental construction of a structured grid auto-tuner discussing the motivation, implementation, and benefit for each optimization. Overall, we saw up to a 16-times increase in performance. It also compares and analyzes the multicore SMPs in the context of LBMHD. Finally, the chapter concludes with future work and insights applicable to auto-tuning other structured grid codes.

Reminiscent of Chapter 5, Chapter 7 expands upon the sparse linear algebra motif. It begins by differentiating sparse linear algebra from its dense brethren. To that end, it discusses the sparse BLAS kernels, as well as both direct and iterative sparse solvers. The chapter proceeds with a discussion of how these kernels and methods are used in the finite difference method. As a primer for the sparse matrix-vector multiplication (SpMV) case study used in the next chapter, we first discuss a variety of sparse matrix formats likely to survive into the multicore era.

Reminiscent of Chapter 6, Chapter 8 applies auto-tuning to SpMV on multicore architectures. In begins with a case study of SpMV that includes an analysis of the reference implementation as well as previous serial auto-tuning efforts. This is followed by sections that use the Roofline Model to analyze SpMV and detail the matrices used as benchmark data. The chapter then proceeds with an incremental construction of a multicore SpMV auto-tuner discussing the motivation, implementation, and benefit for each optimization, with the use of local stores delivering a 22× speedup over the reference PPE implementation. It also compares and analyzes the multicore SMPs in the context of SpMV. Finally, the

chapter concludes with future work and insights applicable to auto-tuning other kernels in the sparse linear algebra motif.

Chapter 9 integrates the results from auto-tuning individual kernels and discusses insights for the entire structured grid and sparse motifs. Moreover, it examines the future directions in auto-tuning including expanding auto-tuning to kernels from other motifs, broadening auto-tuning to motif frameworks, as well as discussing ideas for making auto-tuning more efficient.

Chapter 10 concludes this thesis with an executive summary of the contributions, results, and ideas for possible future work.

Chapter 2

Motivation and Background

This thesis is focused on providing a productive means of attaining good performance for a variety of computational motifs across the breadth and evolution of multicore architectures. To that end, this chapter discusses the motivation for maximizing performance or throughput as well as the requisite background material. Section 2.1 justifies performance as the preeminent metric of success for this work, while Section 2.2 addresses the trends in computing. Section 2.3 introduces dwarfs, patterns, and motifs. Moreover, it dispels certain myths and misconceptions about them. Next, Section 2.4 introduces auto-tuning and discusses previous attempts to optimize kernels from various motifs using auto-tuning. Finally, Section 2.5 unifies the patterns, motifs, and auto-tuning into a productive performance solution given the trends in computing.

2.1 Why Optimize for Performance?

In high performance computing, jobs are executed in batched mode on a large distributed memory machine composed of shared memory parallel (SMP) nodes. Each node is composed of one or more sockets. As it is rare that every job will use the entire machine, the machine is partitioned and jobs are run concurrently. Moreover, it is rare that two jobs will share a single node, however. The primary metric of success for such machines is aggregate throughput. Maximum aggregate throughput for a fixed hardware configuration is attained by maximizing node performance. The latency, as measured by the sum of the time a job spends queued and the time a job spends executing, is of secondary concern to operators. Conversely, users demand low latency to maximize productivity in the hypothesis-execution-analysis cycle. Although adding additional nodes can improve throughput by reducing queue time, it costs both additional money and power. Optimizing performance can reduce both queue time and execution time. At extreme scales, doubling performance via doubling capacity is usually far less cost-effective than paying a small number of programmers to optimize single-node performance and extract both higher throughput and lower latency.

At the opposite end of the spectrum, in handheld or personal computing devices, maintaining soft real-time constraints is often the metric of interest. However, as all personal computing devices for the foreseeable future will be composed of multicore processors, they

can be conceived of as SMPs. As it is impossible to add additional nodes to attain soft real-time performance, optimizing SMP performance is the only feasible solution.

Thus, we have chosen performance — that is, time to solution — as the metric of success for this work. Moreover, we restrict ourselves to performance using an entire shared memory parallel (SMP) node. However, we relax the constraints by allowing offline optimization.

We acknowledge that both power and energy can be as important a metric as performance. In general, we have observed that while increased application performance may be accompanied by increases in total power. However, if the increase in application performance exceeds the increase in power, the total energy required to solve said problem has actually been reduced.

2.2 Trends in Computing

In this section, we examine several of the trends in computing. We limit this examination to a single SMP. These trends act as constraints and guide us to our solution.

2.2.1 Moore's Law

Moore's Law [96] postulates that the number of transistors per cost-effective integrated circuit will double every two years. Each advance in process technology — a technology node — reduces the transistor gate length by about 30%. If one accepts that all other components will scale similarly, then the area required to re-implement an existing design in the next process technology is halved. In practice, this perfect scaling has not been achieved for CPUs since many not all design rules linearly scale with transistor gate length. However, increased yields and wafer sizes has made it feasible to increase the chip area. Thus, if density (transistors per mm^2) increases by 70%, and chip area increases by 17%, then the total number of transistors per integrated circuit has doubled — a reasonable path to fulfill Moore's Law.

As silicon lattice constants are approximately half a nanometer, and transistors must be at least dozens of atoms wide, the ultimate limit to transistor scaling is rapidly approaching. Current transistor gate lengths are around 45nm. As such, there are perhaps only four more technology nodes before planar scaling will completely fail. Moreover, it will become increasingly unlikely that subsequent process technology nodes will deliver quadratically smaller chips.

Stacked designs will stack multiple chips in a package and connect them with *through-silicon vias*. This may provide a stopgap measure to supplement the demand for increasing transistor counts, as technology nodes become more widely spaced in time. Perhaps a new Moore's law will account for smaller transistors, increased chip size, and increasing number of chips per stack. Unfortunately, cost will likely scale with the number of chips per stack. Thus, in the long term, a more efficient 3D technology implementation is required.

2.2.2 Frequency and Power

On average, in the 1990s frequency doubled every 18 months. This rate, more than any other factor, drove commentators to equate Moore's law with ever increasing performance. However, this increase in frequency was only achieved by allowing a steady increase in power. Today, consumer chips are limited to between 80 and 120W. This limit is the practical, cost-effective, range for air-cooling. The limit for liquid cooling is perhaps twice this. Nevertheless, the green computing movement [62, 46] and mobile and embedded computing demands will place ever increasing downward pressure on power. As such, power and frequency will likely not increase in server processors, and will be forced steadily lower in mobile environments.

Today, chip power often constitutes about half of an SMP's total power. Moreover, SMP server power will likely range from 300 to 500W. We don't expect this to dramatically change in the future. When scaling out, power will scale linearly with performance capability.

2.2.3 Single Thread Performance

In the past, single thread performance has been the metric of interest. Over the years, there have been numerous attempts to attain more instruction- or data-level parallelism within a single thread. Additionally, much of Moore's law has been diverted into increasing the cache sizes in an effort to reduce average memory access time.

Superscalar processors have slowly grown to issue four to six instructions per cycle. Much of this evolution has been constrained by the serial, fixed binary requirements of personal computing. Dynamic discovery of instruction-level parallelism is not a power or area-efficient solution. As such, there are severe constraints on the size of the out-of-order window used in superscalar processors — typically less than 150 instructions. Coupled with near flat increases in frequency and a lack of further ILP, single-threaded application performance has nearly saturated [68].

As multimedia applications have become increasingly important over the last decade, manufactures have scrambled to exploit some form of data-level parallelism. Virtually all have embraced single-instruction, multiple-data (SIMD) instructions as the solution [41, 74, 75, 123]. In a little over 12 years, the x86 implementations will have increased from 64-bit SIMD registers and datapaths to 256-bit SIMD registers and datapaths [74]. Doubling peak performance every 6 years is a very slow, but noticeable effect, especially in the era of near constant frequencies. However, unlike true vector implementations, every generation of SIMD instructions requires re-optimization and recompilation. One can only reap the benefit of transitioning from half-pumped (or perhaps quarter-pumped in the case of AVX) to fully pumped, after recompiling for AVX.

2.2.4 The Multicore Gambit

Multicore — the integration of multiple processing cores onto a single piece of silicon — has become the de-facto solution to improving peak performance given the power and frequency-limited single-thread performance. The result is that all computers have become shared memory parallel (SMP) computers. Assuming all CMOS scaling is directed

at increasing core counts, it has been postulated that the number of cores will double every two years [1, 7, 44].

However, perfect linear scaling is not possible, and doubling the core count will require a substantial increase in chip area. For example, in migrating from 90 nm to 65 nm, NVIDIA nearly doubled the core count in their top-of-the-line GPUs at the expense of a 20% increase in chip area [114]. Conversely, fixed designs will not see chip area cut in half by migrating to a smaller process technology. Since its introduction, the fixed-design 8-core Cell processor has migrated from 90 nm to 65 nm to 45 nm. In doing so, both chip area and power have only been reduced by half [128]. Thus, the chip market has bifurcated into commodity chips constrained by area and power, and an extreme chip market where customers are willing to pay for giant chips and high power. Core counts in the commodity world might increase at only 20 to 25% per year, where core counts in the extreme world may increase at up to 40% per year.

Multicore does not invalidate either the single source or single binary model — the ability to maintain one code base and distribute one binary. However, it does require applications be written in a manner that is scalable with the number of cores in the machine the application will be run on. This doesn't require an agnostic approach, as the application can query the OS to determine how many threads are available. Thus, that one binary, could query the OS to determine the number of available threads, and select the appropriate algorithmic implementation.

There are three main styles for multicore architectures: homogeneous superscalar multicore, homogenous simple multicore, and heterogeneous multicore. Homogeneous superscalar multicore means taking existing giant superscalar cores and integrating more and more onto a single chip. Typically, innovation is implemented outside of the cores in the cache hierarchy. These designs can still deliver very good single thread performance as they exploit all the architectural paradigms of the last 20 years. Homogenous simple multicore integrates many more scalar or simple in-order cores together. As the cores are simpler and thus much smaller, many more can be integrated in a fixed area of silicon. These designs have clearly shifted their focus from single thread performance to multithreaded throughput. As such, applications must be rewritten to express as much thread-level parallelism as possible. Nevertheless, the mass of simple cores running multithreaded applications will likely deliver better performance than their superscalar counterparts. Heterogeneous multicore achieves the best of both worlds by integrating many scalar or simple in-order cores with one or more complex superscalar cores. Single thread performance will remain good without recompilation, but when multithreaded applications are available, they can exploit the bulk of the computing capability.

Today, most multicore chips are limited to about eight cores. It is difficult to extrapolate the software requirements on processors with 32 or more cores using only 8. There are two possible solutions to this dilemma. First, multi-socket SMPs are available. However, as discussed below, it is best not to exceed two sockets on a SMP using a snoopy cache coherency protocol. At the very least, a dual-socket SMP provides a glimpse into multicore architectures two to four years from now. Second, there are designs in which each core is hardware multithreaded. Multithreading will provide nearly an order of magnitude increase in thread counts and provide insights into multicore chips 6 to 12 years from now.

2.2.5 DRAM Bandwidth

Although DRAM capacity will continue to scale as well if not better than the number of cores, the bandwidth per channel between the DRAM modules and the processor will scale at perhaps 20% per year [105]. Remember, markets unbridled by area constraints may scale the number of cores per chip at 40% per year. To compensate for this potential discrepancy in scaling trends, manufactures are slowly increasing the number of channels per socket. Today, low-end processors still have only one channel per socket, but consumer processors often have two or three, and high-end designs have between four and eight channels per socket. If the number of cores increases at 40% per year, and the bandwidth per channel increases at 20% per year, then to maintain a constant FLOP:byte ratio, the number of channels per socket must also increase at 20% per year — or double every four years. Such a trend is not cost-effective in the long term. As a result, without an innovative, cost-effective interface to main memory, cost-effective computers will be increasingly memory-bound.

2.2.6 DRAM Latency

In the 1990s, with processor frequencies doubling every 18 months, much play was given to the fears of DRAM latencies approaching 1000 core clock cycles. This trend is unlikely to happen, as core clock frequencies have saturated and most designs currently have integrated memory controllers with DRAM latencies including coherency checks under 200 ns. We suggest that multithreaded applications have transformed the challenge from latency-limited computing to throughput-limited computing. By throughout-limited, we mean reducing memory, cache or instruction latency will not improve performance because either the memory, cache or result bus is fully utilized. As such, the challenge can be succinctly expressed via Little's Law [10]. Little's Law states that the requisite concurrency expressed to the memory subsystem to achieve peak performance is the latency-bandwidth product. We believe that multicore is an effective and scalable technique in addressing Little's Law. The number of cores dictates the concurrency expressed to the memory subsystem. As the number of cores might increase by as much as 40% per year, the concurrency that can be efficiently expressed to the memory subsystem will increase by as much as 40% per year. Latency to DRAM is actually decreasing. As socket bandwidth is increasing at 20 to 40% per year, the concurrency expressed through multicore can easily cover Little's Law's latency-bandwidth product now and in the future.

2.2.7 Cache Coherency

Snoopy cache coherent SMPs do not scale beyond two to four sockets due to the quickly increasing latency and bandwidth demands over networks with low bisection bandwidth. Beyond this point, only directory-based protocols are appropriate, and up to perhaps 512 sockets. Unfortunately, to achieve good performance at such scales, one must program for locality and thus obviate much of the need for cache coherency. Although on-chip bandwidth and latency are orders of magnitude better than off-chip, they are not free. As such, there is a multicore scale at which on-chip snooping and directory protocols will fail or become prohibitively expensive. This has motivated some multicore vendors to

explore local store or scratchpad architectures to eliminate the need for coherency. This approach is seen in IBM's Cell Broadband Engine and NVIDIA's GPUs. We believe that it is possible to share a local store among several cores, but these will likely be grouped into on-chip shared memory clusters.

2.2.8 Productivity, Programmers, and Performance

The drive for increased productivity has placed more and more layers of software between programmers and hardware. Moreover, architectures have become more and more complex and opaque to programmers. As such, most programmers have no effective means for predicting or understanding performance. As more and more computing cycles are consumed by cloud and server computing, the fixed ISA requirements become secondary and are replaced by a unified source requirement.

Compilers are very adept at handling a two-level memory hierarchy: main memory and registers. Moreover, they work best when latencies are small and deterministic. Unfortunately, this means they are inadequate from a performance standpoint given today's multi-level cache hierarchies and out-of-order execution. Worse still, compilers have utterly failed at auto-parallelization. As future performance is premised on multicore, parallel efficiency is far more pertinent than single-thread performance.

Of course, no one is suggesting all programmers must be architectural experts to efficiently program a multicore computer. However, they must be given a means by which those concerned with poor performance can understand the bottlenecks and address them. The result is a bifurcation of the programmer population into two camps: those programmers concerned with parallel, architectural, and power efficiency — "efficiency programmers" — and those programmers concerned with features — "productivity programmers." Thus, in the future, we expect the efficiency programmers to encapsulate various components into frameworks that can then be easily used by the productivity layer programmers without requiring them to have the detailed understanding of parallel programming.

2.3 Dwarfs, Patterns, and Motifs

One of the more subtle trends in computing is a transition away from control-intensive computation to data-intensive computation. In essence, the problems and data sets have scaled much faster than their respective control requirements. Nowhere else is this more obvious than in scientific computing. Moreover, it has been postulated that there are as few seven key numerical methods in this field — coined the Seven Dwarfs [31]. To be clear, these are not kernels or methods in the traditional sense, but broad fields or domains in computational science comprising many kernels. The underlying implementations of the kernels are free to evolve, but the high-level mathematics remains the same. The Seven Dwarfs are: dense and sparse linear algebra, calculations on structured and unstructured grids, spectral methods, particle methods, and Monte Carlo simulations. A general purpose, capacity supercomputer or cluster will likely be required to efficiently process all of these dwarfs, but domain specific computers may only be required to process a subset well.

In this section, we track the evolution of the Seven Dwarfs from the attempts to expand them to other fields to their generalization into patterns and motifs.

2.3.1 The Berkeley View

In part, the Berkeley View [7] attempted to map benchmarks for embedded computing and general purpose computing onto the existing Seven Dwarfs. Two problems arose: first mapping the kernels of a benchmark to dwarfs doesn't provide insight into the fundamentally important problems. Second, there were several kernels that simply didn't map to any of the Seven Dwarfs.

As a result, a broader approach was taken. A survey of several additional fields of computing was conducted including machine learning, databases, graphics, and games. From these, the key computational methods were extracted. The result was the creation of six new dwarfs: combinational logic, graph traversal, dynamic programming, backtrack and branch-and-bound, and graphical models. When there were seven, one could easily tie them to the well-known fairy-tale. However, given 13, a new name was required — motifs.

2.3.2 The Case for Patterns

Dwarfs are useful for programs in which it is obvious that the underlying computation is captured by a dwarf. However, many programmers might not realize their computation is actually a dwarf in disguise. Moreover, many programs are still control-intensive. As such, pontificating about data-intensive computational methods is irrelevant. Finally, by no means are there only a small set of dwarfs now and forever in all of computing. As new problems arise, new dwarfs will be created and older ones will become obsolete. To be productive, most programmers will need a methodology that allows them to exploit the dwarfs when necessary, but will still provide some parallel efficiency for their typical programs.

After examining programs and kernels, some have postulated that there is a basic set of structural program patterns that appear over and over [2, 54, 92]. These include pipe-and-filter, agent and repository, event based, bulk-synchronous, and map reduce among others. Clearly, these are high-level, "whiteboard" conceptualizations of program structural organization. They provide no insight as to how one could efficiently implement such patterns on any parallel machine. However, below these structural patterns, one could envision a layering of both the styles of data and task partitioning as well as the parallel building blocks required to implement any such program. These parallel components include shared and distributed arrays, queues, and hash tables as well as routines for task creation and completion. Below these components are the most basic building blocks including communication and synchronization routines. This structure can be implemented recursively; that is, bulk-synchronous within each node of a pipe-and-filter structure. Figure 2.1 shows the result: a layered conceptualization of program organization, from the broadest "whiteboard" conceptualization down to the implementation details.

No one expects many programmers to be capable of implementing all of these patterns, components, and routines efficiently on all multicore machines. Thus, the splitting of programmers into productivity and efficiency programmers is exploited. The efficiency

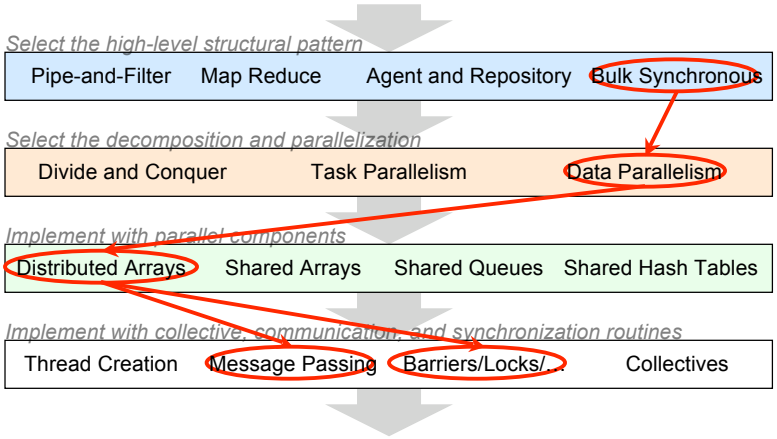


Figure 2.1: A high-level conceptualization of the pattern language. Programmers step through each level selecting the appropriate pattern or components.

programmers will implement and encapsulate the parallel components and basic routines. If it can be done in a general extensible manner, then it is likely the productivity-layer programmers will be able to reuse this work over and over. Figure 2.1 shows the loose decision tree the productivity programmers might then follow. First, they decide on the appropriate pattern, then the appropriate decomposition, then components, and finally routines to implement those components.

Although simplistic in its structured design and regimented decision tree, this approach demands that productivity programmers make the correct decisions as they navigate through the tree for the given platform. Failure to do so may have significant performance and scalability ramifications. Moreover, this approach simply punts all the performance challenges of parallel programming to the efficiency programmers.

2.3.3 The Case for Motifs

Recall that the Dwarfs are commonly used numerical methods. In the context of the patterns, hundreds of researchers working for decades have found the best structural patterns, decompositions, and data structures for each Dwarf. In essence, they have agreed upon the best know traversal of the pattern tree for a specific computational method and black boxed the result into a library. For a number of reasons including increasing the number of Dwarfs beyond seven, and thus poorly correlated to the well-known fairy-tale, Dwarfs have been renamed motifs [6]. By recognizing a particular motif or kernel, the productivity programmers can achieve good efficiency by leveraging the work conducted by hundreds of researchers. Figure 2.2 on the next page shows the integration of the motifs into the pattern language stack. Realization of the motif nature of computation allows productivity programmers to bypass the decision tree pattern-based approach to parallel program implementation and use a black boxed best-known traversal and implementation.

Some of the motifs are far more mature and developed than others. These include

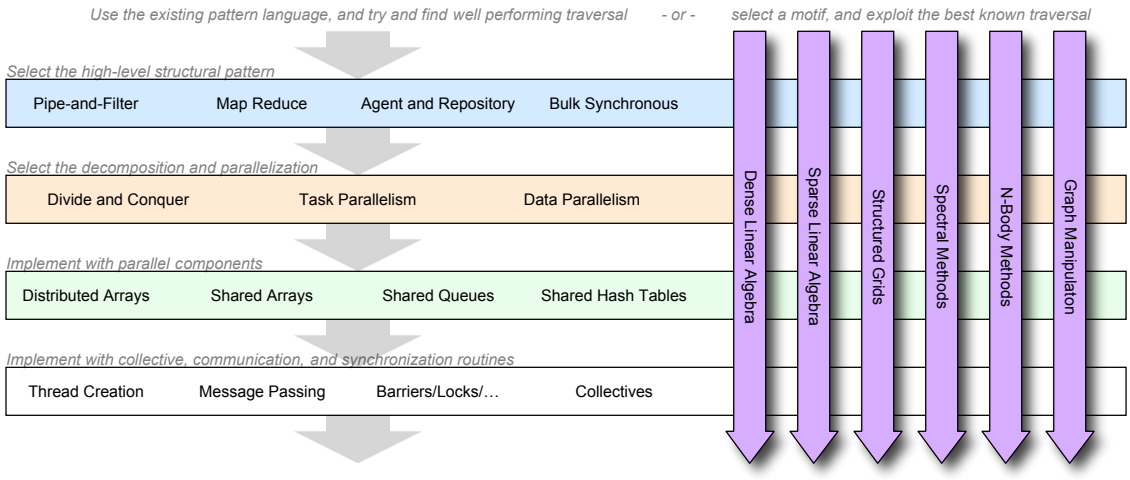


Figure 2.2: Integration of the Motifs into the pattern language. By selecting a motif, one selects the best know traversal of the pattern language decision tree for that method.

six of the original Seven Dwarfs: dense and sparse linear algebra, computations on structured and unstructured grids, spectral methods, and particle methods. To these motifs, it is clear that graph manipulation and finite state machines must be added. In the future, we expect the acknowledgment of existing important motifs. Ultimately, the value in motifs lies in their extensibility, not only in problem size, but also in data type, data structure, and the operators used.

2.4 The Case for Auto-tuning

Potentially, motifs provide productivity programmers a productive solution to parallel programming. However, it falls to the efficiency programmers to ensure that the kernels within the motifs each deliver good performance. By no means is this a easy task given the breadth of both motifs and architectures they must run efficiently run on. This approach is further complicated by the fact that simply running well on all of today's machines is insufficient. With a single code base, we must deliver good performance on any possible future evolution of today's architectures.

2.4.1 An Introduction to Auto-tuning

Automated tuning or *auto-tuning* has become a commonly accepted technique used to find the best implementation for a given kernel on a given single-core machine [16, 138, 52, 135, 23]. Figure 2.3 on the following page compares the traditional and auto-tuning approaches to programming. Figure 2.3(a) shows the common Alberto Sangiovanni-Vincentelli (ASV) triangle [95]. A programmer starts with a high-level operation or kernel he wishes to implement. There is a large design space of possible implementations that all deliver the same functionality. However, he prunes them to a single C program representation. In doing so, all high level knowledge is withheld from the compiler which in turn takes the

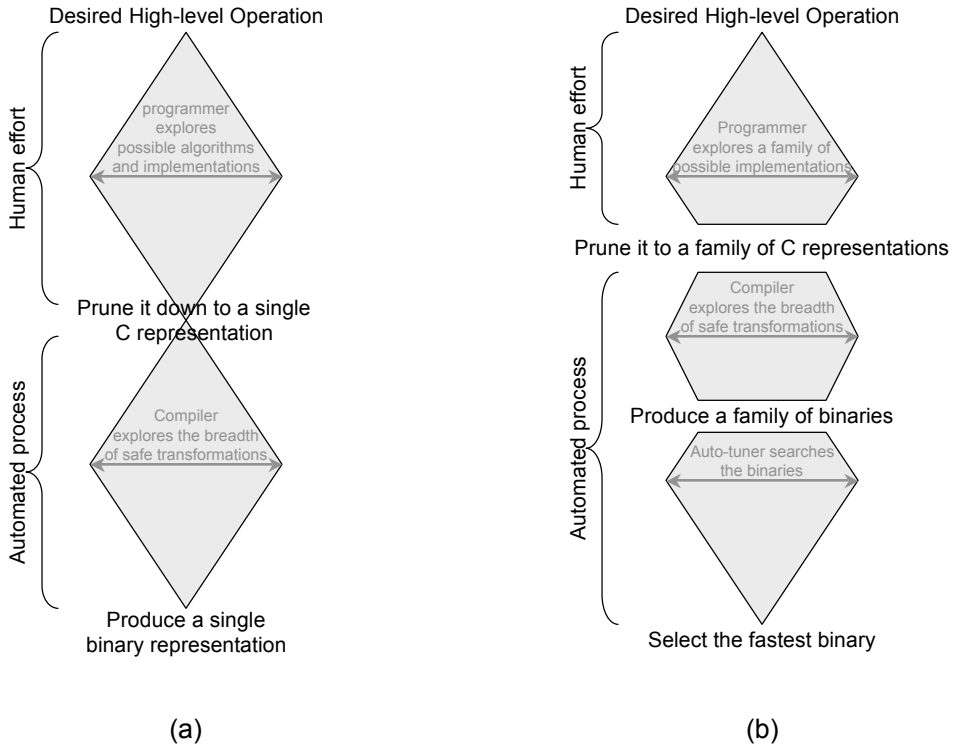


Figure 2.3: ASV triangles for the conventional and auto-tuned approaches to programming.

C representation, and explores a variety of safe transformations given the little knowledge available to it. The result is a single binary representation. Figure 2.3(b) presents the auto-tuning approach. The programmer implements an auto-tuner that rather than generating a single C-level representation, generates hundreds or thousands. The hope is that in generating these variants some high-level knowledge is retained when the set is examined collectively. The compiler then individually optimizes these C kernels producing hundreds or machine language representations. The auto-tuner then explores these binaries in the context of the actual data set and machine.

There are three major concepts with respect to auto-tuning: the optimization space, code generation, and exploration. First, a large optimization space is enumerated. Then, a code generator produces C code for those optimized kernels. Finally, the auto-tuner proper explores the optimization space by benchmarking some or all of the generated kernels searching for the best performing implementation. The resultant configuration is an auto-tuned kernel.

Figure 2.4 on the next page shows the high-level discretization of the optimization space. The simplest auto-tuners only explore low-level optimizations like unrolling, reordering, restructuring loops, eliminating branches, explicit SIMDization, or using cache bypass instructions. These are all optimizations compilers claim to be capable of performing, but often can't due to the lack of information conveyed in a C program. More advanced auto-tuners will also explore different data types, data layouts, or data structures. Compilers

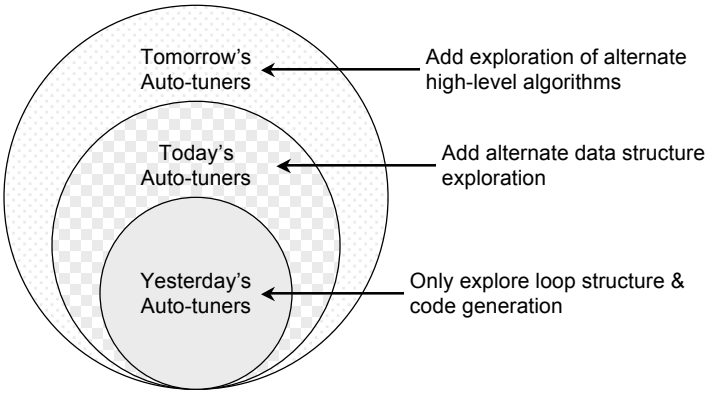


Figure 2.4: High-level discretization of the auto-tuning optimization space.

have no hope of performing these optimizations. Finally, the most advanced auto-tuners also explore different algorithms that produce the same solution for the high-level problem being solved. For example, an auto-tuner might implement a Barnes-Hut-like particle-tree method instead of a full N^2 particle interaction.

The second aspect of auto-tuning is code generation. The simplest strategy is for an expert to write a Perl or similar script to generate all possible kernels as enumerated by the optimization space. A more advanced code generator could inspect C or FORTRAN code, and in the context of a specific motif generate all valid optimizations through a regimented set of transformations [80]. This differs from conventional compilation in two aspects. First, compilers are incapable of making optimizations they cannot verify are always safe. In essence, we have added a `-motif=sparse` compiler flag. Second, this method produces all kernels, where a compiler will only produce one version of the code.

The third aspect of auto-tuning is the exploration of the optimization space. There are several strategies designed to cope with the ever-increasing search space. We wish to clearly differentiate an optimization from its associated parameter. For example, unrolling is an optimization; the degree to which a loop is unrolled is the parameter. For certain optimizations like whether to SIMDize or not, the parameter is just a Boolean variable.

The most basic approach is an exhaustive search of all parameters for all optimizations. For each optimization an appropriate parameter is selected, and the appropriate kernel is benchmarked. If the performance is superior to the previous contender for best implementation, then the new performance and parameters are recorded as the best implementation. Clearly, when this approach becomes intractable when the number of combinations exceeds a few thousand.

Second, one can use heuristics or models of architectures to decide the appropriate parameters for certain optimizations. For example, a heuristic may limit the parameter search space for cache blocking so that the resultant working sets consume 80 to 99% of the last level cache. In general, these search techniques can be applied to a subset of the optimizations.

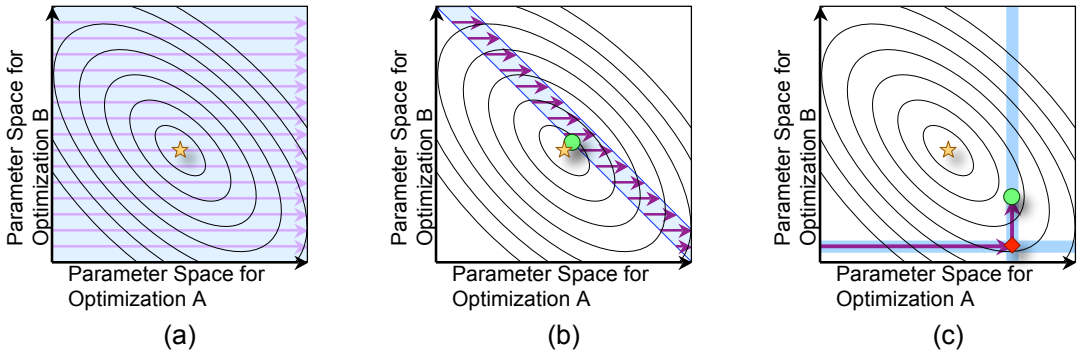


Figure 2.5: Visualization of three different strategies for exploring the optimization space: (a) Exhaustive search, (b) Heuristically-pruned search, and (c) hill-climbing. Note, curves denote combinations of constant performance. The gold star represents the best possible performance.

Finally, in a hill climbing approach, optimizations are examined in isolation. An optimization is selected. Performance is benchmarked for all parameters for that optimization and the best-known parameter for all other optimizations. The best configuration for that optimization is determined. The process continues until all optimizations have been explored once. Previous work [37] has shown this approach can deliver good performance. Unfortunately, this approach may still require thousands of trials.

Figure 2.5 visualizes the three different strategies for exploring the optimization space. For clarity, we have restricted the optimizations space to two optimizations, each with their own independent range of parameters. In practice, the number of optimizations will likely exceed 10. The curves shown on the graph denote lines of constant performance. We have assumed a smoothly varying performance curve where the local minimum is the global minimum. The gold star represents the best possible performance. In Figure 2.5(a), an exhaustive approach searches every combination of every possible parameter for all optimizations. Clearly, this approach is a very time consuming, but is guaranteed to find the best possible performance for the implemented optimizations. Figure 2.5(b) heuristically-prunes the search space, and exhaustively searches the resultant region. Clearly, this will reduce the tuning time, but might not find the best performance as evidenced by the fact that the resultant performance (green circle) is close but not equal to the best performance (gold star). Figure 2.5(c) uses a one pass hill-climbing approach. Starting from the origin, the parameter space for optimization A is explored. The local maximum performance is found (red diamond). Then, using the best known parameter for optimization A, the parameter space for optimization B is explored. The result (green circle) is far from the best performance (gold star), but the time required for tuning is very low.

Perhaps the most important aspect of an auto-tuner's exploration of the parameter space is that it is often done in conjunction with real data sets. That is, one provides either a training set or the real data to ensure the resultant optimization configuration will be ideal for real problems.

	Optimizations Explored				Exploration of the Optimization Space			
	Low Level	Data Structure	Other Algorithms	multi-core	Data-oblivious Heuristics	Data-oblivious Search	Data-aware Heuristics	Data-aware Search
Traditional Compilers	✓			†	✓			
"auto-tuning" Compilers	✓			†	✓	✓		
Yesterday's Auto-tuners	✓							✓
Today's Auto-tuners	✓	✓					✓	✓
Tomorrow's Auto-tuners	✓	✓	✓	✓			✓	✓

Table 2.1: Comparison of traditional compiler and auto-tuning capabilities. Low-level optimizations include loop transformations and code generation. †only via OpenMP pragmas.

Table 2.1 provides a comparison of the capabilities of compilers and auto-tuners. Traditional compilers can only perform the most basic optimization and choose the parameters in a data-oblivious fashion. Some have proposed compilers perform some exploration of the generated kernels. Nevertheless, these compilers are limited by the input C program and are oblivious to the actual data set. As a result, even yesterday's traditional auto-tuner is more capable. Today, some auto-tuners explore alternate data structures and use heuristics to make exploration of the search space tractable. Tomorrow's auto-tuners will likely also explore algorithmic changes. In doing so, they may trade vastly improved computational complexity (total FLOPs) for slightly worse efficiency (FLOP/s). As a result, the time to solution will be significantly improved.

The next three sections examine previous efforts to auto-tune various representative kernels of three motifs. Chapter 7 includes a section on previous auto-tuning work within the sparse motif. Collectively, these provide background and related work to our current and future multicore auto-tuning endeavors on other motifs presented in Chapters 5 through 9.

2.4.2 Auto-tuning the Dense Linear Algebra Motif

As the name suggests, the dense linear algebra motif encapsulates operations and methods on dense linear algebra. For purposes of this discussion, we restrict ourselves to the Basic Linear Algebra Subroutines (BLAS) [45, 42] which are grouped into three basic categories (labeled BLAS1 through BLAS3) based on whether they perform vector-vector, matrix-vector, or matrix-matrix operations.

The BLAS3 operations typically have the highest computational complexity and thus have the longest time to solution. The canonical BLAS3 operation is DGEMM — essentially matrix-matrix multiplication. In fact, this operation is the core component of the LINPACK benchmark [89], a metric by which many supercomputers are judged [131]. Figure 2.6 on the following page presents the reference C implementation of matrix-matrix multiplication. Such naïve implementations place high demands on cache capacity and bandwidth as matrix B is read N times. In fact, there is no reuse in the register file as

```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    double cij = 0;
    for(k=0;k<N;k++){
      cij += a[i][k] * b[k][j];
    }
    C[i][j] = cij;
  }
}
```

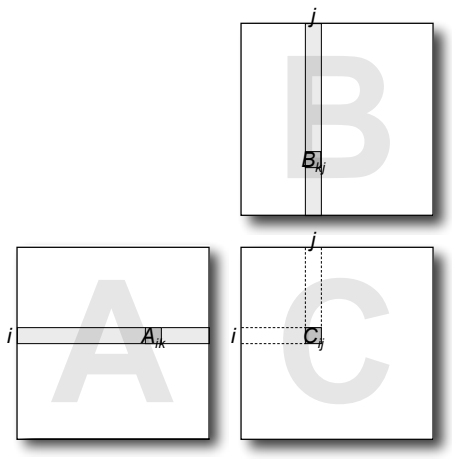


Figure 2.6: Reference C implementation and visualization of the access pattern for C=A×B, where A, B, and C are dense, double-precision matrices.

`a[i][k]` and `b[k][j]` are used only once in the inner loop.

It is possible to explicitly unroll all three loops and create 2×2 register blocks for A, B, and C. Figure 2.7 on the next page shows this optimization. Observe, `a[i][k]` is loaded once in the inner loop, but used twice. If larger register blocks were used, more locality could be exploited. However, if one were to exceed the register file size, then spills to the stack would occur, and the benefit would be lost. This blocking technique can be applied hierarchically to all levels of the memory hierarchy: register file, L1 cache, L2 cache, TLB, and so on. However, one typically won't use temporary variables for cache blocks but rather rely on the nature of caching to exploit locality and thus add loop nests. Unfortunately, as each microarchitecture may have different register file and cache sizes, the optimal blockings will vary from one machine to the next. In the past, vendors have poured enormous efforts into hand optimizing DGEMM for their latest architecture.

Bilmes, *et al.* observed that if one could enumerate and generate all possible code variants, then given the capabilities of modern microprocessors, they could all be explored for varying problem sizes, and the optimal configuration for a given problem size could be determined. The result, PhiPAC [16] produced a portable implementation of DGEMM capable of achieving a high fraction of peak on a wide variety of architectures and is considered the progenitor of auto-tuners. ATLAS [138] expanded the optimization space and extended its breadth to all of the BLAS routines. Upon installation, the target machine is benchmarked, and the results are used for selecting the appropriate optimized routine at runtime. Today, although the vendors still produce "vendor tuned" implementations, they have all embraced auto-tuning as a means to facilitate their somewhat restricted optimizations. Externally, these routines must preserve the existing interface. As such, any data structure transformations are internal and temporary.

```
for(i=0;i<N;i+=2){
  for(j=0;j<N;j+=2){
    double c00=0;double c10 = 0;double c01=0;double c11=0;
    for(k=0;k<N;k+=2){
      c00 += a[i+0][k+0]*b[k+0][j+0] + a[i+0][k+1]*b[k+1][j+0];
      c10 += a[i+1][k+0]*b[k+0][j+0] + a[i+1][k+1]*b[k+1][j+0];
      c01 += a[i+0][k+0]*b[k+0][j+1] + a[i+0][k+1]*b[k+1][j+1];
      c11 += a[i+1][k+0]*b[k+0][j+1] + a[i+1][k+1]*b[k+1][j+1];
    }
    C[i+0][j+0] = c00;
    C[i+1][j+0] = c10;
    C[i+0][j+1] = c01;
    C[i+1][j+1] = c11;
  }
}
```

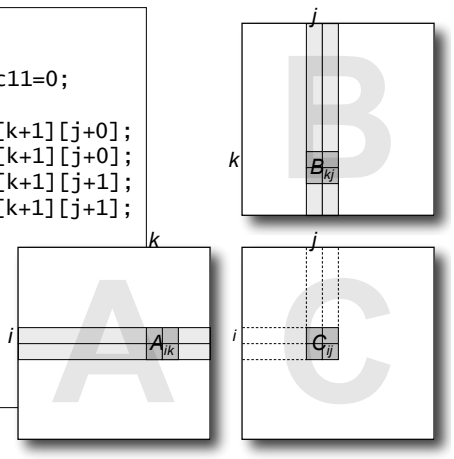


Figure 2.7: Reference C implementation and visualization of the access pattern for C=A×B using 2×2 register blocks.

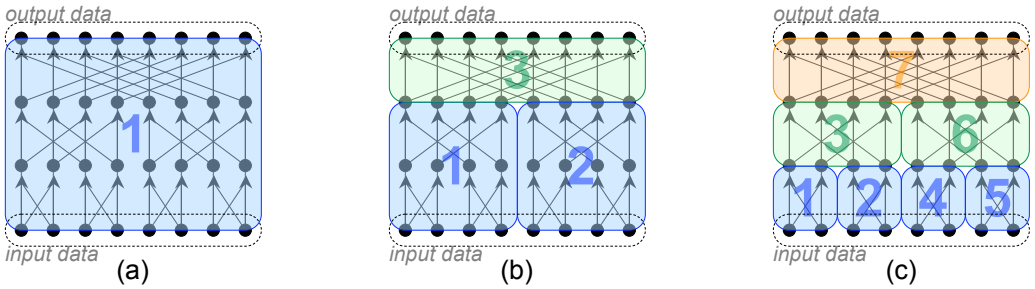


Figure 2.8: Visualization of the cache oblivious decomposition of FFTs into smaller FFTs exemplified by FFTW.

2.4.3 Auto-tuning the Spectral Motif

The canonical kernel for the spectral motif is the Fast Fourier Transform (FFT) [32]. Like DGEMM, vendors have poured enormous efforts into hand tuning the FFT for their processors. Auto-tuning has been applied to this kernel in order to provide performance portability across the breadth of processors.

Unlike DGEMM, where the traversal and block size is specified, cache oblivious algorithms [53, 50] attempt to recursively subdivide the problem into two or more sub-problems and solve them individually. If one were to blindly apply the cache oblivious technique to a n-point FFT, then one would solve two $\frac{n}{2}$ point FFTs and perform the combining butterfly. When the recursion is carried to completion, the base case is a 2-point butterfly. FFTW [52] applies auto-tuning by benchmarking both the recursive approach and the blindly naïve approach offline for every problem size. Thus, at runtime FFT can use this data to decide whether naïvely solving a n-point FFT is faster than recursively solving two $\frac{n}{2}$ point FFTs. Figure 2.8 shows three different approaches to solving a 8-point FFT

and represents them as a DAG. The order of computation is labeled. Figure 2.8(a) directly solves the FFT performing each of the three stages successively. Figure 2.8(b) decides that it is faster to solve two 4-point FFTs individually and thereby exploit locality either in the register file or cache. Finally, Figure 2.8(c) extends the recursion to the 2-point base case.

Although cache oblivious approaches guarantee a lower bound to cache misses, they do not guarantee an optimal implementation. The performance on any modern architectures is not solely determined by cache misses. As a result, cache oblivious algorithms do not guarantee peak performance [81]. Modern microprocessors require long streaming accesses, software prefetching, array padding, and SIMDization to achieve peak performance. Unfortunately, all future microprocessors will require another key set of optimizations to achieve peak performance: efficient parallelization for multicore.

The SPIRAL project [97, 124] has taken a high-level approach to library generation for the spectral motif, specifically linear transforms. Instead of restricting the library to simple exploration of generated C code variants, SPIRAL takes as an input a high-level, declarative representation of a linear transform. It then performs a series of divide-and-conquer or rewrites using a library of over 50, hardware-conscious transformation rules. When coupled with efficient parallelization strategies, SPIRAL produces a well-performing library routine. We believe SPIRAL should be viewed as a template for motif-wide auto-tuning to which we must ultimately aspire.

2.4.4 Auto-tuning the Particle Method Motif

The particle method motif typically simulates a continuing all-to-all interaction between N particles. The canonical kernel for this motif is a Newtonian force calculation in 2- or 3-dimensions. Simply put, time is discretized into steps. At each time step, for each of the N particles, one sums the N-1 forces acting on it as well as any external forces and calculates the resultant acceleration. Then, given the current positions, velocities, and accelerations, one calculates the new position and velocity for all particles. As each force calculation may require dozens of floating-point operations, there are N^2 force calculations per time step, and there are thousands if not millions of time steps, this method is extremely computationally demanding for even modest numbers of particles. Thus, even the fastest GPUs are limited to perhaps only tens of thousands of particles.

Clearly, many interesting problems demand N to be much larger. If one is willing to sacrifice some accuracy for improved performance, then there are two approaches depending on whether the particles are spatially clustered or not. If they are clustered, then particle-tree methods are used; otherwise, a particle-mesh approach is used. In either case, a large auxiliary data structure is used. These algorithmically superior approaches provide better time to solution, but often lower FLOP rates.

Particle-tree methods recursively tessellate 3-space into an octree until there is only one particle per cell (leaf in the octree). Observe that the force from a cluster of particles (intermediate node in the tree) is well-approximated by its center of mass when the cluster is both sufficiently small and sufficiently distant. In the Barnes-Hut [12], one calculates the forces between nodes in the octree, and projects them onto the particles. With $O(N \cdot \log(N))$ computational complexity, it is clear that for sufficiently large N, this method is superior to the naïve N^2 approach. However the inefficiency of tree manipulation and traversal

results in a very large scaling constant. Another particle-tree method is the Fast Multipole Method (FMM) [63]. Rather than directly calculating forces, it approximates the potential throughout each leaf via an spherical harmonics expansion. Then, it differentiates the potential to push the particles. Subtly this method has a O(N) computational complexity, albeit with an enormous constant. Thus, the value of N for which the O(N) Fast Multipole Method is superior to the O(N·log(N)) Barnes-Hut approach or the $O(N^2)$ naïve approach is both architecture and problem dependent. An approach similar to the FMM is Anderson's Method [5] which uses numerical integration and often delivers superior time to solution.

Particle-mesh methods discretize 3-space into uniform 3D mass and potential grids. Often there are tens to hundreds of particles per grid point. Each particle then deposits its mass or charge on to the bounding grid points. The result is a grid representing the distribution of mass. Solving Poisson's equation results in a grid of the potential throughout space. The force on each particle is calculated by differentiating the potential. When there are sufficiently many particles the O(N) charge deposition and pushing dominates the computational complexity of the Poisson solve. Unfortunately, the charge deposition phase involves a scatter with conflicts. As a result, it is not efficiently parallelized on existing architectures.

The N^2 methods work best when a working set of particles is kept in the cache. As such, one could auto-tune such kernels to look for the appropriate loop blocking sizes. Low-level auto-tuning of particle-tree and particle-mesh approaches is uncommon. Instead, Blackston, *et al.* studied the high-level algorithmic auto-tuning to find the appropriate method, expansion size, and whether or not to use supernodes as a function of architecture, the number of particles, and their clustering [18]. Clearly, with enough particles, O(N·log(N)) methods win out over $O(N^2)$ and eventually O(N) methods win out over O(N·log(N)) methods. The challenge is finding the break-even point *a priori*.

2.5 Summary

In this chapter, we discussed trends in computing, and two potential solutions to mitigate their pitfalls. As discussed in Section 2.2, it is clear that all future gains in performance will come from multiple cores on a chip. Moreover, the number of cores will likely double every 2 to 4 years. It is unlikely that without major technological advances, cost-effective memory bandwidth will be able to keep pace. The result, without algorithms and optimizations known only to domain experts, will be that more and more codes will be limited by memory bandwidth.

To address the parallel programming problem, Section 2.3 describes a layered, parallel pattern language. Programmers focused on features and productivity can navigate through the resultant decision tree, selecting patterns, decompositions, and components as needed. A second group of programmers — efficiency programmers — will implement the components and ensure that any combination of patterns and components will work correctly. However, it is unlikely that the productivity programmers can select the patterns and components that will deliver scalable performance in the multicore era. As such, the efficiency programmers will also implement a set of extensible motifs based on well-established numerical methods. Such motifs will provide productivity programmers with access to the

algorithms and implementations known only to the domain experts.

Hence, we have passed the buck to efficiency programmers. Thus, the final question we must answer is how efficiency programmers can attain good performance on these motifs given the trends in computing. As detailed in Section 2.4, auto-tuning has been shown to provide performance portability on single core processors for several of the motifs. Thus, we propose extending auto-tuning to multicore architectures and broaden it to other motifs. In the following chapters, we discuss the experimental setup, performance modeling, background material, our approach, and our results.

Chapter 3

Experimental Setup

This chapter provides the background material on the computers and programming models used throughout this work, especially in Chapters 6 and 8. In Section 3.1 we discuss the computers and architectures used for benchmarking. For the architectures that exploit novel or less well-understood concepts, we elaborate. In Section 3.2 we discuss possible programming models and the reasoning behind our selection of a bulk-synchronous, single-program, multiple-data (SPMD) `pthreads` approach as well as details of the barrier, affinity, and cycle timers used. The chapter wraps up with a discussion of the compilers and compiler flags used. Finally, Section 3.3 summarizes the experimental setup.

3.1 Architecture Overview

In this section, we discuss the five computers and the six microarchitectures used throughout this work. We assume the reader is familiar with the basic principles of computer architecture [68]. However, as some architectures use novel or poorly documented features, we describe them here. To avoid redundancy, we organize this section by architectural topic rather than by computer. For clarity, we specify both the processor product names and code names as well as the computer system product name. Generally, we refer to machines by the processor code names.

3.1.1 Computers Used

In this dissertation we used five dual-socket shared-memory multiprocessors (SMPs) as the testbed for our auto-tuning experiments. We believe that these architectures span the bulk of architectural paradigms and allow us to perform some rudimentary technology analysis. Table 3.1 on the next page provides a summary of the architectures' floating-point capabilities.

Intel Quad-Core Xeon E5345 (Clovertown)

The Intel Quad-Core Xeon® (Clovertown) is a dual-socket capable implementation of the Core2 Quad 64-bit processor [71, 70, 43]. The Core2 Quad is actually a

Core Architecture	Intel Xeon E5345 (Clovertown)	AMD Opteron 2214 (Santa Rosa)	AMD Opteron 2356 (Barcelona)	Sun T2+ T5140 (Victoria Falls)	STI Cell (PPE)	STI Cell (SPE)
threads/core	1	1	1	8	2	1
issue width	4	3	4	2†	2	2
FPU	MUL+ADD	MUL+ADD	MUL+ADD	MUL or ADD	FMA	FMA
DP SIMD	✓	✓	✓	—	—	✓
Clock (GHz)	2.33	2.20	2.30	1.167	3.20	3.20
DP GFLOP/s	9.33	4.40	9.20	1.167	1.83	6.40

System	Dell PowerEdge 1950	Sun X2200 M2	AMD internal	Sun Victoria Falls	IBM QS20 Blade	
# Sockets	2	2	2	2	2	
Cores/Socket	4	2	4	8	1	8
DP GFLOP/s	74.66	17.60	73.60	18.66	12.80	29.25

Table 3.1: Architectural summary of Intel Clovertown, AMD Opterons, Sun Victoria Falls, and STI Cell multicore chips. †Each of the two thread groups may issue up to one instruction.

multi-chip module (MCM) solution in which two separate Core2 Duo chips are paired together in a single socket. These four dual-core processors are marketed as two quad-core processors. Each 2.33 GHz Core2 Duo implements the Core™ microarchitecture. The Core microarchitecture decodes x86 instructions into RISC micro-ops that it then executes in an out-of-order fashion. To this end, the L1 instruction cache can deliver 16 bytes per cycle to the decoders, which in turn can decode four x86 instructions per cycle into seven micro-ops of which only four can be issued per cycle. The SSE datapaths are all 128 bits wide allowing an SSE instruction to be completed every core clock cycle. Thus, the peak double-precision floating-point performance per core is 9.33 GFLOP/s. For this thesis, we use a Dell PowerEdge 1950 with two Xeon E5345 quad-core processors.

AMD Dual-core Opteron 2214 (Santa Rosa)

The AMD Opteron™ 2214 (Santa Rosa) is a dual-socket capable 64-bit dual-core rev.F Opteron [4]. Like the Xeon, the Opteron decodes x86 instructions into RISC micro-ops and executes them in an out-of-order fashion. It may decode up to 3 instructions per cycle and issue 6 micro-ops per cycle. The cores execute 128-bit SSE instructions every two core clock cycles using both a 64-bit floating-point multiplier and a 64-bit floating-point adder. Therefore, the maximum throughput of packed double-precision multiply instructions is one every other cycle. Thus at 2.2 GHz, the peak performance per core is 4.4 GFLOP/s. We use a Sun X2200 M2 with two Opteron 2214 dual-core processors.

AMD Quad-core Opteron 2356 (Barcelona)

The Opteron™ 2356 (Barcelona) is a dual-socket capable 64-bit quad-core rev.10h Opteron [4, 3]. Barcelona can fetch 32 bytes of instructions from the instruction cache and

can decode up to four x86 instructions per cycle into six RISC micro-ops. Furthermore, the cores can execute 128-bit SSE instructions every core clock cycle using both a 128-bit floating-point multiplier and a 128-bit floating-point adder. Thus at 2.3 GHz, the peak performance per core is 9.2 GFLOP/s. We use an AMD internal development computer powered by a pair of Opteron 2356 quad-core processors.

Sun UltraSPARC T2+ T5140 (Victoria Falls)

The Sun UltraSparc® T2 Plus is an eight-core processor referred to as Victoria Falls [122, 107]. Each core is a dual-issue, eight-way hardware multithreaded (see Section 3.1.4) in-order architecture. There are four primary functional units per core: two ALUs, a FPU, and a memory unit. Although the cores are dual-issue, each thread may only issue one instruction per cycle and resource conflicts are possible. Our study examines the Sun UltraSparc T2+ T5140 with two T2+ processors operating at 1.16 GHz. They have a per-core and per-socket peak performance of 1.16 GFLOP/s and 9.33 GFLOP/s respectively, as there is no fused-multiply add (FMA) functionality. Inter-core communication is only possible through memory. A large crossbar connects the eight cores per socket with eight L2 banks as well as I/O. Although the L2 bandwidth is high, so too is the L2 latency.

IBM QS20 Cell Broadband Engine

The Sony Toshiba IBM (STI) Cell Broadband Engine™ was designed to be the heart of the Sony PlayStation 3 (PS3) video game console. Unlike all other computers used in this work, the Cell exploits a heterogeneous approach to multicore integration. Each chip instantiates one conventional RISC Power Processing Element (PPE), and eight Synergistic Processing Elements (SPEs) [79, 49, 106, 64, 65]. The PPE provides portability and performs all OS and control functions, while the SPEs are designed to be extremely efficient computational engines. In this work, we use a QS20 Cell blade with two 3.2 GHz Cell Broadband Engine chips.

The PPEs are dual-issue, dual-threaded, in-order 64-bit PowerPC processors. Although they support single-precision AltiVec (SIMD) instructions, all double-precision computation is scalar. As such their peak floating-point performance is 6.4 GFLOP/s per PPE using fused multiply add (FMA).

The SPEs contain both a dual-issue in-order 32-bit core (SPU) and a memory flow controller (MFC). The memory flow controller is essentially a programmable microcontroller that handles all DMA transfers into and out of the SPE. The SPU's instruction set is entirely SIMD with some control instructions. Unlike conventional dual-issue architectures, to achieve peak instruction throughput, computational instructions must be placed at even addresses, while memory or branch instructions must be placed at odd addresses. Any deviation from this stalls the SPE for one cycle. Thus, at the low-level, instructions must be scheduled like those for a short VLIW processor. Although single-precision floating-point performance is an impressive 25.6 GFLOP/s per core using SIMDized FMAs, double-precision performance is considerably lower. Not only is the pipeline half-pumped (serializing the operations encoded within a SIMD instruction), but the latency is considerably longer than the instruction forwarding network. To maintain correct hazard detection

and forwarding, every double-precision instruction stalls subsequent instruction issues by 6 cycles. As such, the peak double-precision SIMDized FMA performance is 1.83 GFLOP/s per SPE — far less than Santa Rosa's 4.4 GFLOP/s or the Clovertown's 9.3 GFLOP/s. The most notable advantage of the SPEs is their use of a disjoint, DMA-filled 256 KB local store memory (see Section 3.1.2) instead of the conventional cache hierarchy. Although a potential productivity killer, the performance gains on memory-intensive kernels can be significant.

Unlike conventional architectures, the PPE, SPEs, I/O, and memory controllers are all interconnected through four 128-bit rings known as the Element Interconnect Bus (EIB). All nodes are connected to all four rings. Two rings move data counter clockwise, and two move data clockwise. Instead of cores communicating through a shared cache, they may directly transfer data to or from their local memories or directly transfer data to or from the shared main memory.

3.1.2 Memory Hierarchy

For SMPs, there are two modern styles of memory hierarchies: cache-based and local store-based. In this work, every core uses either a cache hierarchy or a local store, but never both. In the following sections, we discuss both the design and motivations for both styles.

Cache-based Hierarchy

As caches are transparent to the programmer, they may seamlessly improve performance. Modern cache topologies are hierarchical — that is, there are multiple levels of cache, each somewhat larger but somewhat slower. The misses of the caches nearer the cores are to be serviced by the next cache closer to DRAM. In addition to the standard cache parameters of associativity, capacity and line size, one must also consider how the caches are shared either by multiple cores or by multiple caches lower in the hierarchy. Moreover, most cache topologies are inclusive, while some are exclusive.

Table 3.2 on the following page summarizes the cache hierarchy of the computers used in this work. Clearly, only Barcelona has an L3 cache and the Cell SPEs have no caches. Most line sizes are 64 bytes, but Victoria Falls uses 16 bytes for the L1, and the PPEs use 128 bytes throughout the hierarchy. Note, that Little's Law [10] applies to cache access just as it does to DRAM. As such, be mindful of the computers where $\frac{bandwidth \times latency}{threads}$ is large, as this is the cumulative size of the requests per thread (in bytes) that must be pipelined to the cache. More troubling is the very low capacity and associativity per thread in the Opteron's and Victoria Falls' L1's. This problem persists on Victoria Falls in the L2 where 64 threads share a 4 MB 16-way cache.

Local Store-based Hierarchy

Cache-based memory hierarchies are often referred to as two-level memory hierarchies, as both the register file and DRAM are treated as addressable memories. The user or compiler is responsible for explicitly transferring data from DRAM to the register file. A

Level	Cache Parameter	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	(SPE)
L1	capacity	32 KB	64 KB	64KB	8 KB	32 KB	N/A
	associativity	8-way	2-way	2-way	4-way	4-way	
	line size	64 bytes	64 bytes	64 bytes	16 bytes	128 bytes	
	latency	3 cycles	3 cycles	3 cycles	3 cycles	2 cycles	
	bandwidth	16+16 B/c	16+8 B/c	32+16 B/c	8 B/c	16 B/c (VMX)	
	shared by	one core	one core	one core	8 threads	2 threads	
	λBW/threads	48+48 Bytes	48 Bytes	96 Bytes	3 Bytes	16 Bytes	
	notes	WB	WB, exclusive	WB, exclusive	WT	WB	
L2	capacity	4 MB	1 MB	512 KB	4 MB	512 KB	N/A
	associativity	16-way	16-way	16-way	16-way	8-way	
	line size	64 bytes	64 bytes	64 bytes	64 bytes	128 bytes	
	latency	14 cycles	≈20 cycles	12 cycles	>20 cycles	≈40 cycles	
	bandwidth	32 B/c	8+8 B/c	16+16 B/c	8×(16+8) B/c	16+16 B/c	
	shared by	two cores	one core	one core	64 threads	one socket	
	λBW/threads	224 Bytes	≈320 Bytes	384 Bytes	>60 Bytes	≈640 Bytes	
	notes	WB	WB, exclusive	WB, exclusive	WB	WB	
L3	capacity	N/A	N/A	2 MB	N/A	N/A	N/A
	associativity			32-way			
	line size			64 bytes			
	latency			≈40 cycles			
	bandwidth			16+16 B/c			
	shared by			four cores			
	λBW/threads			≈320 Bytes			
	notes			WB, semi-exclusive			
Capacity per thread		2 MB	1.06 MB	1.06 MB	64 KB	256 KB	N/A

Table 3.2: Summary of cache hierarchies used on the various computers. Note: B/c = bytes per cycle peak bandwidth, WB = write back, WT = write through

cache can sit in front of DRAM to capture spatial and temporal locality. The programmer only needs to address DRAM and the register file, not the cache. Moreover, hardware is responsible for moving data in and out of caches as a response to load and store instructions. Three-level memory hierarchies instantiate a third addressable memory (a local store) between the register file and DRAM. The user must then explicitly transfer data from DRAM to the local store, and then separately and explicitly transfer data from the local store to the register file. Clearly, such an approach requires significant programming effort or tool support. From a hardware point of view, however, it is far simpler and more power efficient as the hardware doesn't have to maintain coherency, manage transfers, or tag locations with physical addresses.

It is imperative the reader keeps the concepts of local stores and caches distinct. To be clear, DRAM, local stores, and register files are completely disjoint address spaces. As such, you must consider the data in the local store as a copy of data in DRAM. Similarly, data in the register files is a copy of data in the local store. When it comes to caches, locations in the cache are aliased to locations in DRAM. Although you cannot address a DRAM line without implicitly accessing a cache line, you can access a DRAM address without accessing a local store address.

As seen in Table 3.3 on the next page, the Cell SPE is the only architecture in this work that uses local stores. Moreover, they are private to each SPE and are quite

Level	Local Store Parameter	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
L1	capacity	N/A	N/A	N/A	N/A	N/A	256 KB
	line size						128 bytes
	latency						6 cycles
	bandwidth						16 B/c
	shared by						one SPE
	λBW/threads						96 Bytes

Table 3.3: Summary of local store hierarchies used on the various computers. Note: B/c = bytes per cycle peak bandwidth

Level	Cache Parameter	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
L1	entries	16†	32	48	128	1024	256
	working set	64 KB	128 KB	192 KB	512 MB	4 MB	1 MB
L2	entries	256‡	512	512	N/A	N/A	N/A
	working set	1 MB	2 MB	2 MB			
Default page size		4 KB	4 KB	4 KB	4 MB	4 KB	4 KB

Table 3.4: Summary of TLB hierarchies in each core across on the various computers. †Only for loads. ‡Shared by two cores.

large compared to a L1 cache. However, the latency is somewhat longer. Thus, one SPE must express 6 independent SIMD loads to saturate the local store bandwidth. Although the local store can be filled on quadword (16 byte) granularities, it is optimally filled in granularities aligned to the 128-byte line size. There is no reason why future architectures can't hierarchically include multiple, potentially shared local stores or even a separate cache hierarchy to main memory.

TLB design

All computers used in this work use a hardware page walker to service TLB misses. However, the computers have different TLB structures and default page sizes. As such, their structure can have severe performance ramifications. The Cell SPE's TLB is somewhat unique. The local stores are physically addressed and thus are not paged. However, DRAM is paged as usual. Thus, address translation must occur in the memory flow controllers as they process each DMA. The advantage is that other queued DMAs can be processed while waiting for a TLB miss to be serviced. On Barcelona, software prefetched data will generate a TLB miss, but not a page fault.

Table 3.4 summarizes the TLB hierarchies within each core across all architectures. Observe that each architecture has a different maximum working set that it can process without capacity TLB misses between benchmark trials. Clearly, Victoria Falls can map a dramatically larger problem. As such, optimizations for the TLB will only be required on large problems.

The number of entries is indicative of the maximum number of arrays than can be accessed in a kernel assuming the working set is larger than the product of page size and

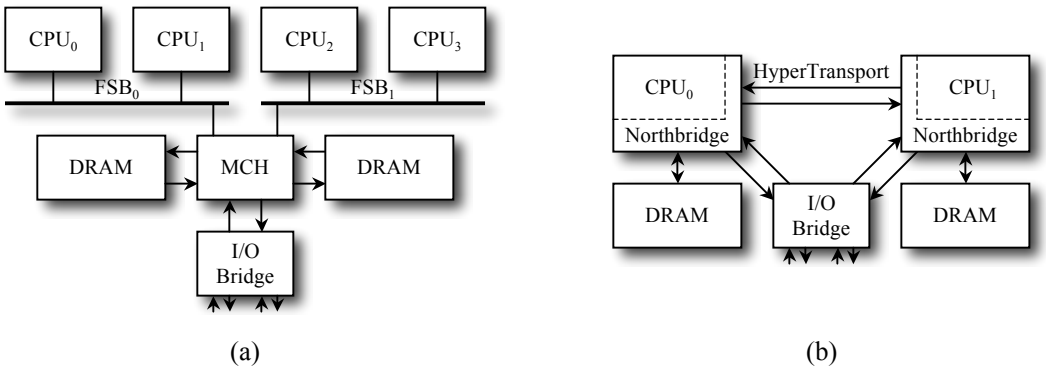


Figure 3.1: Basic Connection Topologies. (a) Four chips attached to two different front side buses with a discrete Northbridge. (b) the Northbridge has been partitioned and distributed among two chips.

the number of TLB entries. If the inner kernel exceeds the mappable problem size, then TLB capacity misses will occur through the execution of the loop. However, if the inner kernel also touches more arrays than TLB entries, TLB capacity misses will likely occur on every memory access.

3.1.3 Interconnection Topology

Computers must contain three basic components: processors, main memory, and I/O. As the computers in this work each have two sockets, a given socket can communicate with DRAM, another socket, or I/O. In this section, we examine each of the connections. Older computers like the Clovertown have a separate Northbridge chip that acts as a hub between sockets, DRAM, and I/O. VLSI integration has allowed this to be included on chip for the Opterons, Victoria Falls, and Cell. However, this integration has been complicated as in two-socket computers, the Northbridge must be distributed across multiple chips. Note, Clovertown's Northbridge chip is also called the memory controller hub (MCH). Figure 3.1 shows the two basic connection topologies.

Access to DRAM

Table 3.5 on the following page shows the DRAM specs for each computer as well as the interconnection topology. Clearly, the Clovertown is the only architecture still using an external memory controller hub. In this case, only the bandwidth between the DIMMs and the MCH is shown.

Remember, as the computers used here are all dual-socket SMPs, all DIMMs are addressable from any socket, not just the DIMMs directly attached to the originating socket. To this end, memory transactions and coherency is forwarded along the inter-socket network and data can be returned on the reciprocal path. We have configured all such computers to operate in a non-uniform memory access (NUMA) mode, rather than a uniform memory access (UMA) mode. Note, although not a natural match for those architectures, UMA

Connection Topology	DRAM Parameter	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade
DRAM directly attached to each socket	type capacity bandwidth	N/A	667MHz DDR2 8 GB 10.66 GB/s (read or write)	667MHz DDR2 8 GB 10.66 GB/s (read or write)	667MHz FBDIMM 16 GB 21.33 GB/s (read) 10.66 GB/s (write)	XDR 512 MB 25.6 GB/s (read or write)
DRAM attached to an external MCH	type capacity bandwidth	667MHz FBDIMM 16 GB 21.33 GB/s (read) 10.66 GB/s (write)	N/A	N/A	N/A	N/A
aggregate DRAM pin bandwidth		21.33 GB/s (read) 10.66 GB/s (write)	21.33 GB/s (read or write)	21.33 GB/s (read or write)	42.66 GB/s (read) 21.33 GB/s (write)	51.2 GB/s (read or write)

Table 3.5: Summary of DRAM types and interconnection topologies by computer.

can be achieved by interleaving physical addresses on cache line or similar granularities between the sockets. Thus, UMA implies every other cache line accessed is accessed at a substantially slower rate. The Clovertown computer is naturally a UMA SMP as all DRAM accesses are centralized through the MCH. For the naturally NUMA SMPs, the aggregate DRAM pin bandwidth is twice the bandwidth attached to each socket in Table 3.5.

Although it appears there are three different DRAM technologies employed, there are in fact only two. FBDIMMs [47] leverage DDR2 [38] technology by placing DIMMs on a ring rather than a bus. There are three issues for this approach. First, as there is an extra chip per DIMM to act as a node in the ring, the DIMMs consume much more power. Second, there are separate read and write lines sustaining a 2:1 bandwidth ratio. Finally, on DDR2-based computers, as the number of DIMMs per channel increases, the bus load increases. At a critical load, the DIMMs are clocked at a lower frequency. If too few DIMMs are attached, however, then there is insufficient concurrency on the channel to hide the overhead. FBDIMM eliminates capacitive load as a concern. We have judiciously balanced the number of DIMMs on each computer to maximize performance. The other type of DRAM technology employed is XDR [145]. Although this technology promises much higher bandwidth, a comparable cost DRAM capacity is more than a order of magnitude less than DDR2.

Access to Other Sockets

Requested data may not reside either in a local (same socket) cache or directly attached DRAM. In such a case, access to the remote socket is required. Table 3.6 on the next page shows the inter-socket interconnect topologies used by the SMPs in this work. Note, there are three possible communication paths — two for the Intel computer, and one for the others.

Intel computers of this era all rely on an external memory controller hub that centralizes all accesses to DRAM, I/O and other sockets. The computer used in this work is a Dell PowerEdge 1950, and uses the Intel 5000X memory controller hub (MCH). This MCH is quite different than most in that it uses a dual independent bus (DIB) architecture. Each MCM has access to its own front side bus (FSB). However, as each MCM is in reality

Parameter	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade
type	Dual FSB	HyperTransport	HyperTransport	custom by address	custom
topology	dual buses	direct connect	direct connect	direct connect	direct connect
bandwidth	2×10.66 GB/s† (read or write)	≈4 GB/s (each direction)	≈4 GB/s (each direction)	4×6.4 GB/s (each direction)	<20 GB/s (each direction)

Table 3.6: Summary of inter-socket connection types and topologies by computer. For the Opterons, we show practical bandwidth rather than pin bandwidth. †FSB bandwidth also shared with DRAM and I/O access

just two chips, each FSB has three agents: two chips, and the MCH. Alternate designs that use a single FSB would thus have the load of five agents. By reducing the number of agents, the frequency of the bus may be increased to 333 MHz. All data transactions are quad pumped (i.e. 4 bits per clock) resulting in a raw per FSB bandwidth of 1333 MT/s (10^6 transfers/second) or 10.66 GB/s. Two chips on Clovertown socket may communicate through via their shared front side bus (FSB). However, if communication between sockets is required, then communication is handled via the first FSB to the MCH, then from the MCH over the second FSB to the second socket. Clearly, the latter consumes twice the number of bus cycles and incurs substantial latency.

The third possible communication path is a direct connection between two sockets. The Opterons use HyperTransport with separate sustained 4 GB/s per direction per link. Cell uses a similar approach with a much higher frequency bus. Finally, instead of a single link on which all accesses are sent, Victoria Falls defines four separate coherency domains based on the lower bits of the cache line address [107]. Each of the four domains has a pair of 6.4 GB/s links (one per direction).

Access to I/O

Although not required for this work, access to I/O is handled differently on each computer. On Clovertown, I/O access is handled indirectly by the MCH via the FSB. On the Opterons, there are additional HyperTransport links to a I/O PCI-e bridge or hub. Both Cell and Victoria Falls have a separate, dedicated I/O bus.

3.1.4 Coping with Memory Latency

Memory latency has become one of the most severe impediments to performance. As such, architects have endeavored to devise novel techniques to avoid or hide memory latency. Be mindful of the difference. Avoiding memory latency is principally handled through caches. The standard techniques for hiding memory latency include out-of-order execution, software prefetching, and vectors [68]. All three x86 computers used in this work exploit out-of-order execution, and all cache-based architectures can exploit software prefetching. No computer used here exploits long vector execution. In this section, we discuss three alternate and novel techniques to hiding memory latency.

Detect misses from:	Prefetch Parameter	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
load queue	pattern streams request from store into	unit-stride 1? L1 line buffers	—	—	—	—	—
load queue (by PC)	pattern streams request from store into	strided 1 per PC L1 line buffers	—	—	—	—	—
L1	pattern streams request from store into	strided 12+4 per core L2 L1	unit-stride ? DRAM L2	unit-stride ? L2 & L3 L1	—	—	N/A
L2	pattern streams request from store into	strided ? DRAM L2	—	—	—	—	N/A
L3	pattern streams request from store into	N/A	N/A	strided ? DRAM dedicated buffer	N/A	N/A	N/A

Table 3.7: Summary of hardware stream prefetchers categorized by where the miss is detected. Only data prefetchers are shown. Note, Cell has no caches thus cannot prefetch from one. However, it does have local stores. The architects accepted the 6-cycle latency rather than the complexity of a local-store load queue prefetcher.

Hardware Prefetching

The first technique, hardware stream prefetching, is entirely a hardware solution. It requires no programming effort. In essence, it is a speculative technique as it attempts to predict future cache misses and generate a transfer early enough to ensure memory latency is not exposed. Initially, hardware prefetchers could only detect a pattern if accesses to consecutive cache lines resulted in misses. The prefetchers have been improved on Clovertown and Barcelona to detect arbitrary strides. In addition to the detectable patterns, the other parameter is the number of concurrent streams that can be differentiated. Modern architectures can employ multiple stream prefetchers, one for each level of the cache. As hardware stream prefetchers almost invariably operate on physical addresses, they cannot cross page boundaries, which are 4 KB on the x86 computers. Thus, the latency of the first access to a page is always exposed. Disturbingly, this implies that as memory bandwidth increases, the time to access one page using one core asymptotically approaches the ratio of page size to DRAM latency. That is, $BW_{average} = \frac{BW_{pin} \cdot PageSize}{BW_{pin} \cdot Latency + PageSize}$. As BW_{pin} increases, $BW_{average}$ tends to $\frac{PageSize}{Latency}$. The only viable solution is to have multiple hardware prefetchers simultaneously engaged so that this latency can be hidden. This concurrency is achieved through multicore and padding of arrays.

Table 3.7 summarizes the hardware prefetch capabilities of the computers. Closest to the core, Clovertown can detect requests from the L1 for both unit-stride patterns from the line buffers as well as strided patterns by matching them with the IP (program counter) of the load instruction. The Santa Rosa Opteron prefetches data from DRAM into the L2.

As such, additional software prefetching may be necessary to hide the latency to the L2. Note, Barcelona's DRAM prefetcher detects L3 misses, but prefetches data on idle DIMM cycles into a dedicated L3 buffer that the L3 will subsequently read from.

Hardware Multithreading

As hardware prefetchers are only effective when presented with certain address patterns, hardware multithreading has become a common solution when there is no discernible pattern, but ample thread-level parallelism. Hardware multithreading virtualizes the resources of a core by instantiating multiple hardware thread contexts. This includes the state required for a thread's execution, including register files, program counters, and privileged state. On every cycle the fetch and issue logic must select a thread, and fetch the corresponding instructions from the cache and then issue them down the pipeline. There are three basic granularities at which threads may be interleaved: coarse, fine, or simultaneous [68].

In this work, only two architectures implement hardware multithreading: Victoria Falls and the Cell PPEs. Victoria Falls groups four hardware thread contexts into a thread group. It then groups two thread groups into a core. There is fine-grained multithreading between threads within a thread group. Essentially, on every cycle one ready-to-execute instruction from each thread group is issued to that thread group's pipeline. When the two instructions reach the decode state, resource conflicts are resolved for the FPU and memory units that are shared between thread groups. The Cell PPE is also fine-grained multithreaded. However, unlike the more dynamic approach employed by Victoria Falls, the PPE has only two hardware thread contexts and tasks even cycles to the even thread and odd cycles to the odd thread.

Both memory or functional unit latency can be hidden by switching to ready threads. In today's world, hiding memory latency is the preeminent challenge. Multithreading can satisfy the concurrency demanded by Little's Law by providing one independent cache line per thread. Ideally, on Victoria Falls, this would provide 64 $threads \times 64\ bytes/thread = 4$ KB of concurrency per socket to DRAM — enough to hide 190 ns of DRAM latency. However, on the Cell PPEs, two-way multithreading can only cover 10 ns of DRAM latency — clearly a far cry from the nearly 200 ns of memory latency. This technique can be applied to the L2 cache as well. On Victoria Falls, the L1 line sizes are only 16 bytes. As such, only 1 KB ($8\ cores \times 8\ threads\ per\ core \times 16\ bytes$) of concurrency is expressed to the L2. Given the L2 read bandwidth of 128 bytes per cycle, and a latency of 20 cycles, multithreading may only utilize 40% of the L2 bandwidth.

The biggest pitfall of multithreading is the fact that a huge number of independent accesses are generated. As a result, conflict misses and bank conflicts become common. Moreover, page locality within a DIMM becomes a challenge as other threads are contending for limited resources. Careful structuring of the memory access patterns or severe overprovisioning is required.

Direct Memory Access (DMA)

The one final technique for coping with memory latency is direct memory access (DMA). As discussed with the previous topics, the goal is expression of concurrency to the memory subsystem. Unlike hardware prefetchers, the programmer is required to detect and express the concurrency, and unlike multithreading, all the memory-level parallelism must be derived from one thread, although user-level software multithreading is a viable solution in some cases [115, 9]. In the simplest DMA operation, the user specifies a source address, a destination address, and the number of bytes to be copied. As DMA operations are typically asynchronous, completion of a DMA is often detected though interrupts or polling. More complex DMAs can realize scatter or gather operations. In a single command, the user specifies an address of a list of DMA stanzas (addresses and sizes) as well as the packed address. The DMA engine then asynchronously processes the list, either unpacking the array and scattering stanzas or gathering stanzas into a packed array. The memory latency is amortized by the total concurrency expressed by the DMA. Moreover, the latency can be hidden with multiple cores or multi-buffering.

In theory, there is no reason why DMA can't be coupled with cache architectures or why multithreading can't be coupled with local store architectures. In the former cases, one must specify how deep in the cache hierarchy the DMA data should be cached. If the DMA exceeds the cache capacity, then the equivalent of write backs would be generated. Nevertheless, the only architecture in this work to use DMAs is the cacheless Cell SPEs. Thus, DMA is coupled with a local store. Cell implements four basic types of DMA:

- **GET** — copy one stanza from DRAM to the local store.
- **PUT** — copy one stanza from the local store to DRAM.
- **GETL** — gather a list of stanzas from DRAM, and pack them contiguously in the local store.
- **PUTL** — take a packed local store array, and scatter a series of stanzas to DRAM.

To satisfy Little's Law, one only need to ensure that the aggregate sizes of all the DMAs in flight expresses sufficient concurrency.

3.1.5 Coherency

Most computers in this work use some variant of a snoopy protocol [68] for inter-socket cache coherency. On the Opterons and Victoria Falls, the MOESI [4] protocol is handled over the inter-socket network. However, Clovertown's and Cell's cache coherency protocol is somewhat different and requires some additional explanation. On all computers, the latency for the coherency protocol is likely to be greater than that for physical access to DRAM. As such, the latency in Little's Law is the coherency latency. Moreover, without sufficient concurrency in the memory subsystem, this latency might limit bandwidth.

Snoop Filter

On Clovertown, coherency is handled via a snoopy MESI protocol. However, chips may only snoop on the bus to which they are attached. As the Clovertown employs a dual bus architecture, every memory transaction may result in a coherency transaction being forwarded to the second bus. To minimize FSB transactions, a rather large cache coherency filter (deemed a *snoop filter*) [73] is included in the MCH. The goal of the snoop filter is to eliminate superfluous coherency traffic, thereby retasking the available bandwidth for data transfers.

The snoop filter is divided into two 8K sets $\times$ 16-way affinity groups (one per FSB). Although the snoop filter only holds tags, it's designed to track 16MB of L2 cache lines. Each entry attempts to record the MESI state as well as which socket, if either, has ownership. The snoop filter replacement policy is not directly tied to L2 replacement policy. Moreover, as L2 line states can change without a bus transaction (e.g. an eviction of shared or exclusive data), it is possible that the snoop filter may fall out of sync and believe an entry is present when it is not. Conversely, the snoop filter may replace and evict an entry when the cache selected a different entry. Thus, it is improper to say the snoop filter generates a snoop when it is required to. Rather, the snoop filter won't generate a snoop when it know its not required to. This subtle distinction can have far ranging ramifications on application performance.

If the data sets of interest are significantly larger than either the caches or the snoop filter, the snoop filter is likely to be ineffective. As such, a transaction on the first FSB will invariably generate snoop traffic on the second, and vice versa. This might be acceptable if not for the combination of small cache lines and a quad pumped data rate. When these two are combined, the cycles required for a data transfer are comparable to those required for coherency. Thus, for problems with large datasets, roughly 50% of the raw FSB bandwidth must be dedicated to coherency [61]. The combination of a slightly higher bus frequency and double the number of buses comes with a huge price: twice the traffic. As a result, the effective bandwidth available to a dual-socket quad-core Xeon is only slightly greater than that available to a single socket Core2 Quad.

Cell Broadband Engine

The SPE local stores are not caches, but rather disjoint address spaces. As such, each line in the local store is a unique address and the data itself cannot be cached elsewhere. As a result, no coherency traffic is generated as a result of SPE reads and writes to the local store. This saves a huge amount of inter-core coherency traffic and provides a very scalable solution. Through DMA, however, the SPEs may transfer data between their local stores and the shared main memory (DRAM). At this point cache coherency becomes an issue. As each PPE can cache main memory data in its caches, all caches must be kept coherent with all DMAs. IBM's engineers employed a very straightforward approach to this. Before a DMA may execute, a coherency protocol snoops both caches (one per socket) and evicts matching addresses back to DRAM. The DMA may then execute and read the address whose most up to date data will always be in DRAM. Thus, it is impossible for a SPE to directly read from a PPE's cache. To some extent, this limits a programmer's ability to

exploit heterogeneity. The solution is to have the PPEs write the SPE local stores rather than having the SPEs read from the PPE's caches. PPE accesses to the local stores aren't cached and thus coherency is not an issue.

3.2 Programming Models, Languages and Tools

This thesis makes no attempt to develop or evaluate programming models, languages, or compilers. We simply specify our choices and do not dwell on the innumerable alternatives. Sufficive to say, we use C with intrinsics for every kernel on every architecture. Intrinsics are essentially individual assembly instructions than can be used within a C program as if they were simple functions or macros. This section discusses the programming model, scaling model, and in addition, details the implementations of our shared memory barriers and the affinity routines we used.

3.2.1 Programming Model

Throughout this work, we employ a bulk-synchronous, single-program, multiple-data (SPMD) shared-memory parallel programming model. Moreover, we use POSIX Threads (`pthreads`) [129] as the threading library of choice. It should be noted that the Cell SPEs require an additional `libspe` threading library. Although the model is multi-threaded, the SPMD label is still appropriate. All threads will execute the same function, but on different data.

We exploit heterogeneity for control and productivity rather than simultaneous computation. Thus, while the Cell PPE is performing computation, the SPEs are waiting. When the Cell SPEs are performing computation, the PPE is waiting.

In this work, we make no statements about other programming models or communication approaches like OpenMP [103], UPC [15], or MPI [119], other than to say much if not all the work presented here is likely applicable.

3.2.2 Strong Scaling

In the single-core era, weak scaling is the methodology by which one fixes the problem size per processor (an MPI task), and scales the number of processors used in a super computer. Strong scaling is the reverse. The problem size per processor is inversely proportional to the number of processors. In the multicore-era, this taxonomy is more complicated. Assuming each socket is assigned an MPI task, one may:

- keep the total problem size constant (traditional *strong scaling*).
- scale the total problem size proportional to the number of cores per socket.
- scale the total problem size proportional to the total number of sockets (traditional *weak scaling*).
- scale the total problem size proportional to product of the number of sockets and cores per socket (proportional to the total number of cores).

All experiments in this dissertation only explore single node multicore scalability performance on applications designed for weak scaling. As such, we examine the third bullet. That is, the problem per SMP remains constant, but the number of hardware thread contexts employed is scaled in a very regimented manner. Initially, the serial case is run. Second, within one core on one socket, the number of hardware thread contexts is scaled to the maximum number. Third, within one socket, the number of fully threaded cores is scaled from one to the maximum. Finally, the number of sockets is scaled from one to two while using all threads and cores.

There are several motivations for using this approach. First, it is a significantly more difficult challenge, as the balance between a kernel's components may not scale well with the number of threads. Second, unlike large supercomputers, whose memory capacity scales with the number of processors, memory capacity on multicore processors will likely scale more slowly than the number of cores. Finally, load balancing can be more challenging.

3.2.3 Barriers

A barrier is a collective operation in which no participating thread or task may proceed beyond the barrier until all participating threads have entered it. Barrier performance acts as an upper limit to scalability when conducting strong scaling experiments. Naïvely, speedup is $NThreads$, assuming perfect load balance on a strong scaling experiment. That is, the time spent per thread scales as $\frac{TotalWork}{NThreads}$. Let us define *barrier time* as the time between when the last thread enters the barrier and the last may leave. In the best case, barrier time may add a constant to the compute time per thread and the time spent per thread is $\frac{TotalWork}{NThreads} + BarrierTime$. As such, the best speedup ($\frac{TotalWork}{\frac{TotalWork}{NThreads}+BarrierTime}$) is $\frac{TotalWork}{BarrierTime}$. Alas, barrier time may scale linearly with the number of threads. As such, there would be an optimal number of threads beyond which performance will degrade simply because execution time is dominated by the barrier time. Alternatively, one could interpret this as placing a relationship between the size of a function, and the maximum parallelization that can be employed on it. That is, functions that are too small can't be parallelized.

Although POSIX threads provides a barrier routine or other means to realize such functionality among threads, their performance is abysmal. Despite the fact that our kernels will require relatively infrequent barriers when using two threads, as the number of threads scales, performance can still be substantially limited. To mediate this, we implemented two versions of a fast shared memory barrier — one for cache-based computers and one for Cell.

In the Cell version of the barrier, a `waiting` variable is created in each local store. Both the PPE and all SPEs must call the barrier routine. Upon entry, each SPE will set its `waiting` variable within its local store. It will then spin waiting for it to be cleared. When the PPE enters the barrier, it spins waiting for all SPE `waiting` variables to be set. When this condition is met, the PPE resets all SPE `waiting` variables, and exits the barrier. Within each SPE, they will observe the `waiting` variable has been cleared, and will exit the barrier. To ensure PPE-SPE communication doesn't impair performance, we limit PPE accesses to the SPEs to once per microsecond. For our purposes, an exponential back off was unnecessary.

```
void barrier_wait(barrier_t *barrier, int threadID){
  double x = 2.0;
  int i;
  if(barrier->WaitFor==1)return;  // 1 thread waiting for itself
  barrier->ThreadIsWaiting[threadID] = 1;
  if(threadID==0){
    // thread 0 is the master thread
    // (it has sole write control on the barrier)
    int ThreadsWaiting = 0;
    while(ThreadsWaiting != barrier->WaitFor) {
        // not all threads are done
        ThreadsWaiting = 0;
        for(i=0;i<barrier->WaitFor;i++)
          ThreadsWaiting+=barrier->ThreadIsWaiting[i];
    }
    // master thread now resets all other threads
    // (same way PPE does it on cell)
    for(i=0;i<barrier->WaitFor;i++)barrier->ThreadIsWaiting[i]=0;
  }else{
    // other threads just wait for the master thread to release them
    // use divide to ensure spinning doesn't sap cycles on MT cores
    while(barrier->ThreadIsWaiting[threadID]){x=1.0/x;}
  }
  return;
}
```

Figure 3.2: Shared memory barrier implementation. Note that the Cell implementation is very similar.

The cache-based implementation is quite similar and is shown in Figure 3.2. However, there are N identical threads instead of one PPE and N SPE threads. Thus, we re-task thread_0 to handle the functionality of the PPE. Thus $\text{thread}_0..\text{thread}_{N-1}$ will set their `waiting` variable, and thread_0 will reset all of them. On single threaded architectures, spinning (load, compare, branch) only wastes power. However, on multithreaded architectures, these instructions sap instruction issue bandwidth from other threads. Thus, to ensure spinning doesn't impair hardware multithreading, each thread executes a non-pipelined floating-point divide when spinning. Future work could implement a less portable, architecture-specific solution such as x86's `mwait` instruction.

Unfortunately, both barrier implementations easily scale linearly with the number of threads. On going research by Rajesh Nishtala *et al.* aims to auto-tune barriers and other collectives [20]. His approach explores a variety of tree topologies. Initial results show roughly logarithmic scaling in time. Such an approach increases the attainable concurrency or reduces the minimal quanta for parallelization.

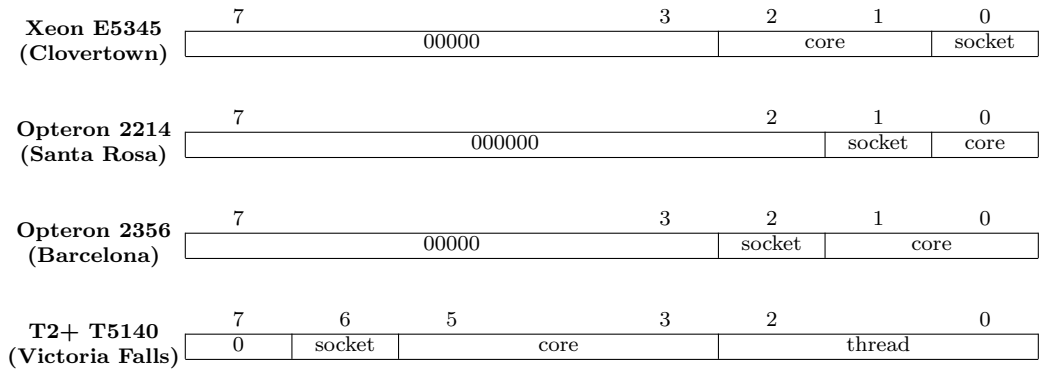


Table 3.8: Decode of an 8-bit processor ID into physical thread, core, and socket

3.2.4 Affinity

To ensure consistent and optimal performance, we pin threads to cores. Moreover, we use the affinity routines to ensure data is placed in memory attached to the socket tasked to process it. Each operating system provides different routines by which such affinity functionality may be implemented. On the x86 architectures, we use the Linux scheduler's `sched_setaffinity()` routine and presume a first touch policy [48]. On Victoria Falls, we use the complimentary Solaris scheduler's `processor_bind()` routine and also assume a first touch policy. For Cell, we use `libnuma`'s `numa_run_on_node()` to bind SPE threads to one socket or the other and `numa_set_membind()` to bind memory allocation to one socket or the other.

The mapping of Linux processor IDs to physical cores is computer dependent. In addition, the mapping of Solaris processor to physical thread is generally not well known. In both cases, we present how one could decode the processor ID into physical thread, core, and socket. Table 3.8 shows how an 8-bit Linux/Solaris processor ID would be decoded into physical thread, core, or socket. Clearly, using processors 0 and 1 is a very different mapping on Clovertown compared to Barcelona. As such, the affinity routines were made cognizant of the mapping. As this mapping is unique to each computer (not just to each processor), future work should investigate a more portable solution.

3.2.5 Compilers

Before performing our experiments, we performed several preliminary experiments (not presented here) to determine the appropriate compiler and compiler flags for each computer. Table 3.9 on the next page lists those parameters. Note that all architectures except the 32-bit Cell SPEs were compiled for a 64-bit environment. Our version of gcc didn't have a tuning option for the Core microarchitecture. Surprisingly, gcc tuned for Nocona (a 64-bit Pentium4) often produced better performance on memory intensive kernels than icc compiled for the Core microarchitecture. The Sparc gcc 4.0.4 was significantly faster than gcc 4.0.3. Apparently, the backend was replaced and many optimizations enabled in

Computer	Compiler	Flags
Xeon E5345 (Clovertown)	icc 10.0 gcc 4.1.2	-O3 -fno-alias -fno-fnalias -xT -O4 -march=nocona -mtune=nocona -msse3 -m64 -funroll-loops
Opteron 2214 (Santa Rosa)	gcc 4.1.2	-O4 -march=opteron -mtune=opteron -msse3 -m64 -funroll-loops
Opteron 2356 (Barcelona)	gcc 4.1.2	-O4 -march=opteron -mtune=opteron -msse3 -m64 -funroll-loops
T2+ T5140 (Victoria Falls)	gcc 4.0.4	-fast -m64 -xarch=v9 -xprefetch=auto,explicit
Cell QS20 (PPE)	xlc 8.2	-O3 -qaltivec -qenablevmx -q64
Cell QS20 (SPE)	xlc 8.2	-ma -O3 -qnohot -qxflag=nunroll

Table 3.9: Compilers and compiler flags used throughout this work. On Clovertown, icc delivered comparable performance for sparse matrix-vector multiplication.

this minor revision change. Note that the Cell PPE compiler flags were those used when running the PPE in standalone mode.

3.2.6 Performance Measurement Methodology

Throughout this work we measure average performance over ten trials. In the HPC world, where these kernels will be executed thousands if not millions of times, average is the appropriate metric. In other domains, minimum, or median performance might be more appropriate. Nevertheless, two quantities must be measured: the total number of floating-point operations per trial and the time for ten trials. We determine the former by manually inspecting the computational kernel as well as the dataset or problem size. We calculate the latter using each computer's cycle counter. We define *GFLOP/s* as billions (10^9) of floating-point operations per second.

Each computer used in this work has a low-level cycle counter. On all architectures except Cell, this cycle counter counts core cycles. On Cell and PowerPC, the counter is called the TimeBase. The TimeBase frequency is implementation dependent. Playstations, QS20's and QS22's each run at a different frequency. The counters run continuously and are independent of task switches. Thus, to measure the time required by a kernel, we read the counter both before and after and take the difference. To calibrate the cycle counter frequency, we perform a `sleep(1)`, and measure the delta in the cycle counter. As a basis, we use the FFTW [52] cycle counter implementations reproduced in Table 3.10 on page 42.

3.2.7 Program Structure

Figure 3.3 on the next page shows the basic benchmark program structure. The cache-based and Cell implementations are remarkably similar. The primary difference is that on the cache-based implementation, Thread0 is tasked with the work performed by the Cell PPE and SPE0. Thus, it handles all barrier management and any serial code in addition to the computation handled by SPE0.

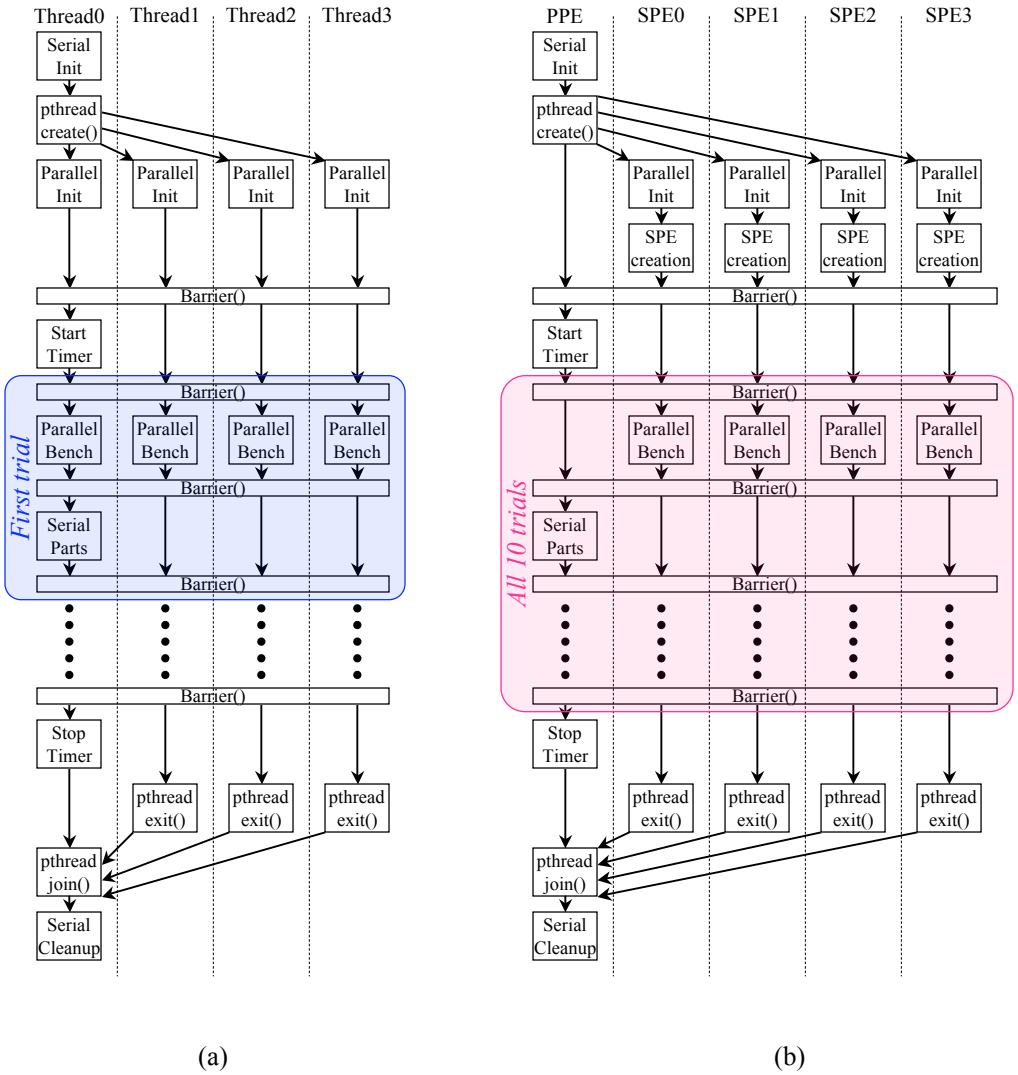


Figure 3.3: Basic benchmark flow. Auto-tuning adds a loop around the ten trials to explore the optimization space. In the cache implementations (a) Thread0 is tasked with the same work as the PPE and SPE0 in the Cell version (b).

ISA	Code Snippet
x86/64	`__asm__ __volatile__ ("rdtsc" :  "=a" (lo), "=d" (hi));` `return( (((uint64_t)hi) << 32) | ((uint64_t)lo) );`
SPARC	`__asm__ volatile("rd %%tick, %0" :  "=r" (ret));` `return ret;`

PowerPC	`do {`
	`  tbu0 = __mftbu();`
	`  tbl = __mftb();`
	`  tbu1 = __mftbu();`
	`}while (tbu0 != tbu1);`
	`return (((volatile uint64_t)tbu0) << 32) | (volatile uint64_t)tbl;`

Table 3.10: Cycle counter implementations.

We now provide a brief walkthrough of the basic structure and parallelization scheme for this bulk-synchronous, single-program, multiple-data benchmark. Upon program invocation, any argument processing or serial initialization is performed. This preprocessing does not include allocation of any data structures requiring some affinity. Next, the main thread then creates either N-1 additional `pthreads` for the cache-based computer or N `pthreads` for Cell. As Cell has an extra core (the PPE), it always runs with one extra thread. The main thread then performs a function jump to the same function as the target of the `pthread_create()`. At this point any initialization or allocation requiring memory affinity is performed. On the Cell implementation the N `pthreads` each create one SPE thread. A global barrier ensures all threads have completed initialization. Thread0 or the PPE starts the timer and another barrier is performed. The threads then make 10 passes through the benchmark. The benchmark consists of two phases: a parallel part and a serial part; after each is a barrier. After the second barrier on the 10^{th} trial, the timer is stopped and performance is calculated. The created threads then perform a `pthread_exit()` while the creating thread does a return and a `pthread_join()`. Any final cleanup is performed serially.

Although this is the structure of the benchmarks used in this work, we believe it is the appropriate template for many performance-oriented threaded codes. Moreover, we believe thread creation should be performed only once. Subsequently, functions should be dispatched enmasse to the entire thread pool. A barrier instead of a join would be placed at the end of each of these functions. If threads aren't tasked with work, they will reach the barrier quickly and then spin or sleep.

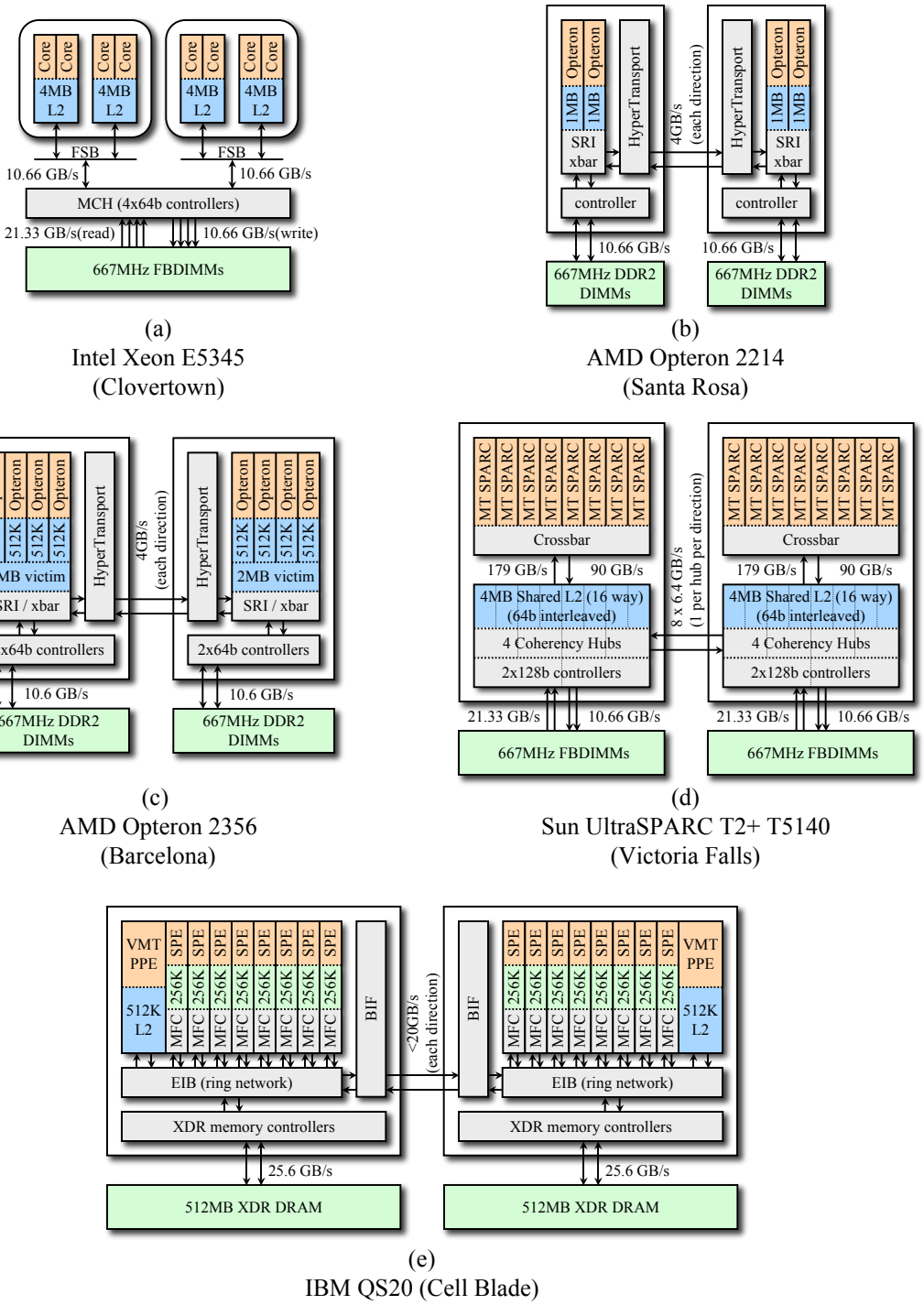


Figure 3.4: The five computers used throughput this work. Note, the Cell blade contains two architectures — the PPE and the SPEs. As such, it will be used for both cache-based and local store-based experiments. Note, L1 caches are not shown.

3.3 Summary

In this chapter we discussed the five computers used throughout this work: Intel's Xeon E5345 (Clovertown), AMD's Opteron 2214 (Santa Rosa), AMD's Opteron 2356 (Barcelona), Sun's T2+ T5140 (Victoria Falls), and IBM's QS20 Cell Blade. Figure 3.4 on the preceding page provides a visual representation of these five computers showing their bandwidths and topologies. Note, L1 caches are not shown. In addition, we provided some requisite background material on the novel or relatively unfamiliar architectural features of these computers. We then provide an overview and some implementation of the bulk-synchronous, single-program, multiple-data (SPMD) shared memory parallel (SMP) strong scaling programming model we implemented using POSIX threads. Similarly, we provided in-depth details on our affinity, barrier, and timing routines.

Chapter 4

Roofline Performance Model

This chapter presents the visually-intuitive, throughput-oriented Roofline Model. The Roofline Model allows a programmer to model, predict, and analyze an individual kernel's performance given an architecture's communication and computation capabilities and the kernel's arithmetic intensity. When used in the context of tuning, the model clearly notes which optimizations and architectural paradigms must be exploited to attain performance. Given a kernel's observed performance, one may quantify further potential performance gains. Qualitatively, one can compare the Roofline Models for a set of machines to gain some insight into the requisite software complexity and productivity. Ultimately, the Roofline Model allows one to reap much of the potential performance benefit with relatively little detailed architectural knowledge. We use this model to understand and qualify the performance of the auto-tuned kernels in Chapters 6 and 8.

In this chapter, we use memory-intensive floating-point kernels as the primary motivator for the Roofline Model. In the latter sections we generalize the Roofline Model to other communication and computation metrics. Section 4.1 discusses a few related and dependent performance models. Section 4.2 clearly defines the work and performance metrics as well as the concept of arithmetic intensity used throughout the rest of this work. Section 4.3 synthesizes communication, computation and locality into a performance graph using bound and bottleneck analysis — a naïve Roofline Model. Unfortunately, such a simplified graph is of little value in the performance analysis and optimization world. As such, Sections 4.4 through 4.6 expand the Roofline Model by noting the computer's response to perturbations in the parameters contained in Little's Law [10] as well as its response to capacity or conflict cache misses. Section 4.7 unifies the previous three sections into a single model, and makes several qualitative assessments of it. Section 4.8 speculates on how one might extend the Roofline Model to other communication or computation metrics. Section 4.9 describes how one could use performance counter data to construct a runtime-specific rather than architecture-specific Roofline Model. Finally, we summarize the chapter in Section 4.10.

4.1 Related Work

Enormous effort has been invested on the part of the performance optimization and analysis community in performance modeling. More specifically these efforts can be divided into two main categories: performance prediction, in which software is used to predict the performance of future hardware, and performance analysis, in which one attempts to understand observed performance on existing or future hardware.

Perennially, architects have written software simulators to predict performance as hardware parameters are tweaked. Although such approaches may be extremely accurate in predicting hardware performance years before a machine is built, in themselves they do not provide any insight into why an architecture performs well or not. Nor do they provide any insight into how one would optimize code rather than redesign hardware. Finally, their abysmal performance (simulated cycles per second) mandates either enormous simulation farms or restricts their use to small kernels. The former is very expensive, and the value from the latter is small. The work presented in this thesis is focused on understanding a machine's performance, and adapting software to real hardware. Thus, we don't require a simulator to analyze or predict performance.

Accurate performance counter collection from real hardware only provides a flood of data. Although one could observe that changes in hardware or software might result in substantially different numbers of, say TLB misses, a separate approach is required to understand why there are so many misses.

More recently, statistical methods have been developed to predict performance and identify hardware bottlenecks [118, 28]. These methods attempt to produce a machine signature by running a training set on them and deriving the parameters from performance. To predict application performance, one must also characterize the access patterns and computation of the application and use an operator to combine the resultant application profile with the machine signature. In the simplest form, this is a linear combination of the parameters. In many ways, this is somewhat of a black art. Moreover, although one maybe able to predict performance and perhaps how fast future architectures could run the same program, little valuable insight is provided into how architects should change future hardware and nor is any insight afforded to programmers on how to restructure their code for current or future processors. One can estimate existing bottlenecks by comparing performance-correlated application and architecture signatures.

Our recent paper [36] theorized that the behavior of hardware prefetchers can severely impair the conventional blocking strategies. When a hardware prefetcher is engaged, the miss time is a function of bandwidth. When they are not effective, however, then the miss time is a function of exposed memory latency. To that end, a microbenchmark was created to quantify the fast and slow miss times. A performance model could then predict stencil performance as a function of blocking as well as fast and slow miss times. Clearly, such an approach was only possible because the authors had substantial knowledge about the kernel and well founded theories as to the performance bottlenecks. However, such a performance model would have been completely oblivious to software optimization strategies such as software prefetching or a hierarchical restructuring of the data.

Rather than modeling an architecture's performance response to code (either blindly or with some knowledge), some have reversed the process and begun by analyz-

ing code. When a code's demands exceed an architecture's capabilities, performance will suffer.

The compiler community uses similar models for code generation. *Machine balance* is often defined as the ratio of memory bandwidth (in words) to FLOP/s (see paper [26, 27]. A complimentary term is *loop balance*. Loop balance is the steady state ratio of a loop's total number of loads and stores to total FLOPs. Without caches, one might conclude that when a loop balance exceeds machine balance, the machine is bandwidth limited (*i.e. memory-bound*). The presence of caches and thread-level parallelism coupled with the idiosynchronies of the memory substsystems devalues this concept.

A similar approach is bound and bottleneck analysis. It is a very simple approach that yields high and low bounds to performance. One example applies bound and bottleneck to sizing of service centers [85]. Their basic units are the number of users and demands per center. As the number of users increases, performance increases. However, performance will eventually saturate at the inverse of the most heavily bottlenecked service center.

In this chapter, we leverage the simplicity of bound and bottleneck analysis with the concept of machine balance as the basis for the Roofline model. Thus, in our work the service centers of bound and bottleneck analysis have been transformed from datacenters into memory controllers and FPUs. Alternate approaches like simulation or statistical methods do not provide the performance, simplicity, or insight we desire.

4.2 Performance Metrics and Related Terms

In this section, we define the terms used as inputs to the Roofline Model. These include work, performance, and arithmetic intensity. We limit ourselves to memory-intensive floating-point kernels.

4.2.1 Work vs. Performance

Every kernel has a computation performance metric of interest. Each kernel will perform some units of work over a given period of time. Work can include computation (*i.e.* transformation or combination of data) or data movement.

For our kernels, the computational work performed by the various kernels is floating-point operations. These include add, subtract, multiply, and sometimes divide. Thus, our throughput performance metric is floating-point operations per second (FLOP/s) or billions (10^9) of floating-point operations per second (GFLOP/s). It is conceivable that higher-level metrics are possible: lattice updates per second or matrix multiplications per second. Unfortunately, there are several reasons why such metrics are inappropriate in the context of performance optimization. For example, there are many lattice methods, and each method might require a different number of floating-point operations per lattice update. Lattice updates per second alone provides no insights into the resultant architectural bottlenecks, whereas GFLOP/s does. As such, for purposes of clarity in the performance optimization arena, it is easier to relate application performance measured in GFLOP/s.

When it comes to communication between storage (DRAM) and computational resources (CPU), we define work as the total number of bytes transfered. This includes all

Kernel	Characteristic Parameter(s)	Compulsory FLOPs	Compulsory Memory Traffic	Compulsory Arithmetic Intensity	Requisite Cache Capacity
Dense Vector-Vector Addition	Vector Size (N)	N	24·N	$\frac{1}{24}$	O(1)
Dense Matrix-Vector Multiplication	Matrix Rows (N)	$2 \cdot N^2$	$8 \cdot N^2 + 16 \cdot N$	$\frac{1}{4}$	O(N)
Dense Matrix-Matrix Multiplication	Matrix Rows (N)	$2 \cdot N^3$	$24 \cdot N^2$	$\frac{N}{12}$	$O(N^2)$
Sparse Matrix-Vector Multiplication	Nonzeros (NNZ) Matrix Rows (N)	2·NNZ	12·NNZ + 16·N	$\frac{1}{6}$	<O(N)
3D Heat Equation PDE (Jacobi)	Grid Dimension (N)	$8 \cdot N^3$	$16 \cdot N^3$	$\frac{1}{2}$	O(1)
1D Radix-2 FFT	Sample Size (N)	5·Nlog(N)	32·N	$\frac{5}{32}\log(N)$	O(N)
N-body Force Calculation	Number of Particles (N)	$O(N^2)$	O(N)	O(N)	O(N)

Table 4.1: Arithmetic Intensities for example kernels from the Seven Dwarfs. Arithmetic intensity is the ratio of compulsory FLOPs to compulsory memory traffic. Note, to asymptotically achieve such arithmetic intensities, very large cache capacities are required.

memory traffic arising from compulsory, capacity, and conflict misses. In addition, it include all speculative transfers — *e.g.* hardware initiated prefetching. We define bandwidth (communication performance) as bytes transfered per second (B/s) or more commonly billions (10^9) of bytes transfered per second (GB/s). In the absence accurate performance counter data, we may approximate memory traffic by assuming there are no conflict or capacity misses.

4.2.2 Arithmetic Intensity

We redefine *arithmetic intensity* to be the ratio of compulsory floating-point operations to the total DRAM memory traffic; that is, the ratio of computation work to communication work. Total DRAM memory traffic is all memory requests after being filtered by the cache. Similarly, we define *compulsory arithmetic intensity* to be the ratio of compulsory floating-point operations (minimum number of FLOPs required by the algorithm) to the compulsory DRAM memory traffic. The latter is a characteristic solely of the kernel, where the former is a characteristic of the execution of a kernel on a specific machine. In many ways, compulsory arithmetic intensity is a broad measure of the locality within a kernel.

If one were to consider the canonical example kernels from a subset of the Seven Dwarfs [31], one would notice that some kernels have compulsory arithmetic intensity that is constant regardless of problem size, while others have arithmetic intensity that grows with the problem size. Table 4.1 clearly shows that arithmetic intensity has substantially different scaling as a function of memory traffic not only for kernels within a dwarf but

also across dwarfs. Clearly, many of these arithmetic intensities are rather low — less than the conventional wisdom design point of 1 FLOP per byte. Moreover, to asymptotically achieve such arithmetic intensities, substantial cache capacities are required. Many of these arithmetic intensities could be applied hierarchically. That is, the arithmetic intensity of dense matrix-matrix multiplication could be applied to storage and transfers from DRAM or storage and transfers from the L2 cache. Clearly, in the latter case the L2 arithmetic intensity can only grow until limited by L2 cache capacity.

4.3 Naïve Roofline

Using bound and bottleneck analysis [85], Equation 4.1 bounds attainable kernel performance on a given computer. We label this well known formulation to be a naïve Roofline Model. In this formulation, there are only two parameters: peak performance and peak bandwidth. Moreover, there is a single variable: arithmetic intensity. As we are focused on memory-intensive floating-point kernels in this section, our peak performance is peak GFLOP/s as derived from architectural manuals. Peak bandwidth is peak DRAM bandwidth obtained via the Stream benchmark [125]. Technically, the Stream benchmark doesn't measure bandwidth, it measures iterations per second, then attempts to convert this performance into a bandwidth based on the compulsory number of cache misses on non-write allocate architectures. To avoid the superfluous allocate traffic, we modified Stream.

$$\text{Attainable Performance} = \min \begin{cases} \text{Peak Performance} \\ \text{Peak Bandwidth} \times \text{Arithmetic Intensity} \end{cases} \tag{4.1}$$

Figure 4.1 on the next page plots Equation 4.1 on a log-log scale resulting in a Roofline Model for memory-intensive floating-point kernels for the machines introduced in Chapter 3. The log-log scale is particularly useful as it ensures details are not lost despite data points across a range of more than two orders of magnitude. If the next generation of microprocessors doubles peak FLOP/s by doubling the number of cores, they can easily be drawn and visualized without a loss of detail on a log-log scale. Clearly, as arithmetic intensity increases, so does the bound on performance. However, at a critical arithmetic intensity, performance saturates at the peak performance level. We call this point the *Ridge Point*. In essence, this is the minimum arithmetic intensity required to achieve peak performance.

Figure 4.1 also shows the DRAM pin bandwidth. Stream bandwidth can be substantially lower than DRAM pin bandwidth for a variety of reasons. Most notably, the Intel Clovertown uses a front side bus (FSB) not only with less pin bandwidth than DRAM, but on which all coherency traffic is also present. The IBM Cell was designed so that bandwidth could only be fully exploited by the SPEs, but not by the PPE alone. Finally, the Stream benchmark is a suboptimal implementation for the Sun Victoria Falls.

Despite its simplistic nature, even this representation has significant value. Given a kernel and its arithmetic intensity (obtained either through inspection or simulation), one may query the Roofline Model and bound performance. Additionally, given the kernel's

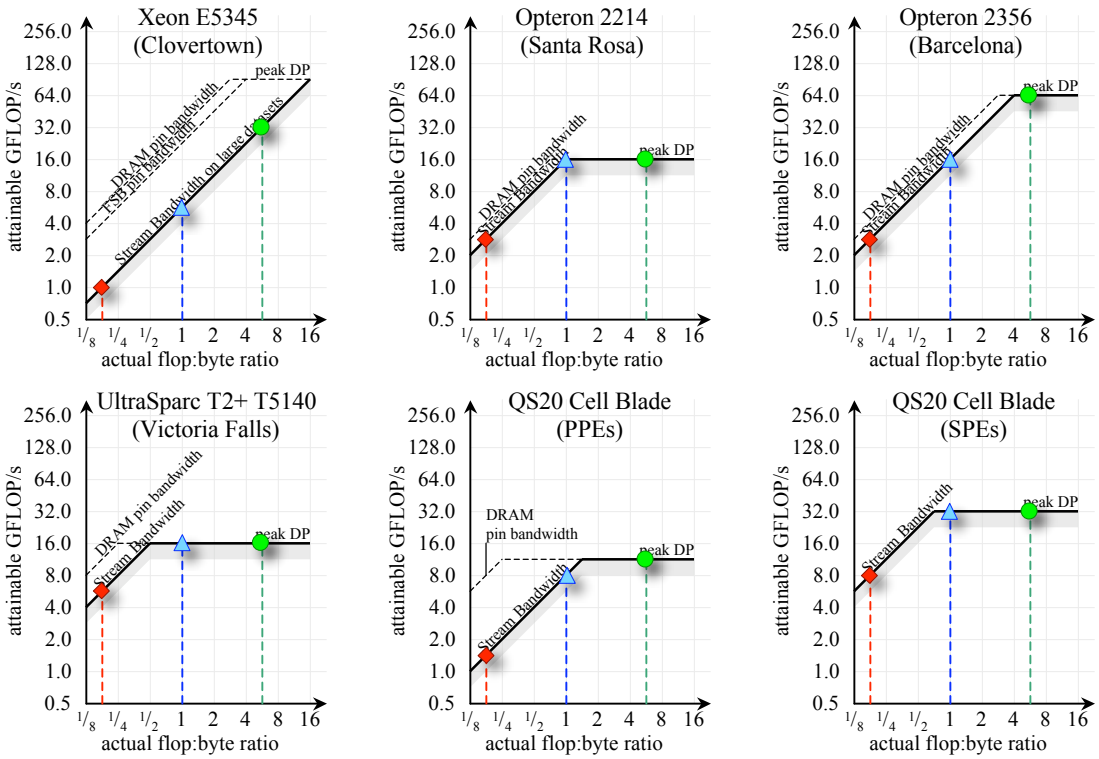


Figure 4.1: Naïve Roofline Models based on Stream bandwidth and peak double-precision FLOP/s. The arithmetic intensity for three generic kernels is overlaid. For clarity, we also note DRAM pin bandwidth. Note the log-log scale.

actual performance, one may quantifiably bound further performance gains. However, this formulation has certain, perhaps unrealistic, assumptions; most notably that a machine may always obtain either peak bandwidth or peak FLOPs. Additionally, although this simplistic version can quantify further potential performance gains, it fails to specify which optimizations must be implemented to achieve better performance. These issues will be addressed in Sections 4.4 through 4.7.

Imagine three nondescript kernels with arithmetic intensities of $\frac{1}{6}$, 1, and 6. Figure 4.1 overlays the arithmetic intensities of these kernels — represented by red diamonds, blue triangles, and green circles — onto the Roofline Models for the machines used in this work. On all machines, the first kernel is clearly memory-bound; one would expect performance to be proportional to the machine's Stream bandwidth. Despite having six times the arithmetic intensity, the kernel represented by the blue triangle would still be memory-bound on the Clovertown, Barcelona, and Cell PPEs. The final kernel, with an extremely high arithmetic intensity of 6, would still be memory-bound on the Clovertown. Although simple inspection of DRAM pin bandwidth would suggest a compute bound state, the fact that Stream bandwidth is substantially lower than DRAM pin bandwidth invalidates this hypothesis. Very few floating-point kernels would have arithmetic intensity higher than 6.

Clearly, this formulation assumes one can always achieve either peak bandwidth

or peak FLOP/s. No architecture/compiler combination can guarantee this on every code. Architectures have non-zero latencies for both instructions and memory access, as well as substantial memory and compute bandwidth that may only be achieved if all components are utilized. Finally, non-compulsory cache misses can substantially reduce the expected arithmetic intensity and thus limit performance. In the following four sections we address the implications for realistic bandwidth, computation, and cache parameters.

4.4 Expanding upon Communication

We define *memory bandwidth* to be the rate data can be transfered from DRAM to on-chip caches assuming stalls within a core do not impair performance. This bandwidth can be diminished if Little's Law is not satisified. In addition to the three components included in Little's Law — memory latency, raw memory bandwidth, concurrency in the memory subsystem — we also include the reduced bandwidth due to cache coherency. If concurrency isn't sufficiently exploited, sustained bandwidth will drop. We express deficiencies in these low-level concepts as deficiencies in software optimizations. We discretize deficiencies in software optimization as either none or total deficiency. A total deficiency will result in a roofline-like curve beneath the roofline proper. We call these curves *Bandwidth Ceilings*. This section discusses the impact of cache coherency, the components of memory bandwidth (parallelism), memory latency, and spatial locality. Finally, these concepts are unified into the concept of bandwidth ceilings.

4.4.1 Cache Coherency

All machines used in this work rely on a snoopy cache protocol to handle inter-chip cache coherency. Remember, the Clovertown is comprised of two multichip modules (MCM) each comprising two chips. All four chips are connected to the memory controller hub (MCH) via dual independent front side buses (FSB). This implies that only pairs of chips within an MCM may directly snoop transactions on their respective FSBs. To remedy this, a snoop filter was instantiated in the MCH and was discussed in Chapter 3. Nevertheless, the fact that FSB bandwidth must be partially retasked for coherency bandwidth implies that the Clovertown's bandwidth is dependent on the effectiveness of the snoop filter. Thus, on Clovertown, one would see substantial differences between the raw DRAM pin bandwidth, the raw FSB pin bandwidth, the sustained application bandwidth when the snoop filter is effective, and the sustained application bandwidth when the snoop filter is ineffective. The naïve Roofline Model is premised on only peak Stream bandwidth and ignores the others. An enhanced Roofline Model would be cognizant of all of them.

All other architectures in this work have a dedicated network for coherency and inter-socket transfers. Thus, for well-structured programs on these two socket SMPs, neither the latency nor limited bandwidth of the network are expected to impede performance. However, large snoopy SMPs will require an inordinate amount of inter-chip bandwidth to cover the resultant explosion in cache coherency traffic. Without such hardware, performance will be significantly below the raw DRAM pin bandwidth. The ideal cache coherency solution is a directory protocol, since it only intervenes when there is a coherency issue.

However, this would be overkill for the dual-socket SMPs used here. It should be noted that on many SMPs, including these, the coherency latency will be comparable if not larger than the local DRAM latency. In the end, one must be mindful that cache coherency may set the Stream bandwidth component of the Roofline far below the aggregate DRAM pin bandwidth.

4.4.2 DRAM Bandwidth

Raw DRAM pin bandwidth comes from a variety of sources that in simplest terms can be categorized into the product of parallelism and frequency. Parallelism itself can be subdivided into: bit-level parallelism (channel bus width $\times$ channels per controller), parallelism across multiple memory controllers on a chip, and parallelism across multiple chips. Thus, we calculate the raw DRAM pin bandwidth as the product of chips, controllers per chip, channels per controller, bits per channel, and frequency. Clearly, if hardware or software is deficient in fully exploiting any of these forms of raw bandwidth, then Stream bandwidth can be substantially diminished.

4.4.3 DRAM Latency

Access to DRAM requires latency of hundreds of core clock cycles. This latency arises from several sources outside the cache hierarchy. Not only is there substantial latency required to transfer a cache line, but there is also substantial overhead to open a DRAM page. These latencies may be hidden via pipelining, and the overheads can be amortized via high DRAM page locality. Moreover, the overhead may be hidden with concurrent accesses to different DRAM ranks. One should be mindful that to pipeline requests, significant parallelism must be expressed not only to the memory subsystem, but to each controller. Each controller can service multiple requests; easily the number of attached DRAM ranks. Thus, two nearly identical machines with the same number of memory controllers but different numbers of attached DIMMs may have different Roofline models simply because they have a different number of ranks. One cannot hide the overheads if there are not enough ranks.

Little's Law dictates that the concurrency that must be expressed to the memory subsystem to achieve peak bandwidth is the latency-bandwidth product. Assuming load and store instructions are rearranged to access independent cache lines, the requisite number of independent cache lines in flight is the latency-bandwidth product divided by the cache line size in bytes. For the machines used in this work, this might range from 64 to 200 independent and concurrent accesses; quite a challenge given the number of cores per SMP. However, these machines provide a number of strategies designed to express this vast *Memory-Level Parallelism* to the memory controllers. These paradigms include: extreme multithreading, out-of-order execution, software prefetching, hardware unit-stride stream prefetching, hardware strided stream prefetching, block DMA transfers, and DMA list transfers. Some of the machines can exploit more than one of these techniques to better express concurrency. If no combination of techniques can satisfy the latency-bandwidth product, the architecture can never achieve peak DRAM pin bandwidth.

Clearly, some of these techniques are dependent on hardware detecting certain access patterns and generating speculative requests. Under ideal conditions, these techniques require no software modifications, but may not be applicable for all applications. Other techniques like software prefetching or DMA may require significant software modification or compiler support without which performance will be substantially degraded. Finally, for single-program, multiple-data (SPMD) codes, multithreading is relatively easily exploited for a variety of applications without further software modifications. Given that consecutive loads are likely to touch the same cache line, limited memory-level parallelism is actually expressed to the memory controllers. Thus, the out-of-order capabilities of the superscalar architectures cannot be brought to bear, as they are more readily used for hiding access to the last level cache rather than main memory.

In the case where there is no inherent memory-level parallelism within an application (*e.g.* sequential pointer chasing), memory latency is completely exposed, and performance will suffer greatly. That is, one could only expect to load a 64 byte cache line every 200 ns or roughly 20 MB/s instead of the 10's of GB/s for unit-stride streaming accesses. Thus, it is key that software and hardware collaborate to express, discover, and exploit as much memory-level parallelism as possible.

4.4.4 Cache Line Spatial Locality

Spatial locality is defined as the use of consecutive data values in memory, specifically those within a cache line. Clearly, this is a high-level conceptualization. High bandwidth is a high utilization of the memory controller and DRAM capabilities and is oblivious to such high-level locality conceptualizations. It is imperative that the reader keeps these distinct concepts separate.

Kernels with unit-stride stream accesses have high spatial locality within a cache line; every byte loaded is used. However, there are other access patterns for which a small fraction of the data loaded will be used. This should be not viewed as a reduction of bandwidth commensurate with the fraction of data used, but rather a reduction in arithmetic intensity commensurate with the lack of sub-cache line spatial locality. Small stride memory access patterns might require every consecutive cache line, but might not require every byte. As such, they may achieve good bandwidth, but poor arithmetic intensity. Similarly, random access patterns might achieve neither good bandwidth due to a lack of exploited memory-level parallelism, nor good arithmetic intensity due to a lack of spatial locality.

4.4.5 Putting It Together: Bandwidth Ceilings

For single-program, multiple-data (SPMD) memory-intensive floating-point kernels, there are two big pitfalls that will reduce effective memory bandwidth: exposing memory latency and nonuniform utilization of memory controllers. In this section, we perform a sensitivity analysis in which we examine the impact on performance as we satisfy less and less memory concurrency (expose more and more memory latency) as well as not uniformly utilizing all memory controllers. We remove these "optimizations" one by one from a highly optimized implementation of the Stream benchmark in a prescribed order corresponding to the optimization's likely exploitation in a typical SPMD kernel using existing compiler

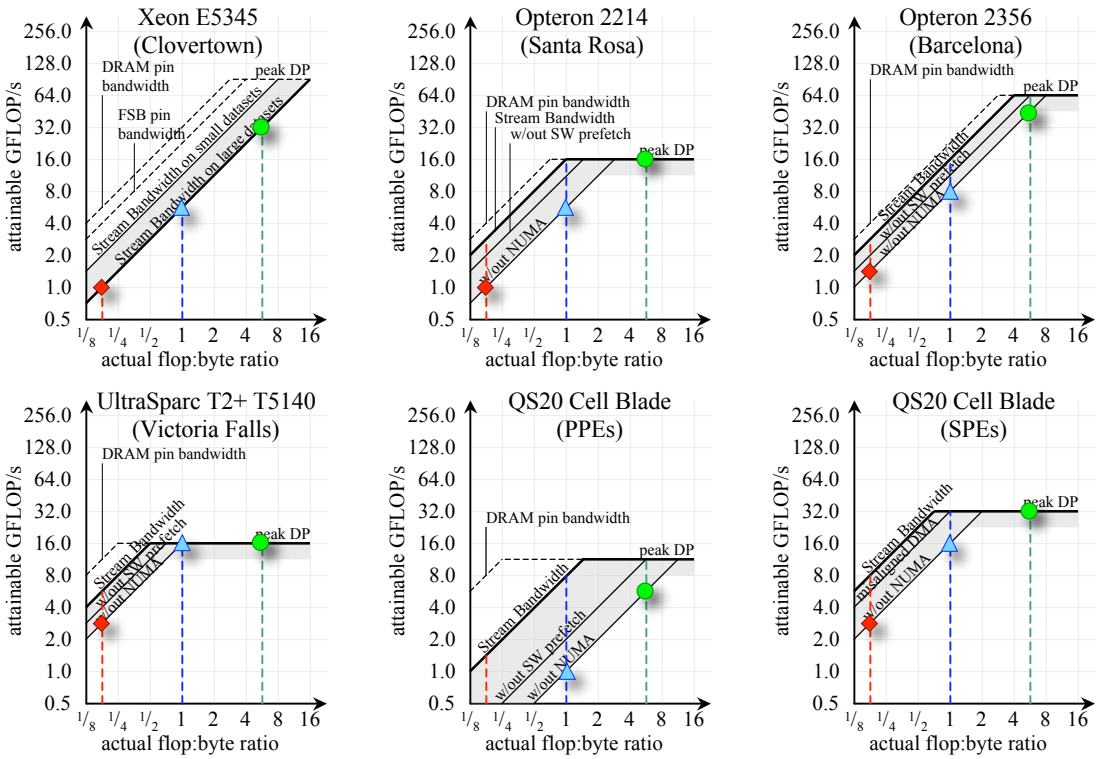


Figure 4.2: Roofline Model with bandwidth ceilings. The arithmetic intensity for three generic kernels is overlaid. For clarity, we also note DRAM pin bandwidth. Note the log-log scale.

technology. Those least likely exploited are removed first.

As previously discussed, on some architectures, it is impossible for the Stream benchmark to achieve a bandwidth comparable to the DRAM pin bandwidth. Thus, the Stream bandwidth diagonal component of the Roofline is below the DRAM pin bandwidth diagonal. Similarly, as we remove optimizations, new bandwidth diagonals will be formed below the idealized Stream bandwidth diagonal. We call these interior Roofline-like structures *Bandwidth Ceilings*. Figure 4.2 includes these ceilings within the Roofline Model for the six architectures. Without the corresponding optimizations, these ceilings constrain performance to be below them. We detail these ceilings below.

It is possible on the Clovertown for bandwidth to exceed the Stream bandwidth, but only when the snoop filter within the memory controller hub is effective in eliminating superfluous FSB snoop traffic. Thus, a bandwidth ceiling appears above the Roofline. When the dataset is sufficiently small, this ceiling provides a more reasonable bound on performance than the Stream benchmark ceiling. However, this is almost impossible to achieve in a real application as software optimizations can rarely reduce problem sizes sufficiently. As such, we place the effective snoop filter ceiling at the top.

On many architectures, software prefetching can express more concurrency than is normally expressed through scalar loads and stores, through hardware prefetchers, or

through out-of-order execution. However, their use requires either the compiler or the programmer to include these instructions in the program. This is an optimization rarely implemented, and thus the stream bandwidth without software prefetching is the next ceiling we draw. Figure 4.2 on the previous page clearly shows that the Clovertown is insensitive to a lack of software prefetching, while Victoria Falls, Barcelona, Santa Rosa and the Cell PPEs show progressively higher sensitivity to a lack of software prefetching. In fact, the Cell PPEs see only $\frac{1}{4}$ the bandwidth without software prefetching.

The Cell SPEs use DMA rather than software prefetching to express memory-level parallelism. However, if these DMAs are not aligned to 128-byte addresses in both DRAM and the local store, performance is diminished substantially. Although restructuring a program to exploit DMA is not a trivial task, restructuring a program to use 128 byte aligned DMAs can be an insurmountable challenge. As such, on Cell, we place the DMA alignment ceiling immediately below the Roofline as it is the most challenging for either programmers, middleware, or compilers to exploit.

All machines used in this work are dual-socket SMPs. On all but Clovertown, the memory controllers are distributed among the chips. Peak bandwidth is achieved when memory transactions are generated by the same socket as their target memory controllers. Moreover, the total memory traffic must be uniformly distributed among the memory controllers both within a socket and across sockets. Failing to satisfy the former will expose the limited bandwidth, high latency inter-socket interconnect, while failing to satisfy the latter will result in idle cycles on one or more of the controllers. Any idle bus cycle detracts from peak bandwidth. We describe these optimizations designed to modify a kernel to adhere to these requirements collectively as "NUMA optimizations." Although occasionally challenging, for SPMD kernels these are often easier to implement than many of the other bandwidth-oriented optimizations. As such, we place the ceiling marking performance without these optimizations at the bottom. On a dual-socket NUMA SMP, these ceilings should be about half the next ceiling's bandwidth. If the no-NUMA ceiling is substantially below this, then the inter-socket network is likely a major performance impediment. If the no-NUMA ceiling is significantly better than half of the Roofline, then the machine is likely over provisioned with bandwidth. Obviously, Clovertown is not sensitive to a lack of NUMA optimizations, while the other architectures may see about a factor of two.

Note, on many NUMA machines, it is possible to configure the machine for either NUMA interleaving or UMA interleaving in the BIOS. In the latter, physical addresses are interleaved between sockets on cache line or column granularities rather than gigabyte-size boundaries. In such a case, one should generate a separate Roofline model with a new Stream bandwidth-derived Roofline that may be substantially lower. On the UMA Roofline, there will not be a NUMA ceiling. Throughout this work, we leave the NUMA BIOS option enabled.

Without optimizations, performance is constrained by the ceilings. As such, a lack of memory optimizations moves the effective ridge point to the right. Thus, a much higher arithmetic intensity is required to achieve peak FLOP/s. If one were to inspect only the naïve Roofline Model or worse, just machine balance, they might erroneously conclude a kernel's performance is limited by the core's performance when in fact, the delivered memory bandwidth is the bottleneck.

Consider our three nondescript kernels with arithmetic intensities of $\frac{1}{6}$, 1, and 6 shown in Figure 4.2 on page 55. Without software prefetching or NUMA optimizations, kernels to the left of the original ridge point will see the largest drop in performance. This is most pronounced on the Cell PPEs, where the kernel with an arithmetic intensity of $\frac{1}{6}$ drops off the scale. As arithmetic intensity increases from the original ridge point to the new ridge point, kernels will see an ever smaller loss in performance until they pass the new ridge point and see no loss in performance. As the Clovertown is not sensitive to these optimizations, performance isn't degraded.

4.5 Expanding upon Computation

In Section 4.4 we showed that without substantial software optimization, bandwidth could drop substantially. In this section, we perform a similar analysis with respect to computation. We define *in-core performance* to be the performance when data is in registers and no memory accesses are required; not even to the L1. We assume there is sufficient capacity. There are two primary limiters to in-core performance. First, Little's Law applies just as much to the functional units as the memory subsystem. Second, instruction issue bandwidth is finite. Every non-floating-point instruction required by the code is potentially one less cycle that can be used to issue a floating-point instruction. We discretize the parameters that determine if these functions are satisfied. This also results in a series of roofline-like curves beneath the roofline proper. We call these *In-core Performance Ceilings*. In Section 4.5.1 we discuss the impact of not satisfying an in-core version of Little's Law on performance, and in Section 4.5.2 we discuss the impact of non-floating-point instructions on in-core floating-point performance. We combine these and define the in-core performance ceilings in Section 4.5.3.

4.5.1 In-Core Parallelism

In its general form, Little's Law states the concurrency required to achieve peak performance is equal to the latency-bandwidth product. When dealing with in-core performance, the "concurrency" is the number of independent operations in flight — deemed *in-core parallelism*. The "bandwidth" is the number of operations that may be completed per cycle, and the "latency" is the instruction latency measured in cycles. We first examine the different forms of parallelism, *i.e.* "concurrency."

Each core incorporates one or more floating-point functional units. Each of these units may be general-purpose or restricted functionality. For purposes of this work, we examine four types of floating-point functional units: general-purpose add or multiply, multiply-only, add-only, and general-purpose fused multiply-add (FMA). The FMA datapath may execute multiplies, adds, or under ideal conditions a fused multiply-add ($A \times B + C$). Obviously, if an architecture uses an add-only datapath, for completeness it must also include a multiply-only datapath. Typically, floating-point divides are executed in a multiply or fused multiply-add datapath in an non-pipelined iterative fashion. Table 4.2 on the next page lists both the number of datapaths and their latencies for each type within each core of each architecture. Although machines like Clovertown and Opteron are superscalar, they

Functional Units by type	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	QS20 Cell Blade (SPE)
MUL or ADD	—	—	—	1 × 6c	—	—
MUL-only	1 × 3c	1 × 4c	1 × 4c	—	—	—
ADD-only	1 × 5c	1 × 4c	1 × 4c	—	—	—
FMA	—	—	—	—	1 × 6c	1 × 13c†
SIMD register width	128-bit	128-bit	128-bit	—	—	128-bit
datapath width	128-bit	64-bit	64-bit	64-bit	64-bit	128-bit
requisite independent FLOPs in flight (per thread)	16 (16)	8 (8)	16 (16)	6 (0.75)	12 (6)	8 (8)

Table 4.2: Number of functional units × latency by type by architecture. The resultant latency-bandwidth product is shown. This is the number of in flight FLOPs that must be maintained. †Every double-precision instruction stalls subsequent instruction issues for 6 cycles. Thus, back-to-back double-precision instructions have stalled execution by more than the functional unit latency.

only have one add-only functional unit and one multiply-only functional unit. When one or more of these datapaths are included in an architecture, they express either general-purpose or specialized instruction-level parallelism. No machine in our work exploits the former as none have multiple general-purpose functional units. Note, we distinguish balance between multiples and adds and exploitation of fused multiply-add from the conventional instruction-level parallelism by labeling them individually.

Every instruction has a nonzero latency associated with it. Hardware interlocks ensure dependent instructions are not issued down the pipeline until their operands are available or can be forwarded. In order to issue instructions at the peak rate, there must be at least as many independent instructions as functional unit latency. There are two means to accomplish this: instruction-level parallelism within a thread or parallelism across threads. The latter is only applicable on hardware multithreaded architectures; that is, those that have multiple hardware contexts per core (or per shared FPU). Typical floating-point instruction latencies are 4 to 7 cycles. On some architectures, the requisite parallelism may be difficult to come by within a single thread, but on Niagara with 8 thread contexts per core, its easy to cover its 6 cycle floating-point latency. Cell is somewhat unique as the double-precision pipeline is significantly longer than the forwarding network. To ensure correct behavior, all subsequent instructions are stalled by 6 additional cycles. As such, only two independent floating-point instructions are required to cover the 13 cycle latency.

There is another way to improve in-core performance: data-level parallelism. Vector and SIMD architectures may encapsulate multiple floating-point operations into the same instruction. Each operation is independent of the others. Subtly, the maximum throughput is neither the maximum vector length nor SIMD register width as specified in the ISA, but rather the number of lanes implemented in each microarchitecture. The x86 and Cell SPE instruction set architectures define a SIMD register width of 128 bits. In double-precision, this is a pair of doubles. Although, machines like the Clovertown, Barcelona, and the Cell SPEs have two double-precision lanes per functional unit, the Santa Rosa Opteron has only one. As a result, on the Santa Rosa Opteron, the datapath is

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	QS20 Cell Blade (SPE)
Total instruction issue bandwidth per core	4	3	4	2	2	2
FP bandwidth required to achieve peak FLOP/s	2	1	2	1	1	2

Table 4.3: Instruction issue bandwidth per core. In addition, we show the sustained floating-point issue bandwidth required to achieve peak FLOP/s under ideal conditions.

occupied for two cycles for a SIMD operation. Thus, executing a SIMD operation on this machine reduces the requisite instruction-level parallelism. PowerPC, and SPARC don't implement double-precision SIMD instructions.

Combining these different forms of parallelism produces both the "bandwidth" and "latency" terms in Little's Law. However, the fact these forms cannot be interchanged results in a heterogeneous "concurrency" term. That is, only so much concurrency is required from each of several styles. Too much of one form cannot make up for a deficiency in the other. Using Table 4.2 on the preceding page, one may calculate all the requisite forms of parallelism by taking the sum over the products of the number of pipelines, their depth, and their SIMD widths. For example, by inspection of Table 4.2, Clovertown requires 3 SIMD multiplies and 5 SIMD adds available for execution every cycle to achieve peak performance. Thus, it must keep 16 double-precision operations in flight per thread to achieve peak performance (last row). Without SIMD, performance would be cut in half, regardless of how much instruction-level parallelism remained present. Note: although an FMA may be encoded as one instruction, it is counted here as two floating-point operations. Clearly, to achieve peak performance, most architectures are required to exploit many forms of parallelism and keep many operations in flight.

4.5.2 Instruction Mix

Table 4.3 shows both the maximum instruction issue bandwidth per cycle and the floating-point instruction issue bandwidth required to achieve peak FLOP/s. Although some architectures have substantial slack, as measured by the difference in available and requisite bandwidth, others don't. As the ratio of non-floating-point instructions to floating-point instructions increases, a tipping point is reached, and the floating-point units are starved of instructions. Assuming memory bandwidth is not an issue and one can satisfy all forms of in-core parallelism, Equation 4.2 estimates the impact of progressively less instruction bandwidth available for floating-point instructions. As the floating-point fraction of instruction decreases, eventually the floating-point units become starved for instructions, and performance suffers.

$$\text{Fraction of Peak FLOP/s} = \min \begin{cases} \text{FP Fraction} \times \dfrac{\text{Total Issue Bandwidth}}{\text{FP Issue Bandwidth}} \\ 1 \end{cases} \tag{4.2}$$

On Cell, where double-precision floating-point instructions consume 7 issue cycles, the equation is somewhat different:

$$\text{Fraction of Peak FLOP/s} = \frac{14 \times \text{FP Fraction}}{1 + 13 \times \text{FP Fraction}} \tag{4.3}$$

Clearly, on Cell, 100% of the instruction mix must be floating-point to achieve peak performance. However, as the floating-point fraction of the instruction mix decreases, performance drops very slowly. As such, one might conclude that Cell is less sensitive to the instruction mix than other architectures.

4.5.3 Putting It Together: In-Core Ceilings

We have discussed the two principal factors that constrain in-core performance: satisfying all forms of in-core parallelism, and ensuring the non-floating-point instructions don't suck up all the issue bandwidth. In this section, we perform a straightforward sensitivity analysis in which we examine the impact on performance as we satisfy less and less of the in-core parallelism or increase the non-floating-point instructions. These Roofline Models presume load-balanced, single-program, multiple-data (SPMD) memory-intensive floating-point kernels. Thus, multithreading, multicore, multisocket parallelism is assumed to be load balanced.

First, let us consider the impact on performance as an application fails to express sufficient instruction, data, and functional-unit parallelism. We remove these in a prescribed order typical of many codes. For many kernels, it is impossible to achieve balance between multiplies and adds or always exploit fused multiply-adds. As such, these are the least likely forms of parallelism to be exploited. Second, many compilers are challenged by the extremely rigid nature of many SIMD implementations. As such, it is very likely that despite the presence of data-level parallelism within an application, neither the compiler nor the programmer will exploit SIMD. Finally, we believe that some instruction-level parallelism inherent in the kernel can readily be discovered and exploited by the compiler. Thus, ILP is the most readily exploited form of in-core parallelism.

Every time we remove one of these forms of parallelism, performance is diminished. As a result, we form a new *in-core Performance Ceiling* below the Roofline. These ceilings act to constrain or limit how high performance can reach. As an impenetrable barrier, performance cannot exceed a ceiling until the underlying lack of parallelism is expressed and exploited. Figure 4.3 on the next page presents an expanded Roofline Model where all the in-core parallelism ceilings are shown. The ceilings are derived from architectural optimization manuals rather than benchmarks. Clearly, Victoria Falls' chip multi-threading is very effective in hiding instruction level parallelism. As Victoria Falls requires neither data-level nor functional unit parallelism, there are no additional ceilings. However, other architectures are heavily dependent on instruction-level parallelism being inherent in the code, discovered by the compiler and exploited by the architecture.

Alternately, an architecture may be much more sensitive to the floating-point fraction of the dynamic instruction mix. Figure 4.4 on page 62 clearly shows that architectures like Victoria Falls are far more sensitive to the instruction mix than the degree of in-core

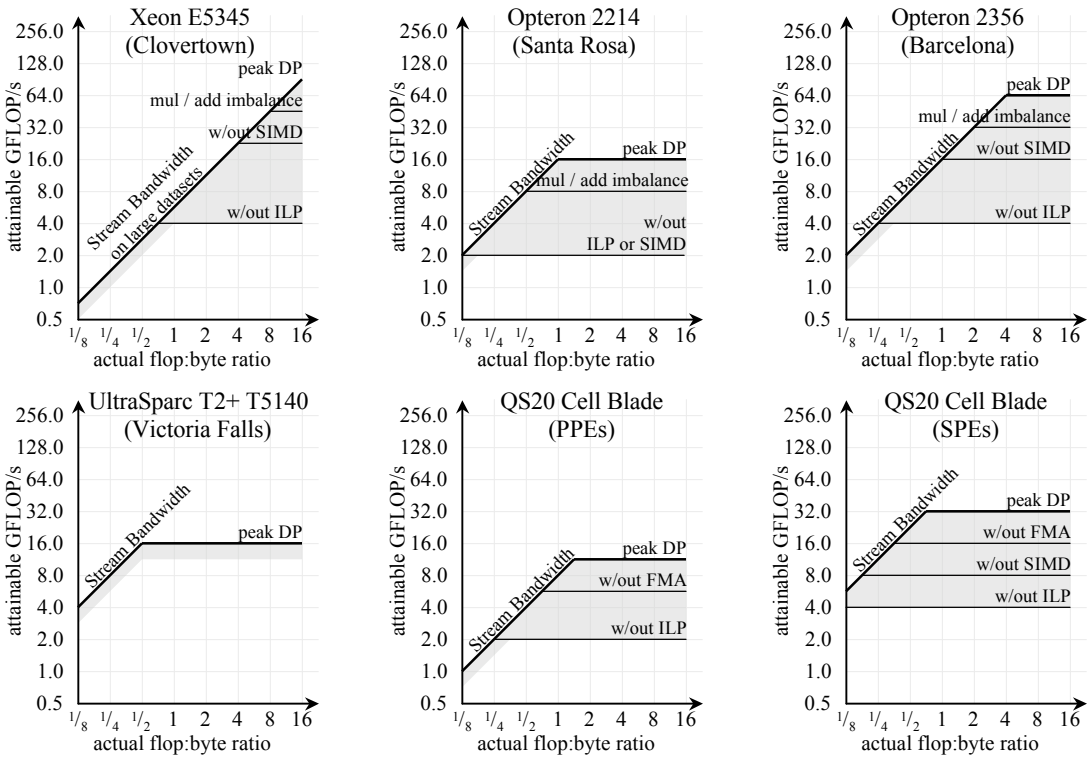


Figure 4.3: Adding in-core performance ceilings to the Roofline Model. Note the log-log scale.

parallelism expressed within a thread. Conversely, architectures like the Cell SPEs are remarkably insensitive to the instruction mix showing only a factor of two loss in performance when only 1 in 16 instructions is floating-point. Most of the codes we deal with should have much higher than a 10% floating-point fraction. As a result, the typical performance loss should be much less than 4×.

Combining the previous two figures, it is clear that each architecture has an *Achilles' Heel* to performance: either an architecture's sensitivity to a lack of in-core parallelism or its sensitivity to the instruction mix. Figure 4.5 on page 63 shows the appropriate Roofline Model in-core ceilings for each architecture. Clearly, Victoria Falls seems to be more sensitive to the instruction mix balance than to a lack of in-core parallelism on SPMD codes. Conversely, satisfying the requisite degree of in-core parallelism is the preeminent challenge on the single-threaded architectures.

Returning to our three nondescript example kernels, it is clear that the red diamond kernel on the left requires relatively little in-core parallelism to achieve peak performance regardless of architecture. In fact, a complete lack of in-core parallelism and poor instruction mix may only impair performance by a factor of two on the Cell SPEs. As arithmetic intensity increases, performance is much more dependent on there being sufficient in-core parallelism. For the kernel with an arithmetic intensity of 6, most machines must rely on full ILP and quite possibly both DLP and FMA or balance between multi-

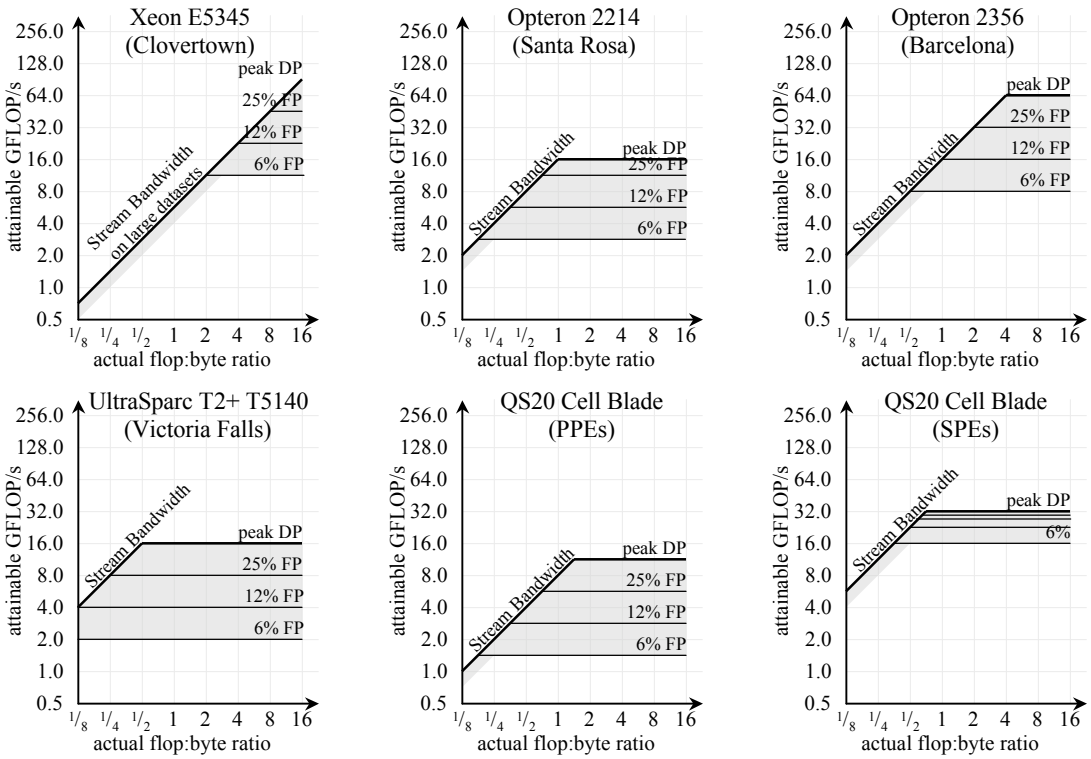


Figure 4.4: Alternately adding instruction mix ceilings to the Roofline Model. Note the log-log scale.

plies and adds. Interestingly, without in-core parallelism, all machines deliver comparable performance — an artifact of similar process technologies, core counts, and power envelopes.

Across machines, if in-core parallelism is not expressed, then the ridge point is reduced so substantially that virtually any kernel with an arithmetic intensity greater than $\frac{1}{4}$ would be compute bound at a substantially reduced performance. If one were to use only the naïve Roofline Model, one might have erroneously concluded a kernel was memory-bound, when in fact it was bound by not satisfying the in-core version of Little's Law. Examination of a Roofline Model including the in-core ceilings should make this mistake obvious.

4.6 Expanding upon Locality

Thus far, we have assumed that arithmetic intensity is solely a function of the kernel. However, this presumes an infinite, fully associative cache — clearly unrealistic. In this section, we lay the framework as to how one would incorporate cache topology into the Roofline Model. The result will be that arithmetic intensity is unique to the combination of kernel and architecture. It may be substantially less than the compulsory arithmetic intensity. This may result in a degradation in performance depending on the

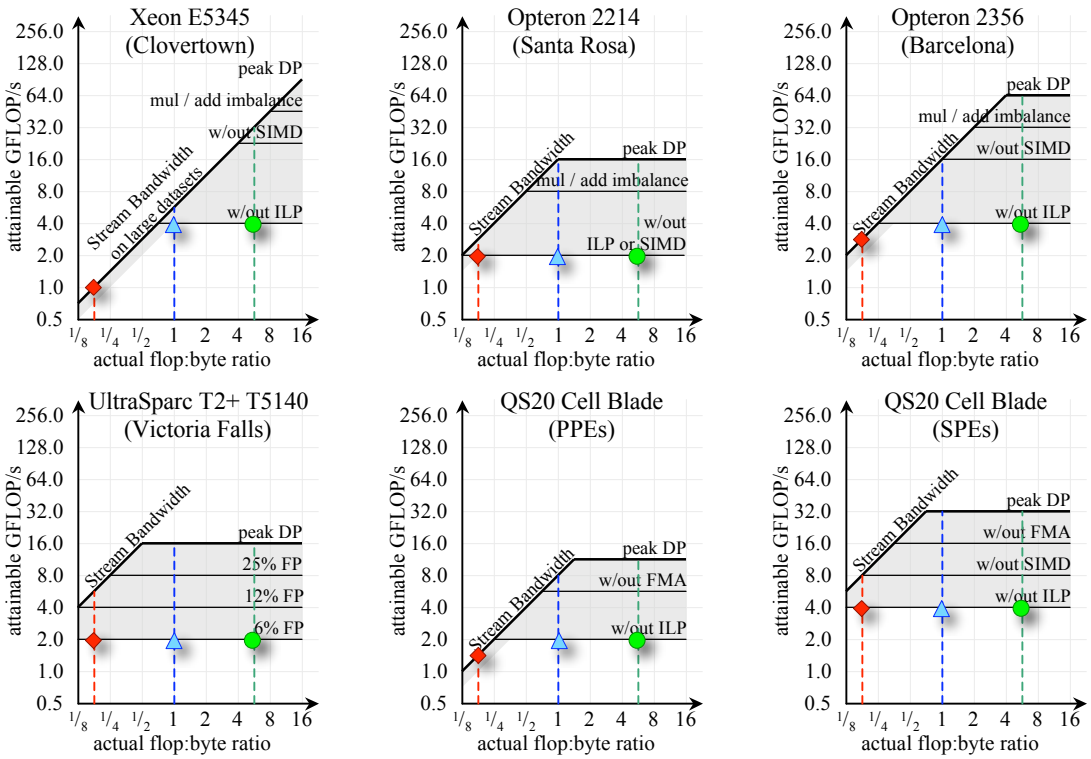


Figure 4.5: Roofline model showing in-core performance ceilings. Each architecture has its own *Achilles' Heel*: either its ability to satisfy in-core parallelism or its sensitivity to the instruction mix. Note the log-log scale.

balance between the resultant arithmetic intensity and the ridge point.

4.6.1 The Three C's of Caches

Cache misses can be classified into three basic types [69]: compulsory, capacity, or conflict misses. Thus far, we have assumed that the only cache misses produced by the execution of a kernel on a machine were compulsory misses. This simplification allowed us to calculate the arithmetic intensity of a kernel once and apply it to the Roofline Models for any number of architectures. In practice, this is only applicable for simple streaming kernels with working sets smaller than any cache in existence today. A better solution would be to include the effects of limited cache capacity and associativity when calculating the total number of cache misses. This would allow us to calculate a *true arithmetic intensity* rather than just a compulsory arithmetic intensity. Unfortunately, without a cache simulator, one can only determine the total number of cache misses using performance counters: see Section 4.9.2.

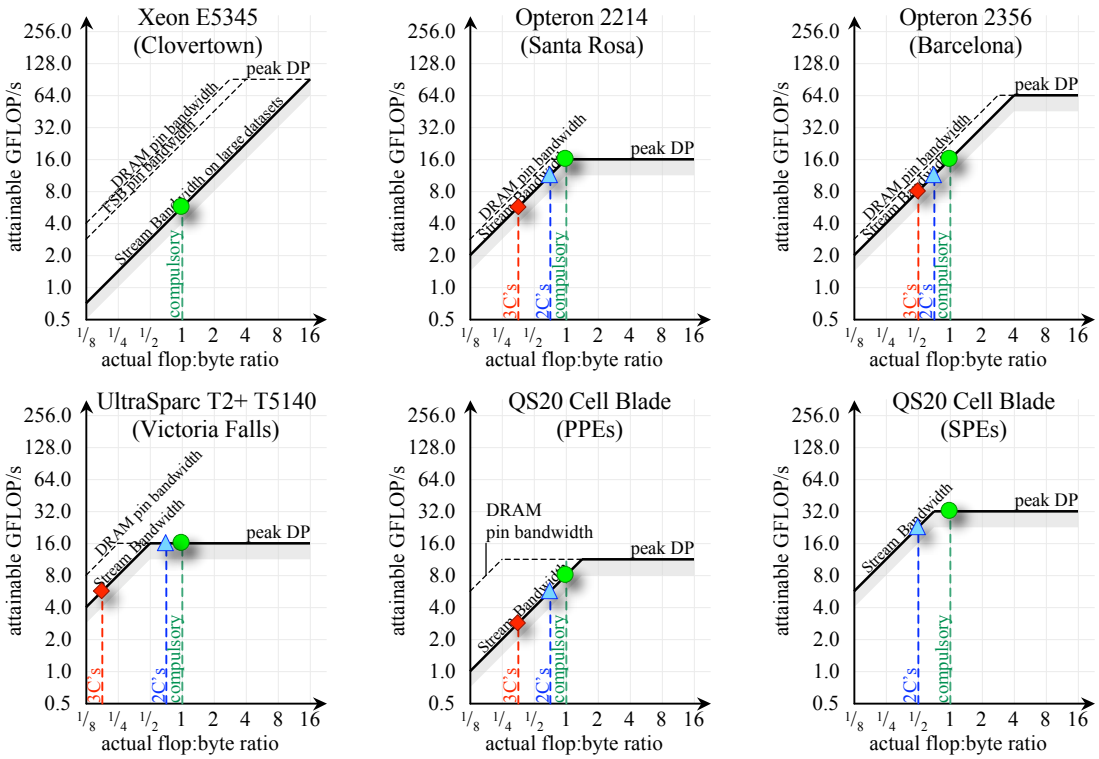


Figure 4.6: Impact of cache organization on arithmetic intensity. Each architecture has might be more sensitive than others to capacity or conflict misses. 2C's implies either compulsory and capacity or compulsory and conflict misses are still present. 3C's implies conflict, capacity, and compulsory are present. Note the log-log scale.

4.6.2 Putting It Together: Arithmetic Intensity Walls

In much the same way the realities of bandwidth and in-core performance act to constrain performance through the creation of ceilings interior to the Roofline, the realities of caches act to constrain arithmetic intensity. Unlike ceilings, these structures are vertical barriers through which arithmetic intensity may not pass without optimization. As such, we describe them as *Arithmetic Intensity Walls*. As with the ceilings, the locations of these walls are unique to each architecture. However, unlike ceilings, their locations are also dependent on the kernel's implementation.

Figure 4.6 overlays not only the compulsory arithmetic intensity, but also the resultant arithmetic intensities when cache capacity and conflict misses are included for an arbitrary kernel. Generally, Clovertown's large high associativity caches make it relatively insensitive to moderate cache working sets and dramatically reduce the probabilities of conflicts. The moderate cache capacities and associativities of the Opterons and Cell PPEs can result in a substantial difference between the compulsory and true arithmetic intensities. Victoria Falls is likely the most sensitive to cache capacity and conflict misses due to the very small L2 working set sizes and associativities per thread. The Cell SPEs use a local

store architecture. As such, the equivalent of capacity misses must be handled through software, but there are no conflict misses.

A loss of arithmetic intensity will have a clear impact on performance if the resultant arithmetic intensity is left of the ridge point. In such cases, performance may drop by a factor of four or more. Subtly, if the latency resulting from said misses isn't covered via the previously discussed latency hiding techniques, then bandwidth will suffer as well.

As with ceilings, the order of the arithmetic intensity walls is loosely defined. Generally they should be ordered from those easiest to compensate for to those one can never surpass. Clearly, compulsory misses should be the rightmost. Additionally, the leftmost wall is the worst case where no effort has been made to address conflict or capacity misses. It is somewhat kernel dependent as to whether conflict or capacity misses are easier to reduce. We discuss this further in Section 4.7.3.

4.7 Putting It Together: The Roofline Model

In Sections 4.3 through 4.6 we developed the major components for the Roofline Model: the Roofline, the in-core ceilings, the bandwidth ceilings, and the locality walls. In this section, we put the individual components into a single unified framework and then discuss the interplay between architecture and optimization.

4.7.1 Computation, Communication, and Locality

The Roofline Model presumes an idealized form where either communication or computation can be perfectly hidden by the other. As such we may independently incorporate the ceilings from both communication and computation. Equation 4.4 incorporates both types of ceilings. Performance$_i$ denotes in-core performance exploiting architectural paradigms 1 through i. Performance(FP fraction) denotes in-core performance as a function of the FP fraction of the instruction mix assuming one has exploited all architectural paradigms. Bandwidth$_j$ denotes attained memory bandwidth exploiting optimizations 1 through j. Finally, the true arithmetic intensity includes all cache misses, not just compulsory.

$$\text{Attainable Performance}_{ij} = \min \begin{cases} \text{Performance}_i \\ \text{Performance(FP fraction)} \\ \text{Bandwidth}_j \times \text{True Arithmetic Intensity} \end{cases} \tag{4.4}$$

Figure 4.7 on the following page plots Equation 4.4 for the six architectures used in this work. We have chosen to include in-core performance as a function of instruction mix for only Victoria Falls. Such a formulation can be useful in not only qualifying (good, bad) performance and quantifying how much further performance can be had, but also noting whether bandwidth or in-core optimizations should be applied. Simply put, given an arithmetic intensity, or range of intensities, one can examine the Roofline Models and not only determine the expected performance, but also the programming effort required to deliver that level of performance.

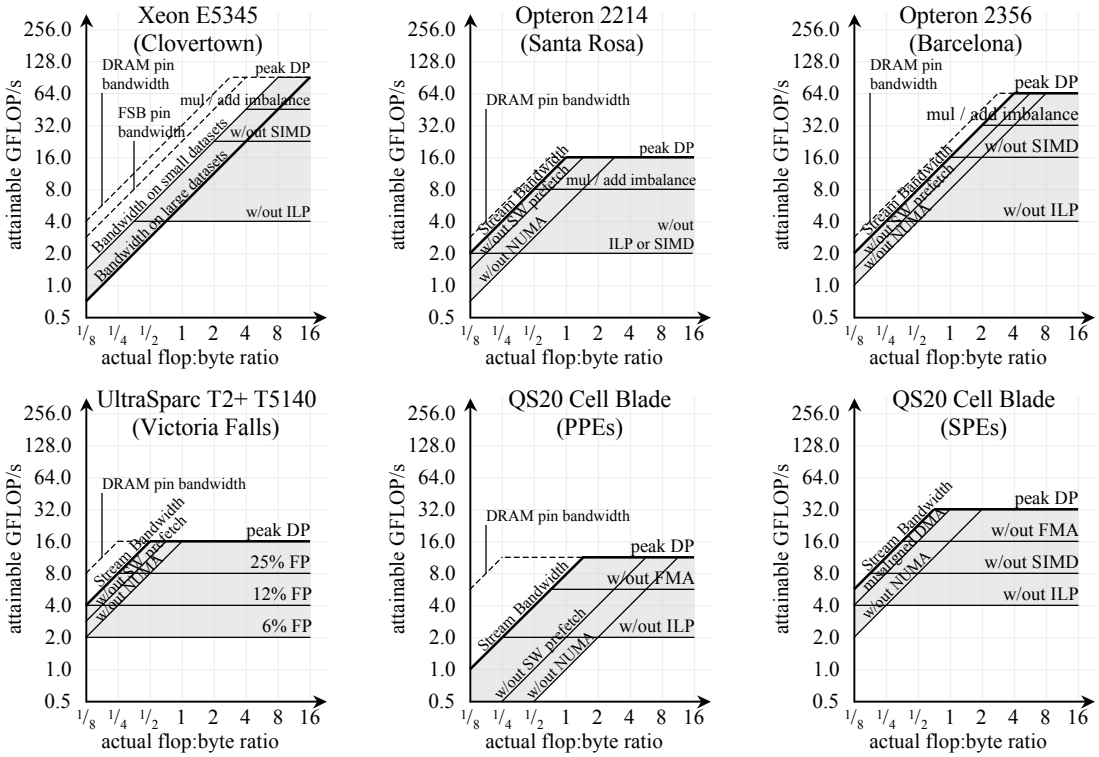


Figure 4.7: Complete Roofline Model for memory-intensive floating-point kernels. Note both bandwidth and in-core performance ceilings are shown. Note the log-log scale.

4.7.2 Qualitative Assessment of the Roofline Model

Examining the Roofline Models in Figure 4.7, we may make several qualitative statements about the architectures used. Such statement should act as general guidelines to the reader.

First, Stream bandwidth far short of DRAM pin bandwidth is a harbinger of continuing poor performance for that architecture. Clearly, this is the case of the Cell PPEs. It suggests the architecture lacks the concurrency to cope with either the latency to DRAM or the latency for the cache coherency protocol. Of course, this assumes sufficient DIMMs (and thus ranks) are installed in the machine to amortize or hide the overhead for DRAM page accesses.

Second, the ridge point marks the minimum arithmetic intensity (locality) required to achieve peak performance. Architectures with ridge points far to the right will be hard pressed to achieve peak performance on any kernel. Conversely, architectures with ridge points to the left have so much DRAM bandwidth that, all things being equal, they should achieve a high fraction of peak performance on a much wider set of kernels. Architects must be mindful of the benefit (sustained performance) and the cost (both increased unit cost and power) required to move a ridge point to the left.

Third, let us define productivity as the software optimization work required to

reach the Roofline. Notice that most architectures are relatively insensitive to a lack of memory optimizations (less than a factor of three). However, the simplest, least parallel architecture — the Cell PPEs — are incredibly sensitive to bandwidth optimizations. When it comes to in-core performance, Victoria Falls is somewhat insensitive to a lack of in-core per thread parallelism as multithreading can compensate for it. Generally, one could interpret the thickness and more specifically the number of ceilings below the Roofline as an assessment of the requisite software, middleware, and compiler complexity. The lower the complexity, the more productive one is likely to be. Subtly, some architectures may be far more sensitive to cache capacities and associativities. As such, they may see a substantial swing in true arithmetic intensity. Such is the case on Victoria Falls. If one can compensate for the cache characteristics, then it is a productive architecture.

Fourth, the Roofline Model is the ideal form in which either computation or communication can be completely hidden by the other. An architecture's departure from this form is a commentary on its inability to overlap communication and computation. In practice, one would expect the Roofline to be smoothed on some architectures. In the worst case, no overlap, performance at the ridge point would be only half of the Roofline — see Section 4.8.8.

Finally, it is interesting that the performance curve defined by the lowest ceilings is remarkably similar across most architectures. Thus, without optimization one expects all architectures to deliver comparable performance. In many ways the lowest ceilings are the latency limit (*i.e.* solely thread-level parallelism). This should be no surprise since latency is dictated by technology, where peak performance — in the form of parallelism or deep pipelining — is an architectural decision. As all machines used here are either based on a 65nm or 90nm technology, they should all deliver similar sequential performance.

Figure 4.8 on the next page shows the interplay between architects and programmers. Over the last couple decades, architects have exploited deep pipelining and every imaginable form of parallelism to provide ever higher levels of potential performance — Figure 4.8(a). This approach has pushed the Roofline up by more than a order of magnitude by inserting additional ceilings. In addition, this has moved the ridge point to the right. In the coming decade, as manycore becomes the battle cry, we expect at least another order of magnitude increase. However, this performance increase has not come without a price. In a Faustian bargain, performance is now contingent upon programmers, middleware, and compilers completely exploiting all these forms of parallelism. Figure 4.8(b) shows that performance may come crashing down without their collective help.

4.7.3 Interaction with Software Optimization

Figure 4.8(b) suggests that without the appropriate optimizations, the ceilings will constrain performance into a narrow region. In the context of single-program, multiple-data (SPMD) memory-intensive floating-point kernels, we may classify an optimization into one of three categories: maximizing in-core performance, maximizing memory bandwidth, and minimizing memory traffic. Figure 4.9 on page 69 shows that when these optimizations are applied, they remove ceilings as constraints to performance. In doing so, performance may increase substantially. In Section 4.8, we discuss other optimizations as we extend the applicability of the Roofline Model.

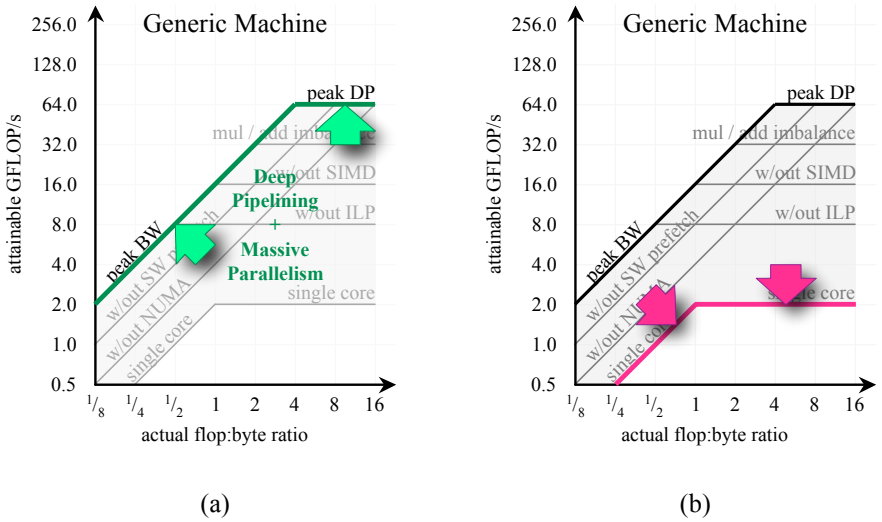


Figure 4.8: Interplay between architecture and optimization. (a) Architects have exploited deep pipelining and massive parallelism to dramatically raise the Roofline. (b) Without commensurate optimization, performance is significantly constrained. Note the log-log scale.

Optimizations categorized as maximizing in-core performance include optimizations such as software pipelining, unroll and jam, reordering, branchless or predicated implementations, or explicit SIMDization. We discuss our use of these optimizations in Sections 6.3 in Chapter 6 and 8.4 in Chapter 8. Broadly speaking, some of these optimizations express more fine-grained instruction- or data-level parallelism, while others amortize loop overhead. In doing so, they maximize the floating-point fraction of the instruction mix. To be fair, one might also include optimizations designed to mitigate L2 cache latency, or L1 associativity and bank conflicts. Nominally these don't generate additional main memory traffic, but may impair in-core performance. Multiply/add balance can be supremely difficult to achieve. However, on multiply-heavy codes, one might consider additions instead of small integer multiplications. For example, instead of $y = 2 * x$, one might implement $y = x + x$. This is appropriate until balance is reached, or instruction issue bandwidth is saturated.

Once again, it is imperative to distinguish improving memory bandwidth from reducing memory traffic. There are a number of code generation and data layout techniques designed to maximize memory bandwidth. The most obvious optimization is to change the data layout in main memory by using various affinity routines to ensure that the data and processes tasked with accessing them are collocated on the same socket. In addition, either the compiler or the programmer may insert software prefetch instructions. On the Cell processor, bandwidth can be significantly increased when both the local store and DRAM addresses for a DMA are aligned to 128 byte boundaries. There are also a few subtle optimizations that can enhance memory bandwidth. These include restructuring

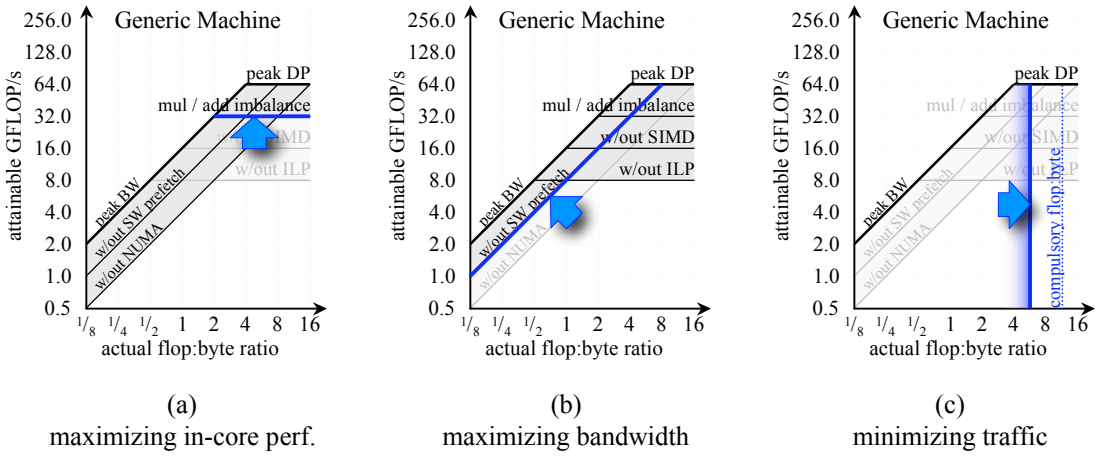


Figure 4.9: How different types of optimizations remove specific ceilings constraining performance. (a) expression of in-core parallelism eliminates in-core performance ceilings as constraints to performance. (b) removal of bandwidth ceilings as constraints to performance. (c) minimizing memory traffic maximizes arithmetic intensity. When bandwidth limited, this improves performance. Note the log-log scale.

loops so that there are a small, finite number of long memory streams. This can be critical, as hardware prefetchers require page-sized granularities of sequential locality to effectively hide memory latency. In addition, they can only track a few (less than a dozen) streams. They fail dramatically when the number of streams exceeds the hardware's capability to track. Subtly, unroll and jam can actually express more memory level parallelism as it results in accesses to disjoint cache lines.

To achieve peak performance on bandwidth-limited code, one must maximize memory bandwidth and minimize the total memory traffic. Minimizing memory traffic will improve arithmetic intensity. In order to improve arithmetic intensity, one must address the 3C's of caches [69]. To that end, there are several standard software optimizations designed to fix or minimize each type of cache miss. To ensure fast hit times, caches may have low associativities. Thus, for problems that access many disjoint arrays, or implement higher dimensional access patterns on power of two problem sizes, caches are highly sensitive to cache conflict misses. Array padding is the standard technique designed to address conflict misses. Essentially, this optimization can convert a power of two problem size to a non-power of two problem size. Alternately, in the context of stencils, this can spread streaming accesses throughout the cache's sets so that the full cache capacity can be exploited.

Capacity misses often arise when, in a standard implementation, there are a large number of intervening accesses before data is reused (*i.e.* a large working set). If the number of intervening accesses exceeds the cache capacity, locality cannot be exploited. Loop restructuring, also known as cache blocking, can reduce the capacity requirements to the point where the requisite cache capacity is less than the cache capacity of the underlying architecture. Such optimizations can dramatically increase arithmetic intensity. The goal of such optimizations is to move the true arithmetic intensity closer and closer to the

compulsory arithmetic intensity.

One might think that compulsory misses cannot be helped. There are several fallacies with such an assumption. First, write-allocate architectures load the cache line in question on a write-miss. Under the conditions where the entire line is destined to be overwritten, this cache line load is superfluous. In the optimal circumstance, the use of a cache bypass or similar instruction could cut the memory traffic in half. In doing so, arithmetic intensity could be doubled. As writes become the minority of memory traffic, there will be less and less potential benefit for such optimizations. Another way to eliminate compulsory memory traffic is to change data types for large, frequently accessed arrays. A switch from 64-bit to 32-bit data types could double arithmetic intensity and thus potentially double performance.

In this section, we have discussed a number of different optimizations, but the question of which optimization should be applied arises. One could blindly apply all optimizations, but this might be time consuming with little benefit for the work involved. Simply put, given a kernel's true arithmetic intensity, programmers can scan up along said arithmetic intensity wall and determine which ceiling is impacted first. If it is a bandwidth ceiling, they must decide whether it is easier and more important to increase arithmetic intensity and slide along that ceiling, or remove the ceiling by addressing the underlying lack of optimization. If the impacted ceiling was a in-core performance ceiling then performing any other optimization will show little benefit. They will then iterate on the process of identifying the constraint, applying the corresponding optimization, and benchmarking the resultant performance. Such a strategy assumes that such analysis is possible. Section 4.9 will discuss how one would use performance counters to achieve this.

4.8 Extending the Roofline

Sections 4.3 through 4.7 built a Roofline Model for single-program, multiple-data (SPMD) memory-intensive floating-point kernels. This is perfectly adequate for the kernels presented in Chapters 6 and 8. However, for more diverse kernels, we need a broader metric. In this section, we address several directions by which the Roofline Model could be greatly extended to handle a much broader set of kernels.

4.8.1 Impact of Non-Pipelined Instructions

The in-core ceilings of Section 4.5 dealt solely with pipelined instructions of equal maximum throughput. However, on many architectures, reciprocal, divide, square root, and reciprocal square root are not pipelined or are executed at a substantially reduced rate. As such, they will stall execution of subsequent instructions for dozens of cycles. For example, consider an instruction mix of SIMDized multiplies and divides. On many architectures, as the fraction of divides or reciprocal square roots increases, FLOP/s will drop dramatically. Many N-body particle codes rely on a distance calculation including a reciprocal square root. An non-pipelined implementation may severely impair FLOP/s.

Figure 4.10 on the next page shows performance on the Clovertown as the fraction of SIMDized divide instructions (`DIVPD`) increases. Assume the floating-point instructions

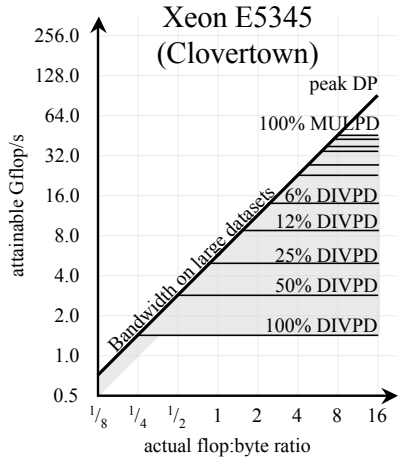


Figure 4.10: Impact of non-pipelined instructions on performance. As double-precision SIMD divides predominate, performance is drastically impaired. Note the log-log scale.

are either multiplies or divides. Clearly, if the code is entirely divides, performance may drop by a factor of 32× over the multiply-only code, but more importantly, if the code is as little as 25% divides, performance has dropped by nearly a factor of 10×. Not only is this a dramatic drop in performance, but it has shifted the effective ridge point far to the left. Thus, codes that nominally would be considered memory-bound would become compute bound.

4.8.2 Impact of Branch Mispredictions

A mispredicted branch will result in a flush of the pipeline. In essence, all functional units will be stalled for the duration of the branch mispredict penalty. Given the average branch penalty, one can define a series of ceilings beneath the roofline enumerated by the fraction of branch mispredicts per floating-point operation in much the same way one visualizes non-pipelined floating-point instructions.

4.8.3 Impact of Non-Unit Stride Streaming Accesses

Although not shown in Figure 4.2 on page 55, one could contemplate additional bandwidth ceilings. For example, consider the attained bandwidth delivered from a Stanza Triad [82] memory access pattern. As the stanza size decreases, hardware prefetchers become increasingly ineffective and more and more memory latency is exposed. The ultimate result is a code that randomly accesses cache lines. In such a case, one would expect bandwidth to be roughly $\frac{LineSize \times Threads}{Latency}$. Future work should include a benchmark to automatically calculate all bandwidth ceilings for the Roofline Model.

4.8.4 Load Balance

In this section, we deviate from the perfectly load-balanced SPMD programming model. As a result, load imbalance will occur. In the context of the Roofline Model, load balance within a socket can broadly be categorized as either imbalance in the memory accesses generated per core or imbalance in the computation performed per core.

Memory imbalance occurs when there is not a uniform distribution of memory traffic across threads, cores, or sockets. As previously discussed, if there is an imbalance in the memory traffic generated among sockets, there will be a drop in memory bandwidth. We can expand on this observation by examining imbalance across cores or threads. The memory level parallelism that a socket must express to satisfy Little's law typically often cannot be satisfied by a single core. As such, if a subset of the cores don't expresses any memory-level parallelism, the socket as a whole may be incapable of satisfying Little's Law. Thus, it cannot achieve peak Stream bandwidth. We can bound this by the extremes. First, determine the achievable bandwidth assuming all memory requests are uniform across all sockets, cores, and threads. Then determine the achievable bandwidth assuming all memory requests come from one socket, but are uniform across all cores and threads within it. Then determine the achievable bandwidth assuming all memory requests come from one core, but are uniform across all threads within it. Finally, determine the achievable bandwidth assuming all memory requests are generated by one thread.

Computational imbalance arises from the fact that some cores are tasked to perform more work than others. As such, the more heavily loaded cores will take longer to finish. In some cases, it is easier to satisfy all forms of in-core parallelism in a kernel than to load balance it, but on other codes load balance is very easy to achieve. In the context of the Roofline Model, the latter would simply form load imbalance ceilings below the in-core parallelism ceilings of Section 4.5. At the extreme, assuming communication is proportional to computational work, if a socket doesn't perform any computation, then it will not generate any memory traffic. Thus, computational imbalance ceilings can result in bandwidth imbalance ceilings, but bandwidth imbalance ceilings don't result in computational imbalance ceilings.

Figure 4.11(a) on page 73 shows the impact of memory imbalance on the Barcelona Opteron assuming fully optimized in-core performance. Clearly, as memory traffic is shuffled onto one single socket or eventually onto a single core, bandwidth will be diminished. Let compute imbalance among sockets imply all computation is performed on one socket, and similarly compute imbalance among cores imply all computation is performed on one core. Thus, Figure 4.11(c) shows the impact on performance if one were to attempt to optimize single thread performance only after attempting to compute balance a kernel. For low computational intensity kernels (less than $\frac{1}{4}$), in-core optimizations are irrelevant, but compute imbalance can result in memory imbalance. In turn, memory imbalance can limit performance. Figure 4.11(b) shows performance when the order of optimization is reversed. As arithmetic intensity exceeds 2.0, computational balance becomes even more important.

One must balance the difficulty in producing balanced parallelization or optimization of in-core performance with the desire for better performance. Consider the case of no in-core optimizations compared with imbalanced code. When arithmetic intensity is less than $\frac{1}{8}$, memory balance is critical, and in-core optimizations and computational balance

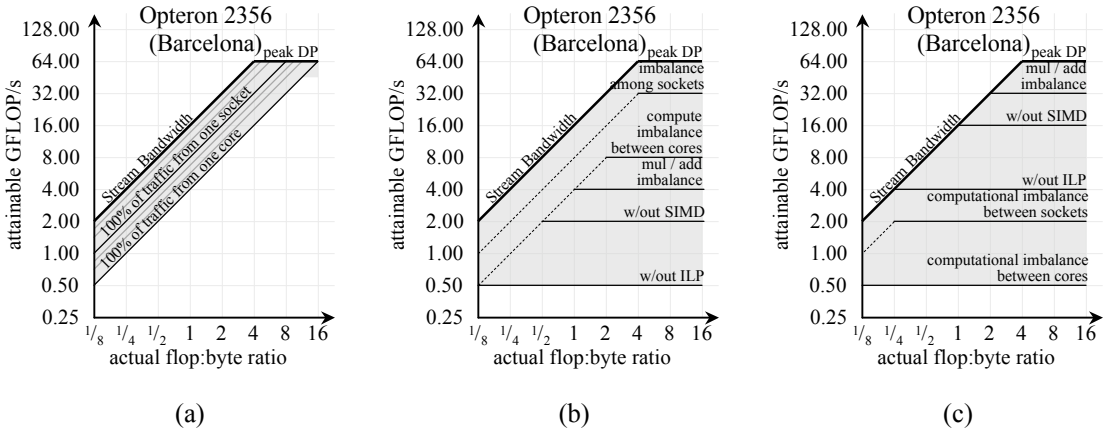


Figure 4.11: Impact of memory and computation imbalance. (a) memory imbalance assuming computational balance and full in-core and bandwidth optimizations. (b) impact of compute imbalance on optimized code. (c) impact of compute imbalance on unoptimized code. The latter two assume memory traffic is proportional to computation. Note the log-log scale.

is irrelevant. However, computational balance quickly becomes important but ultimately can only deliver 4 GFLOP/s without in-core optimizations. In-core optimizations alone could only double-performance when arithmetic intensity exceeds 2. Thus, for such large arithmetic intensities, both balanced execution and optimizations are necessary.

4.8.5 Computational Complexity and Execution Time

Thus far, we have equated performance with throughput (GFLOP/s). However, this ignores the possibility that two different implementations or algorithms might require significantly different numbers of floating-point operations. As such, reduced work might be preferable over increased throughput. If performance is measured in units of time as opposed to units of throughput, then a direct comparison can be made. Assuming computational complexity does not vary with arithmetic intensity, the execution time of a kernel would be its computational complexity (measured in floating-point operations) divided by its throughput (measured in FLOP/s). A nice benefit of plotting on a log-log scale is that if one inverts a function, the only effect is that this negates the slopes of the curves. As such, we may reuse all our previous work including ceilings. All we must do is graphically flip the figures and change the Y-axis labels. However, unlike the performance-oriented Rooflines, such an approach quantifiably ties the resultant figure to a specific kernel's computational complexity.

Figure 4.12 on the next page presents both the throughput-oriented Roofline model as well as an execution time-oriented Roofline Model for a kernel requiring 16 million floating-point operations regardless of arithmetic intensity. The execution time for such a kernel is simply 16 million divided by the floating-point throughput attained in Fig-

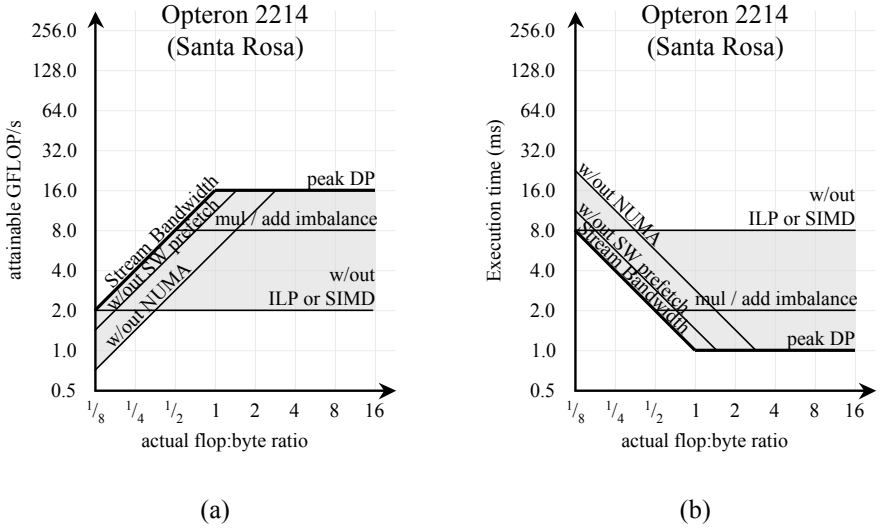


Figure 4.12: Execution time-oriented Roofline Models. (a) standard throughput-oriented Roofline Model. (b) time-oriented Roofline Model for a kernel with a computational complexity of 16 million floating-point operations. Note the log-log scale.

ure 4.12(a). Execution time is a function of arithmetic intensity. Thus it is a dependent on cache misses and in-core optimizations. Once the code is compute bound, no further memory traffic optimizations will improve the execution time.

4.8.6 Other Communication Metrics

Thus far, we have assumed all kernels are "memory-intensive." By that, we mean kernel performance is tied closely to the balance between peak FLOP/s, DRAM bandwidth, and the FLOP:DRAM byte ratio. For the kernels presented in Chapters 6 and 8 this is certainly true. However, the Roofline Model can easily be extended to handle other communication metrics.

Codes such as dense matrix-matrix multiplication can readily exploit locality in any cache. As such, L1 or L2 bandwidth may constrain performance more than DRAM bandwidth. Thus, one could create a Roofline Model based on L2 bandwidth and the FLOP:L2 byte ratio. The in-core roofline and ceilings would be the same as before, and one could modify the Stream benchmark to run from the L2 cache. However, the bandwidth ceilings must be recast in the light of locality within the L2 cache. Bandwidth ceilings might include bandwidth without prefetching, bandwidth to another core's L2 or bandwidth to another socket's L2 cache. One should be mindful as to whether aggregate across all cores or individual bandwidth to a core is being utilized as they will connect to different in-core rooflines.

There is no reason to limit bandwidth to the cache hierarchy. One could imagine Roofline Models based on I/O, network, or disk bandwidth. In each case, one must bench-

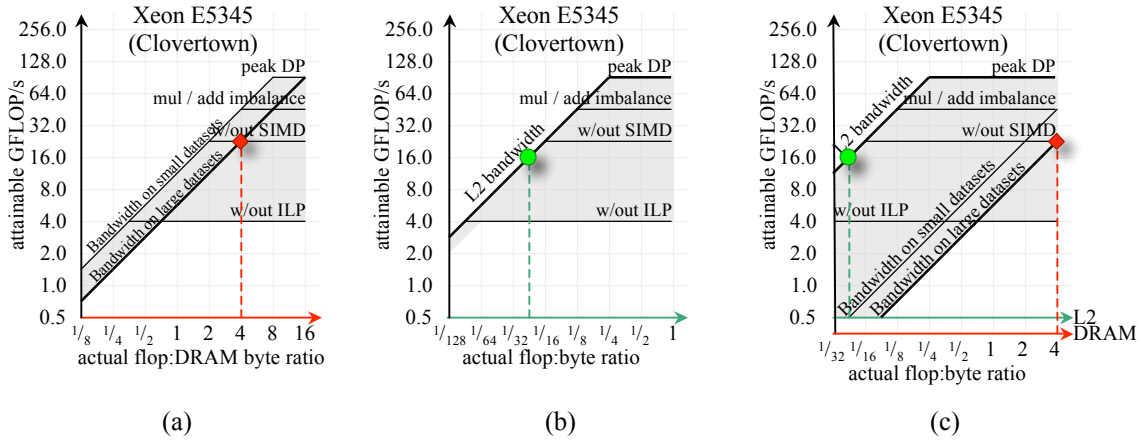


Figure 4.13: Using Multiple Roofline Models to understand performance. (a) DRAM-only Roofline Model. (b) L2-only Roofline Model. (c) combined DRAM+L2 combined model. Note the log-log scale.

mark the bandwidth to define the roofline. To estimate performance, one would also need to know the FLOP:byte ratio for whichever communication metric was selected.

Although it is possible that some kernels on some machines are limited by a single communication bandwidth for a wide range of arithmetic intensities, there are some kernels on some machines for which a small increase in the FLOP:DRAM byte ratio will move an architecture from being memory-bound to L2 bound. Remember, the FLOP:L2 byte ratio is only proportional to the FLOP:DRAM byte ratio on certain kernels. Hence, improving one doesn't necessarily improve the other. As a result, one must simultaneously inspect two Roofline Models. These can either be separate figures, or as per Jike Chong's suggestion, a combined figure with multiple X-axes. Thus, Figure 4.13 shows how one would use multiple Roofline Models to understand performance. Given a nondescript kernel that has a FLOP:DRAM byte ratio of 4.0 and a FLOP:L2 ratio of 0.04, if one were to use only the DRAM model the L2-bound nature would not be evident. By either using a second model (Figure 4.13(b)) or combined model (Figure 4.13(c)) it becomes evident that the low FLOP:L2 byte ratio in conjunction with the low L2 arithmetic intensity results in a constraint on performance greater than DRAM bandwidth. Note there is no reason why the arithmetic intensity x-axes of Figure 4.13(c) must be aligned to the same value. In addition, when using a combined figure, one must be mindful to match arithmetic intensities and bandwidths corresponding to the same communication type.

4.8.7 Other Computation Metrics

The Roofline Model could also be extended to other computational metrics including sorting, cryptography, logical bitwise, graphics, or any other low-level operations. Although graphics operations are commonly expressed in throughput metrics, sorting's nonlinear computational complexity implies that one must note the data set size when noting

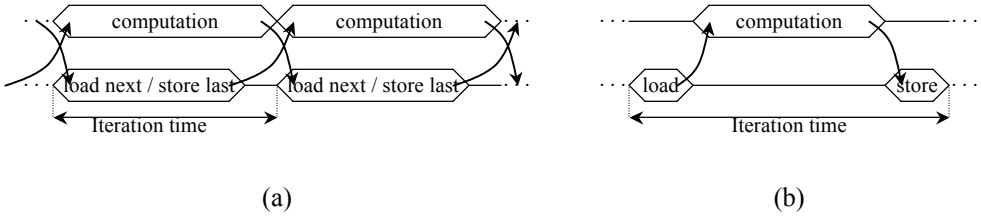


Figure 4.14: Timing diagram comparing (a) complete overlap of communication with computation, and (b) serialized communication and computation. Note the iteration time is up to a factor of 2 smaller when communication and computation are overlapped.

performance. In the context of the Roofline Model, two possibilities arise: first, use the time-oriented form and express performance in units of time. Second, use a throughput-oriented metric for performance. In the case of the latter, we suggest a reasonable metric for performance would be *pairwise sorts per second*. Clearly, use of such a metric is dependent upon one calculating either the algorithmically-dictated number of such sorts required depending on the sorting algorithm, or explicitly counting the number performed. The latter adds accuracy at the expense of software overhead.

In general, one could create a Roofline Model based on any combination of computation and communication metrics. One simply must benchmark the communication bandwidths with differing optimizations, and then benchmark the computation metrics with their appropriate optimizations. Given the resultant pair of tables (metric $\times$ optimization), with some knowledge, experienced computer scientists could pick the relevant metrics for their application and model performance.

As we move from floating-point arithmetic operations to the more generic operations, we should define a generic arithmetic intensity. We define *operational intensity* as the ratio of computational operations to total traffic for the appropriate level of memory. This could be the conventional FLOP:DRAM byte ratio or it could be the pairwise sorts:L2 byte ratio.

4.8.8 Lack of Overlap

Thus far, we have assumed that there is sufficient memory-level parallelism within a kernel that communication and computation can be overlapped. Moreover, we have assumed that given sufficient memory-level parallelism, an architecture has the ability to overlap communication and computation. In this subsection, we examine how one would modify the roofline if such an assumption fails. A motivating example would be transfers from CPU (host) memory to GPU (device) memory over PCIe. Unlike Cell's DMA programming model, in NVIDIA's CUDA programming model, kernel invocations and transfers cannot be overlapped. As such, they are serialized. Thus, the best we can hope for is that computation amortizes the transfer time rather than hiding it.

Conceptually, Figure 4.14 shows two timing diagrams depicting the idealized over-

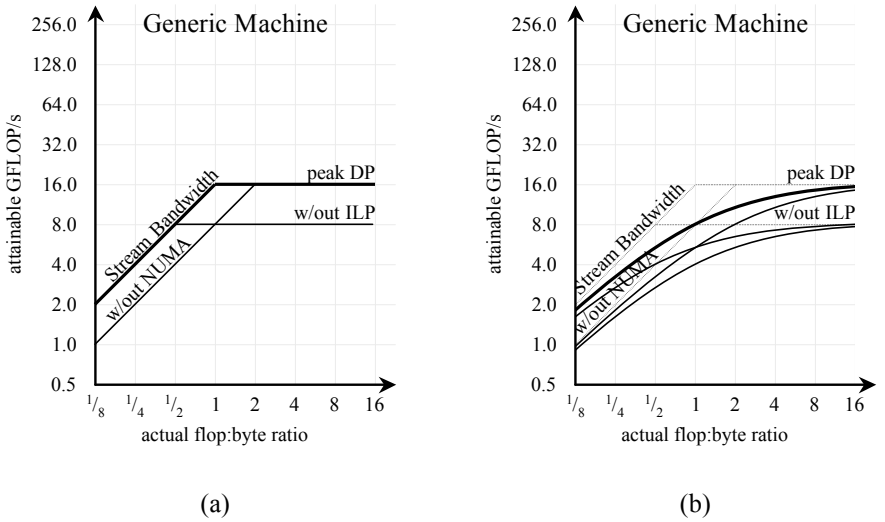


Figure 4.15: Roofline Model (a) with and (b) without overlap of communication or computation. Note the difference in performance is a factor of two at the ridge point. Note the log-log scale.

lap of communication with computation and the serialized form. Clearly, the time per iteration in Figure 4.14(b) is considerably longer than that of Figure 4.14(a). As such, average performance is diminished. Equation 4.5 quantifies the average throughput in the no overlap, no overhead case. Clearly, the concept of a ridge point is muted as performance is always dependent on both bandwidth and computation.

$$\text{Attainable Performance}_{ij} = \frac{\text{Arithmetic Intensity}}{\frac{\text{Arithmetic Intensity}}{\text{Performance}_i} + \frac{1}{\text{Bandwidth}_j}} \tag{4.5}$$

Figure 4.15 presents a Roofline Model for each case. The sharp roofline structure of Figure 4.15(a) has been smoothed in Figure 4.15(b). At the limits of arithmetic intensity, performance is basically dependent on bandwidth with certain optimizations or computation with certain optimizations. However, near the ridge point, performance may drop by as much as a factor of two: computation requires as much time as computation. Examining the ceilings, there is a clear switch as to the importance of certain optimizations near the original ridge point.

The original Roofline formulation is the ideal case. However, not every architecture or machine can effectively overlap communication or computation. Moreover, not every kernel expresses sufficient parallelism to allow this. As such, in some cases it may be appropriate to use this no-overlap Roofline Model to understand the nearly factor of two loss in performance.

Although examples of I/O bandwidth to a GPU may seem compelling, one should also consider the applicability to CPU performance. Some architectures must expend computational capability to express communication — *i.e.* software prefetch. As such, communication and computation cannot be perfectly overlapped.

4.8.9 Combining Kernels

Applications are invariably more than a single kernel. The Roofline is premised on analyzing a single kernel at a time. Thus, the substantial benefits that may be perceived by including an optimization based on the Roofline Model may be muted in the context of a multi-kernel application.

To correctly analyze a multi-kernel application, one should first benchmark the application noting the time required for each kernel. On could then pick the critical kernel and analyze it assuming the time required for the other kernels is invariant. Application performance now becomes a time-weighted harmonic mean of kernel time. As a result, an application's Roofline Model for optimizing one kernel at a time would have ceilings and ridge points that are likely to be muted and smoothed.

4.9 Interaction with Performance Counters

Sections 4.3 through 4.7 built the Roofline Models based on microbenchmarks. The resultant Roofline Models could then be used to gain some understanding of kernel performance, as well as to quantify the potential performance gains from employing more optimizations. Unfortunately, such an approach still requires substantial architectural knowledge and significant trial and error work. In addition, in many cases, bounds were defined as all or nothing ceilings, and arithmetic intensity was often calculated based on compulsory memory traffic. In this section we detail how the Roofline could be enhanced through the use of performance counters. We expect it to provide a framework and stepping stone for future endeavors. Performance counters will allow us to remove much of the uncertainty in the Roofline Model.

4.9.1 Architectural-specific vs. Runtime

When multiple ceilings are present, we calculate performance assuming we reap the benefits from all ceilings up to a point but no benefit thereafter. In the real world this is atypical. One can partially exploit both DLP and ILP without reaping the full potential of either. Thus, we motivate a switch from architectural Rooflines and ceilings to one based on runtime statistics. These runtime ceilings will show the performance lost by not completely exploiting an architectural paradigm.

4.9.2 Arithmetic Intensity

Accurately calculating total memory traffic is the easiest and most readily understandable use of performance counters. Previously, we were forced to either assume only

compulsory misses or use a costly cache simulator to estimate all cache misses. Using performance counters, assuming they provide the requisite functionality, one could calculate the total memory traffic. This should include all compulsory, conflict, capacity, and speculative misses; the latter includes hardware stream prefetched data and should be readily differentiable from the former three. Ideally this will allow a transition from a simple less-than bound on arithmetic intensity to an exact calculation.

Memory traffic is only half of the data required to calculate arithmetic intensity — the other being the total number of floating-point operations. Just as one could either use compulsory memory traffic or the performance counter measured memory traffic, one could also use either the algorithmically derived compulsory floating-point operations or the performance counter measured number of floating-point operations. The compulsory FLOP:compulsory byte ratio provides an ideal arithmetic intensity. When the performance counter measured FLOPs exceed the compulsory number of FLOPs, wasted work has been performed. If compute bound, the programmer should consider optimizing wasted work away. When the performance counter measured memory traffic exceeds the compulsory memory traffic, bandwidth has been squandered to transfer extra data. When memory-bound, the programmer should be encouraged to block, pad, bypass the cache, or change data structures to minimize the volume of data.

4.9.3 True Bandwidth Ceilings

In addition to determining the true arithmetic intensity, one could use performance counters to accurately determine the spacing of various bandwidth ceilings. In doing so it becomes evident how much performance is lost by not implementing or sub-optimally implementing certain optimizations.

Simply put, peak bandwidth is only possible if data is transfered on every bus cycle. Not transferring data every cycle diminishes the sustained bandwidth. By counting the number of data bus cycles, one can calculate the true bandwidth. In conjunction with true arithmetic intensity, performance is more readily understandable.

Unfortunately, simply stating that the true bandwidth is 57% of the Stream bandwidth doesn't aid in performance tuning. We still need to understand why performance is lost. Thus, by inspecting the cause for each idle cycle one could determine not only the true bandwidth, but also what fraction of performance may be lost due to various factors. By quantizing the loss of performance by category, we have once again in effect created a number of bandwidth ceilings between the true bandwidth and the roofline.

Consider the example presented in Figure 4.16 on the following page. Assume all bandwidth optimizations have been attempted. Figure 4.16(a) shows the conventional architecture-oriented Roofline Model. The gold star denotes the attainable performance given the compulsory arithmetic intensity. However, the red diamond marks the observed performance. One might erroneously conclude that attempts to exploit both NUMA and software prefetching were completely ineffective. To resolve this confusion, performance counters might be used to determine that the true arithmetic intensity was half the compulsory arithmetic intensity — shown in Figure 4.16(b). Moreover, performance counters could be used to show that the observed performance corresponds to the product of the true bandwidth and true arithmetic intensity. The question arises, why is performance only half

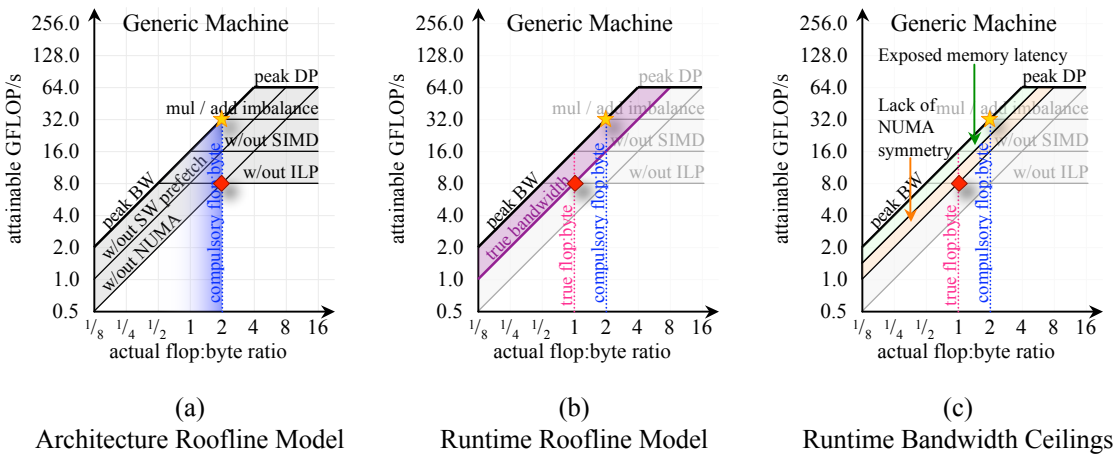


Figure 4.16: Runtime Roofline Model showing (b) performance counter calculated true arithmetic intensity and (c) bandwidth ceilings representing the true rather than maximum loss in performance. Note the log-log scale.

what's expected given the level of optimization? To that end, Figure 4.16(c) shows that performance counters could be used to show that NUMA and software prefetching were only partially effective. Not only is there asymmetry in the memory traffic produced by each socket, but it is clear that software prefetching didn't fully cover the memory latency. Thus, performance counters can be effectively utilized to show why observed performance was only $\frac{1}{4}$ the Roofline bound.

Although we may be able to categorize and quantify many of the causes for reduced bandwidth, some may remain as combinations of unknown given the limitations of performance counters. Thus, with realistic limitations of performance counters, we may be forced to label a bandwidth ceiling as "unknown." This could include problems buried deep in the memory controllers like frequently exposing the DRAM page access latency.

4.9.4 True In-Core Performance Ceilings

In much the same way performance counters could be exploited to understand the performance when multiple memory bandwidth concepts are partially exploited, we can use them to understand in-core performance.

Clearly, performance counters could be used to understand the dynamic instruction mix. By simply counting the number of floating-point instructions issued and dividing by the total number of instructions issued, we can arrive at the instruction mix. Given the true mix, one could query an architecture Roofline mode to determine if the mix is actually constraining performance.

Similarly, performance counters could be used to determine what fraction of the floating-point instructions were SIMDized, what fraction of the instructions exploit FMA, on what fraction of the issue cycles are both multiplies and adds issued, and on what fraction of the cycles are instructions not issued because of in-core instruction hazards. Given this

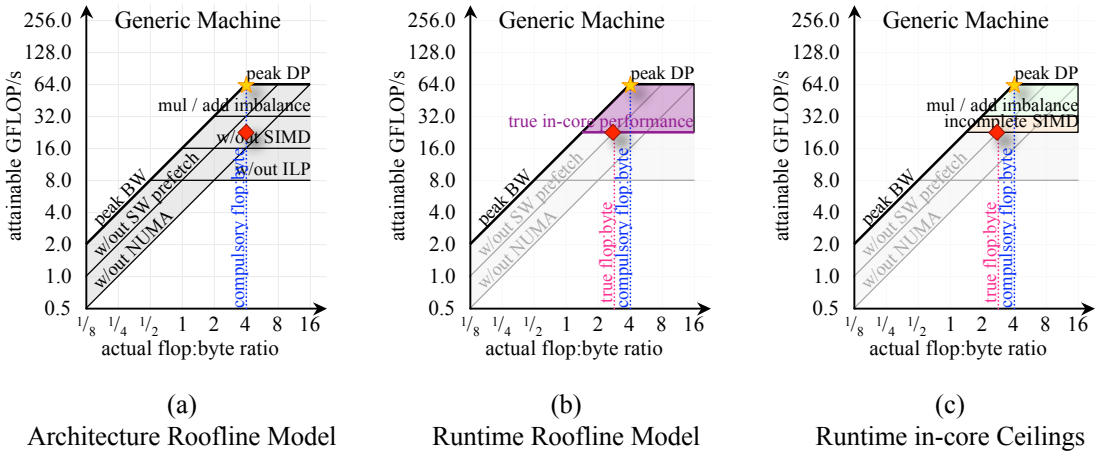


Figure 4.17: Runtime Roofline Model showing (b) performance counter calculated true arithmetic intensity and (c) in-core performance ceilings representing the true rather than maximum loss in performance. Note the log-log scale.

information, we could construct a series of runtime in-core ceilings analogous to the runtime bandwidth ceilings in Figure 4.16 on the previous page.

Figure 4.17 presents an in-core example. In this case, assume all optimizations have been applied and communication has been determined not to be the bottleneck. As shown in Figure 4.17(b), performance counters could be used to determine both the true arithmetic intensity and the true in-core performance. To facilitate optimization, performance counters could be used to determine how effectively ILP, DLP, and functional unit parallelism have been exploited. Clearly, in this example the compiler was able to deliver a little less than half the potential performance. Upon examination, it is clear the compiler fully exploited ILP, partially exploited SIMDization, but was not able to balance multiples and adds. As a result, the latter two remain as ceilings in Figure 4.17(c).

4.9.5 Load Balance

Runtime load balance is far from the all or nothing (perfect or worst case) balancing presented in Section 4.8. One could imagine that at each level of the integration hierarchy — within a core, within a socket, within a SMP — there is some distribution of work (computation or memory traffic) across threads, cores, or sockets. That is, there is a distribution of computation among the sockets of an SMP, within each socket there is a distribution of computation among cores, and within each core there is a distribution of work among hardware thread contexts.

As the distribution of work becomes more imbalanced, certain threads, cores, or sockets are given a disproportionate fraction of the work. Up to an architecturally dependent critical imbalance, there is no performance drop, but beyond this critical imbalance, performance begins to drop. Clearly, for computational balance at the multicore or multisocket level, the critical balance is uniform. However, at the multithreading level, the

critical balance may be significantly different than uniform. A similar condition arises when dealing with memory imbalance. Each core or thread may not be capable of satisfying the socket-level Little's Law. As such, if too few are used, the subset will not satisfy the socket-level Little's Law.

As load imbalance represents current and on going research, we haven't defined how one could distill performance counter derived imbalance into one or more ceilings.

4.9.6 Multiple Rooflines

Performance counters would also facilitate the analysis of multiple Roofline Models. Just as we might collect DRAM related performance counter information including total DRAM memory traffic, or exposed DRAM latency, we could have just as easily collected L2 memory traffic or exposed L2 latency. With such information in hand, bottleneck analysis would be greatly enhanced.

4.10 Summary

In this chapter, we introduced and defined the Roofline Model. This model comes in two basic forms: an architecture-specific form, constructed in Sections 4.3 through 4.7, and a runtime form, described in Section 4.9. We use the former model throughput the rest of this paper to model, predict, analyze, and qualify kernel performance.

The architecture-oriented model is an idealized representation of an architecture's potential performance based on bound and bottleneck analysis. The Roofline Model is further enhanced by performing multiple analyses based on progressively higher exploitation of an architecture's features. Such a formulation can be very useful in modeling, predicting, and analyzing kernel performance. However, its value is diminished when the user cannot quantify which aspects of an architecture have not been fully exploited.

To that end, the runtime form exploits performance counter information to precisely detail in what aspects the software, middleware, and compiler failed to implement the optimizations required by a specific kernel to fully exploit the machine. To be clear, it is no longer a model, but a visualization of performance counter information structured to resemble the Roofline Model.

We believe that programmers with all levels of background, specialization, experience and competence can use the Roofline Model to understand performance. From this understanding it is hoped that they can effectively optimize program performance or redesign hardware.

Chapter 5

The Structured Grid Motif

Structured grid kernels form the cores of many HPC applications. As such, their performance is likely the dominant component for many applications. Structured grid kernels are also seen in many multimedia and dynamic programming codes. Thus, their value cannot be underestimated.

This chapter discusses the fundamentals of structured grid computations. By no means is it comprehensive. In addition, we do not analyze the mathematics or computational stability of any kernel. For additional reading, we suggest [109]. The chapter is organized as follows. First, Section 5.1 discusses some of the fundamental characteristics of structured grids. In Section 5.2, we discuss the common characteristics of structured grid codes. Section 5.3 then discusses several techniques that have been developed to accelerate the time to solution of various structured grid codes. Finally, we provide a summary in Section 5.4.

5.1 Characteristics of Structured Grids

Motifs typically do not have well defined boundaries. As such, based on a kernel's characteristics, one might categorize a given kernel into one or more motifs. In this section and the next, we discuss several agreed upon characteristics of the structured grid motif. Generally, kernels adhering to these characteristics would be classified as structured grids. We discuss these characteristics not only to differentiate structured grids from other motifs, but also to allow one to understand the breadth of and differences between structured grid codes. Most importantly, we discuss the characteristics here to provide clarity to the auto-tuning effort in Chapter 6.

Conceptually, one can think of a structured grid as a graph. There are a number of nodes where data is stored and computation is performed and a number of edges connecting them. The edges represent the valid paths data traverses across throughout the calculation. The nodes directly connected to a node are the "neighboring" nodes. In addition to simple Boolean connectivity, the edges encode distance or weight. Conceptually, a graph might be derived from a N-dimensional physical problem. One of the inherent characteristics of structured grids compared to graphs is that they retain knowledge of the global connectivity or dimensionality and periodicity rather than blindly interpreting the data as an arbitrary graph. The next subsections discuss the principal characteristics of

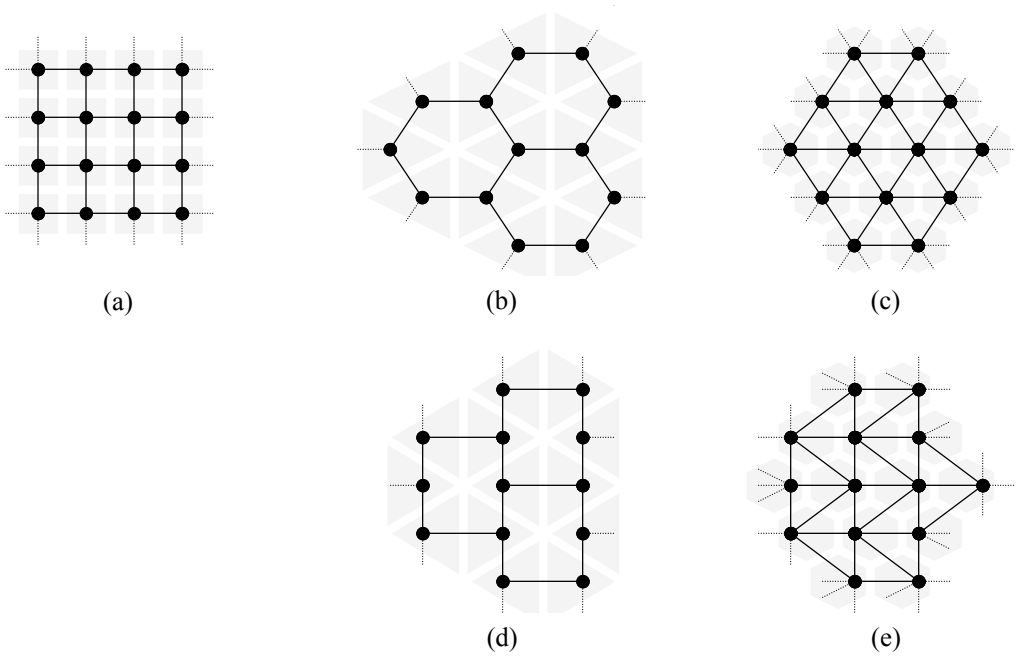


Figure 5.1: Three different 2D node valences: (a) rectangular, (b) triangular, (c) hexagonal. (d) and (e) are just (b) and (c) aligned to a Cartesian grid to show geometry and connectivity are disjoint concepts.

structured grid codes: node valence, topological dimensionality, periodicity, and physical geometry. node valence deals with the local connectivity among nodes, where topological dimensionality and periodicity deals with the global connectivity. In addition, we discuss the ease in which neighbor addressing is handled. The geometry maps this topology to physical space. Although the breadth of such characteristics is extremely wide, we only discuss the common configurations.

5.1.1 Node Valence

The first characteristics we discuss are the topological characteristics. More specifically, we first describe the individual or local connectivity between nodes. Typically in structured grid codes, all nodes have the same node valence. That is, every interior node in the grid is connected to the same number of nodes. Moreover, the lengths or weights of the connective edges are identical or uniform. Such characteristics distinguishes the structured grid motif from both the unstructured grid motif and the sparse motifs. In both of the latter cases, connectivity is explicit and encoded as data at each node.

There are several common node valences or connectivities including cases where each node connects to 2, 3, 4, 6, or more other nodes. Dimensionality is typically disjoint from realizable connectivities. For example, 4-way connectivities appear in both 2 and 3 dimensions.

Figure 5.1 shows three different 2D node valences. We name them based on their

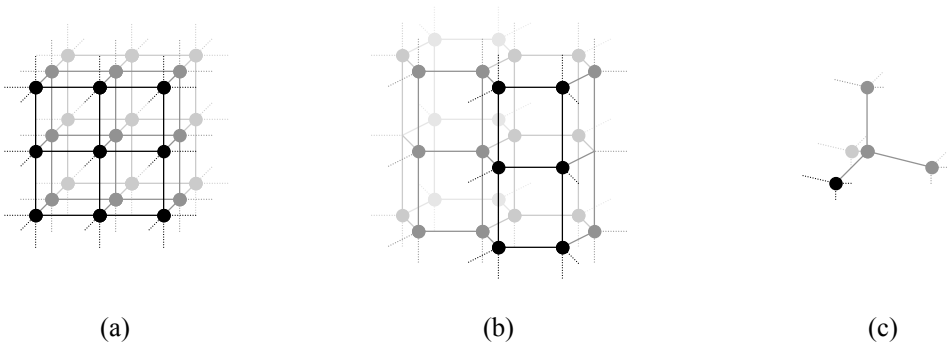


Figure 5.2: Three different 3D node valences: (a) hexahedral, (b) triangular slabs or prisms, (c) tetrahedral.

duals. In rectangular connectivity, every node is connected to four other nodes. Triangular connectivity mandates every node is connected to three other nodes, whereas in hexagonal connectivity, every node is connected to six other nodes. Observe that node valence is disjoint from topological dimensionality and periodicity. As such, despite dramatically different appearances, Figure 5.1(b) and (d) are both considered triangular connectivities and Figure 5.1(c) and (e) are both considered hexagonal connectivities.

Figure 5.2 shows three different 3D node valences. Figure 5.2(a) and (b) are simply layered or slab extensions of Figure 5.1(a) and (b), where Figure 5.2(c) is a more natural and uniform extension of Figure 5.1(a).

In scientific computing, rectangular and hexahedral topologies are by far the most common simply because this meshes well with the underlying mathematics. However, there are cases where the desire to facilitate a mapping onto a geometry overrides the mathematical challenges. In such cases, tetrahedral or hexagonal slabs are often used. In image processing and dynamic programming, 2D rectangular topology predominates.

5.1.2 Topological Dimensionality and Periodicity

The next characteristics we discuss are the topological dimensionality and periodicity. These define the global connectivity among nodes. For simplicity, we will refer to these combinations by their physical coordinate system cousins. There are a plethora of geometries we may map a given node valence onto just as there are a number of coordinate systems as a function of dimensionality. Often the dimensionality and periodicity are derived from a physical coordinate system. We discuss several common cases starting with the trivial one-dimensional forms.

In one dimension, there are two common coordinate systems: linear and circular. These represent the canonical 1D Cartesian and fixed radius polar (angular) coordinates. In essence, a circular geometry is implementing periodic boundary conditions without resorting to explicit copies of the boundaries (ghost zones are discussed in Section 5.2.2). Note that topology can be restricted by dimensionality. For example, in 1D, topology is restricted to only left and right neighbors. Typically, a single topological index *(i)* is used.

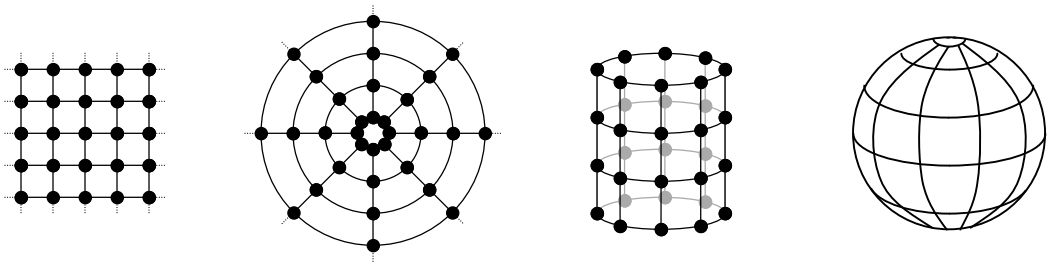


Figure 5.3: Four different 2D geometries, all using a rectangular topology: (a) Cartesian, (b) polar, (c) cylindrical (surface), (d) spherical (surface). Note, in (b) there is no node at the center, and the innermost radius can be arbitrarily small. A similar condition exists in (d).

In two dimensions, degenerate forms of Cartesian, polar, cylindrical or spherical coordinate systems are typical. One may restrict the 3D cylindrical or spherical coordinate systems to 2D by fixing one coordinate. Thus the topologic index *(i,j)* coordinate can be interpreted as a *(x,y)* in a Cartesian coordinate system, *(r,θ)* in polar, *(θ,z)* in cylindrical, and *(θ,ϕ)* in spherical. Note that the possible connectivities in two dimensions are much richer than in one dimension.

Connectivity, dimensionality, and periodicity, are disjoint concepts. Figure 5.1 on page 84 can be thought of as mapping three different connectivities onto a 2D Cartesian plane. Similarly, Figure 5.3 shows four different mappings of the rectangular node valence. When examining any interior node, it is clear that it is always connected to four other nodes regardless of geometry as there is no vertex at the origin of Figure 5.1(b) or the poles of Figure 5.1(d). Although we show only the rectangular connectivity, we could have mapped any of the topologies in Figure 5.1 onto the geometries of Figure 5.3. Note, to make the Cartesian nature of the triangular and hexagonal grids obvious, the nodes have been aligned to a Cartesian grid in Figure 5.1(d) and (e).

The number of possible mappings expands further in three dimensions. In addition to the standard Cartesian *(x,y,z)*, spherical *(r,θ,ϕ)*, and cylindrical *(r,θ,z)* coordinate systems, one could easily add toroidal, among others.

5.1.3 Composition and Recursive Bisection

When presented with a complex geometry, it is common for one to jump to an unstructured grid calculation. However, simpler solutions exists. First, it is possible to compose two structured grids with different node valences together. The challenge is that a new node valence is introduced where the two grids meet. In addition to composing grids of different connectivities, one could compose grids of different physical geometries or compose similar grids into a complex, higher-dimensional grid. Second, one may recursively recursively bisect the edges of the faces of a solid into uniform grids. The end result is the

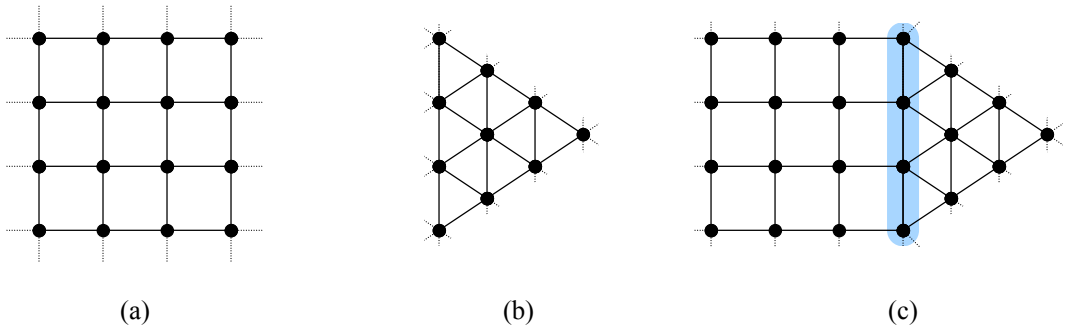


Figure 5.4: Composition of a Cartesian (a) and hexagonal (b) mesh. Note, the mathematics will be different at the boundary in (c).

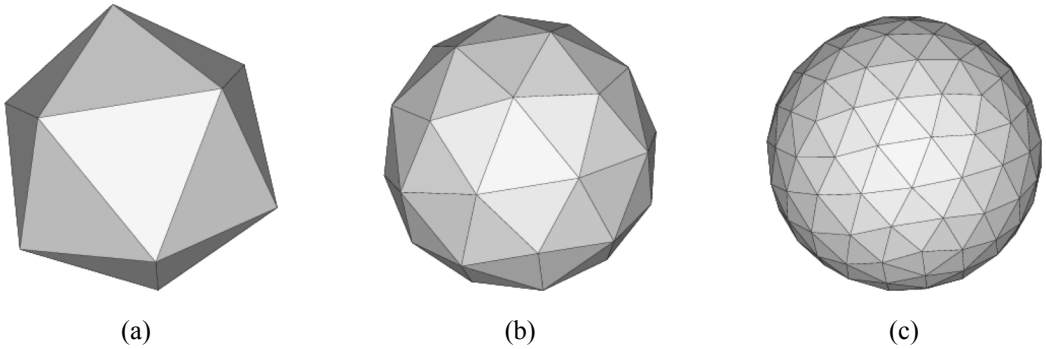


Figure 5.5: Recursive bisection of an icosahedron and projection onto a sphere. Each of the original 20 faces becomes a hexagonal mesh. Reproduced from [76]

same as composing several grids together.

Figure 5.4 shows the composition of a rectangular and a hexagonal grid. At the interface, each node has five neighbors rather than the four in rectangular or six in hexagonal. Thus, the computation at the interface would be different.

Although there are standard coordinate systems for spherical geometries, they lack the uniformity scientists and mathematicians desire. This recursive bisection approach has become a common technique for generating a nearly uniform topology and geometry. Thus, to approximate a sphere, one commonly geodesates a cube, octahedron, or icosahedron.

Figure 5.5 shows the recursive bisection of an icosahedron. At each step, each edge is bisected, and three new edges are inserted. A projection is applied when mapping to physical coordinates. Notice that there are two node valences: pentagonal and hexagonal. Moreover, there are always 12 nodes with pentagonal connectivity. However, the nodes with hexagonal connectivity soon dominate. In essence, 20 hexagonal Cartesian grids with triangular boundaries have been composed together into a 3D geometry. Despite the apparent complexity of this approach, notice the edges are all uniform lengths — a quality

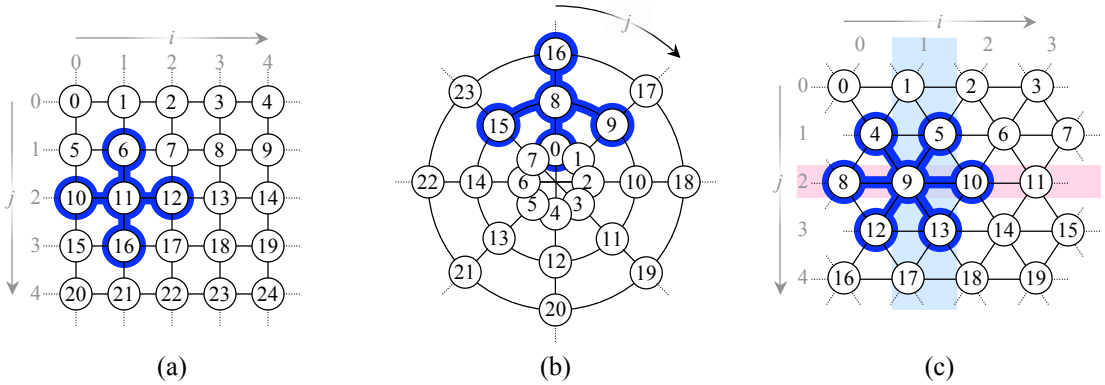


Figure 5.6: Enumeration of nodes on different topologies and periodicities: (a) rectangular Cartesian, (b) rectangular polar, and (c) hexagonal Cartesian. The value at each node is its array index.

not present in spherical coordinates. This approach is used in some climate codes [66, 67]. Others use a similar approach in which a cube is geodesated [112].

5.1.4 Implicit Connectivity and Addressing

If the connectivity is uniform across the grid, one can choose an enumeration of the nodes such that one can arithmetically calculate a node's neighbors. That is, given a grid's topological dimensions and a node's coordinate in topological space *(i,j,k)* or address in memory `(&Node)` — an *array index*, one can directly calculate the coordinates or addresses of the connected nodes. For the more complex geometries in Section 5.1.3, such calculations may require several cases. Nevertheless, in structured grid codes, the addresses of the connected nodes are never explicitly stored. Explicit storage of nodes is required in the unstructured grid, graph algorithm, and sparse linear algebra motifs.

Consider Figure 5.6. We show two different connectivities and two different geometries. Each node has a topological index *(i,j)*.

Figure 5.6(a) shows a rectangular connectivity with a Cartesian geometry. The neighbors of a node at *(i,j)* are: *(i-1,j)*, *(i+1,j)*, *(i,j-1)*, and *(i,j+1)*. If we choose an enumeration of the nodes such that the index of the node at *(i,j)* is $i + 5 \times j$, then the indices of the neighboring nodes are offset by ± 1 and ± 5.

Figure 5.6(b) shows a rectangular connectivity with a polar geometry. Let *(i,j)* represent *(r,θ)*. As it is still a rectangular connectivity, the neighbors are still: *(i-1,j)*, *(i+1,j)*, *(i,j-1)*, and *(i,j+1)*. However, the periodicity of the coordinate system makes the addressing of the neighbors more challenging. Although offsets in the radial direction are simply ± 8, offsets in the angular direction can be ± 1, -7 and -1, or $+1$ and $+7$ depending on *j*.

Figure 5.6(c) shows a hexagonal connectivity with a Cartesian geometry. There are now six neighbors, and depending on j, they can be either:

(i-1,j), *(i+1,j)*, *(i-1,j-1)*, *(i-1,j+1)*, *(i,j-1)*, and *(i,j+1)*

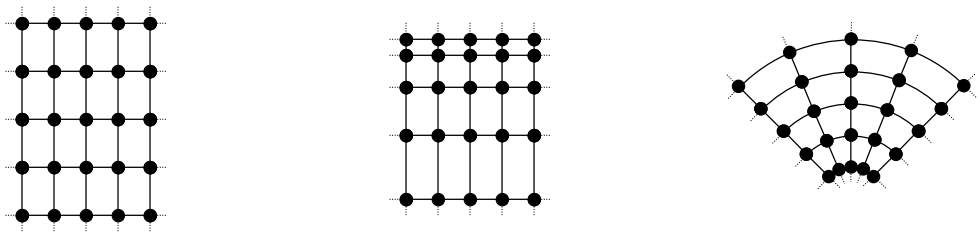


Figure 5.7: Mapping of a rectangular Cartesian topological grid to different physical coordinates using (a) uniform, (b) rectilinear, or (c) curvilinear approaches.

— or —

(i-1,j), *(i+1,j)*, *(i,j-1)*, *(i,j+1)*, *(i+1,j-1)*, and *(i+1,j+1)*

Clearly, this necessitates two different cases to handle the addressing. Although neighbor offsets of ± 1 and ± 4 are common to all points, depending on j, offsets of -3 and $+5$ or -5 and $+3$ are possible.

Rectangular connectivity and Cartesian geometry requires trivial arithmetic when calculating the neighboring addresses, but even the work required for other connectivities and geometries is still easily tractable.

5.1.5 Geometry

Thus far, we have only discussed topology, dimensionality and periodicity. However, we must now map to a physical geometry. Given topological indices *(i,j,k)*, a mapping function will produce physical coordinates *(x,y,z)* or *(r,θ,φ)* depending on the mathematics of the underlying coordinate system. These transformations allow simulation of complex geometries using simple arithmetic operations. Although it is possible to realize any geometry with any topology by mixing and matching topologic and physical coordinate systems, it is inherently undesirable. Figure 5.7 maps a single node valence, dimensionality, and periodicity combination into three different geometries using three different approaches.

Uniform

The simplest transformation is a uniform scaling of the topological indices into physical coordinates. Thus, one could transform topological index *(i,j,k)* by scaling each such that $(x,y,z) = (\Delta x \cdot i,\ \Delta y \cdot j,\ \Delta z \cdot k)$. Similar transformations apply in all coordinate systems. In all cases, the scaling is uniform for all ranges of a given coordinate. As such, the topological coordinate system must be the same as the physical coordinate system. Figure 5.7(a) shows that the y scaling is twice the x scaling. That is, the nodes are further apart in the y-dimension than in the x-dimension.

Note, topological coordinates, physical coordinates, and the units of the underlying mathematics may be dramatically different. For example, *(i,j,k)* may only be small integers

in the range 0 through 255, but the physical coordinates might be real numbers 0.0 through 99.9. Moreover the physical coordinates might be in units of gigaparsecs.

Rectilinear

Rectilinear coordinates are those in which the physical coordinates have a non-linear relationship with the topological index. For instance, one could conceive of physical coordinates that are the exponential of the topological index: $(x,y,z) = (10^i, 10^j, 10^k)$. Alternately, consider the case where the scaling is dependent on the topological index: $(x,y,z) = (f(i) \cdot i, g(j) \cdot j, h(k) \cdot k)$. Figure 5.7(b) shows that the y scaling is dependent on the j topological index, but the x scaling is uniform. In general, this could be extended to other topologies, dimensions or periodicities.

Curvilinear

In Cartesian coordinates, the lines of constant coordinate values are straight lines. In a curvilinear coordinate system, these lines are curved. One may map a Cartesian coordinate system to a curvilinear physical coordinate system through the appropriate transformation. Clearly, such an approach pushes the complexity of the periodicity of coordinates from addressing to boundary conditions.

Figure 5.7(c) maps a rectangular Cartesian grid into a curved space. The physical coordinates of each node and the spacing between nodes are obviously far more than a simple scaling of the topological index.

5.2 Characteristics of Computations on Structured Grids

In this section, we discuss the common characteristics of computations on structured grids. We commence by discussing the data stored and computation performed at each node. Finally, we conclude this section with a discussion of the breadth of code structure and inherent parallelism within structured grid codes as well as common memory access patterns.

5.2.1 Node Data Storage and Computation

There are no restrictions as to the number of variables or their type stored at each node other than that every node has identical storage requirements. Similarly, the structured grid motif places no mandates on the computation performed at each node other than the computation being the same at every node.

Data

In the simplest codes, one floating-point number or integer is stored at each node. Such scalar representations are appropriate for scalar fields like temperature or potential. As codes become more complex, each node may be required to store a Cartesian vector like velocity or alternately a pixel (RGB color tuple). As codes become even more complex, a small matrix or lattice distribution may be stored at each node.

Lattice methods evolved from statistical mechanics [126]. A lattice distribution maintains up to 3^D higher dimension phase space components (velocities). Often these velocities are scalar floating-point numbers, but they could also be Cartesian vectors. Uniquely, lattice methods describe their distributions as DdQq, where d is the dimensionality and q is the number of velocities. For example, D3Q27 would be a complete 3D lattice distribution. The most complex codes will maintain several grids of varying data types. Rectangular or hexahedral are the most common connectivities for lattice methods.

Within the structured grid motif, there is no mandate as to how data must be stored. It may either be stored by node or by component, either in one array or multiple arrays. These two extremes are the canonical array-of-structures (AOS) and structure-of-arrays (SOA) styles. Typically, nodes are stored consecutively within the corresponding arrays. They are indexed by their enumerations or array index, for example, `grid[component][node]` or `grid[node][component]`. This make addressing nodes and components computationally trivial.

Computation

The basis behind all structured grid computations is the *stencil*. A stencil is a specific pattern, centered on the current node, that specifies not only which neighboring nodes must be accessed, but also which components must be gathered from each neighbor. Stencils may be expanded to include neighbors of neighbors. The only real restrictions are that the number of neighbors be fixed and finite, and that the same stencil be applied to all nodes in the grid.

Once the data has been gathered, the computation proper may be performed. There are no real restrictions on the computation at each node other than the same computation being performed at every node, just on different data. It can be linear or non-linear functions on bitwise, integer, or floating-point data.

Examples

Figure 5.8 on the next page presents two different stencils on three different 2D Cartesian grids. By no means are they representative of all possible stencils or all possible grids, but they should be illustrative of the concepts. Any computation could be performed on the data gathered.

Figure 5.8(a) shows a 5-point 2D stencil on a scalar rectangular Cartesian grid. The term point refers to the number of nodes that will be accessed. Its five points are: the center, the left, right, top, and bottom neighbors. From each point, the scalar value stored at that node is gathered.

By contrast, Figure 5.8(b) shows a 5-point 2D stencil on a Cartesian vector grid. At each point in the grid a 2D Cartesian vector *(x,y)* is stored. From the left and right neighbors, the x component is gathered, but from the top and bottom neighbors, the y component is required. From the center point, both components are used. Clearly, not all 5-point stencils are the same, nor are all 5-point stencils on vector grids.

Finally, Figure 5.8(c) shows a 9-point stencil on a lattice. Each node stores a distribution (array) of velocities (components). The stencil operator gathers a very specific

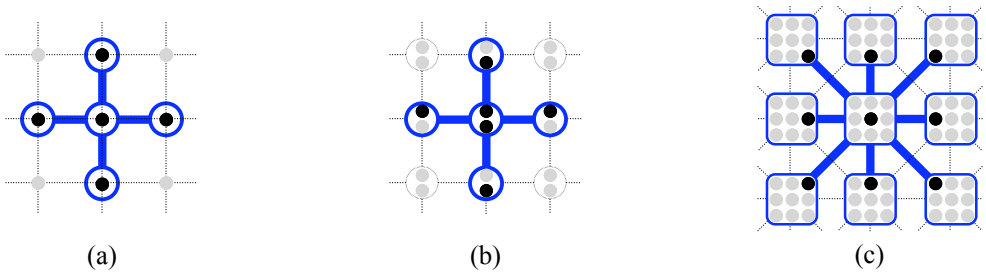


Figure 5.8: 5- and 9-point stencils on scalar, vector, and lattice grids: (a) 2D 5-point stencil on a scalar grid, (b) 2D 5-point stencil on a Cartesian vector grid, (c) 2D 9-point stencil on a lattice distribution (D2Q9). Remember, the DdQq notation is only used for lattice methods.

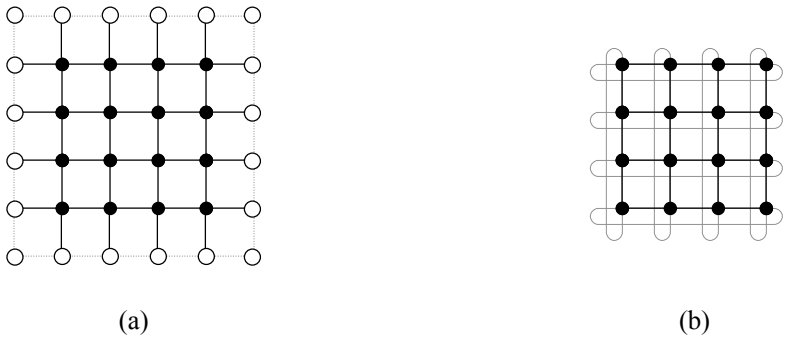


Figure 5.9: 2D rectangular Cartesian grids (a) with and (b) without the addition of ghost zones.

pattern of velocities from the eight neighbors and the center point.

5.2.2 Boundary and Initial Conditions

Depending on the periodicity, some problems have a boundary and others don't. Problems with boundaries leave some nodes partially connected — exceptions to uniform node valence. Consider the mappings in Figure 5.3 on page 86. Clearly, the spherical geometry has no boundaries. However, the cylindrical geometry has a boundary around the top and bottom, the polar geometry has a boundary at the outer radius, and the Cartesian grid has a boundary around all four sides. There two issues that must be addressed. First, how is a stencil applied to a node on such a boundary, and second, what values are employed for the stencils on the boundary?

Figure 5.9 shows two solutions to applying a stencil on the boundary. The most common solution is to augment the grid with a *ghost zone*. A ghost zone is an additional set of nodes around the perimeter of the grid. Often additional ghost cells are added to facilitate addressing. The values at these nodes are set with the appropriate boundary conditions,

thereby allowing the same stencil to be applied to every interior point. However, there is no need to apply the stencil to the ghost zone. An alternate approach would be to define a series of additional stencils for the nodes on the boundary. Although in the 2D polar and cylindrical geometries only one addition stencil would be required, the Cartesian grid would mandate the addition of at least 8 additional stencils — four for the sides, and four for the corners. The latter approach is less common as it requires a location-aware conditional to decide which stencil should be applied. As such, ghost zones are the common case.

Typically, the mathematics applied on a structured grid dictates how these boundaries are treated. One could classify these boundary conditions into four common types:

- Constant
- Periodic
- Ghost zones created through parallelization
- Other

Constant boundary conditions mean the boundary to the grid is constant in time, but free to vary in space. Thus, each point could have a different value, but that value will never change. Constant boundaries can be implemented either by designating a ghost zone and filling it at the beginning of the kernel or by designating boundary stencils. The former is the common solution. The stencil operator must not be applied to the boundary.

Periodic boundaries are somewhat restricted in their application. Consider a Cartesian grid. A periodic boundary suggests the values above the top edge of the grid are the values along the bottom edge of the grid, and vice versa. This could be extended to a more complex mapping. Thus, there is a very regimented manner in which the values are computed, but those values will change over time. Once again, these can be implemented either with explicit ghost zones or with stencils designed for the boundaries, or as simple as a stencil using a modulo operator for the neighbor calculation. When implemented using ghost zones, the values must be updated every time a boundary node is updated.

On distributed memory machines, the nodes assigned to one computer are not directly addressable by a different computer. Figure 5.10 on the following page shows the standard solution of introducing a ghost zone around each computer's portion of the grid. Figure 5.10(b) shows part of the operation. At each time step, each computer sends its boundary to the "neighboring" computers. In turn, they receive the boundaries of the neighboring computers' grids. They then copy this data into their ghost zones. Thus, the values in these ghost zones vary in space and time. Depending on what constitutes a "neighboring" computer, one could implement a periodic boundary through this technique.

The final category is perhaps appropriately labeled as "other." It includes boundaries that must be locally recalculated, not simply copied, at every point in space and at every point in time — a driving function. Typically, they are implemented with ghost zones.

A given structured grid may include one or more of these boundaries. For example in the parallel, distributed memory case, processors on the boundary of the problem could have both ghost zones from parallelization and constant boundary condition ghost zones.

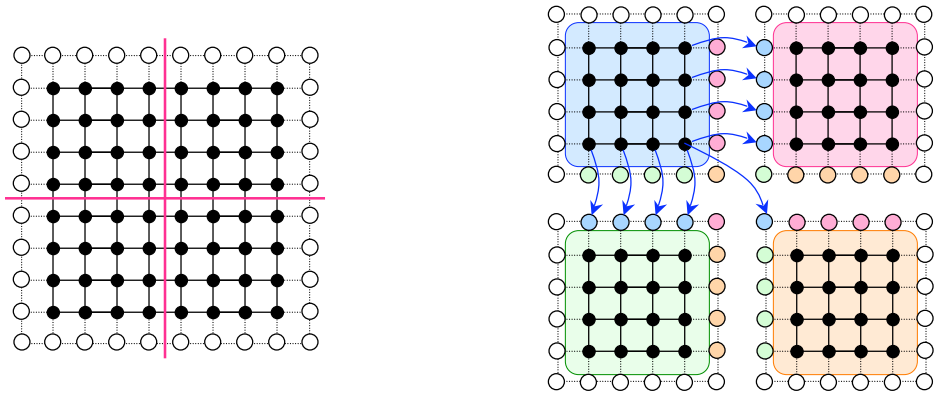


Figure 5.10: Ghost zones created for efficient parallelization.

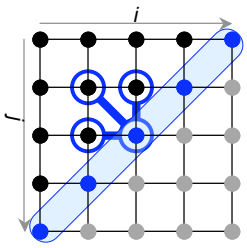
Initial conditions refer to the initial values of the nodes. For some problems, this data can be left uninitialized. However, for most problems it must be seeded with specific or reasonable values. We assume setting the grid for the initial conditions can be sufficiently amortized over the execution of a method. As such, the time required for initial conditions does not significantly adversely affect performance. Thus, we do not include this time in any performance calculation.

5.2.3 Code Structure and Parallelism

The basic building block of a structured grid code is a stencil. Stencils are then grouped together into grid sweeps. In such a sweep, all of the nodes are updated using the appropriate stencil operator. Structured grid codes often perform a large number of such sweeps. In codes like parabolic or hyperbolic PDEs, sweeps simulate the time evolution of a problem. In elliptic PDEs, they are used for convergence. In multigrid or graphics codes they can coarsen or restrict the grid to a different resolution. Finally, for some dynamic programming codes, sweeps are used for different boundary conditions. Typically, between each complete update of a grid, the ghost zones are updated in accordance with their type. Rather than defining a new set of cross-domain classifications, we typically use the names from scientific computing and note when the concepts apply to structured grid codes from other domains.

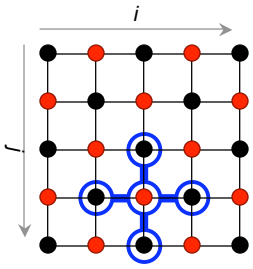
Inherent parallelism can vary widely from one sweep to another. Consider five common examples: Gauss-Seidel method, upwinding stencils, Red-Black Gauss-Seidel method, Jacobi's method, and the restriction and prolongation pair found in multigrid. We focus on the code structure and inherent parallelism of these methods, not the numerical stability or convergence time.

In the pure Gauss-Seidel method, there is only one grid. Moreover, the structure of the typical stencil mandates that there is data hazard between stencils. As such, there is one correct ordering in which the stencils must be applied. Critically, for some stencils




```
1  while(!done){
2    // sweep thru diagonals
3    for(d=0;d<5;d++){
4      for(j=0;j<d;j++){
5        i = d-j;
6        grid[i,j] = stencil(grid,i,j);
7    }}
8  }
```

Figure 5.11: Visualization of an upwinding stencil. the loop variable "d" denotes the diagonal as measured from the top left corner. Note each diagonal is dependent on the previous two diagonals. Black nodes have been updated by the sweep, gray ones have not.



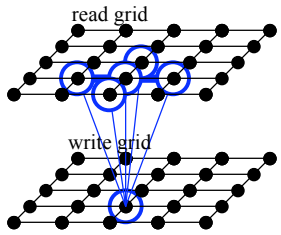

```
1  while(!done){
2    // for all colors, i, j
3    for(red=0;red<2;red++){
4      for(j=0;j<5;j++){
5        for(i=0;i<5;i++){
6          NodeIsRed = (i+j)&1;
7          // node color == sweep color?
8          if( red==NodeIsRed ){
9            grid[i,j] = stencil(grid,i,j);
10 }}}}}
```

Figure 5.12: Red-Black Gauss-Seidel coloring and sample code.

there is no parallelism within each sweep, as all successive stencil updates are dependent on the current one.

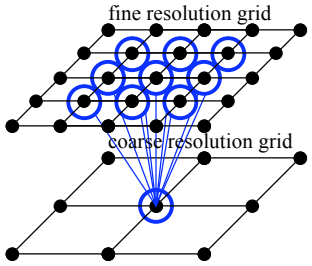
However, if the stencil used in a Gauss-Seidel-like method were ideally constructed, then there are some loop orderings that avoid the data dependencies. As such, those loops could be parallelized. Figure 5.11 shows such a stencil, similar to those in upwinding and dynamic programming [98, 117] codes. Note, there is only one grid, and the stencil only looks backward. Diagonals are enumerated from the top left corner. Thus, there is a clear read-after-write data hazard when looking at previous diagonals. However, there are no data hazards among stencils along a diagonal. As such, all nodes along the highlighted diagonal can be executed in parallel. In higher dimensions, it is possible that an entire plane could be executed in parallel.

The Red-Black Gauss-Seidel method is applicable to certain stencils. The mathematics behind the Red-Black Gauss-Seidel method is slightly different than the normal Gauss-Seidel. In contrast, the nodes of the grid are colored into two or more colors. Figure 5.12 shows such a coloring on a 2D rectangular Cartesian grid. One could alternatively color the nodes by rows, columns, or planes. Notice the neighbors of a 5-point stencil centered on a black node are all red, and the neighbors of a red node are all black. As such, one could restructure the loops in a Red-Black Gauss-Seidel sweep to update all black nodes, then update all red nodes. In doing so, all data hazards within a sweep are eliminated. Thus, the nested for loops on lines 4-5 of Figure 5.12 can execute in any order or entirely




```
1  while(!done){
2    // sweep thru grid
3    for(j=0;j<5;j++){
4      for(i=0;i<5;i++){
5        write[i,j] = stencil(read,i,j);
6      }}
7      // swap read and write pointers
8      temp=read;read=write;write=temp;
9  }
```

Figure 5.13: Jacobi method visualization and sample code. The stencil reads from the top grid, and writes to the bottom one.




```
1  // sweep thru grid
2  for(j=0;j<3;j++){
3    for(i=0;i<3;i++){
4      coarse[i,j] = stencil(fine,2*i,2*j);
5  }}
```

Figure 5.14: Grid restriction stencil. The stencil reads from the top grid, and writes to the bottom one.

in parallel. As a result, the parallelism available is half the total number of stencils.

The same mathematics could be alternately implemented with Jacobi's method. Figure 5.13 shows such a method in which two copies of the grid are maintained. One represents the current state of the grid, and the other is a working or future version. Like Red-Black, the benefit is the elimination of data hazards. To that end, one grid is read-only, while the other is write-only. Unlike Red-Black, any stencil can easily be applied. At the end of each sweep, pointers are swapped, and the working grid becomes the current grid. The for-loop nest on lines 3-4 of Figure 5.13 can be parallelized — that is, all stencils may be executed in parallel. The downside is that the total storage requirements are doubled.

Although the previously discussed Jacobi method utilized a second grid, the resolution of the two grids is the same. However, in many codes, a stencil is applied to coarsen or restrict the resolution of a grid, multigrid, or image resampling for example. Figure 5.14 shows such an example in which the resolution in both dimensions are coarsened by a factor of 2. In this example, there is no data dependency between stencils. As such, the loop nest on lines 2-3 of Figure 5.14 can be executed in parallel.

Table 5.1 on the next page compares the parallelism, storage, and spatial locality for each method given an N^3 input problem. Remember, one only needs enough parallelism for the machine one is running on. Methods with good spatial locality, low storage requirements, and enough parallelism will perform well. Moreover, in multigrid, the restriction and prolongation operators always appear together and in conjunction with one of the relaxation

Method	Parallelism	Storage Requirement	Spatial Locality
Gauss-Seidel	1	N^3	good
Gauss-Seidel (upwinding)	N^2	N^3	poor
Gauss-Seidel (red-black)	$\frac{N^3}{2}$	N^3	fair
Jacobi's Method	N^3	$2 \cdot N^3$	good
Restriction Operator	$\frac{N^3}{8}$	$1.125 \cdot N^3$	good
Prolongation Operator	$8 \cdot N^3$	$9 \cdot N^3$	good

Table 5.1: Parallelism, Storage, and Spatial Locality by method for a 3D cubical problem of initial size N^3. In multigrid, the restriction and prolongation operators always appear together and in conjunction with one of four relaxation operators.

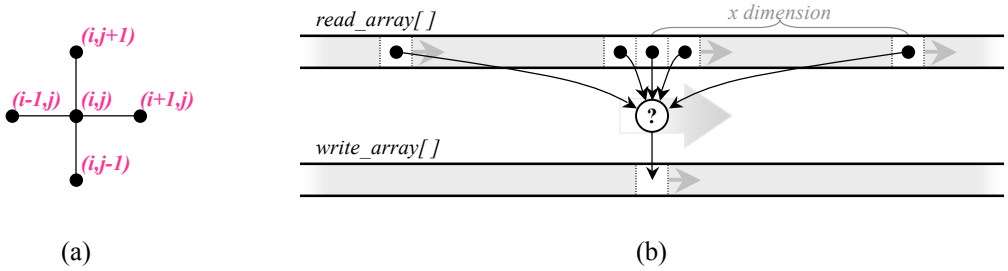


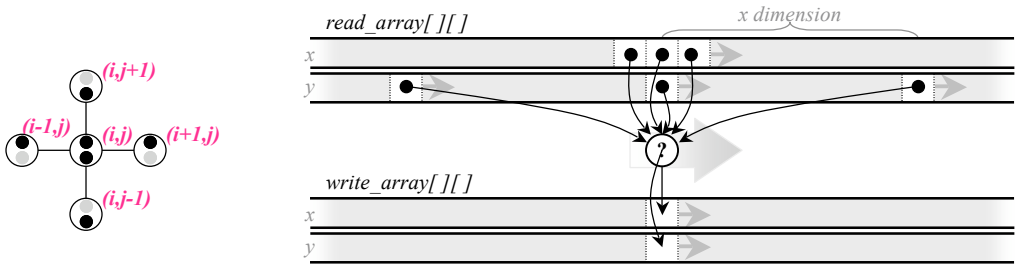
Figure 5.15: A simple 2D 5-point stencil on a scalar grid: (a) Conceptualization of the stencil as seen in 2D space, (b) mapping of the stencil from 2D space onto linear array space. The stencil sweeps through the arrays from left to right maintaining the spacing between points in the stencil. Scanning along a component, it is clear there is both and temporal and spatial locality.

operators.

5.2.4 Memory Access Pattern and Locality

A grid traversal is often either ordered by topological dimensions or by the enumerated index. When combined with the data storage format, the memory access pattern of the stencil can have dramatic impacts on performance. In this section, we examine the memory access patterns and cache locality for three generic stencil types: stencils on scalar grids, stencils on vector grids, and stencils on lattice distributions. For simplicity, we limit ourselves to 2D problems on rectangular Cartesian grids and allow the reader to contemplate more complicated problems.

Figure 5.15 shows the memory access pattern for a 2D 5-point stencil on a scalar rectangular Cartesian grid using the Jacobi method. When mapped to the linear addresses of main memory, the points of the conceptual stencil are all within a single array. The



(a) (b)

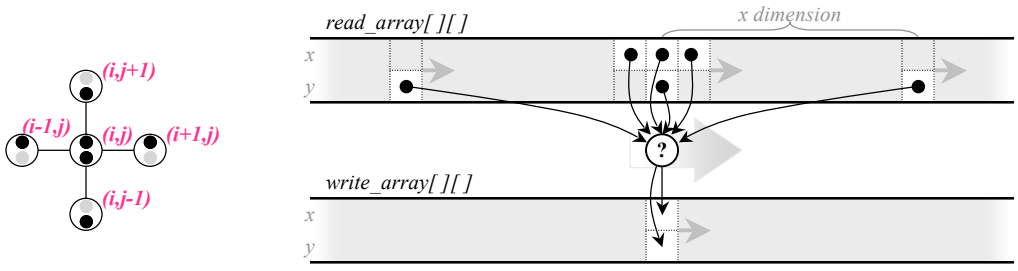
Figure 5.16: A simple 2D 5-point stencil on a 2 component vector grid: (a) Conceptualization of the stencil with the vector components superimposed on it as seen in 2D space, (b) mapping of the stencil from 2D space onto linear array space (structure-of-arrays). The stencil sweeps through the arrays from left to right maintaining the spacing between points in the stencil. Scanning along a component, it is clear there is both and temporal and spatial locality.

dimensions of the grid dictate the separation of the points of the stencil in main memory. These offsets are constant for all nodes in the grid. As the conceptual stencil sweeps through the grid, the stencil in main memory will sweep from left to right through progressively higher addresses. Observe that the address touched by the leading point in the stencil will be subsequently reused by all other points in the stencil. As such, if the cache capacity is sufficiently large (twice the x-dimension), then it will remain in the cache and no capacity misses will occur. In three dimensions, the distance between the leading and trailing points in the stencil may be twice the plane size. Such sizes may present challenges to cache capacities.

Figure 5.16 extends Figure 5.15 to a vector grid. The data is stored in a structure-of-arrays format. That is, the x and y components are stored in disjoint arrays. Those arrays, although separate in memory, have had the addresses of their first elements aligned to each other in the figure. Once again, in the linear address space, the stencil will sweep from left to right. Observe that little cache capacity is required to capture the temporal locality associated with the x component. However, substantial cache capacity is still required to capture locality of the y component.

Figures like Figure 5.16 motivate exploration of different storage formats. Figure 5.17 on the following page shows the same stencil if the data were stored in an array-of-structures, as one would expect if the data were RGB pixels. Clearly, there is a lack of spatial locality. The leading element of the stencil would only touch the y component. The cache hierarchy will load the corresponding x, but it will not be used immediately. Sufficient cache capacity must be present to keep it in the cache until needed; that is, sufficient cache capacity to avoid capacity misses.

Alternately, one might consider storing the data by diagonals rather than by rows (diagonal-major rather than row-major). It could remain a RGB-friendly array-of-structures



(a) (b)

Figure 5.17: A simple 2D 5-point stencil on a 2 component vector grid: (a) Conceptualization of the stencil with the vector components superimposed on it as seen in 2D space, (b) mapping of the stencil from 2D space onto linear array space (array-of-structures). The stencil sweeps through the arrays from left to right maintaining the spacing between points in the stencil. Scanning along a component, it is clear there is temporal, but poor spatial locality.

format, but the ordering of pixels must change. One would then traverse the grid by diagonals. Observe two diagonally adjacent stencils exhibit good spatial locality. Such an approach would likely result in significantly lower cache requirements.

Figure 5.18 on the next page shows the memory access pattern for a lattice method's `collision()` function's 2D 9-point stencil on a lattice distribution. Observe not only are significantly more arrays accessed, but within each array there is no temporal locality. Once an element is accessed, it is never used again. As the arrays are disjoint in memory, it is likely that a different TLB entry must be allocated for each array. Although many architectures have sufficiently large TLBs for 2D lattice methods, as dimensionality increases, the requisite number of entries grows rapidly.

When comparing Figures 5.15 through 5.18, one should observe there is less and less potential reuse of data. In Figure 5.15 the data used by the leading point in one stencil will eventually be reused by four other stencils. In Figure 5.18 it is clear that there is no reuse of data between stencils. As such, one should be extremely concerned about the appropriate cache blocking on scalar grids and virtually oblivious of it on lattice methods.

5.3 Methods to Accelerate Structured Grid Codes

A number of techniques have been developed to accelerate the time to solution on structured grid codes. These can be divided into implementation changes and algorithmic changes. An implementation change simply changes the loop structure, but overall, performs exactly the same operations. Algorithmic changes will dramatically change the number of operations required.

In this section, we discuss four different strategies. The first two, cache blocking and time skewing, are implementation-only optimizations in which the loops are restruc-

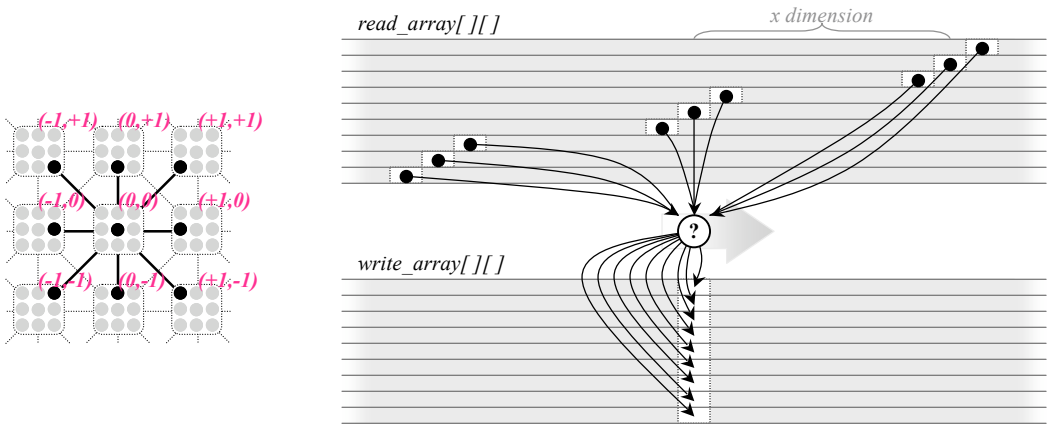


Figure 5.18: A simple D2Q9 lattice: (a) Conceptualization of a grid with the lattice distribution superimposed on it as seen in 2D space, (b) mapping of the lattice-stencil from 2D space onto linear array space. The stencil sweeps through the arrays from left to right maintaining the spacing between points in the stencil. Scanning along a component, it is clear there is good spatial locality, but no temporal locality.

tured to improve performance. The last two we discuss all make algorithmic changes: multigrid and adaptive mesh refinement.

5.3.1 Cache Blocking

Consider a stencil sweep. As discussed in Section 5.2.4, a minimum cache capacity is required to avoid capacity misses. If we consider the example of stencil sweeps in Section 5.2.3, it is possible for the codes in which the loop nests can be parallelized to block the loops in a manner that maintains a useful working set in the cache. This is analogous to the well known cache blocking techniques applied to dense matrix-matrix multiplication. In two dimensions, only the unit-stride loop needs to be blocked. However, in 3D either the unit-stride, the middle dimension, or both are blocked to maintain a cache-friendly working set. Although individual stencils may be executed in a different order, neither the total number of such stencils nor the resultant values are different. Ultimately, one might choose to enumerate the data differently so that for a natural traversal, good cache behavior is guaranteed.

5.3.2 Time Skewing

Cache blocking only blocks the spatial loops within a single sweep. However, if we take a step back, and incorporate the time or iteration loop in the stencil kernel, then we may choose to block this loop as well. In essence, this is blocking in space-time. Thus, once the nodes of a subgrid are in the cache, they are advanced several time steps. This can dramatically increase the arithmetic intensity. However, this technique is ultimately

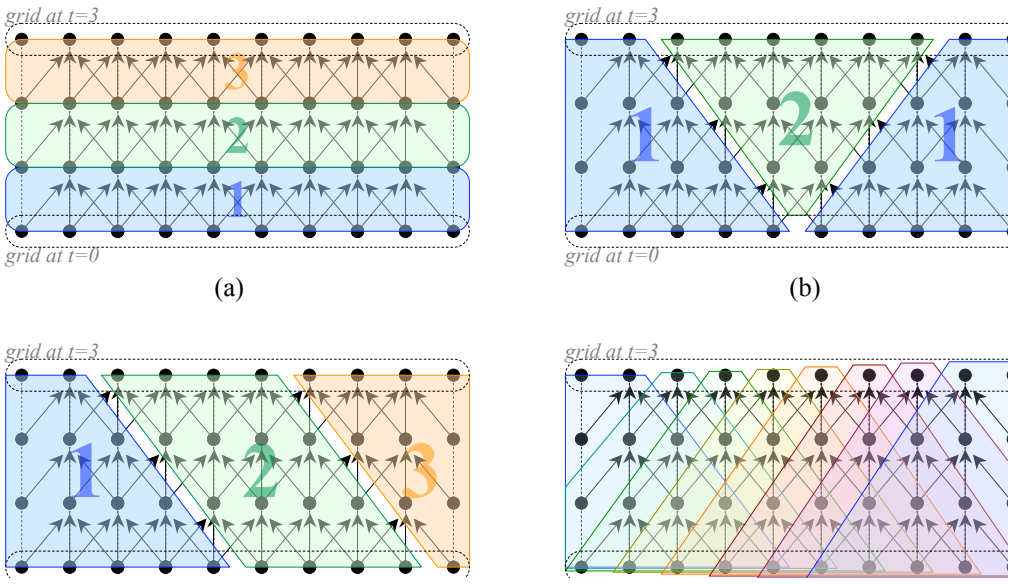


Figure 5.19: Visualization of time skewing applied to a 1D stencil: (a) Reference implementation where an entire sweep is completed before the next it started, (b) one style of time skewing tessellates space-time into non-overlapping trapezoids. Clearly, some can be executed in parallel. (c) Other time skewing approaches tessellate space-time into both trapezoids and parallelpipeds. Clearly, there is a dependency that prevents parallelization. (d) The circular queue approach creates small auxiliary structures and tessellates space time into overlapping trapezoids. Although easily parallelized, some work is duplicated.

limited by the bandwidth to the cache and the in-core performance. We use *time skewing* as a blanket term that covers a number of such implementation techniques. Figure 5.19 shows the most common approaches to time-skewing. Figure 5.19(a) shows naïve grid traversals. If the cache is smaller than the grid, then the first points updated by sweep "1" will have been evicted from the cache by the beginning of sweep "2."

Cache oblivious codes have been made famous with FFTW [52]. They tessellate the data and computation and traverse it in a recursive ordering. This ordering can be achieved either with recursive function calls or with a code generator that unfolds the recursion into straight-line code. The memory references in the straight-line code maintain the ordering of a recursive traversal. Cache oblivious algorithms have been applied to structured grid codes [50, 81]. These codes tessellate space-time into trapezoids and parallelepipeds. They then traverse them in a recursive ordering. However, the complexity of the traversal typically negates the reduction in cache misses. As such, they often run slower. The first level of this recursion is shown in Figure 5.19(c).

Cache aware implementations [144, 93, 116, 81, 121] maintain the trapezoid and parallelepiped tessellation of space-time, but abolish recursion in favor for complex loop

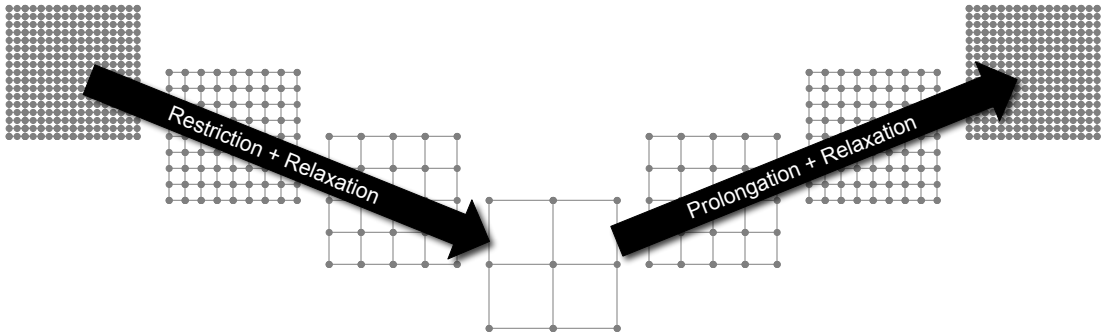


Figure 5.20: Example of the multigrid V-cycle.

nests. In the simplest case [35], only non-overlapping trapezoids are employed — Figure 5.19(b). Although the code is very simple and easily parallelized, its complexity dramatically increases as the number of dimensions that are blocked increases. The more complex code shown in Figure 5.19(c) uses parallelepipeds but is not easily parallelized. Figure 5.19(d) tessellates space-time into overlapping trapezoids that will be executed in parallel through the use of temporary arrays. Clearly, redundant work will be performed. In the sparse and unstructured grid world, these methods are generalized as A^k methods [40].

5.3.3 Multigrid

Multigrid [22, 24] has become a popular solution for accelerating structured grid problems. Like time skewing, it takes a holistic view of the structured grid code rather than a narrow view of only a grid sweep. In essence, one could solve a coarser grid and use that as a starting point for the fine resolution grid. Multigrid applies this recursively in what is known as a *V-cycle*. As one travels down the V-cycle, a series of *restriction* operators are applied that progressively coarsen the grid by cutting the resolution in half. In addition, at each level, one or more relaxation sweeps are applied to improve the solution. On the way back up the V-cycle, a series of *interpolation* operators return the grid to its original fine resolution. Let $\mathbb{N}$ denote the total number of points in the discretized space of D dimensions. Although the solution at all $\frac{log(\mathbb{N})}{D}$ stages of the V-cycle must be stored, they are geometrically smaller at each level. As such, both the storage and computational requirements of multigrid are linear in the number of nodes at the fine resolution. Such approaches dramatically reduce the number of floating-point operations from being proportional to the number of nodes squared ($\mathbb{N}^2$) in Red-Black Gauss-Seidel or $\mathbb{N}^{1.5}$ in successive over-relaxation (SOR) to being proportional to the total number of nodes in the grid — $O(\mathbb{N})$.

Figure 5.20 shows the multigrid V-cycle. Clearly, two new stencils must be introduced. The *restriction* stencil takes the grid at a finer resolution and coarsens it, where the *interpolation* or *prolongation* stencil takes a coarser grid and interpolates it onto a fine grid. It is possible to combine the relaxation sweeps to improve cache behavior [116]. One could extend this by combining it with the restriction or prolongation stencils.

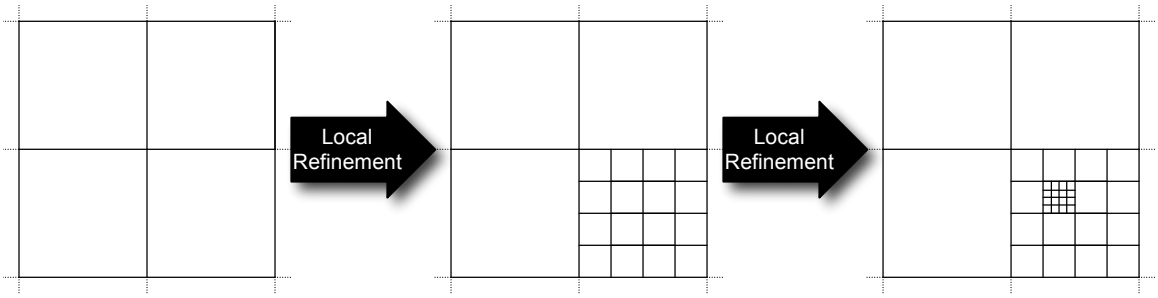


Figure 5.21: Visualization of the local refinement in AMR. Only three levels of refinement are shown.

5.3.4 Adaptive Mesh Refinement (AMR)

Consider that there are problems of such enormous scale that no machine could ever store a uniform, fine resolution grid. Thus, despite multigrid's algorithmic and storage advantages over conventional approaches, it still requires the storage of a fine resolution grid. As such, there are situations where it cannot be used. Adaptive mesh refinement (AMR) [14] is a novel approach that locally adapts the grid resolution as needed. In doing so, it can simulate problems for which no fine grid could ever be stored. Often, when regridding, the resolution is increased by a factor of 32 rather than the factor of 2 associated with multigrid. This creates tremendous storage and load balancing challenges on a distributed memory machine. Remember, grids twice as fine must take time steps half as big [33]. Thus, creation of a 32^3 subpatch on an existing 32^3 grid will require $32\times$ as much computation as the original grid. These fine grids must be dynamically and recursively created and destroyed as needed.

Figure 5.21 visualizes the local mesh refinement that is present in AMR codes. The grid cells at the finest resolution are updated four times for every update at the middle resolution and 16 times for every update of the coarsest resolution. In practice 10 or more levels of refinement are often seen.

5.4 Conclusions

In this chapter we provided an overview of the breadth of the structured grid motif by first discussing many of the common characteristics of its kernels. The most important characteristics discussed in Section 5.1 and Section 5.2 are the uniform topology, topological dimensionality, topological periodicity, data storage, and computation. The code structures of the kernels within the motif are broadly similar and are characterized by grid sweeps of a common stencil. However, the parallelism and storage requirements within such a sweep can vary widely. We may view structured grid computations as a restricted DAG with these particular characteristics. The edges of the DAG represent the gather operation of a stencil, and the nodes represent both computation and the resultant storage. However, it is much easier to construct a DAG after the fact based on these restrictions than attempt to infer or detect them by inspecting the DAG.

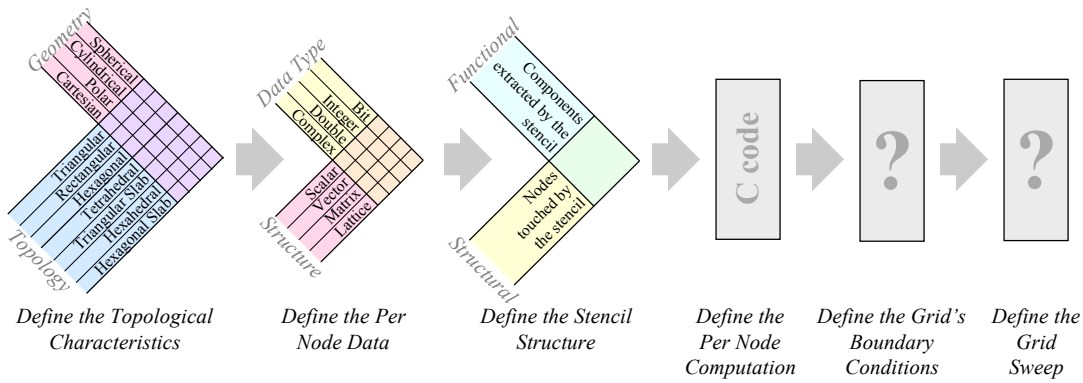


Figure 5.22: Principle components for a structured grid pattern language.

Figure 5.22 presents the primary components of pattern language for describing a structured grid kernel. One may start at the high level by synthesizing together the node valence, topological dimensionality, and periodicity to describe the grid. One then augments this description by specifying the data stored at each node both in terms of the data type and the data structures. Now that the grid is described, we must describe the per node computation. First, we describe the stencil both in its structural node connectivity and the data it must access from each of said nodes. Once this data has been gathered, one then specifies the computation as if it were entirely local operating on a number of input parameters. Although per node data and computation is described for the interior of the grid, we must also describe the boundary conditions. Finally, we must describe the collective method that sweeps stencils through the grid. This can range from the simplest Gauss-Seidel to the most complex upwinding stencils.

Unfortunately, a detailed pattern language alone doesn't ensure performance. As the sizes of the grids are very large, they do not remain in cache between sweeps. Moreover, given typical cache sizes, they can be so large that any inherent reuse within one sweep cannot be exploited using a naïve traversal. When coupled with stencil computations amounting to nothing more than simple linear combinations, grid sweeps invariably have low arithmetic intensities. As such, they often generate many capacity misses, are bandwidth-limited, and deliver low performance. Section 5.3 provided an overview of several common and novel techniques that improve the time to solution for structured grid codes, the simplest of which eliminate capacity misses. As the complexity of the per node data structure increases, a similar technique should be applied in component or velocity space rather than grid (physical) space. Acceleration techniques progress to the point of taking a holistic view of structured grid methods, rather than the narrow view of sweep optimization. In doing so, they may restructure loops to dramatically improve arithmetic intensity. In doing so, one sacrifices the ability to inspect the grid between sweeps as these values are now considered temporaries. Ultimately, the acceleration techniques can make algorithmic changes that drastically reduce the total number of floating-point operations required for the method.

In Chapter 6 we use the insights gained from this case study to successfully ap-

ply auto-tuning to a structured application: Lattice Boltzmann Magnetohydrodynamics (LBMHD). Succinctly, Chapter 5 provides the fundamental knowledge required to understand Chapter 6.

Chapter 6

Auto-tuning LBMHD

This chapter presents the results of extending auto-tuning to multicore architectures and the structured grid motif. To that end, we select the Lattice Boltzmann Magnetohydrodynamics (LBMHD) application as it will require a super set of optimizations likely required by smaller kernels. Although we see that auto-tuning provides a performance portable solution across cache-based microprocessors, it is clear that compilers cannot fully exploit the power of existing SIMD ISAs. Thus, architecture specific optimizations still provide a further boost to performance.

Section 6.1 delves into a case study of LBMHD. Section 6.2 uses the Roofline model introduced in Chapter 4 to estimate attainable LBMHD performance, as well as enumerate the optimizations required to achieve it. Section 6.3 walks through each optimization as it is added to the search space explored by the auto-tuner. At each point, performance and efficiency are also reported and analyzed. In addition, the final fully-tuned performance is overlaid on the Roofline model. Section 6.4 summarizes, analyzes, and compares the performance across architectures. In addition, a brief discussion of productivity is included. Although significant optimization effort was applied in this work, Section 6.5 discusses a few alternate approaches that may be explored at a later date. Finally, Section 6.6 provides a few concluding remarks.

6.1 Background and Details

In our examination of auto-tuning on structured grids presented in this chapter, we chose to restrict ourselves to lattice methods as they will likely show a great diversity in the optimizations required. To that end, we chose Lattice Boltzmann Magnetohydrodynamics (LBMHD) [90] as an example lattice method and extend the work presented in [139]. This section performs a case study LBMHD, and the rest of the chapter is dedicated to the study of auto-tuning LBMHD on multicore architectures.

Although superficially similar to simple differential operators, Lattice Boltzmann methods (LBM) form an important and distinct subclass of structured grid codes. They emerged from the use of statistical mechanics to develop a simplified kinetic model designed to maintain the core physics while reproducing the statistically averaged macroscopic quantities [126]. The popularity of the application of LBM to computational fluid dynamics

Kernel	Topological Parameters	Node Parameters	Boundary Conditions	Sweep
`collision()`	Vertex Valence: Hexahedral Geometry: Cartesian Domain: Cubical	Data: D3Q27 Lattice (SOA) Stencil: 27-point Computation: nonlinear operator	Periodic (via ghost zones)	Jacobi's Method

Table 6.1: Structured grid taxonomy applied to LBMHD.

(CFD) has steadily grown due to their flexibility in handling irregular boundary conditions. Recently, LBM has been extended to magnetohydrodynamics (MHD) [91, 39].

LBMHD was developed to study homogeneous isotropic turbulence in dissipative magnetohydrodynamics (MHD) — the macroscopic interaction of electrically conducting fluids with an induced magnetic field. MHD turbulence plays an important role in many branches of physics [17] from astrophysical phenomena in stars, accretion discs, interstellar and intergalactic media to plasma instabilities in magnetic fusion devices. There are three principal macroscopic quantities of interest at each point in space: density (a scalar), momentum (a Cartesian vector), and the magnetic field (also a Cartesian vector).

Table 6.1 uses the structured grid taxonomy introduced in Chapter 5 to describe LBMHD's `collision()` operator. Although the topological parameters are rather mundane, the complexity of the node parameters is the source of the challenge.

As LBMHD couples computational fluid dynamics (CFD) with Maxwell's equations, two (phase space) distribution functions are required. The first is a momentum distribution arising from the CFD part of the physics to reconstruct density and momentum. The second is a Cartesian vector distribution function included to reconstruct the additional macroscopic quantity — the magnetic field. As the magnetic field may be resolved with only the first moment, only 15 (velocities 12 through 26) discrete velocities are required. These are enumerated in Figure 6.1(c). For simplicity, a D3Q27 quantization is used for both the momentum and magnetic distributions, although only the relevant subset of velocities are stored and computed for the latter. LBMHD only simulates a 3D hexahedral Cartesian volume with periodic boundary conditions despite the ease with which LBM methods can be implemented with complex boundary conditions and geometries.

Figure 6.1 on the following page illustrates how LBMHD is applied to a 3D volume. For every point in space 6.1(a), two higher dimensional lattice distribution functions are stored: momentum 6.1(b), and magnetic 6.1(c). Thus, to reconstruct three macroscopic quantities of interest — density, momentum, and the magnetic field — an additional 27 scalar and 15 Cartesian vector quantities must be stored and operated upon. Tallying this up, over 1 KB of storage is required for every point in space. This means a 64^3 problem requires about 330 MB, where a 128^3 requires more than 2.5 GB; far more than a QS20 Cell blade can accommodate.

6.1.1 LBMHD Usage

LBMHD has been extensively used for MHD simulations. In fact, Figure 6.2 on page 109 is reproduced from one of the largest 3D LBMHD simulations conducted to date [29]. The goal was to further understanding of the turbulent decay mechanisms

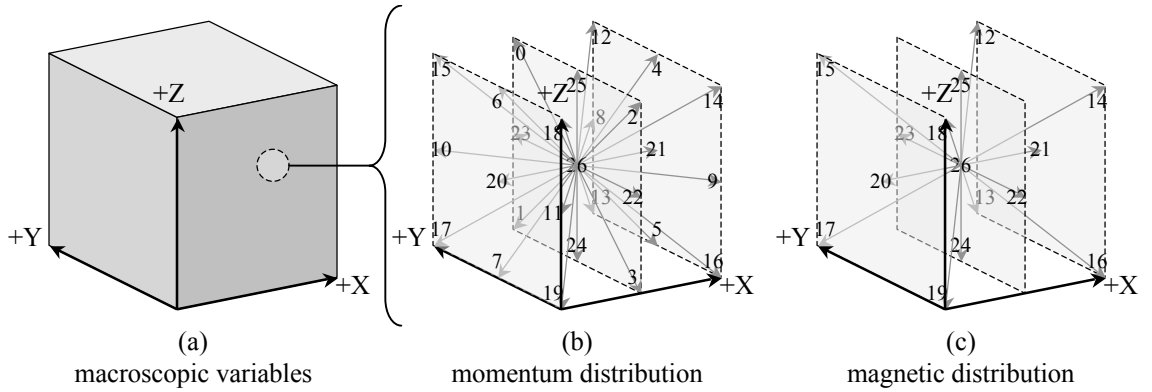


Figure 6.1: LBMHD simulates magnetohydrodynamics via a lattice boltzmann method using both a momentum and magnetic distribution. Note, each velocity in the momentum distribution is a scalar, and each velocity in the magnetic distribution is a Cartesian vector.

starting from a Taylor-Green vortex. This astrophysical simulation shows the development of turbulent structures in the z-direction.

6.1.2 LBMHD Data Structures

LBMHD was originally written in Fortran and parallelized onto a 3D processor grid using MPI. It used a Jacobi method structure-of-arrays approach — storing not only even and odd time steps separately, but also each velocity of each distribution. This approach achieved high sustained performance on the Earth Simulator, but a relatively low percentage of peak performance on superscalar platforms [102]. For this work, the application was rewritten in C using two different threading models to exploit our multicore architectures of interest. It retained the Jacobi approach, ghost zones, and periodic boundary conditions of the original.

As noted, to facilitate vectorization on the Earth Simulator, the previous LBMHD implementation utilized a structure-of-arrays approach. As that approach delivers good spatial locality and was easily vectorized, we extended that approach here. As seen in Figure 6.3 on the following page, each element of the data structure points to a conceptual 3D array surrounded by a ghost zone that we may pad or align as needed. To simplify indexing, the 36 (12 Cartesian vector pointers) unused lattice elements of the magnetic component are simply unused NULL pointers. As ghost zones are normally required, each N^3 3D grid is allocated as a $(N+2)^3$ 3D grid.

6.1.3 LBMHD Code Structure

Modern Lattice Boltzmann implementations have been restructured to perform gather rather than scatter operations [137]. Nevertheless, they still iterate between two phases during each time step. The first phase, `stream()`, handles ghost zone exchanges via a three phase exchange [104]. The second phase, `collision()`, evolves the local grid one step in time.

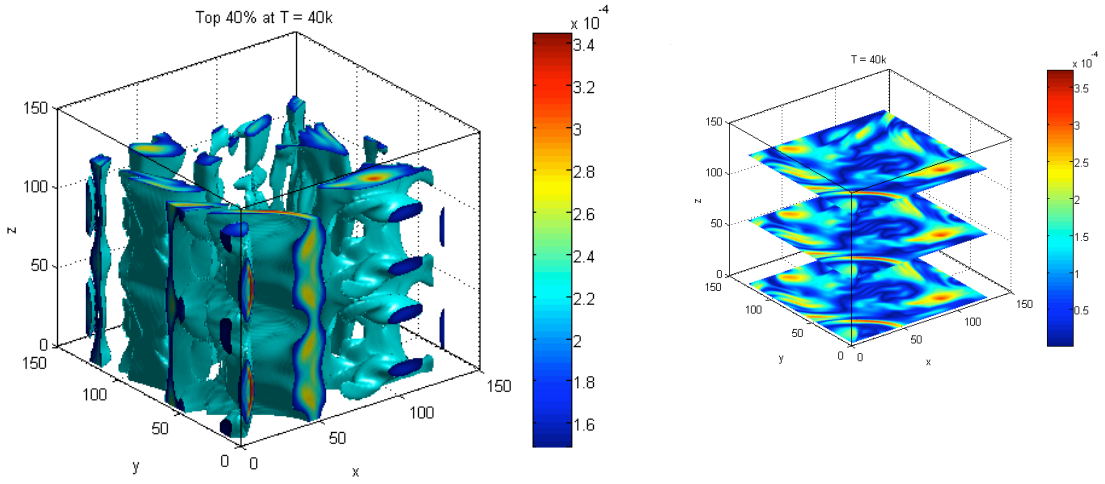


Figure 6.2: Visualization from an astrophysical LBMHD simulation. Figure reproduced from [29] which conducted simulations performed on the Earth Simulator.

```
struct{
  // macroscopic quantities
  double * Density;
  double * Momentum[3];
  double * Magnetic[3];
  // distributions used to reconstruct macroscopics
  double * MomentumDistribution[27];
  double * MagneticDistribution[3][27];
}
```

Figure 6.3: LBMHD data structure for each time step, where each pointer refers to a N^3 3D grid.

The `stream()` function simply extracts the outward directed velocities on the surface of the lattice distributions and packs them into buffers. It then performs the typical MPI `isend() / irecv()` to send this data to the conceptually neighboring processors. The function then unpacks the buffers into the inward directed velocities on the boundaries of the distributions. However, as this work examines only single node performance, all MPI calls were replaced with pointer swapping. That is, we retained the surface extraction into the MPI buffers, but rather than communicating among nodes, we implement periodic boundary conditions on a single node using pointer swapping.

The most interesting aspects of the code are within the `collision()` function. Figure 6.4 on the next page provides an overview of its structure. The code sweeps through all points in 3-space, and updates each individually. For the momentum distribution, a 27-point stencil is used, but for the magnetic distribution, a 15-point stencil is used. The

```
collision(...){
  for(all points in 3-space){
    // reconstruct macroscopic quantities:
    //   weighted reduction over all distribution velocities
    //   ~73 reads from DRAM
    //   ~7 writes to DRAM

    // update distributions:
    //   each is a function of the previous value and the macroscopics
    //   ~72 writes to DRAM
  }
}
```

Figure 6.4: The code structure of the collision function within the LBMHD application.

first sub-phase of the update involves reconstructing the three macroscopic quantities. Unfortunately, this involves a high volume of read memory traffic (in the form of a gather) for relatively few FLOPs serialized into a series of reductions. In the second phase, each velocity of both distributions is evolved individually. Each velocity update requires the recently reconstructed macroscopic quantities as well as the previous values of that velocity for both distributions. In addition, a few projection constants are used. This sub-phase performs the bulk of the nearly 1300 floating-point operations per lattice update, and writes about 600 bytes of data — making it relatively computationally intense. However, on some architectures, a FLOP:byte ratio of 2.25 is still memory bound. Thus, both phases might require comparable time despite the disparity in computation.

The memory access pattern is well visualized with the lattice example in Figure 5.18 on page 100. However, the number of arrays for LBMHD is far larger. Essentially, there are 73 read and 79 write arrays. As with most lattice methods, there is no reuse of the data from one stencil by any other stencil.

6.1.4 Local Store-Based Implementation

As code written for a conventional cache-based memory hierarchy cannot be run on Cell's SPEs, we wrote a local store-based implementation. As with the cache-based implementation, at the beginning of a time step, both the macroscopic quantities as well as the distributions are in the main memory. At the end of a time-step, the updates must be committed back to main memory. The difference is that processing within a time step is processed in three phases. First, the read data required for a lattice update is copied into the local store via DMA. Second, the lattice update is performed within the local store: reading from an input buffer and writing to an output buffer. Finally, the output buffer is copied back to DRAM via a DMA. As this Jacobi implementation can be readily parallelized within a time step, it is possible to overlap three phases associated with three different lattice updates. Thus, an SPE may load via DMA the data associated with the

next (in space) lattice update, compute within the local store the current update, and store via DMA the data for the previous lattice update. The downside to this DMA / computation pipeline is that the pressure on the local store is doubled.

6.2 Multicore Performance Modeling

Before attempting to run the LBMHD application, we chose to perform some rudimentary performance analysis based on the application and architectural characteristics. This analysis will not only provide performance expectations for each architecture, but will also offer insight into the capabilities and productivity challenges associated with each architecture. We choose only to model `collision()`. We believe that for sufficiently large problems, the `stream()` operator will constitute a small and manageable fraction of the execution time.

6.2.1 Degree of Parallelism within `collision()`

A lattice update is one iteration of the innermost spatial loop of the `collision()` function. It is divided into two phases: reconstructing the macroscopic quantities and advancing the velocities. There is very limited instruction-level parallelism (ILP) when reconstructing the macroscopic quantities as they take the form of scalar reductions. For any reasonably sized instruction window, the same is true when advancing the velocities in the second phase. We also observe that there is a significant imbalance between multiplies and adds, and fused multiply-add (FMA) cannot always be used. This imbalance may halve attainable performance on the x86, PowerPC, and Cell architectures.

For the baseline implementation, within a single lattice update, there is no data-level parallelism (DLP). DLP only arises from inter-lattice updates; that is, between points in space. As problems are composed of millions of points, there is multi-million-way DLP in the outer loop. This parallelism cannot be readily exploited in the original implementation without the compiler or programmer restructuring the loops. Loops must also be restructured to effectively exploit the rigid SIMD capabilities on the x86 and Cell architectures.

Although there is no explicit thread-level parallelism (TLP) in the original implementation, we may recast ILP — or more typically DLP — as TLP; essentially an OpenMP [103] inspired approach to loop parallelization implemented with `pthreads`.

Memory-level parallelism (MLP) is a more nebulous concept. As a structure of arrays implementation sweeps through each array in a unit stride fashion, we are only limited by the expressibility of the architecture in finding MLP. For cache-based architectures, we are limited by the load store queue to at most a few kilobytes of data. Hardware prefetching will likely not work as the number of load streams in the structure of arrays implementation is far too great. Restructuring the code should allow several TLB pages of data in flight, perhaps as many as eight. This limit arises from the fact that hardware prefetchers do not prefetch beyond page boundaries. Double buffered implementations on DMA-based architectures are limited by the size of the buffer. Thus, it should be possible to express over 100 KB of MLP per SPE on Cell.

collision() Implementation	Instruction-level Parallelism	Data-level Parallelism	Thread-level Parallelism	Memory-level Parallelism	Memory streams per thread
standard	≤7	≈1	≈N	O(N^3)	≈150
vectorized	≤7	≈VL	≈N^2	O(N^3)	≈7

Table 6.2: Degree of parallelism for a N^3 grid for both the naïve and vectorized `collision()` function.

Table 6.2 shows the degree of parallelism within the original and vectorized (Section 6.3.4) versions of the `collision()` function.

6.2.2 `collision()` Arithmetic Intensity

Chapter 4 introduced the Roofline performance model. As the Roofline model suggests, the performance of many kernels is a function of in-core performance, memory bandwidth, and arithmetic intensity. To perform one lattice update, `collision()` must read the neighboring 27 momentum scalars and 15 magnetic Cartesian vectors from main memory. In addition, it must read the macroscopic density. After performing about 1,300 floating-point operations, it must write the local 27 momentum scalars, 15 magnetic Cartesian vectors, and 7 macroscopic quantities back to main memory. Note that most caches are write-allocate. As a result, whenever a write miss occurs, the cache must first fill the cache line in question — that is, read the cache line from main memory. As a result, we expect `collision()` will generate at least 1,848 bytes of main memory traffic for every 1,300 floating-point operations — a FLOP:compulsory byte arithmetic intensity of about 0.70.

Note that the line fill on a write miss is superfluous in LBMHD where every byte in the cache line will be overwritten. For architectures that allow cache bypass or are not write allocate, we expect a FLOP:compulsory byte ratio of 1.07, or about 50% better. This optimization may be directly implemented on Cell via DMA, but requires a special cache bypass store instruction on the x86 architectures.

6.2.3 Mapping of LBMHD onto the Roofline model

Figure 6.5 on page 114 maps LBMHD's FLOP:compulsory byte ratio onto the Roofline performance model discussed in Chapter 4. From this figure, we should be able to predict performance and which optimizations should be important across architectures.

As a reminder, for each architecture there are three types of lines in the Roofline model:

- **In-core Ceilings** denote in-core FLOP rates with progressively higher levels of optimization.
- **Bandwidth Ceilings** denote memory bandwidths with progressively higher levels of optimization.
- **Arithmetic Intensity Walls** denote actual FLOP:byte ratios with progressively higher levels of optimization.

Combined, the ceilings place bounds on performance and constrain it to a region on a Roofline figure.

The red dashed lines in Figure 6.5 denote LBMHD's FLOP:compulsory byte ratio for write allocate architectures, while the green dashed lines mark the ideal (higher) LBMHD FLOP:compulsory byte ratio. The lowest bandwidth ceiling denotes unit-stride performance without any optimization. Out-of-the box LBMHD performance is expected to fall on or to the left of the red dashed vertical lines due to the potential for significant numbers of conflict or capacity misses reducing the arithmetic intensity. Performance should fall below the lowest diagonal, as the original memory access pattern is not unit stride.

6.2.4 Performance Expectations

Inspection of Figure 6.5 on the next page suggests the Clovertown will be heavily memory bound. The inherent ILP in LBMHD will likely be sufficient to ensure Clovertown remains memory bound. Although explicit SIMDization for computation will be unnecessary, SIMDization to facilitate the use of cache bypass intrinsics will likely improve performance. Depending on the sustained bandwidth, we expect to attain between 6 and 12 GFLOP/s with full optimization. This suggests we may trade concurrency for auto-tuned performance.

We see two very different cases between the two Opteron machines. The Santa Rosa Opteron will likely be heavily processor bound. This implies that optimal code generation is essential, but memory optimization is of lesser importance. Recall that there is an inherent imbalance between adds and multiplies in LBMHD. Thus, architectures relying on an even number of adds and multiplies to achieve peak performance will be at a disadvantage. As such, the Santa Rosa Opteron will be ultimately limited by the floating-point adder performance.

Figure 6.5 on the following page suggests Barcelona will likely require significant memory-level optimizations, especially NUMA, to achieve peak performance. Unlike the Clovertown, Barcelona requires either full ILP or full SIMDization with some ILP to attain peak performance. As with all x86 machines, it is possible to increase LBMHD's FLOP:byte ratio to deliver better performance — up to 8 GFLOP/s on Santa Rosa and up to 16 GFLOP/s on Barcelona — through the use of the cache bypass instruction.

On Victoria Falls, LBMHD maps to the region where both instruction and memory-level optimizations will be necessary. As 8-way multithreading is sufficient to hide the

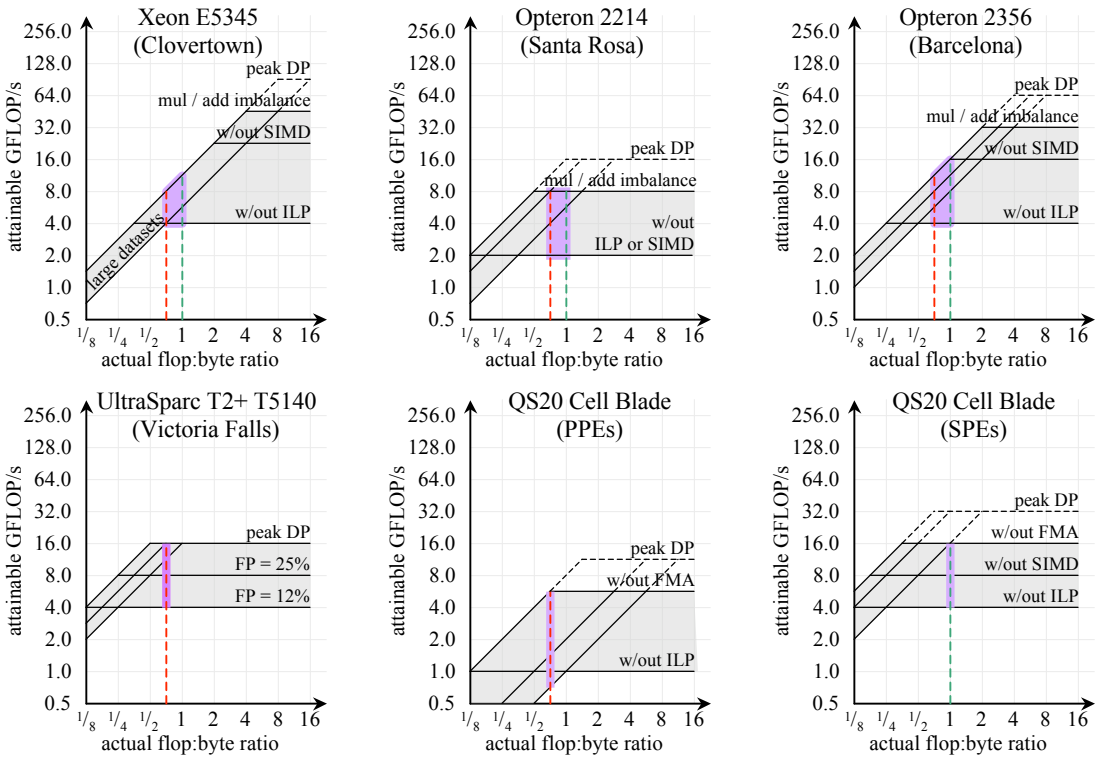


Figure 6.5: Expected range of LBMHD performance across architectures independent of problem size. Red dashed lines denote the realistic FLOP:compulsory byte ratios on write allocate architectures. Green dashed lines denote the ideal FLOP:compulsory byte ratios. The inherent lack of multiplies in the algorithm is noted as multiply/add imbalance is the effective roofline. Note the log-log scale.

latency of a 6 cycle FPU, we expect the challenge to be maximizing the fraction of floating-point instructions issued on this dual-issue architecture. The C compiler will do a reasonable job of ensuring non-floating-point instructions are used sparingly.

When using the PPEs on the Cell Blade, it is clear that both significant memory-level and instruction-level optimizations will be required. We expect the IBM XL/C compiler to efficiently unroll the loops, but experience suggests that without any hardware support for latency hiding, sustained PPE memory bandwidth will be near the lower diagonal. The optimized SPE version will clearly be different than any other architecture, as it lies completely outside the memory-level optimization region. Thus, we must discover sufficient ILP and fully SIMDize the code to achieve peak performance. As FMAs cannot be readily exploited in LBMHD, we expect peak performance to be near 16 GFLOP/s.

6.3 Auto-tuning LBMHD

There are a large number of optimizations to maximize LBMHD performance. Within each of these optimizations is a large parameter space. To efficiently explore this optimization space, we employed an auto-tuning methodology similar to that seen in libraries such as ATLAS [138], OSKI [135], and SPIRAL [97]. A large number of kernel "variations"are produced and individually timed. Performance determines the winner.

The first step in auto-tuning is the creation of a code generator. For expediency, we wrote a Perl script that generates all the variations of the `collision()` function. We also wrote additional architecture specific modules to generate SSE intrinsic laden C code variants. Note that the code generation and auto-tuning process has to date only been implemented for cache-based machines. The Cell implementation is a reasonably well optimized first attempt designed more for correctness rather than performance. Future work will extend this auto-tuning approach to Cell.

The Perl script used in the code generation process can generate hundreds of variations for the `collision()` operator. They are all placed into a pointer to function table indexed by the optimizations. To determine the best configuration for a given problem size and thread concurrency, we run a 20 minute tuning benchmark to exhaustively search the space of possible code optimizations. In some cases, the search space may be pruned of optimizations unlikely to improve performance. In a production environment, the optimal configuration found on one processor is applicable to all identically configured processors in the MPI SPMD version. As the auto-tuner searches through the optimization space, we measure the per time step performance averaged over a ten time step trial and report the best.

In the following sections, we add optimizations to our code generation and auto-tuning framework. At each step we benchmark the performance of all architectures exploiting the full capability of the auto-tuner implemented to that point. Thus, at each stage we can make an inter-architecture performance comparison at equivalent productivity, allowing for commentary on the relative performance of each architecture with a productive subset of the optimizations implemented.

6.3.1 `stream()` Parallelization

In the original MPI version of the LBMHD code, the `stream()` function updates the ghost-zones surrounding the subdomain held by each task with the surfaces of the neighboring tasks. Rather than explicitly exchanging ghost-zone data with the 26 nearest neighboring subdomains, we use the shift algorithm [104]. It performs a three phase exchange where faces are transfered after the first phase, edges after the second phase, and vertices after the third phase. Within each phase, each MPI task only communicates with two neighbors.

Communication with each neighbor requires transferring different subsets of the distributions. To facilitate an MPI implementation, the disjoint distribution faces are first copied into contiguous buffers. Then an MPI `isend()` and `irecv()` is initiated. Once the data is received, it is unpacked, and the process repeats for each additional phase. When auto-tuning, we do not call the MPI routines, but rather do a pointer swap. In effect, we

are implementing periodic boundary conditions via explicit copies.

Each point on a face requires 192 bytes of communication — 9 particle scalars, and 5 magnetic field vectors — from 24 different arrays. We maximize sequential and page locality by parallelizing across the velocities followed by points within each array.

Although `stream()` typically contributes little to the overall execution time, non-parallelized code fragments can become the limiting factor in Amdahl's Law. Thus, one would expect a serial implementation of `stream()` to severely impair performance on Victoria Falls.

6.3.2 `collision()` Parallelization

Although not required, all LBMHD simulations were performed on a cubical volume. Figure 6.6(a) on page 117 shows that `collision()` parallelization uses a 2D decomposition in the Y and Z dimensions. Although parallelization in the unit stride dimension is possible, it is often avoided as this can result in poor prefetch behavior [81]. Load balancing is guaranteed by specifying the per-thread problem dimensions and the number of threads in each dimension. Thus, the full problem size is the element by element product of thread size and the number of threads in each dimension. All benchmarks in this work use strong scaling — the full problem size remains fixed, but the number of participating threads increases. We impose two restrictions when benchmarking: all problem dimensions are powers of two and the number of threads in any dimension is a power of two. When auto-tuning we explore several possible combinations of threads in the Y and Z dimensions with the caveat that there cannot be more threads in any dimension than the problem size. As the total number of threads ($Threads_{YZ} = Threads_Y \times Threads_Z$) is a power of two, we can state that at most $1 + log(Threads_{YZ})$ combinations exist. Thus, exhaustive search along this axis is tractable.

As four of the machines in this work are non-uniform memory access (NUMA) architectures, we must ensure that data allocation is closely tied to the thread tasked with processing it. We rely on a first touch policy to ensure this affinity is guaranteed. To that end, we `malloc()` the data first, then create threads, and finally each thread initializes its piece in parallel. This approach works well when the parallelization granularity (array size per velocity) is much larger than the TLB page size. The arrays of a 64^3 problem are only 2 MB. For most architectures, 2 MB is significantly larger than the default page size. However, Solaris' use of 4 MB pages on the heap implies that grids may not be parallelized across sockets but will be pinned to one or the other. We expect 64^3 problem scalability to be poor beyond 64 threads on Victoria Falls.

Figure 6.7 on page 118 shows initial performance on the cache-based architectures before tuning as a function of the number of threads. Through the use of affinity and pinning routines, threads are ordered to exploit multithreading, then multicore, and finally multisocket parallelism. Note that all Victoria Falls data is shown for fully threaded cores. Note, initial performance is not naïve performance. It includes threading and NUMA optimizations on top of a rich history of LBMHD optimization [102, 29, 90, 137, 101, 139]. For most architectures, the general multicore and multisocket scaling trends are good, but we do see substantial differences in performance as problem size increases as well as between architectures. Table 6.3 on the following page notes the highest sustained floating-point

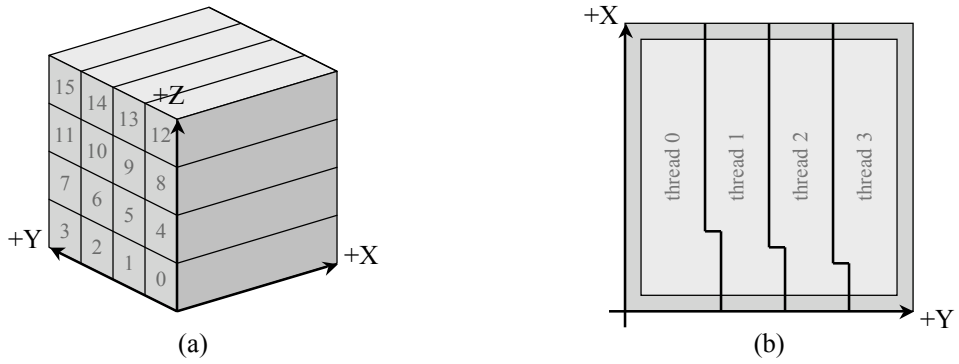


Figure 6.6: LBMHD parallelization scheme: (a) 2D composition of subdomains into a 3D domain, (b) skweing within a plane for alignment to cache lines and vectors.

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	(SPE)
GFLOP/s (% peak)	3.55 (4.8%)	2.46 (14.0%)	4.52 (6.1%)	6.65 (35.6%)	0.13 (1.0%)	—
GB/s (% peak)	5.04 (15.8%)	3.49 (16.4%)	6.42 (30.1%)	9.44 (14.8%)	0.18 (0.3%)	—

Table 6.3: Initial LBMHD peak floating-point and memory bandwidth performance. The Cell SPE version cannot be run without further essential optimizations.

and bandwidth performance, as well as percentage of machine peak, for each architecture. Note, bandwidth is calculated based on the FLOP:compulsory byte ratio. The relative performance seen here is a reasonable proxy for the performance that will be seen on a variety of applications without further optimization.

When examining Clovertown performance in Figure 6.7 on the next page, we see multicore scaling was nearly linear, indicating we are far from saturating a socket's frontside bus (FSB) bandwidth. When using the second socket, we see about 70% better performance, but clearly a drop in parallel efficiency. The dual-independent bus coherency protocol becomes noticeable when a second socket is used as the second socket will begin to generate snoop traffic on the first socket's FSB, and vice versa. Although the 5 GB/s of bandwidth is a small fraction of the raw DIMM bandwidth, it is a substantial fraction of the machine's effective FSB bandwidth of less than 10 GB/s.

The Santa Rosa Opteron, with only the inherent NUMA optimizations, shows very good scaling, but does not deliver the performance of the Clovertown. Of course, this is due primarily to the fact that the Clovertown is an eight core machine, where the Santa Rosa Opteron has only four cores. Comparing quad-core Barcelona performance on a core by core basis with Clovertown, we see a high correlation, until the FSB becomes Clovertown's bottleneck. Without SIMDization, the Opterons have a similar peak performance to that of the Clovertown resulting in similar per core performance for processor bound kernels. We also note that the Santa Rosa Opteron serendipitously achieves comparable utilization of memory bandwidth to that of the Clovertown.

When examining Victoria Falls performance, we see good scaling to 64 threads

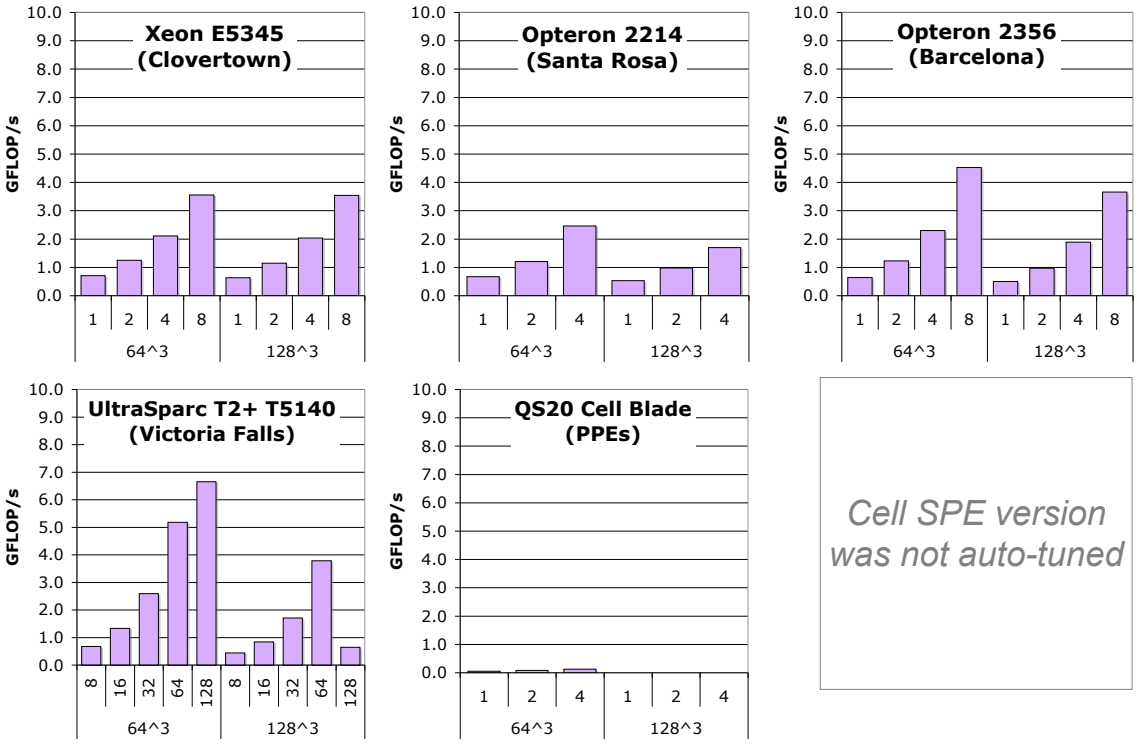


Figure 6.7: Initial LBMHD performance as a function of the number of threads ordered to exploit multithreading, then multicore, and finally multisocket parallelism.

or eight cores, but see little benefit to using the second socket on a 64^3 problem. This sub-linear scaling is due primarily to the interaction of the 4 MB default page size and the desire for parallelization on sub 2 MB granularities on this machine. Although Table 6.3 shows bandwidth utilization as a fraction of total SMP bandwidth, the fact that only one socket's memory controllers are effectively being used implies that the true utilization of the socket's memory controllers is nearly 30%. Section 6.3.8 describes a solution to this bottleneck. We also observe a substantial drop in performance on the large problem at full concurrency. Without accurate performance counter data, we can only speculate as to the cause. Clearly cache capacity is not the culprit as the working size is quite small, and doubling the sockets doubles the cache capacity. It seems more than likely that either limited cache associativity or some idiosyncratic behavior in the memory controllers is the culprit.

As the Cell SPE version cannot be run without significant further requisite optimizations, we initially only examine the Cell PPE performance. We believe examination of the PPE performance is essential not because we believe that it will be good, but rather it will be a limiting factor in a productivity version of Amdahl's Law — maximum productivity is achieved by only running code on the PPE, but maximum performance should be achieved by porting as much code as possible to the SPEs. When examining performance, it is clear that two-way in-order multithreading is wholly insufficient in satisfying the ap-

proximately 5 KB of concurrency required per socket by Little's Law (200ns × 25 GB/s). As a result, the Cell PPE version delivers pathetic performance even when compared to the other multi-threaded in-order architecture — Victoria Falls. It is clear that at the very minimum `collision()` will need to be ported to the SPEs.

6.3.3 Lattice-Aware Padding

LBMHD tries to maintain a working set of points in the cache through the first subphase of `collision()` so that during the second phase, all needed data is still present in the cache. Two major pitfalls may arise with this approach: the possibility of capacity misses, and the possibility of conflict misses. L1 cache working sets can be very small and may not be able to hold a full cache line per distribution velocity. In fact, given Victoria Falls' L1 cache line size of 16 bytes, and the desired working set of more than 150 cache lines, Victoria Falls' 1 KB per thread L1 working set ensures capacity misses will occur through the first phase. However, as the L2 caches on all machines are sufficiently large to avoid L2 capacity misses, L1 capacity misses may not severely affect performance. Conflict misses are a far more dangerous pitfall, as it is possible to thrash in the L1. Remember that a structure of arrays data structure is used for LBMHD. As a result, to update one point, 152 of these arrays must be individually and uniquely indexed. Given a lack of correlation between array addresses, it is quite possible that a conflict miss will occur on a low associativity L1 cache. Victoria Falls' tiny L1 ensures that capacity misses will hide the effects of conflict misses.

Before detailing the solution for LBMHD, let us examine the enlightening solution for a 7-point stencil. As seen from the memory access pattern in Figure 5.15 on page 97 of Chapter 5, there is a fixed offset between the five streams in memory — two are so widely separated they can't be shown on a single figure. Figure 6.8(a) on page 120 maps a cache to polar coordinates to maintain the inherent periodicity, where the angle represents set address, and concentric rings represent associativity. Arcs represent working sets filled by streams of points in a stencil. For any angle, there can never be more overlapping arcs than cache associativity — conflict misses will prevent this. The relative offsets in memory allow us to map each point in the stencil to a different angle in the cache as seen in Figure 6.8(b). The stanzas from streaming in memory are mapped to arcs in the cache. Ideally, we want the arc length to be the size of a plane — the distance between the leading and trailing point in the stencil. Failing that, we wish it to be a pencil (series of points in the unit stride dimension) — the distance between the 2nd and 6th point in the stencil. By padding each pencil and each plane with a few extra doubles, we change the angles corresponding to the mapping of stencil points to the cache from a pathological but common case to an optimal one. The padding that must be applied is a function of the array base, the relative offset arising from the stencil, and the desired position on the cache circle. This stencil-aware padding is a subtle but effective solution. When optimally applied, the resultant arc lengths shown for the 2-way cache in Figure 6.8(c) are 40% of the total cache size. Hopefully, this is sufficiently large to keep several pencils in the cache.

Padding for lattice methods can be significantly more beneficial than padding for stencils. To that end, we want to guarantee that the points accessed by the lattice during each phase are uniformly distributed around the cache circle. Thus, each array is

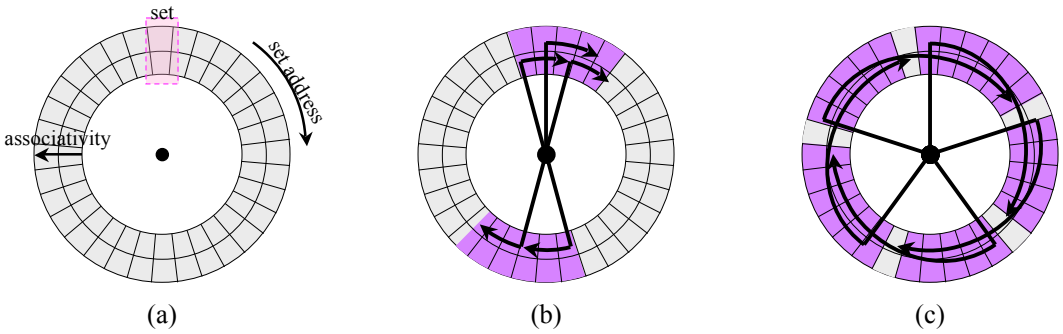


Figure 6.8: Mapping of a stencil to a cache: (a) A 2-way cache represented in polar coordinates, (b) mapping of the original stencil on a near power of two problem size results in poor cache utilization (purple highlighted region), (c) Padding uniformly distributes the points of the stencil in cache space results in good cache utilization (purple highlighted region).

Machine	**Xeon E5345 (Clovertown)**	**Opteron 2214 (Santa Rosa)**	**Opteron 2356 (Barcelona)**	**T2+ T5140 (Victoria Falls)**	**QS20 Cell Blade (PPE)**	**(SPE)**
GFLOP/s (% peak)	4.55 (6.1%)	3.55 (20.2%)	5.82 (7.9%)	6.65 (35.6%)	0.56 (4.4%)	—
GB/s (% peak)	6.46 (20.2%)	5.04 (23.6%)	8.26 (38.7%)	9.44 (14.8%)	0.78 (1.5%)	—
Speedup from this optimization	+28%	+44%	+29%	+0%	+330%	—

Table 6.4: LBMHD peak floating-point and memory bandwidth performance after array padding. The Cell SPE version cannot be run without further essential optimizations. Speedup from optimization is the incremental benefit from array padding.

padded such that `Array Base + Relative Offset + Padding` maps to the desired set on the cache circle. The complexity of LBMHD, with its 73 read arrays, precludes any attempt at drawing this mapping, but one can contemplate it based on the previous stencil example.

Figure 6.9 on the following page shows the performance when the lattice-aware padding heuristic is applied to LBMHD. We see a substantial increase in performance on all machines for which conflict misses are more likely than capacity misses; that is, all but Victoria Falls. It should be no surprise that Clovertown, with an 8-way cache, saw less of a benefit than the Opterons, with their 2-way L1 caches. In fact, for a 128^3 problem, the Santa Rosa Opteron performance doubles. More impressive, although difficult to see, the Cell PPE performance quadruples.

Table 6.4 shows utilization. We see that although all architectures still achieve both a low fraction of peak FLOPs and a low fraction of DRAM bandwidth. Although the Clovertown achieves a tiny fraction of its DRAM bandwidth, it achieves nearly 60% of its effective FSB bandwidth — the bottleneck in its design.

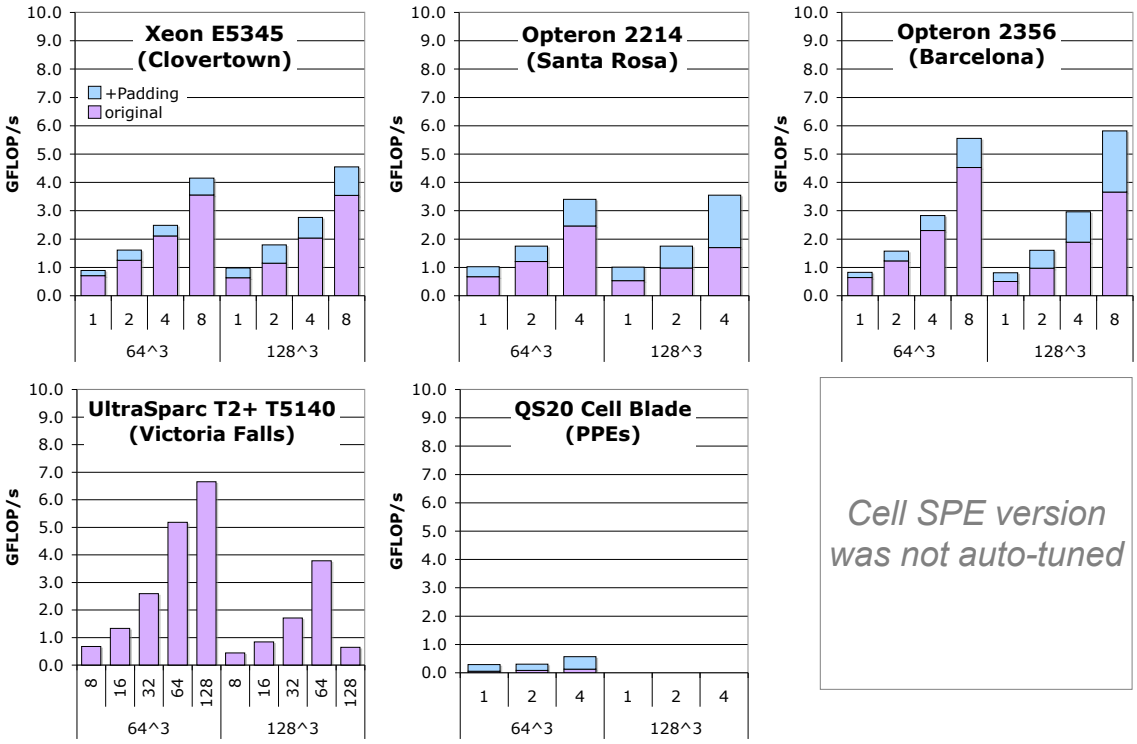


Figure 6.9: LBMHD performance after a lattice-aware padding heuristic was applied.

6.3.4 Vectorization

Although the structure of arrays layout maximizes spatial locality and will facilitate SIMDization, it creates a huge number of streams to memory — 73 read and 79 write. Table 3.4 on page 28 shows most architectures evaluated have small L1 TLBs, fewer entries than the number of streams in LBMHD. Thus, we expect a TLB capacity miss for every array access. These misses would severely impair performance as a table walk is required to service each.

Let us examine Victoria Falls, where Solaris by default uses 4 MB pages for the heap. Even though the architecture only has 128 TLB entries, it can map a 512 MB data structure. For LBMHD this corresponds to a cubicle problem of about 75^3. Thus, as we scale the problem size up, we expect a drop in performance around this size as TLB capacity misses will become common. In fact, we see exactly that. Figure 6.9 shows the 128^3 problems have substantially worse performance and scalability than the 64^3 problems.

Inspired by vector architectures, we can solve this problem by resorting to a loop interchange technique used by vectorizing compilers. In this vectorization technique, we fuse the spatial loops within a plane, and strip mine the resultant loop into vectors. We then exchange the phase space lattice velocity loops with the vector loop. This transformation results in about eight smaller loop nests, each of which performs several simple, but coupled, BLAS1 like operations. This approach increases the page locality as references within a loop nest are likely to touch fewer than 10 pages, thereby ensuring only compulsory TLB misses.

```
collision(...){
  for(all planes){
    for(vectors within this plane){
      // 1. reconstruct macroscopic quantities for VL points
      // 2. update distributions for VL points
  }}
}
```

Figure 6.10: The code structure of the vectorized collision function within the LBMHD application.

Figure 6.10 provides a representation of this code structure.

Conceptually, the original `collision()` function reconstructs the macroscopic quantities for one point one velocity at a time, then updates the distributions for that one point — see Figure 6.11(a) and (b) on page 123. The vectorized `collision()` function reconstructs the macroscopic quantities for VL points one velocity for each at a time, then updates the distributions for those VL points — see Figure 6.11(c) and (d).

There is not a universally optimal vector length. Figure 6.12 on page 124 shows the cache working set footprint is linearly related to vector length. As vector length increases we improve page locality (multiple TLB hits per TLB capacity miss), but we may increase vector length to the point where a vector of points no longer fits in the L1 (or L2). One might believe that keeping data in the L1 is ideal, but the number of TLB misses can be quite large. Trading L1 capacity misses in favor of reducing TLB misses may be acceptable if the relative costs balance. As each architecture has a different relationship between the L1 miss penalty and the TLB miss penalty, we auto-tune by sweeping through vector lengths from one cache line to the maximum number of points that can fit in the cache for the specified concurrency. This exhausitve approach can often find very good performance for non-intuitive vector lengths. For example, Victoria Falls reaches peak performance with a vector length of 24 — clearly much larger than the L1 capacity per thread. To facilitate a future fully auto-tuned vectorization on Cell, we align all vectors to cache line boundaries. That is, we interpret the specified parallelization within a plane as a suggestion rather than a mandate. These jogs in the normally regular parallelization schemes can be seen in Figure 6.6(b) on page 117.

Figure 6.13 on page 125 shows LBMHD performance after the auto-tuned vectorization technique is applied. Clearly the benefit varries greatly among architectures, concurrency, and problem size. Nevertheless, the benefit is substantial.

Interestingly, the Clovertown shows progressively less benefit as the concurrency increases — to the point where all cores of a socket are fully utilized. This is indicative that at full socket concurrency, we're heavily limited by the memory subsystem performance. Conversely, at the single thread concurrency, memory subsystem performance plays a small role, but core performance is the key. Thus, Clovertown performance only increased slightly.

Vectorization had a substantial benefit on both the Santa Rosa and Barcelona Opterons. In fact, they delivered far better performance than the Clovertown despite having

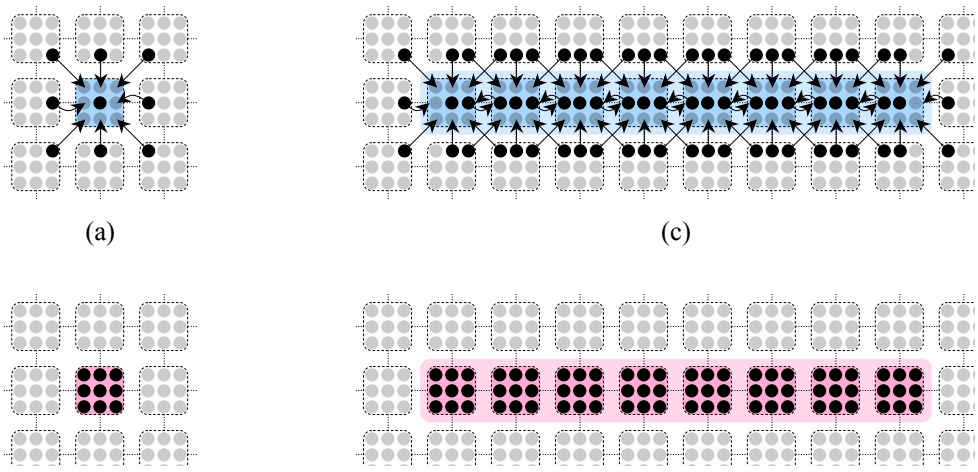


Figure 6.11: Comparison of traditional(a and b) LBMHD implementation with a vectorized version(c and d). In the first phase of `collision()`(a and c) data is gathered, and the macroscopics are reconstructed. In the second phase (b and d), the local velocity distributions are updated. The difference is that the vectorized version updates VL points at a time.

less raw DRAM bandwidth. At this point it is clear that neither Opteron has fully utilized either its raw peak bandwidth, as seen in Table 6.5, or even its nebulous effective peak bandwidth found in the Roofline model. We see that Barcelona uses more that 50% of its peak DRAM bandwidth.

Vectorization was also a major success story on Victoria Falls, improving performance on the largest problems by more than a factor of 15. Vectorization solved both the TLB capacity miss problem as well as a L2 conflict miss problem. We also see that the 128^3 problem no longer suffers from the NUMA effects of the 64^3 problem. As a result, we see near linear scaling from 64 to 128 threads. Remember, vectorization virtualizes the cache hierarchy into a vector register file. As such, far more load and store instructions are required. This tends to depress the floating-point fraction of the dynamic instruction mix. Given that Victoria Falls' performance at this arithmetic intensity is tied to the floating-point fraction and it achieves more than half of its peak FLOPs, it is unlikely further optimization will yield substantially better performance.

The Cell PPE also benefited substantially from vectorization, nearly doubling performance. Nevertheless, the combination of productivity lost via vectorization, and the still dismal performance of the PPE further motivates us to implement a local-store based implementation of `collision()`.

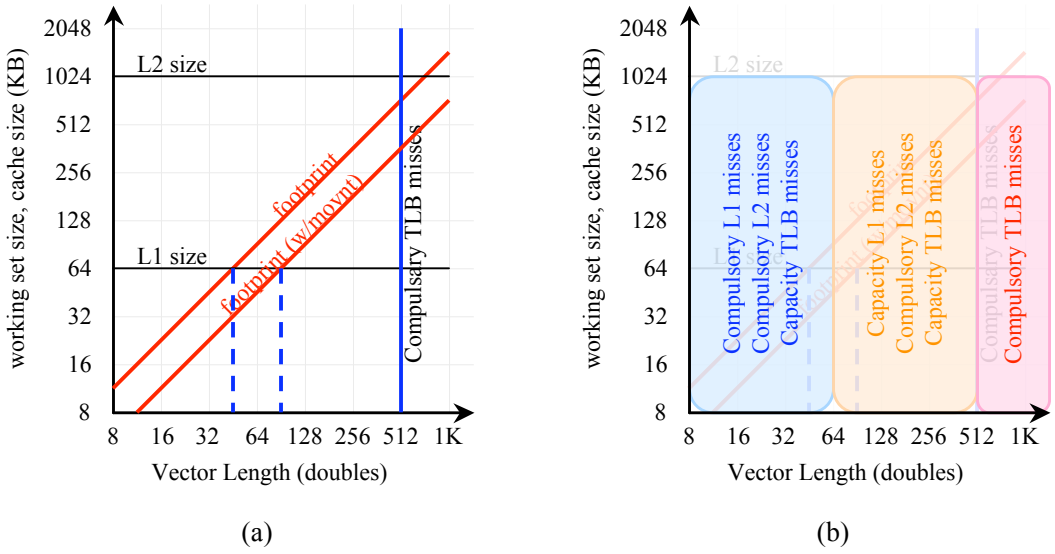


Figure 6.12: Impact of increasing vector length on cache and TLB misses for the Santa Rosa Opteron. (a) Working set footprint as a function of vector length. L1/L2 cache and full page locality limits are shown, (b) Overlay showing regions where different types of cache and TLB misses will occur.

6.3.5 Unrolling/Reordering

Given the vector-style loops produced by vectorization, we modify the code generator to explicitly unroll each loop nest by a specified power of two between one and the cache line size. Although manual unrolling is unlikely to show any benefit for compilers that are already capable of this optimization, we have observed a broad variation in the quality of compiled code on the evaluated systems. As subsequent optimizations will add explicit SIMDization, we expect this unrolling and reordering optimization to become valuable as many compilers cannot effectively optimize intrinsic-laden loops. The most naïve approach to unrolling simply replicates the body of an inner loop to amortize loop overhead. To get the maximum benefit, the statements within the loops must be reordered to group statements with similar addresses, or variables, together to compensate for limited architectural resources and compiler schedulers. These statements are grouped in increasing powers of two, but not more than the unrolling amount. Figure 6.14 on page 126 provides an example. There are $(2 + log(CacheLineSize)) \times (1 + log(CacheLineSize))/2$ combinations of unrolling and reordering. For the architectures in this work, there are only 10 or 15 combinations. As such, an exhaustive search based auto-tuning environment is well-suited at discovering the best combination of unrolling and reordering. The optimal reorderings are not unique to each ISA, but rather to a given microarchitecture, as they depend on the number of rename registers, memory queue sizes, the functional unit latencies, and a myriad of other parameters.

Explicit unrolling and reordering rarely shows any benefit on any architecture with

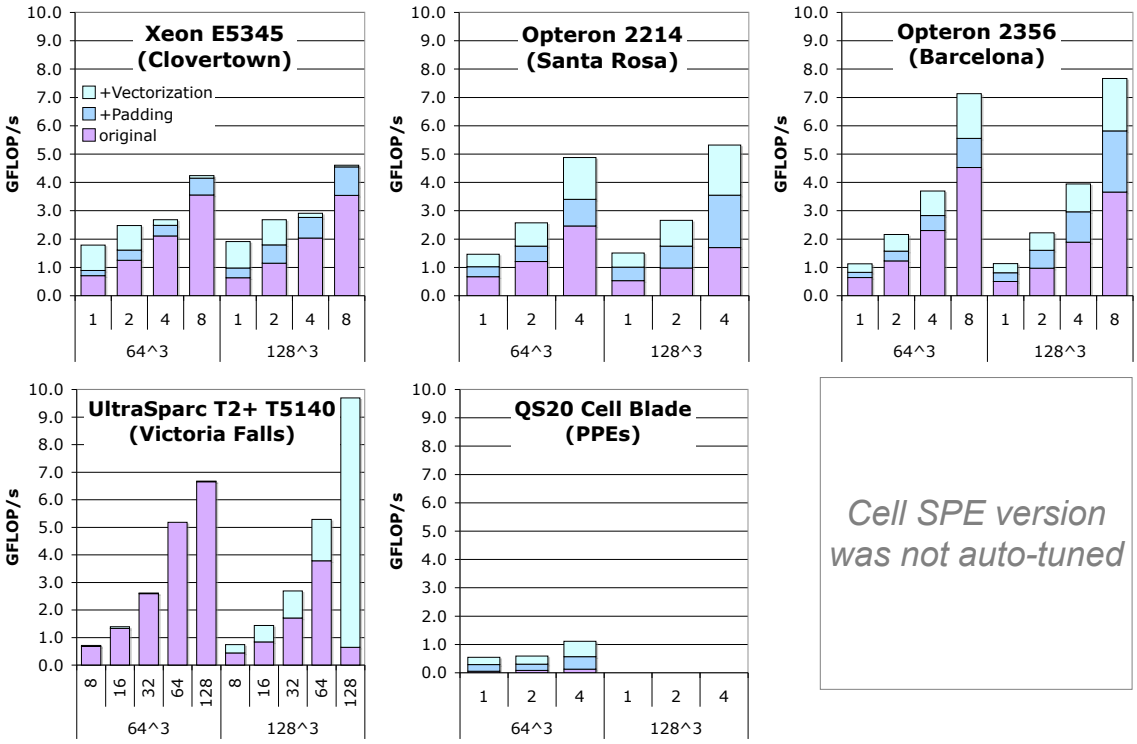


Figure 6.13: LBMHD performance after loop restructuring for vectorizations was added to the code generation and auto-tuning framework.

C code — most compilers can handle these optimizations. They do become somewhat more important after explicit SIMD intrinsics are employed, and are discussed in that section.

6.3.6 Software Prefetching and DMA

Previous work [81, 140] has shown that software prefetching can significantly improve performance on certain superscalar platforms. Although LBMHD is considered memory intensive, it is far less so than many other kernels. As such, we believe that exhaustive search for the optimal prefetch distance will require a significant amount of time but show relatively little benefit. To that end, we modified the code generator to implement three prefetching strategies:

- no prefetching.
- prefetch ahead of each read array by one cache line.
- prefetch ahead of each read array by one one vector length.

Prefetching by one cache line is designed to address L1 latency, while prefetching by a vector is designed to address memory subsystem performance. Thus, in one case, the machine is essentially double buffering in the L1, where the other is double buffering in the

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	(SPE)
GFLOP/s (% peak)	4.60 (6.2%)	5.31 (30.2%)	7.67 (10.4%)	9.69 (51.9%)	1.11 (8.7%)	—
GB/s (% peak)	6.53 (20.4%)	7.54 (35.3%)	10.89 (51.0%)	13.76 (21.5%)	1.54 (3.0%)	—
Speedup from this optimization	+1%	+50%	+32%	+46%	+98%	—

Table 6.5: LBMHD peak floating-point and memory bandwidth performance after vectorization. The Cell SPE version cannot be run without further essential optimizations. Speedup from optimization is the incremental benefit from vectorization.

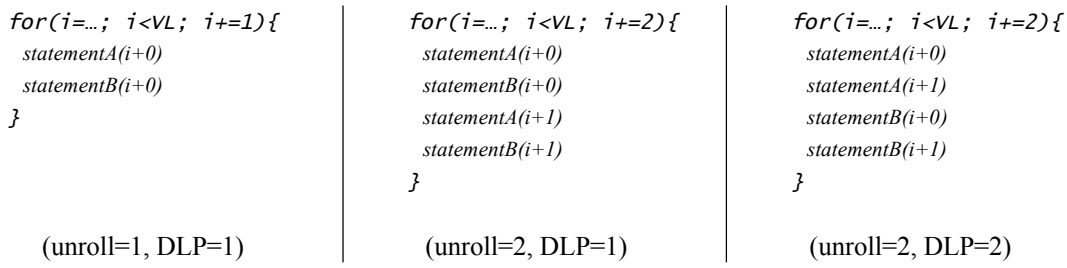


Figure 6.14: Three examples of unrolling and reordering for DLP. Left, no unrolling and no reordering. Middle: unroll by 2, but no reordering. Right: unroll by 2, reorder for DLP by pairs

L2. When double buffering, the current vector is being processed by the execution units, while the next vector is being prefetched into the memory subsystem.

The Cell SPE LBMHD implementation utilizes a similar double buffering approach, but to fill the local store, it uses DMAs instead of prefetching. Thus, each SPE must keep four sets of pencils within the local store: the set of pencils for the current time, y, and z coordinates; the set of pencils being written to for the next time step, but current y and z coordinates; the incoming set from the current time but next y and z coordinates; and the outgoing set for the next time step, but current y and z coordinates.

Software prefetching showed only a small benefit on these architectures, primarily due to the relatively high arithmetic intensity and tuning to maximize cache locality. Obviously, DMA is required on a Cell SPE implementation.

6.3.7 SIMDization (including streaming stores)

SIMD instructions are small data parallel operations that perform arithmetic operations on multiple data values loaded from contiguous memory locations. Although they have become an increasingly popular choice for improving peak performance, their rigid nature makes SIMD difficult to exploit on many codes — lattice methods are no exception. While loop unrolling and code-reordering described previously explicitly expresses data-level parallelism, SIMD implementations typically require memory accesses be aligned to 128-bit boundaries. Structured grids and lattice methods often must access the adjacent point in

the unit stride direction, resulting in an unaligned load. There are several solutions to this problem: SSE allows for slower misaligned loads, while IBM's implementations force the user to load the two adjacent quadwords, and permute them to extract the desired values.

By restricting the unit stride dimension to be even — not an issue as they were already powers of two — we can easily modify the code generators with lattice-aware knowledge of which velocities will be misaligned. Thus, each loop has two variants: one when components are aligned and one when they are not. To facilitate the process, all constants were also expanded into two element arrays. The code generators were modified to replicate all C kernels using SSE intrinsics. They exploit all the previous techniques including prefetching, unrolling and reordering. Thus, these kernels have the advantage of not only exploiting data-level parallelism via SIMD, but are often more cleanly implemented than the code that would be produced by a compiler. When auto-tuning, we benchmark both the C and SIMD kernels for architectures that support both.

SSE2 introduced a streaming store (`movntpd`) designed to reduce cache-pollution from contiguous writes that fill an entire cache line. Normally, a write operation to a write-allocate/write-back cache requires the entire cache line be read into cache then updated and subsequently written back to memory. Therefore, a write requires twice the memory traffic as a read, and consumes a cache line in the process. However, if the writes are guaranteed to update the entire cache line, the streaming-store can completely bypass the cache and output directly to the write combining buffers. This has a several advantages: useful data is not evicted from the cache, the write miss latency does not have to be hidden, and most importantly, the traffic associated with a cache line fill on a write-allocate is eliminated. As LBMHD performs as many compulsory reads from main memory as writes to main memory, the use of streaming stores has the benefit of reducing memory traffic by 33%, and potentially increasing performance by 50%. All SSE kernels use this streaming store. We note that Cell's DMA model was programmed to explicitly avoid the write allocate issue and eliminate the associated memory traffic. However, Cell's weak double-precision performance hides this benefit. We believe it is useful now that the enhanced double-precision implementation of Cell has been introduced.

Figure 6.15 on the next page shows the performance benefit when SIMDization is enabled in the auto-tuning framework on the x86 machines. Clearly, we see varying degrees of improvement. As noted in Section 6.2, we expect SIMD to be capable of not only trading ILP for DLP, but also, through the use of a streaming store, increasing the FLOP:byte ratio. We also cede the point that hand-coded intrinsics are likely faster than compiler generated code.

SSE only afforded the Clovertown — which we generally expected to be memory bound — a 20% increase in performance. If it were solely memory bandwidth bound, we would have expected a 50% increase in performance. Thus, the upper stream limit diagonal in Figure 6.5 on page 114 is likely a substantial overestimate of the bandwidth achievable given the memory access pattern and data set size seen in LBMHD.

Although we see a similar effect on the Santa Rosa Opteron with a roughly 25% increase in performance, we see nearly a 75% increase in performance on Barcelona. Based on Figure 6.5 on page 114, it was clear that the Santa Rosa Opteron would likely not benefit from the increased FLOP:byte ratio, but would readily be capable of exploiting the

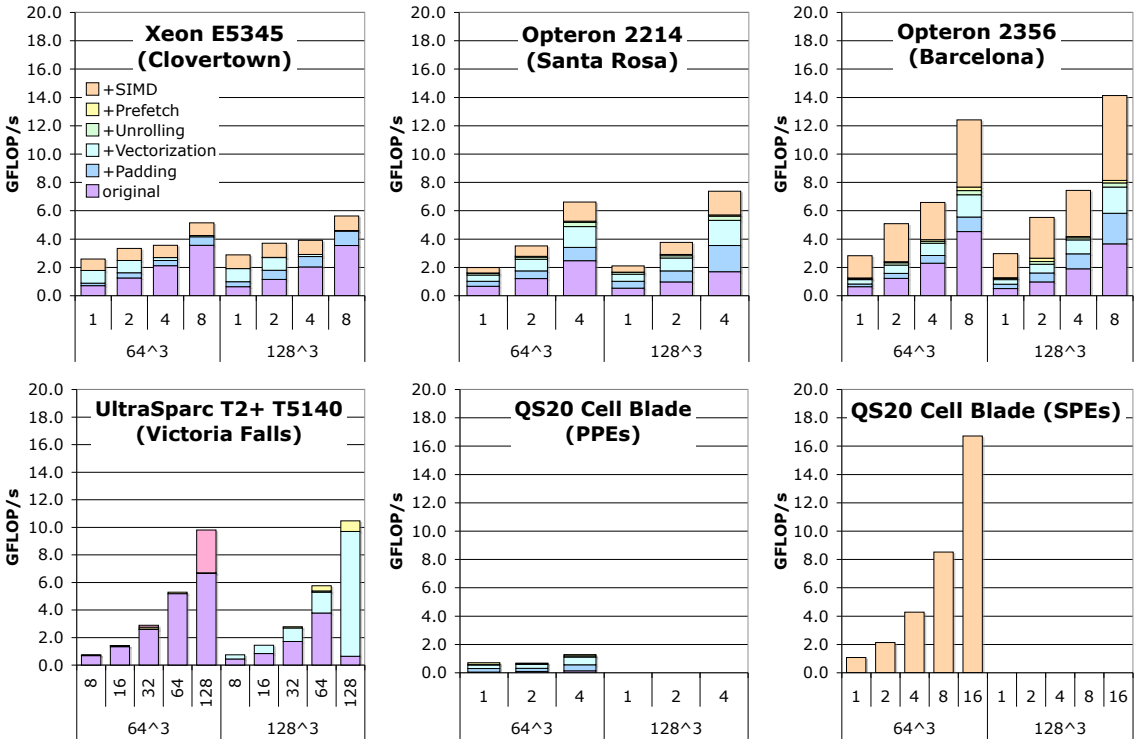


Figure 6.15: LBMHD performance after explicit SIMDization was added to the code generation and auto-tuning framework.

more efficient expression of parallelism in the form of DLP, rather than in ILP. Barcelona, on the other hand, would likely be capable of exploiting both efficient DLP and a higher FLOP:byte ratio. Note that SIMDization exploits reordering and unrolling. On the Santa Rosa Opteron the use of reordering and unrolling yielded about 7% addition performance over SIMDization alone. This small increase in performance came with a nearly 16× increase in tuning time. Furthermore, as architectures become increasingly bandwidth limited (as seen on Barcelona and Clovertown) this benefit will likely shrink.

We also implemented a fully SIMDized Cell SPE implementation. The implementation, although vectorized and SIMDized, is not auto-tuned, and it performs no loop unrolling or reordering. Thus, the only ILP exploited is that which is inherent in a single velocity loop iteration. We should also note that only `collision()` was implemented and benchmarked. As expected, we see great scaling, and are ultimately limited, despite inherent ILP and DLP exploitation, by the combination of the inherent inability to exploit FMA in LBMHD and the stall issue cycles induced by each double-precision instruction.

Table 6.6 shows that after SIMDization, the Clovertown achieves around 25% of its raw DRAM write bandwidth, and a similar fraction of its raw FSB bandwidth. We also see that Barcelona, despite the challenging memory access pattern, achieved nearly 62% of its DRAM bandwidth. As a result, AMD's quad-core Opteron delivers 2.5× the performance of Intel's quad-core machine. We also see Cell, despite its handicapped double-precision,

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	QS20 Cell Blade (SPE)
GFLOP/s (% peak)	5.63 (7.6%)	7.37 (41.9%)	14.13 (19.2%)	10.47 (56.1%)	1.28 (10.0%)	16.72 (57.1%)
GB/s (% peak)	5.26 (25.7%*)	6.89 (32.3%)	13.21 (61.9%)	14.87 (23.2%)	1.79 (3.5%)	15.63 (30.5%)
Speedup from these optimizations	+22%	+39%	+84%	+8%	+15%	—

Table 6.6: LBMHD peak floating-point and memory bandwidth performance after full auto-tuning. *fraction of raw DRAM write bandwidth. Speedup from optimization is the incremental benefit from unrolling, reordering, prefetching, SIMDization, cache bypass, and TLB page size tuning.

still delivers better performance than even Barcelona. When examining Cell's memory bandwidth utilization (30%) we see plenty of room to improve performance with the newly enhanced double-precision implementation.

6.3.8 Smaller Pages

As previously discussed and seen in Figure 6.15 on the preceding page, Victoria Falls has scalability problems on the smaller problem when the second socket is used. This sub-linear scaling is due to the interaction of a first touch policy on a velocity by velocity basis coupled with TLB pages larger than the desired parallelization on a NUMA architecture. The result is the placement of the problem only in the DRAM attached to first socket. The solution, accessible through compiler flags, environment variables, or a wrapper program, is to reduce the default heap page size to 64 KB. This is a trivial solution requiring no coding effort. Clearly 64 KB is far less than the parallelization granularity of 2 MB. Thus, we expect a reasonably equitable distribution of pages among memory controllers. Clearly visible is a roughly 50% increase in performance when auto-tuning is conducted with small pages. To be clear, this performance is achieved in conjunction with vectorization, unrolling, reordering, and prefetching. We see, despite the significantly worse surface:volume ratio, the 64^3 problem achieves nearly the same performance as the 128^3. This is indicative that time within `stream()` has been effectively amortized via parallelization.

6.4 Summary

Table 6.7 details the optimizations and the optimal parameters used by each architecture. Note that the Cell SPE implementation was very primitive and thus did not include many of the optimizations seen on other machines. In addition, many of the parameters were chosen rather than tuned for. The first group of optimizations are focused on maximizing memory bandwidth. The next group addresses attempts to minimize the total memory traffic. Third, we examine the optimizations designed to maximize in-core performance. Finally, we quote the time LBMHD spent in the `stream()` boundary exchange function. As this time was small, we feel confident that further optimization beyond parallelization is currently unnecessary.

Bandwidth Optimization	Auto-tuning approach	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
NUMA Allocation	model	N/A	✓	✓	✓	✓	—
Tuned VL (in doubles)	search	128	128	120	24	80	64*
Prefetch/DMA distance (in doubles)	heuristic	8	8	120	8	16	64*

Traffic Optimization	Auto-tuning approach	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
Lattice-aware Padding	model	✓	✓	✓	✓	✓	—
Cache Bypass	search	✓	✓	✓	N/A	—	✓

In-core Optimization	Auto-tuning approach	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
SIMDized	search	✓	✓	✓	N/A	N/A	✓
Unrolling	search	8	8	8	4	2	2*
Reordering	search	8	8	4	1	2	2*

Function	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
% time in `stream()`	7.7%	9.0%	10.5%	4.8%	8.2%	N/A

Table 6.7: Top: LBMHD optimizations employed and their optimal parameters for the 128^3 problem at full concurrency (64^3 for Cell) and grouped by Roofline optimization category: maximizing memory bandwidth, minimizing total memory traffic, and maximizing in-core performance. Bottom: breakdown of application time by function after auto-tuning. *hand selected.

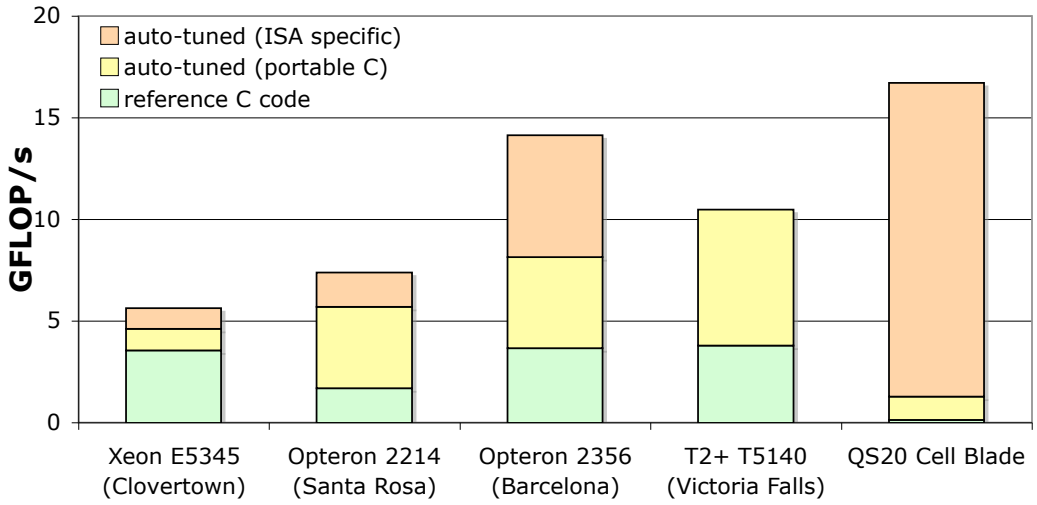


Figure 6.16: LBMHD performance before and after tuning. Performance is taken from the largest problem run.

6.4.1 Initial Performance

Currently, auto-tuners must be written on a per-kernel basis. As an auto-tuner might not be available, comparing unoptimized performance provides insights into the architecture/compiler synergy and as is indicative of what we might expect for most out-of-the-box codes. Figure 6.16 shows that for LBMHD, the Clovertown, Barcelona, and Victoria Falls deliver comparable performance despite the vastly different peak rates. This comparison is somewhat unfair to Victoria Falls, as it attains better performance on the smaller problem with smaller pages. All three deliver nearly twice the out-of-the-box performance of the Santa Rosa Opteron. Clearly, the PPE's on a Cell blade are completely inadequate for LBMHD — with Clovertown being nearly 30× faster. Figure 6.17 on the following page shows their performances are all at the low end of the expected performance range, with the Cell PPE number extremely low. Recall that the bandwidth ceilings all presume unit-stride memory access patterns — something in practice the original implementation of LBMHD does not have.

6.4.2 Speedup via Auto-Tuning

As previously discussed, auto-tuning provided substantial speedup on each architecture. Although the Clovertown attained a nearly 4.5× increase in performance using a single thread, in Figure 6.16 we see only a 60% increase in overall performance. The extremely weak FSB profoundly limited the effectiveness of auto-tuning as the number of threads increased — 3.2× with two threads, 2× with four, and a mere 1.6× with all eight threads. Thus, sheer parallelism won out over code quality. This result was somewhat surprising given the bandwidth characteristics of Figure 6.17 on the following page. Performance seemed to remain bounded by the low end of FSB performance — perhaps due

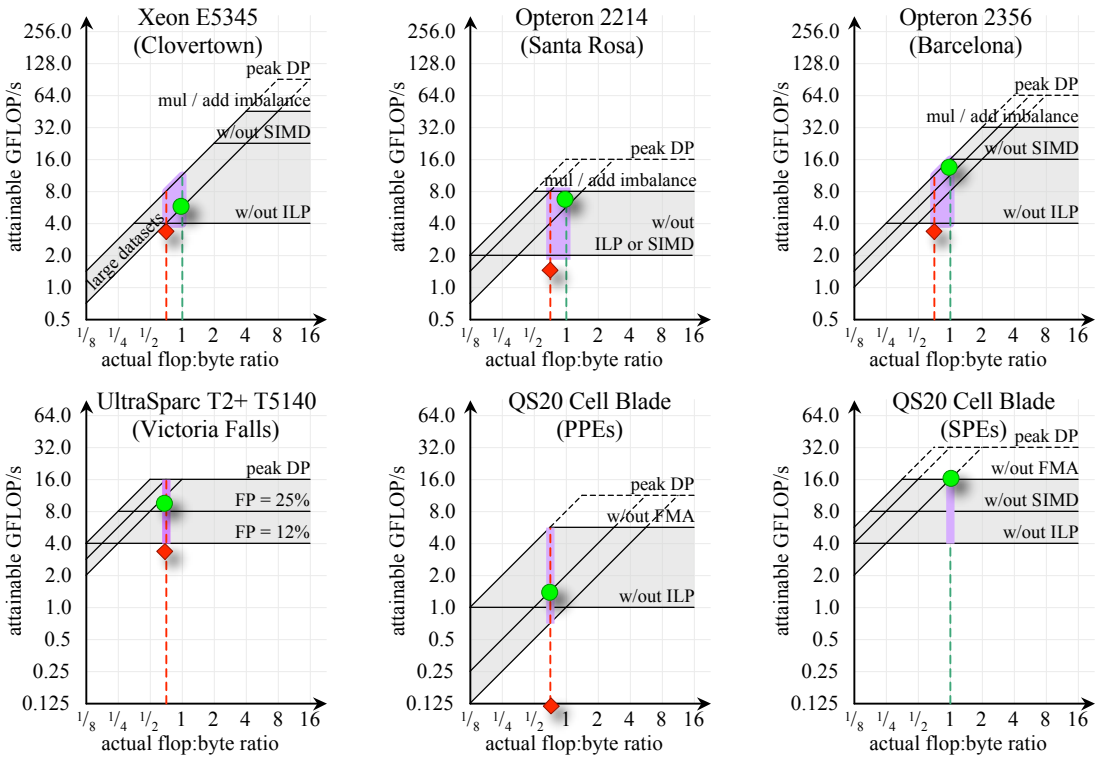


Figure 6.17: Actual LBMHD performance imposed over a Roofline model of LBMHD. Note the lowest bandwidth diagonal assumes a unit-stride access pattern. Red diamonds denote untuned performance, where green circles mark fully tuned performance for the largest problem attempted. Note the log-log scale, and the different scale for the lower three architectures.

to an ineffective snoop filter on problems with large data sets. The primary advantage of ISA-specific auto-tuning was the use of cache bypass instructions to eliminate write fill traffic.

The Opterons, capable of sustaining far greater bandwidth, saw the benefits of the auto-tuning as concurrency increased — better than 4× from one through four threads on Santa Rosa. Despite the fact that Barcelona has no more raw DRAM bandwidth than a Santa Rosa Opteron, and less than Clovertown, we saw that it was extremely effective in exploiting it — also delivering a near 4× speedup via auto-tuning. Thus, in Figure 6.17, we see the Santa Rosa started out processor bound, and ended up bound by the fact that LBMHD is not multiply/add balanced. Conversely, we see Barcelona started out processor bound, but quickly became memory bound with the increased FLOP:byte ratio helping significantly. Figure 6.16 clearly shows the huge advantage of cache bypass ISA-specific tuning on the memory-bound Barcelona, but not the processor-bound Santa Rosa.

On the smaller problem, Victoria Falls showed very good performance before auto-tuning, but ran into limitations due to the page size. The TLB effect was far more dramatic and persisted as concurrency increased on the larger problem. Although auto-tuning im-

proved performance by a respectable 50% on the smaller problem, it improved performance by an astounding 16× on the larger problem. We believe that it remained processor bound as experiments with higher frequency parts showed linear speedups. The SPARC VIS instructions do not currently support double-precision floating-point SIMD. Furthermore, the Sun compiler does not currently provide intrinsics to exploit cache bypass. Thus, the auto-tuner didn't explore SIMDization or the cache bypass optimizations.

Cell required two implementations. The first, an auto-tuned implementation, only ran on the PPEs. Auto-tuning provided a 10× increase in performance, but the raw performance remained very low. When the SPE implementation was included, we saw a phenomenal 13× increase in performance over the auto-tuned PPE implementation. Like the Santa Rosa Opteron, Figure 6.17 on the preceding page demonstrates that the Cell SPE implementation was limited by the inherent inability to exploit fused multiply add (FMA) within the LBMHD kernels.

Table 6.7 on page 130 shows the optimal unrolling was often the cache line size because this provides the optimal number of software prefetches per cache line. The reordering for DLP varied substantially between architectures, although given the range in architected registers, it is not entirely surprising. Regardless, the value of this optimization was small. We also see that after the cache bypass instructions are employed, the footprint associated with the optimal vector length is very close to the L1 size, but far less than the page size — indicative of the relative cost of L1 misses and TLB misses. Although slightly smaller vector lengths slightly degraded performance, larger vector lengths substantially degraded performance.

6.4.3 Performance Comparison

When comparing auto-tuned performance, we see that the Santa Rosa Opteron is slightly faster than Intel's Clovertown, despite the vastly lower FLOP rate. When moving to the quad-core Barcelona, we see nearly a doubling of performance over the Santa Rosa, achieving better than 60% of its memory bandwidth. Clearly, the combination of twice the cores and fully pumped SSE was capable of saturating the machine's attainable memory bandwidth. This makes AMD's quad-core nearly 2.5× as fast as Intel's current quad-core offering. LBMHD is far from Victoria Falls' sweet spot as it is fairly FLOP intensive. Nevertheless, we see Victoria Falls achieve better than 75% of Barcelona's performance despite having a quarter the peak FLOP/s. In the end, Cell's extremely efficient DMA ensured memory bandwidth would not be the bottleneck. Thus, despite having less than half the FLOP rate, Cell delivered better performance than Barcelona. We believe the enhanced double-precision implementaion (eDP Cell) will more than double the current delivered performance. Overall, Cell, without auto-tuning, delivers 1.2× better performance than Barcelona, 1.6× better than Victoria Falls, 2.25× the Santa Rosa Opteron, and almost 3× the performance of the Intel Clovertown. The most disturbing trend we observe on the newer architectures is the increasing dependence on ISA-specific tuning to achieve peak performance. This is not to say portable C tuning isn't valuable, but rather it is insufficient. Clearly Victoria Falls' use of simple multithreaded RISC cores obviated ISA-specific auto-tuning. Thus, from the productivity standpoint, Victoria Falls reaches peak performance with much less work.

```
struct{
  // macroscopic quantities
  double Density;
  double Momentum[3];
  double Magnetic[3];
  // distributions
  double MomentumDistribution[27];
  double MagneticDistribution[3][27];
}
```

Figure 6.18: Alternate array of structures LBMHD data structure for each time step. An N^3 3D array of structures should be allocated.

6.5 Future Work

Although an extensive number of optimizations have been implemented here, significant work specific to LBMHD remains to be explored. This research is divided into four categories: exploration of alternate data structures, exploration of alternate loop structures for vectorization, time skewing, and tuning hybrid MPI-pthread implementations. We discuss motif-wide future work in Chapter 9.

6.5.1 Alternate Data Structures

The major omission in the optimization of `collision()` was the lack of exploration of alternate data structures. The auto-tuned implementation of sparse matrix-vector multiplication (SpMV) presented in Chapter 8 shows significant benefit using alternate data structures. We discuss several possibilities here. Ultimately, if we are at the bandwidth Roofline, only data structures that reduce memory traffic will be valuable, something not easily achieved within the structured grid motif.

Figure 6.18 presents the naïve array of structures format. Use of this approach would reduce the number of memory streams from 152 to ideally 10. The disadvantage is that when gathering data from neighboring points in space, only 8 bytes are used — the one velocity directed at the point to be updated. One can only compensate for this total lack of spatial locality by using a giant cache — as much as 16 MB — to encompass all the neighbors of a point's neighbors. The other disadvantage of this approach is that it cannot be efficiently SIMDized.

Figure 6.19 on the next page shows a hybrid data structure. Spatial locality is maintained without the need for a large cache, and the number of streams in memory is reduced from over 150 to about 56. This approach may obfiscate the need for vectorization on some architectures, but like the array of structures approach, it cannot be efficiently SIMDized.

Rather than storing the problem as a single large N^3 array, one could store the grid hierarchically in several smaller B^3 arrays. The challenge is whether the inter block ghost zones should be explicit and thus require a more complex `stream()`, or implicit and thus require complex addressing in `collision()`.

```
struct{
  double7 *Macroscopics;
  // 7 doubles per point in space
  // 0=Density, 1-3=Momentum[3], 4-6=Magnetic[3]

  double *MomentumOnlyDistributions[27];
  // 1 double-per velocity
  // only velocities 0..11 are non-NULL

  double4 *MomentumAndMagneticDistributions[27];
  // 4 doubles per velocity
  // 0=Momentum, 1-3=Magnetic[3]
  // only velocities 12..26 are non-NULL
}
```

Figure 6.19: Hybrid LBMHD data structure for each time step. Each pointer points to an N^3 3D array of structures.

In general, the auto-tuner should consider or explore all possible data structures and find the one that best matches the architecture being tuned for.

6.5.2 Alternate Loop Structures

Note that only a simple loop interchange was performed in the vectorization optimization. As a result, during the reconstruction of the macroscopic variables, only one velocity is gathered at a time. Although this approach keeps the number of open pages low, it can put enormous pressure on the cache bandwidth, given the very low FLOP:L1 byte ratio. This optimization strategy can be further expaned. First, we can begin grouping several velocities together during the reconstruction phase — gathering two, three or four at a time. Grouping velocities will reduce the cache requirements but increase the TLB capacity requirements. Alternately, we may calculate the momentum and three magnetic variables separately rather than trying to recover them simultaneously. This increases the cache requirements, but reduces the TLB requirements. A search over this vastly expanded space would only be necessary on architectures not already clearly limited by memory bandwidth.

6.5.3 Time Skewing

We observe that many of the machines are memory bound. The trends in computing will only exacerbate this problem in the future. As such, one should consider applying the time skewing techniques introduced in Section 5.3. Previous work [141, 142] has shown that such techniques are applicable in single-precision on Cell for simple PDEs and have also shown benefit for lower dimensional versions of LBMHD [51] on superscalar processors. In essence, each grid sweep would advance the grid by two or more time steps. The challenge is the on-chip memory required to implement such an approach may be far greater than the

cache capacities of some architectures. As such, one could tune for the optimal number of steps to take.

6.5.4 Auto-tuning Hybrid Implementations

Although LBMHD is originally an MPI application, the auto-tuner is only designed to run on an SMP. The auto-tuned framework, but not the exploration, should be back fitted to the application. With this change, one would tune for the optimal single node parameters using a single node, and then scale out to a large distributed memory super computer using MPI. A 4× increase on a single Opteron chip may translate to a 30 TFLOP/s increase in full system performance on a large supercomputer.

Furthermore, in this hybrid implementation, one could tune for the optimal balance between MPI tasks and threads per MPI task. For example, given a dual-socket, quad-core SMP, the naïve approach is to use 8 MPI tasks per SMP. However, one could alternately use 4 MPI tasks of two threads, 2 MPI tasks of four threads (*i.e.* one per socket), or 1 MPI task using 16 threads (*i.e.* one MPI task per SMP). To be fair, the total memory requirements per SMP should remain constant regardless of the decomposition.

6.5.5 SIMD Portability

`collision()` was optimized for SSE through the use of non-portable intrinsics embedded within the C code. A similar approach can be employed on both BlueGene or future Intel machines with the use of double hummer or AVX [74] intrinsics. Although they are not identical to SSE, they are sufficiently similar that the work required is small. Ideally, a common SIMD language would provide portability across SIMD ISAs.

6.6 Conclusions

In this chapter, we examined the applicability of auto-tuning to structured grid kernels on multicore architectures. As structured grids are an extremely broad motif, we chose lattice methods — specifically LBMHD — as an interesting subclass for which many optimizations will need to be tuned. Despite the fact that the original LBMHD implementation is far from naïve, we see auto-tuning provided substantial speedups on all architectures aside from the heavily frontside bus-bound Xeon. Although attempting to specify the appropriate loop unrolling and reordering provides little benefit over current compilation technology, it was clear that these compilers are wholly incapable of efficiently SIMDizing even simple kernels. Some of the largest benefits came from lattice-aware padding to avoid L1 conflict misses and the selective use of cache bypass instructions to avoid write-fill traffic. Both of these optimizations improve the application's actual FLOP:byte ratio.

Before auto-tuning, there is relatively little difference in the performance and efficiency of the cache-based architectures. As we trade productivity to expand the capability of the auto-tuner, Victoria Falls quickly reaches peak performance. Only when SSE intrinsics are included, an extremely unproductive task, does Barcelona achieve peak performance. The Cell implementation, although not auto-tuned, required the same work as the SSE enabled auto-tuner. Thus, we conclude that although Barcelona, Victoria Falls, and Cell

deliver similar performance, we were most productive on Victoria Falls. Finally, we conclude that Cell has the most potential for future performance gains as its extremely weak double-precision implementation is a performance bottleneck. Correcting this architectural limitation is relatively simple compared to increasing memory bandwidth on the other architectures.

Chapter 7

The Sparse Linear Algebra Motif

Sparse methods form the cores of many HPC applications. As such, their performance is likely a key component of application performance. This chapter discusses the fundamentals of sparse linear algebra. We do not discuss derivations or computational stability of any kernel. For additional reading, we suggest [109]. The chapter is organized as follows. Section 7.1 discusses some of the fundamental characteristics of sparse matrices and sparse linear algebra. Next, Section 7.2 discusses the fundamental sparse computational kernels and as well as methods built from such kernels. In addition, it discusses several uses for these methods. Section 7.3 then discusses several common storage formats and what motivated their creation and use. This chapter provides the breadth and depth required to understand our auto-tuning endeavor in Chapter 8. Finally, we provide a summary in Section 7.4.

7.1 Sparse Matrices

Although motifs typically don't have well defined boundaries, sparse linear algebra is an exception. As it evolved from the well-defined dense linear algebra motif, it is also well defined. In this section, we discuss several agreed upon characteristics of sparse matrices and sparse kernels. We do so to provide clarity to the auto-tuning effort in Chapter 8.

An m×n matrix A has elements a_{ij} where i is the row index and j is the column index. By convention, the first row is row 0, and the first column is column 0. Thus, $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$. By definition, there are m×n matrix elements. In many linear algebra matrix operations, the number of element-wise operations is at least m×n and, in many cases, it can be larger than the product of the matrix dimensions. For example, where the computational complexity of square matrix-vector multiplication is $O(n^2)$, the computational complexity of square matrix-matrix multiplication is $O(n^3)$.

A sparse matrix has a large number of elements such that $a_{ij} = 0$. In sparse linear algebra, the properties of addition and multiplication with zeros are exploited. For many kernels, as long as one knows that $a_{ij} = 0$, there is no need to explicitly store or compute on a floating-point zero. Typically, only the values and indices of the nonzeros are stored. All other elements are zero by construction. When performing a matrix operation, one must determine whether a_{ij} is explicitly stored and thus utilize the value or whether

it is implicitly zero and so no computation is required. Such checks can dramatically reduce the floating-point computation rate of sparse matrix operations compared to their dense cousins when implemented on a computer. However, when the number of nonzeros is substantially less than of m×n, the inefficiency of sparse computations is offset by the dramatic reduction in the execution time of matrix operations. As such, there will be both a dramatic performance and storage benefit.

There are several common characteristics and terms associated with sparse matrices. We define them here.

- **rows** and **columns** — All matrices, whether dense or sparse, span a number of rows and columns irrespective of whether or not these rows or columns are empty. A square matrix has the same number of rows as columns.
- **NNZ** — The number of nonzeros (NNZ) in a matrix is the total number of non-zero a_{ij} elements.
- **NNZ/row** — The average number of nonzeros per row. In many kernels, a large number of nonzeros per row amortizes accesses to dense vectors as well as amortizes loop overheads.
- **sparsity** — A qualitative assessment of the pattern and locality of nonzeros. The sparsity pattern of a matrix often provides some insight into a kernel's performance when operating on said matrix.
- **spyplot** — A 2D visual representation of the sparsity pattern of the matrix generated by placing a black pixel at every nonzero.
- **bandwidth** — For any given row, one can calculate the bandwidth of the row as the greatest difference in column indices between any two nonzeros on that row. Moreover, the bandwidth of a matrix is commonly calculated as either the average bandwidth across all rows, or the maximum bandwidth of any row.
- **well structured** — Low bandwidth matrices are often labeled well structured as they are efficiently parallelized and have small cache working sets.
- **symmetric** — square matrices for which $a_{ij} = a_{ji}$
- **hermitian** — square matrices of complex floating-point values for which $a_{ij} = a^*_{ji}$. That is $Re(a_{ij}) = Re(a_{ji})$, but $Im(a_{ij}) = -Im(a_{ji})$

Figure 7.1 on the following page presents spyplots for two different sparse matrices. The matrix in Figure 7.1(a) is well structured, has low bandwidth, and has on average 28 nonzeros per row. Kernels operating on such matrices are often efficiently parallelized. Observe that despite smaller dimensions, Figure 7.1(b) is poorly structured, has a higher bandwidth, and a comparable number of nonzeros per row.

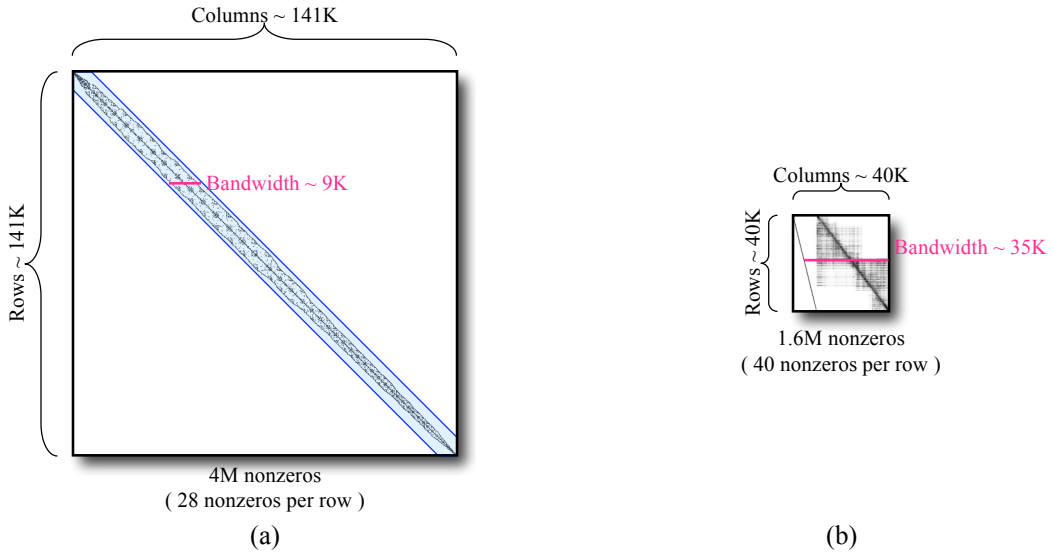


Figure 7.1: Two sparse matrices with the key terms annotated.

7.2 Sparse Kernels and Methods

Sparse kernels and methods are broadly categorized as any routine or set of routines that operates on a sparse matrix or sparse vector. One could consider a sparse vector as one row of a sparse matrix.

Generally, kernels and methods can be categorized into routines that evaluate expressions or routines that solve for a vector or matrix. Typically, solvers are complex routines or methods that use of several primitive kernels. Those primitive kernels can operate on either dense or sparse data. For purposes of this work, we focus on only those methods and kernels derived from dense linear algebra.

One can classify sparse methods into either direct or iterative described in Sections 7.2.2 and 7.2.3 respectively. In turn, these two types of methods are implemented as a series of dense BLAS or sparse BLAS kernel calls. We assume the reader is familiar with the most frequently used dense BLAS kernels [45, 42]. In this section, we begin with a discussion of the sparse BLAS kernels. We follow this with a discussion many common or direct and iterative sparse methods. Finally, we discuss a number of problems solved with these methods.

7.2.1 BLAS counterpart kernels

The dense BLAS kernels are often categorized into level 1, 2, and 3 depending on whether they operate on 0, 1, or 2 matrices. The sparse BLAS [110] have defined sparse versions of many of these routines in which one vector or matrix is sparse. Some dense BLAS routines are actually operating on sparse matrices, but of very restricted sparsity patterns (*e.g.* band matrices).

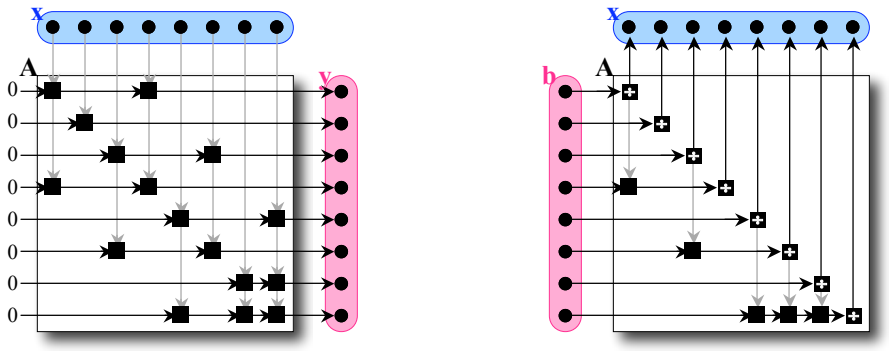


Figure 7.2: Dataflow representations of (a) SpMV ($y = Ax$) and (b) SpTS ($Ax = b$). The boxes represent multiply-add, negative multiply-add, or divide.

The most primitive sparse BLAS operations operate on vectors. The principal operations are addition, dot products, gathers, and scatters. Addition with a dense vector results in a dense vector, but addition of two sparse vectors results in a sparse vector with a number of nonzeros ranging from anywhere between 0 and the sum of nonzeros in the vectors. A dot product always results in a scalar. Gathers convert sparse vectors into dense, and scatters convert dense vectors into sparse.

The more complex sparse BLAS operations are evaluation of sparse matrix-vector multiplication (SpMV) and sparse triangular solve (SpTS). SpMV performs the operation $y = Ax$ where A is a sparse matrix, x is the dense source vector, and y is the dense destination vector. SpTS solves for the dense vector x in the equation $Tx = b$ where T is either a lower or upper triangular sparse matrix, and x and b are once again dense vectors.

SpMV and SpTS are drastically different in their respective implementations and inherent parallelism. Figure 7.2 presents a dataflow representation for the reference implementations of SpMV and SpTS. The evaluation of $y = Ax$ is expressed as $\forall\ a_{ij} \neq\ 0, y_i = y_i + a_{ij} \cdot x_j$. Clearly, all y_i are independent and are thus easily parallelized. This is born out in inspection of the DAG in Figure 7.2(a). The challenge is to attain performance. In a nightmarish pathological case, solving for x_j in SpTS is completely serial. Figure 7.2(b) shows that there is a partial ordering in which the x_j must be evaluated for correctness. Luckily, the there is some parallelism in most matrices for SpTS. Nevertheless, the parallelism and optimal ordering of operations is matrix-dependent. SpTS performance is further hampered as one floating-point divide is required per row.

Both SpMV and SpTS can be extended by operating on multiple right hand sides. In essence, SpMV becomes SpMM (sparse matrix matrix multiplication) where the second matrix is a tall, skinny matrix. In addition, the sparse matrices at the cores of these operations can be transformed via a transpose or complex conjugate. As a result, one could perform the operation $A^T X$ where A is a, say, 1M×1K sparse matrix, and X is a 1M×10 dense matrix. The result would be a 1K×10 dense matrix. Moreover, one could reverse the

order of operations $y = Ax$ vs. $y = x^T A$.

7.2.2 Direct Solvers

Sparse direct methods perform complex operations like solving $Ax = b$ via Gaussian Elimination, LU factorization, or Cholesky Factorization using a number of primitive kernels including SpTS. The selection of the appropriate routine depends on matrix characteristics. For example, Cholesky is only applicable if A is symmetric and positive definite.

Unlike their dense cousins whose requisite computation and storage requirements are fixed and constant, the storage requirements for direct sparse methods are sparsity dependent. To be specific, in the dense world, the storage requirements are $\approx N^2$. However, in the sparse world the size of A is approximately *NNZ*, but the size of the factored matrices can be much larger than *NNZ*. In fact, it is not uncommon for the factored sparse matrices to be more than an order of magnitude larger than the original sparse matrix. Nevertheless, the computational complexity of direct methods is strictly determined by *NNZ* in L and U, and the accuracy is often better than iterative methods. As a result, significant effort has been made in implementing and optimizing sparse direct methods. Although LU results in two sparse matrices that must be stored, Cholesky factorization ($A = LL^T = U^T U$) requires only storage of one matrix. However, SpTS must also be implemented to solve using a transposed matrix: $L^T x = b$ or $U^T x = b$.

7.2.3 Iterative Solvers

Iterative methods take a radically different approach. Rather than directly solving the problem, for which there may be no method, they make an initial estimate of the solution, and then through an iterative process, refine that initial guess. Although the computational requirements per iteration are moderate and easily calculated, the net computational requirements of this method depend on the rate of convergence. Moreover, an accurate result might not be possible. In naïve approaches, the storage requirements are constant, easily predicted, and dominated by the size of A. However, for algorithms that deliver increased accuracy, the storage requirements increase with the number of iterations.

Conjugate Gradient (CG) is a common iterative method to solving $Ax = b$, in which one starts with an initial guess x_0, calculates the residual $b - Ax_0$, calculates a new x_i, and iterates until the residual is sufficiently small. Each iteration of this method requires a sparse matrix-vector multiplication, and a number of dense vector-vector operations. Like Cholesky, CG imposes the symmetric, positive definite (SPD) restrictions on matrices. However, there are a number of other iterative methods such as BiCG that remove some of these restrictions.

7.2.4 Usage: Finite Difference Methods

There are a myriad of uses of sparse kernels and solvers. Motivated by our discussion in Chapter 5 and work in Chapter 6, we discuss the applicability of the sparse motif to partial differential equations on structured and unstructured grids. This is only possible because differential operators become stencils in a finite difference method. Moreover,

these stencil operators are linear combinations of neighboring nodes. Linear combinations map perfectly to linear algebra. If the computation of the resultant stencils were non-linear operators, then one could not use the sparse motif. Moreover, this approach is limited to Jacobi's method as discussed in Chapter 5.

First, let us consider the sweep of a scalar stencil operator on a 2D rectangular structured grid. The first step is to represent the state of the scalar grid as a dense vector: x. We choose a natural row ordered enumeration of nodes. Thus, the value of the grid at (i, j) is element x_k, where $k = j \cdot \ XDimension + i$. The value of the node after the stencil is stored in y_k. In addition, a 5-point linear combination stencil operator performs a linear combination of neighboring nodes. To replicate this functionality, one must perform a linear combination of the "neighbors" in grid space of vector element x_k. This functionality is perfectly realized through a sparse matrix-vector multiplication: $y = Ax$. A has five nonzeros per row (the points of the stencil), the same five values appear on every row, and if properly ordered, the matrix is pentadiagonal. In itself, there is no benefit in this approach as the resultant volume of memory traffic for each sweep is now 76 bytes per node, instead of the structured motif's 16 bytes per node.

Next, consider that the problem we are solving is not a perfect rectangular grid, but has a complex geometry. As such, the addressing of elements may be far too challenging to implement as a structured grid. Moreover, the topological connectivity and edge weights may vary from one node to the next. As such, the structured grid motif is totally inadequate. One could implement such operations either via the unstructured grid motif, or through the sparse motif. In the sparse motif, the number of nonzeros per row varies over the range of possible topological connectivities. In addition, each row's nonzeros may have unique values. If connectivity were still rectangular, the volume of memory traffic is still 76 bytes per node.

At a higher level, the finite difference method can result in either an explicit or an implicit method to model the time evolution of a PDE. That is, given the grid a time t, we apply a function that determines the state of the grid at time t+1.

In an explicit method, the next value at one point in the grid is dependent on a subset of points in the current grid. This is the forward difference, and results in either a stencil sweep or one SpMV. That is, evaluate $x_{t+1} = Ax_t$, where x_t is the known current state of the grid, x_{t+1} is the next state, and A represents the stencil. However, this approach is generally numerically less stable.

As such, the backward difference results in an implicit method, in which the current value of the grid is a function of the next values for a subset of the grid. Such approaches demand that one must solve, rather than evaluate, to determine the next grid values. Depending on structure, one could implement this as a n-diagonal solve or solve $Ax_{t+1} = x_t$. As previously described solving $Ax = b$ can be realized either through direct methods (*e.g.* sparse LU or Cholesky) or iterative methods (*e.g.* conjugate gradient).

In the direct approach, one first factors A into LU. Then for each time step, one must solve $LUx_{t+1} = x_t$. This can be accomplished by solving $Ly = x_t$, then solving $Ux_{t+1} = y$. Both steps require a SpTS. Although this method is generally more stable, sparse LU factorization is expensive, L and U can be substantially larger than A, and SpTS is substantially slower than SpMV.

Classification		Upfront Operation(s)	Steady State Operation(s)	Principle Kernel(s)	Storage Requirement
Explicit		—	$x_{t+1} = Ax_t$	1 × SpMV	≈NNZ(A)
Implicit	Iterative	—	$Ax_{t+1} = x_t$	CG, BiCG, etc... (many SpMV's)	≈NNZ(A)
	Direct	$LU = A$	$LUx_{t+1} = x_t$	2 × SpTS	≫NNZ(A)

Table 7.1: Summary of the application of sparse linear algebra to the finite difference method. The upfront operations only need to be performed once per grid topology.

In the iterative approach, one must attempt to solve $Ax_{t+1} = x_t$ at each time step using a method like conjugate gradient. Although no additional space is required, each solve will require several SpMV's. Ideally, x_t is a reasonable initial guess for x_{t+1}. As such, the number iterations should be well managed.

The question arises, can any structured grid kernel be implemented with a SpMV? For example, can LBMHD's `collision()` operator be implemented with an SpMV? The answer is no. SpMV only performs linear combinations of vector elements. LBMHD requires divides and other structured grid codes may require logs or exponentials. Moreover, LBMHD multiplies lattice velocities together rather than simply taking linear combinations of them.

Table 7.1 summarizes the application of the sparse linear algebra motif to the finite difference method for partial differential equations (PDEs). The computational and storage requirements vary substantially. Moreover, numerical stability is the motivation for implicit methods. Clearly, SpMV and SpTS are critical operations and the value of optimizing them cannot be underestimated.

7.3 Sparse Matrix Formats

Matrices are stored in a variety of formats designed to deliver good performance for the typical kernels that use them. In this section, we discuss formats that will perform well for SpMV. We believe that many of these will likely stand the test of time in the multicore-era

Figure 7.3 on the next page shows a small dense matrix and the most common storage formats. Typically, the 2D m×n structure of a dense matrix is reorganized into a dense 1D array either in a row-major or column major format. Clearly, the resultant memory footprint of such a matrix is simply 8·m·n bytes, or 8 bytes per element. In addition, addressing element a_{ij} is a trivial task, it is either `A[n*i + j]` or `A[m*j + i]`. In addition, a matrix must store the number of rows and the number of columns. Accessing elements on neighboring rows or columns are realized with trivial offsets of ±1, ±m, or ±n, depending on format. Often such approaches are implemented hierarchically. The optimal storage format for a dense matrix is both machine and kernel dependent. That is, the optimal storage format depends on the kernel that will use the matrix and the machine on which the kernel is run.

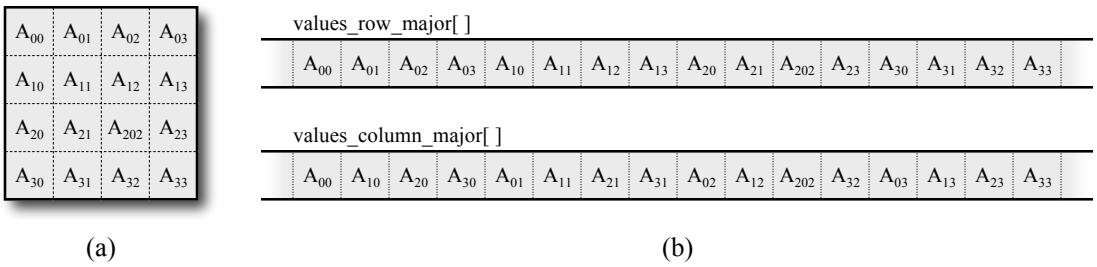


Figure 7.3: Dense matrix storage. (a) enumeration of matrix elements, (b) row- and column-major storage formats.

Storing a sparse matrix presents a number of unique challenges as, unlike dense matrices, the sparsity pattern can dictate the optimal format. That is, the optimal format is machine, kernel, and matrix dependent. To that end, a number of storage formats have been proposed. The simplest strategy is to store a sparse matrix in dense format. This is not to be confused with storing a dense matrix in a sparse format. In such a case, the vast number of zeros would also be stored. As a result, all algorithmic advantages would be lost.

7.3.1 Coordinate (COO)

Figure 7.4 on the following page is the simplest truly sparse format: coordinate (COO). In such a format, in addition to the number of rows and columns in the matrix, three values must be maintained per nonzero: the floating-point value, the row index, and the column index. Typically, a separate array is maintained for each attribute. Given a column and row index, it is expensive to extract the matrix value. However, many kernels can be restructured so that rather than looping through all rows and then all columns, they are data-oriented and stream through the nonzero arrays and perform an operation according to the coordinate. That is, it is easy to extract the row and column of the i^{th} nonzero. The size of a matrix in this format is $16 \cdot NNZ$ bytes in double-precision. Clearly, for large m×n, this approach is orders of magnitude more efficient than dense storage. Although not required, the nonzeros could be sorted by row, column or, hierarchically to exploit locality in certain kernel operations. In doing so, one could observe redundancy among row or column indices. Such observations motivate the use of the next two formats.

7.3.2 Compressed Sparse Row (CSR)

Given a row-sorted COO format matrix, there are blocks of nonzeros all on the same row. As such, there are blocks of elements in the row index array that all have the same row index. One could eliminate an explicit row index array in favor a row pointer array. Figure 7.5 on the next page shows the compressed sparse row (CSR) format. For every row in the original matrix, there is an element in the row pointer array. These pointers denote the indices of the first and last nonzeros in the value and column index arrays for each row. In essence, all nonzeros within a row have been packed together. Subsequently all rows are packed together. Such formats reduce memory traffic and minimize the computation for

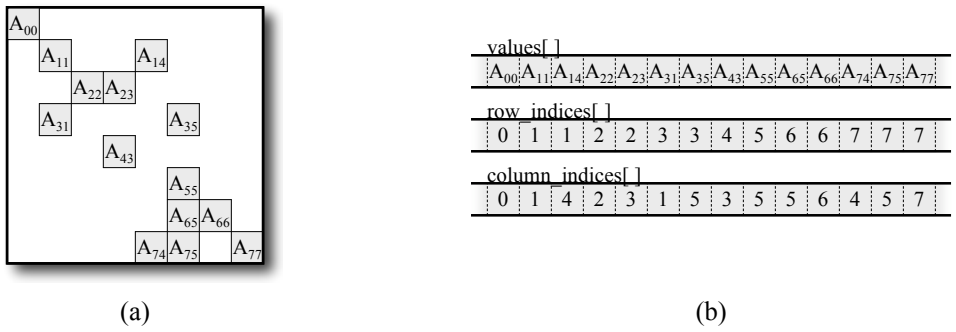


Figure 7.4: Coordinate format (COO). (a) spyplot for a small sparse matrixi, (b) the resultant data structure for COO format.

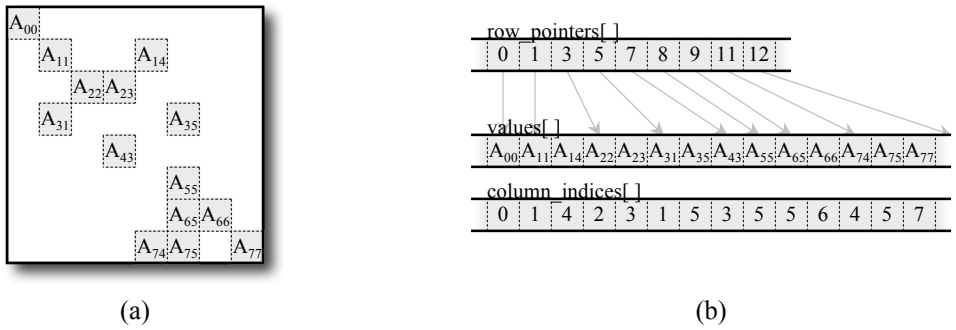


Figure 7.5: Compressed Sparse Row format (CSR). (a) spyplot for a small sparse matrix, (b) the resultant data structure for CSR format.

many kernels. This approach requires 12·NNZ + 4·m bytes to store the matrix. A similar approach for nonzeros sorted by column results in compressed sparse column (CSC).

In itself, many kernels are not efficient operating on matrices with short rows (few nonzeros per row) when the matrix is stored in CSR. Moreover, branchless implementations of SpMV are not possible if empty rows are present. A solution to both problems can be found by eliminating the empty rows and storing a row index for each of the remaining rows — GCSR in OSKI parlance[135].

7.3.3 ELLPACK (ELL)

Many sparse matrices have about the same number of nonzeros per row. By adding explicit zeros until all rows have equal length, the row pointers can be eliminated. The resultant format is known as ELLPACK (ELL). Figure 7.6 on the following page shows the storage of a sparse matrix in ELLPACK. The pointers can be calculated at runtime based on the row index and the new row length. An additional benefit is that it is easy to

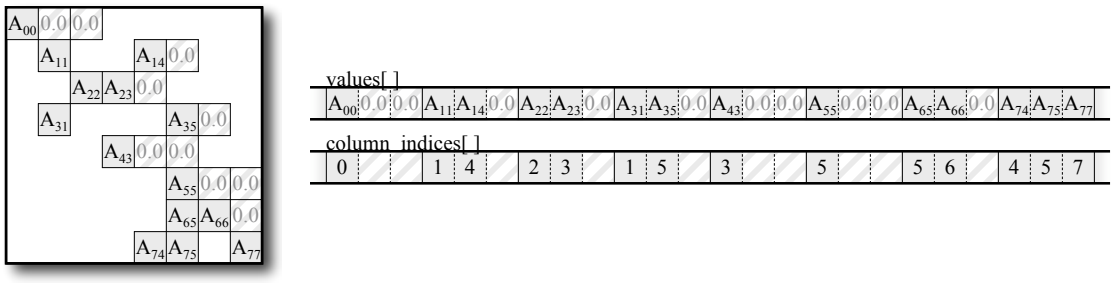


Figure 7.6: ELLPACK format (ELL). (a) spyplot for a small sparse matrix, (b) the resultant data structure for ELL format. Notice, after the addition of explicit zeros, all rows are the same length.

vectorize across several rows at a time. Ideally, this approach only requires 12 bytes per nonzero. However, if the maximum number of nonzeros per row is substantially greater than the average number of nonzeros per row, then this format will require significantly more storage than any other format.

7.3.4 Skyline (SKY)

There are many matrices for which all the nonzeros appear near the diagonal. By adding explicit nonzeros around the diagonal, one could simply maintain a row pointer array and eliminate the explicit column indices as they are now implicit in the format. That is, given a row, one indexes the row pointers array to determine the values. In addition, the row pointer array allows one to determine the column index of the first nonzero within that row. As all nonzeros are densely packed to the diagonal, the column indices of the remaining nonzeros fall out. This format is known as skyline (SKY) and ideally only requires 8 bytes per nonzero and 4 bytes per row. However, this can be substantially increased by the number of explicit zeros that must be added to adhere to this format. Figure 7.7 shows a lower triangular matrix stored in the Skyline format. Notice explicit zeros are inserted to create a dense band up to the diagonal.

7.3.5 Symmetric and Hermitian Optimizations

Symmetric ($A_{ij} = A_{ji}$) or Hermitian ($A_{ij} = A^*_{ji}$) matrices can stored in either a naïve non-symmetric format or an optimized format. The non-symmetric format treats the matrix as if it were non-symmetric and thus stores all NNZ nonzeros. The symmetry-aware formats typically store either the lower or upper triangular matrices including the diagonal. Figure 7.8 on the next page shows an symmetric matrix stored in the optimized format. Ideally, this matrix would only require $6 \cdot NNZ + 4 \cdot m$ bytes as up to half the nonzeros are redundant. Although this format requires substantially less memory traffic, it can only be used in conjunction with highly optimized kernels.

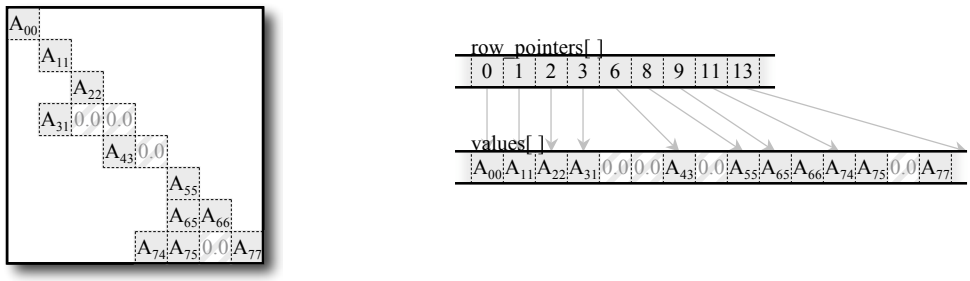


Figure 7.7: Skyline format (SKY). (a) spyplot for a small, lower triangular sparse matrix, (b) the resultant data structure for SKY format.

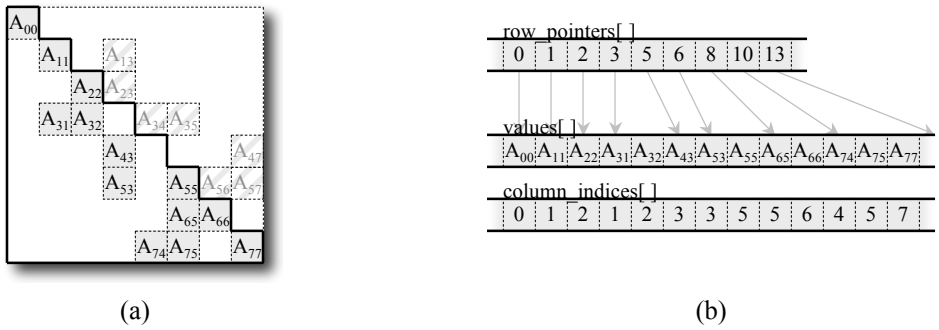


Figure 7.8: Symmetric storage in CSR format: (a) spyplot for a small, symmetric sparse matrix. Note $A_{ij} = A_{ji}$, (b) the resultant data structure for the CSR format after exploitation of symmetry.

7.3.6 Summary

Table 7.2 on the following page provides a brief summary of the sparse matrix formats discussed in this chapter. Clearly, dense provides a lower bound, and CSR and COO are within a factor of two of the dense storage requirement. Symmetric storage is only applicable for symmetric matrices and can nearly cut the storage requirements in half (full row pointers must be maintained). The selection of an optimal format is heavily dependent on matrix sparsity. Moreover, the kernels that operate on these formats should be implemented with the format's natural matrix traversal. Note, these formats are the subset we believe will continue to have value in the multicore era.

Dozens of other formats have been suggested to address the needs of architecture, sparsity, and kernel. However, the justifications for many of these formats have been obviated by obsolescence of various architectures and implementations. Moreover, one must remember, that the implementation of a kernel is completely disjoint from the matrix format

Storage Format	Storage Requirement (bytes)	Natural Matrix Traversal	Motivation and Applicability
Dense (row major)	8·NNZ = 8·N^2	by rows, or blocks of rows	large, dense blocks
Dense (column major)	8·NNZ = 8·N^2	by columns, or blocks of columns	large, dense blocks
COO	16·NNZ	order specified by sorting	extreme sparsity, short rows
CSR	12·NNZ+4·N	by rows	long rows, structured
ELL	12·N·(Max Nonzeros Per Row)	by rows, or blocks of rows	facilitates vectorization, near equal row lengths
SKY	8ΣRowBandwidth$_j$+4·N	by rows	low bandwidth

Table 7.2: Summary of storage formats for sparse matrices relevant in the multicore era. Many other formats have been omitted because multicore obviates the needs for efficient vectorization.

selected. That is, for a given kernel, there are a number of possible implementations that all use the CSR format. For example, SpMV is a kernel, CSR is a format, and segmented scan is an implementation. In Chapter 8 we perform a limited exploration of format and kernel implementation.

Thus far, we have only discussed sparsity and corresponding formats. We have not specified the data type of a_{ij}. It is not uncommon for matrices to be other than simply double-precision floating-point numbers. The nonzeros can be real or complex, and they values can be integers, fixed point, single, double, or double-double-precision floating-point numbers. Moreover, when dealing with complex or double-double numbers the two components can be stored as either an array-of-structures (AOS) or structure-of-arrays (SOA).

7.4 Conclusions

In this chapter we provided an overview of the sparse linear algebra motif. We started with a discussion of the characteristics of sparse matrices in Section 7.1. The primary difference from the dense linear algebra motif, is the fact that most matrix entries are zero. As such, due the reduced storage requirements, the dimensions of the typical sparse matrix is several orders of magnitude larger than the typical dense matrix. Unfortunately, the price for such algorithmic and storage efficiency is architectural inefficiency. That is, the complexity of the code required to implicitly reconstruct matrix structure is very high. As a result, the performance of such code is invariably low.

Section 7.2 discussed a number of sparse kernels and methods and their applicability to partial differential equations on structured and unstructured grid. We observe the two principal kernels — sparse matrix-vector multiplication (SpMV) and sparse triangular solve (SpTS) — have poor locality and little reuse. The resultant low arithmetic intensity implies these kernels will be at best memory-bound on any foreseeable future multicore computer, and without proper expression of memory-level parallelism, they will likely be

latency limited. As such, all efficiency-oriented optimization efforts should focus on expressing sufficient memory-level parallelism, and minimizing the total memory traffic. To that end, we may view sparse computations as a DAG. However, unlike a structure grid DAG, it might be preferable to view the nodes as primitive computation operating on one matrix element rather than an entire row with the prevision that certain rewrite rules exist. Although parallelism in SpMV is obvious, parallelism in SpTS is determined though DAG inspection.

Section 7.3 discussed several common formats for representing sparse matrices selected based on their propensity to minimize memory traffic. Although other formats express more instruction- or data-level parallelism, we believe multicore obviates their need. Ultimately, the selection of matrix format is highly tied to sparsity, computation, and even the underlying computer the computation will be performed on. Thus, when a matrix is defined, it should be inspected and it should be stored in the appropriate representation.

The entirety of the next chapter is dedicated to auto-tuning sparse matrix-vector multiplication (SpMV) on the multicore computers presented in Chapter 8.

Chapter 8

Auto-tuning Sparse Matrix-Vector Multiplication

This chapter presents the results of applying auto-tuning to the Sparse Matrix-Vector Multiplication (SpMV). We observe that auto-tuning provides a performance portable solution across cache-based microprocessors. However, we sacrifice portability and productivity when porting to local store architectures for only modest performance gains.

Section 8.1 delves into the details of the reference SpMV implementation, useful optimizations, and previous serial auto-tuning efforts. Section 8.2 uses the Roofline model introduced in Chapter 4 to estimate attainable SpMV performance, as well as enumerate the optimizations required to achieve it. Section 8.3 describes the benchmark matrices for this kernel. Section 8.4 walks through each optimization as it is added to the search space explored by the auto-tuner. At each step, performance and efficiency are also reported and analyzed. In addition, the final fully-tuned performance is overlaid on the Roofline model. Section 8.5 summarizes, analyzes, and compares the performance across architectures. In addition, a brief discussion of productivity is included. Although significant optimization effort was applied in this work, Section 8.6 discusses a few alternate approaches that may be explored at a later date. Finally, Section 8.7 provides a few concluding remarks.

8.1 SpMV Background and Related Work

In our examination of auto-tuning of sparse methods presented in this chapter, we chose to restrict ourselves to an important kernel that we believe embodies the bulk of the optimizations that would be applicable to any sparse kernel. To that end, we chose sparse matrix-vector multiplication (SpMV) as an example sparse kernel and extend the work presented in [140]. This section performs a case study SpMV detailing the kernel, issues, and some of the previous auto-tuning efforts. The rest of the chapter is dedicated to the study of auto-tuning SpMV on multicore architectures.

```
for(r=0;r<A.m;r++){
  yr = 0.0;
  for(i=A.ptr[r];i<A.ptr[r+1];r++){
    c = A.col[i];
    yr += A.values[i]*x[c];
  }
  y[r] = yr;
}
```

Figure 8.1: Out-of-the-box SpMV implementation for matrices stored in CSR.

8.1.1 Standard Implementation

We restrict this case study to SpMV on a non-symmetric double-precision matrix. Thus, we investigate the evaluation of $y = Ax$ where A is a m×n sparse matrix with NNZ nonzeros, and x and y are dense vectors. Evaluation of $y = Ax$ is defined as $\forall\ a_{ij} \neq\ 0, y_i = y_i + a_{ij} \cdot\ x_j$. We describe x as the source vector and y as the destination vector. Although A is typically so large that it will not fit in cache, x and y might fit in cache. Notice that regardless of sparsity, every y_i can be calculated independently and in any order. Such characteristics make naïve parallelization of SpMV easy. Efficient parallelization can remain a challenge. Contrast this to SpTS ($Lx = b$) in which there are dependencies between x_j's. As such, different orderings and degrees of parallelization are dependent on sparsity.

Decades of research on such a kernel has produced a myriad of representations of A and an even broader variety of implementations of the SpMV kernel. Nevertheless, the most common representation of the matrix is compressed sparse row (CSR) using 32-bit indices and pointers. Moreover, the standard implementation of the SpMV operation on a CSR matrix is a nested loop over all rows and over all nonzeros within said row. Figure 8.1 shows a C implementation of this approach.

Let us consider the performance pitfalls of such an implementation. `A.values[]` and `A.col[]` are very large arrays that are streamed through once per SpMV. As such, they will generate one miss per cache line. Second, for low bandwidth matrices, both `x[c]` and `y[r]` will likely have a high cache hit rates or at least do not significantly impair performance. However, if the rows are short (`A.ptr[r+1]-A.ptr[r]` is small), then the inner loops are short, and the overhead of starting a loop cannot be amortized. This overhead is critical on the Itanium architecture where the overhead involved in starting a hardware-accelerated software pipelined loop is substantial. Finally, there is no instruction- or data-level parallelism within the inner loop. As a result, on deeply pipelined superscalar SIMD architectures, performance will be far from peak.

8.1.2 Benchmarking SpMV

Typically, when benchmarking SpMV performance, one only measures the asymptotic SpMV performance. Thus, we ignore the initial matrix load and preparation time and run enough trials to warm the caches and TLBs. Moreover, to replicate typical usage on a parallel machine, the vectors are swapped between SpMV's. In effect, the typical

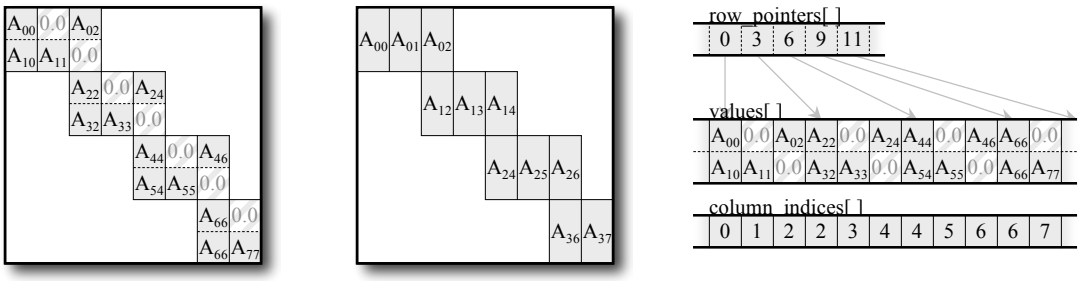


Figure 8.2: Matrix storage in BCSR format: (a) spyplot for a small matrix with zeros added to facilitate register blocking, (b) conceptualization where each matrix entry is a 2×1 dense matrix, (c) the resultant data structure.

benchmark loops over the following operations:

$$\left| \begin{array}{l} y = Ax \\ x = Ay \end{array} \right.$$

If the matrix has a high bandwidth, a significant volume of data (the vectors) will be sent between caches, cores, and processors. As a result, performance will be diminished.

8.1.3 Optimizations

There are a number of commonly used optimizations that have been applied to SpMV. We detail them here.

Branchless implementations restructure the nested loops of Figure 8.1 into a single loop using conditional operations either in the form of predication or via bitwise muxing. The advantage is that this minimizes the performance impact of matrices with few nonzeros per row. Our preliminary investigations have shown this to be an effective solution on architectures like Itanium2 or the Cell SPEs. On predicated architectures, this method can be extended into segmented scan [19]. Such an approach expresses more parallelism within the inner loop and, for appropriate vector lengths, results in better performance.

To express more parallelism, one generally wants to calculate several y_i simultaneously. Moreover, to capture locality within the register, rather than the last level cache, one wants to reuse the element x_j or elements near x_j for successive nonzeros. Blocked Compressed Sparse Row (BCSR) addresses these issues by changing the minimum quanta for an element from a nonzero to an aligned r×c dense matrix. Thus, every "r" rows are grouped into a blocked row, and every "c" columns within that blocked row are grouped into an r×c sub-matrix. The original m×n matrix is now a $\frac{m}{r} \times \frac{n}{c}$ matrix where each entry is an r×c dense matrix, now called a *register block*, in which it is now tolerable to have explicit zeros. Conceptually, the nonzero multiply-add operation has been transformed into a matrix-vector multiply-add on a small r×c matrix and c×1 vector.

Many problems naturally produce a regular blocked structure. Consider a operations on a structured grid of Cartesian vectors. When mapped to a dense vector for sparse

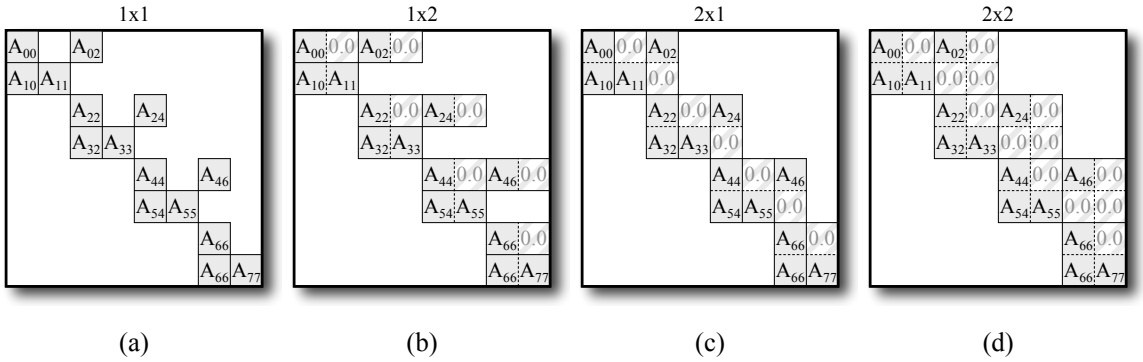


Figure 8.3: Four possible BCSR register blockings of a matrix. Note, (b) and (c) fill in seven zeros, but (d) fills in 13.

linear algebra, the *(i,j,k)* Cartesian components would likely occupy sequential vector elements. Often the stencil operator will calculate a new *(i,j,k)* based on the current *(i,j,k)*. Thus a natural 3×3 blocking arises.

Figure 8.2 on the preceding page illustrates BCSR. Starting with a sparse matrix of nonzeros in Figure 8.2(a), one can reorganize the matrix into small 2×1 dense blocks rather than individual nonzeros — Figure 8.2(b). Only one column index is stored per r×c tile, and only one row pointer is required per blocked row. Figure 8.2(c) shows the typical data structures for BCSR. There are six ways one could store the tiles, as the nonzeros are basically a 3D array indexed by tile index, row within a tile, and column within a tile. Efficient implementations store the nonzeros in as an array of structures with each tile stored in a dense column major format. That is `double values[tile][col][row];`

Typically, the tile×vector product in SpMV is completely unrolled. As such, the inner loop loops over tiles in a blocked row, and the outer loop loops over all blocked rows. Observe that there is r-way explicit data-level parallelism, and only $\frac{m}{r}$ for loop starts. However, the raw number of requisite floating-point operations may have dramatically increased due to the fill of nonzeros. As such, effective performance as measured in $\frac{useful\ FLOPs}{total\ time} = \frac{2 \cdot NNZ}{total\ time}$ is both sparsity and machine dependent.

8.1.4 OSKI

A question arises: what is the optimal register block size for a given matrix on a particular machine? Assume that once an optimal encoding is chosen, thousands of SpMV's will be performed. As such, one can amortize some exploration of possible register blockings. To that end, one could bound the maximum register block size to something like 16×16, convert A to each of the 256 different combinations of r and c, and individually benchmark them (perhaps ten SpMVs each). The optimal choice would need to be used tens of thousands of times to amortize the exploration.

Berkeley's Optimized Sparse Kernel Interface (OSKI) [135] provides an elegant solution to this problem. First, it is an auto-tuned library that encapsulates many tuning and sparse kernels into routines. Upon installation, it benchmarks the performance of every

possible register blocking on the target machine. Then, at runtime, it samples a portion of the matrix to be tuned, and will estimate the useful FLOP rate for every possible register blocking using the benchmark data and projected number of nonzero fills.

For example, Figure 8.3 on the previous page shows a sampling of a matrix blocked with four different register blockings. All four resultant forms are numerically identical. However, their effective SpMV performance may be dramatically different. Suppose, 1×1 BCSR can achieve a FLOP rate of 1 GFLOP/s for a dense matrix stored in sparse format, but 1×2, 2×1, and 2×2 can achieve 1.2, 2.0, and 2.2 GFLOP/s, respectively. We can calculate the effective FLOP rate as the product of the raw FLOP rate and the ratio of nonzeros to matrix entries. Thus, Figure 8.3(a)-(d) should achieve at best 1.0, 0.81, 1.36, and 1.17 GFLOP/s respectively. Thus, OSKI would conclude that for this matrix, 2×1 BCSR will likely deliver the best performance.

OSKI was originally vetted on older single core processors. Nevertheless, it should substantial performance benefits on the Itanium2 and other now obsolete architectures.

8.1.5 OSKI's Failings and Limitations

Despite its successes, there are some key failings and limitations for OSKI that we discuss here. The purpose is to motivate our work rather than deride OSKI.

OSKI is natively a serial library. Although, the auto-tuned kernels it produces can be integrated into an MPI-based parallel distributed memory framework like PETSc [11] with relatively little work, even when using an optimized shared memory MPICH implementation results have shown that scalability of this parallelization strategy was often lacking on many multicore SMPs [140]. There are two principal failings here: OSKI tunes kernels serially and thus assumes the entire socket bandwidth is always available to any and every core. Clearly, when all cores are running, each core can only be guaranteed a small fraction of a socket's bandwidth. Thus, OSKI may choose a register blocking that when run in isolation delivers good performance because the core isn't memory limited. However, that register blocking may have increased the size of the matrix data structures. As such, when run in conjunction with other cores on a memory limited multicore architecture, performance will be reduced because a larger volume of data must be transfered.

The second failing is that PETSc uses explicit messaging to pass newly calculated destination vectors to the other processors for subsequent consumption as source vectors. Every byte of bandwidth used for explicit communication is both redundant on a cache coherent shared memory parallel machine, and strips bandwidth away from where its needed: computation. Finally, many SMPs are NUMA architectures. As such, data must be correctly placed to attain peak bandwidth. It is likely MPI can do this implicitly, but it must be handled explicitly on threaded applications.

OSKI's final limitation is its lack of architecture- or ISA-specific optimizations. For instance, many architectures require software prefetch, cache bypass instructions, or SIMD instructions to attain peak performance. Many modern compilers are incapable of appropriately exploiting these instructions. As such, the responsibility falls to either the library or the user to correctly insert them.

8.2 Multicore Performance Modeling

Before diving into construction of an auto-tuned multicore implementation of Sparse Matrix-Vector Multiplication (SpMV), we first extract the relevant characteristics of the kernel, map those characteristics onto a Roofline model for each architecture, and estimate both performance and the requisite optimizations for each machine. During this analysis, we assume a warm started cache for only the vectors — access to the matrix will generate additional compulsory misses, but the compulsory misses associated with the vectors is amortized.

8.2.1 Parallelism within SpMV

Figure 8.1 on page 152 shows the standard nested loop implementation of SpMV for a compressed sparse row (CSR) matrix format. It performs a multiply accumulate per iteration of the innermost loop. Thus, like many linear algebra routines, the use of fused multiply add (FMA) is explicit. On non-FMA architectures, instead of the typical imbalance between multiplies and adds in many motifs, there is an inherent balance between multiplies and adds. However, for a reasonably sized instruction window — less than a row — there is no other floating-point instruction-level parallelism (ILP). This lack of ILP also implies there is no data-level parallelism (DLP). As discussed in Chapter 4, a total lack of ILP and DLP can profoundly impair performance. In addition, each loop (row) in this nested loop kernel has significant startup overhead. When coupled with the requisite indirect addressing, the floating-point fraction of the total dynamic instruction mix is diminished.

As discussed in Section 8.1.3, OSKI uses register blocking (BCSR) to improve performance. In the context of ILP, DLP, and operation mix, register blocking can significantly increase DLP — and thus ILP — as well as amortize the loop overhead. Each additional row in a register block provides additional DLP, and the loop overhead is executed once per register block rather than once per nonzero.

There is no explicit thread-level parallel (TLP) in the standard CSR SpMV implementation. Nevertheless, we can apply an OpenMP style of loop-level parallelism on the outermost loop. We chose to implement it with `pthreads`.

There is no temporal locality among the value, column index, row pointer, and destination vector arrays of a matrix within a SpMV. Thus, there only needs to be sufficient cache or local store capacity to satisfy Little's Law. For today's typical latency-bandwidth product, we need less than 10 KB across an entire SMP. As these access to these arrays are all unit-stride, there is plenty of spatial locality.

When it comes to the source vector, there is only moderate overall temporal locality — each double is typically used 6 to 50 times. However, given limited cache capacities and large working set requirements, the actual reuse in practice can be significantly lower since cache lines are evicted before the data can be reused. In addition, with larger capacities comes higher spatial locality, as cache lines remain in the cache until the seemingly random accesses associated with the source vector have touched every double in a given cache line.

The degree of memory-level parallelism in SpMV is completely specified when the matrix is created. This includes all matrix values, column indices, row pointers, and all vector accesses. This parallelism — $O(NNZ) + O(N)$ — vastly exceeds the capabilities

Storage Format	FP Parallelism by type				Instructions per multiply-add	Memory streams per thread
	Instruction	Data	Thread	Memory		
standard CSR	≈ 1	≈ 1	$\approx N$	$\approx NNZ + N$	$\approx 11 + \frac{10N}{NNZ}$	2
R×C BCSR	≈ 1	$\approx R$	$\approx \frac{N}{R}$	$\approx NNZ + N$	$\approx 3 + \frac{1}{R} + \frac{7}{RC} + \frac{10N}{R \cdot NNZ}$	2
R×C BCOO	≈ 1	$\approx R$	$\approx \frac{N}{R}$	$\approx NNZ + N$	$\approx 3 + \frac{1}{R} + \frac{7}{RC} + \frac{10N}{R \cdot NNZ}$	3

Table 8.1: Degree of parallelism for a N×N matrix with NNZ nonzeros stored in two different formats. Data- and thread-level parallelism are clearly linked to data structure.

of any architecture. Although a load-store queue will only afford a few dozen accesses, hardware prefetchers can be very effective in prefetching the value, column index, row pointer, and destination vector arrays. The latency associated with randomly accessing source vector elements cannot be covered by either hardware prefetchers or out of order execution. Both multithreading and DMA lists will be effective for all accesses due to the magnitude of MLP inherent in the algorithm.

Table 8.1 summarizes the parallelism in the standard OpenMP style implementations of SpMV on an N×N matrix. The constants are based on disassembly of SpMV kernels on a SPARC architecture. Clearly, DLP increases with register blocking. The second to last column shows the instruction overhead per floating-point multiply-add for three matrix storage formats. The CSR implementations are dominated by two terms: overhead per row and overheads per nonzero. In a sparse linear algebra, a long row is a row with a large number of nonzeros. When the average row is long ($\frac{NNZ}{N}$ is large), the overhead per nonzero dominates ($11 \gg \frac{10N}{NNZ}$), and the floating-point mix approaches one FMA per 11 instructions. However, for short rows ($NNZ \sim N$), then the two terms are approximately equal, and the floating-point mix is cut in half. Clearly, when examining the register blocked implementations, there are four terms: outer loop overhead, index calculation, source vector loads, and FMAs. Register blocks spanning multiple rows decrease the overall number of outer loop iterations, as well as drive down the average number of indexing and source vector loads per nonzero. Register blocks spanning multiple columns also amortize the array indexing.

The conclusion is that register blocking can significantly increase performance by increasing data level parallelism and the floating-point instruction mix. However, its effectiveness is bounded by the instruction latencies, register pressure, and the fact that very large register blocks may significantly increase the total memory traffic since zeros must be explicitly added to the block.

8.2.2 SpMV Arithmetic Intensity

Chapter 4 introduced the Roofline performance model. Given an arithmetic intensity, one can use the Roofline model to predict performance. Each iteration of the innermost loop of a CSR SpMV implementation multiplies one nonzero matrix entry by one vector element and accumulates the result — two floating-point operations. Thus, a SpMV performs $2 \times NNZ$ floating-point operations. In addition, each nonzero is represented by a

double-precision value and a column index. The column index is typically a 32-bit integer. As a result, each SpMV must read at least $12 \times NNZ$ bytes. This lower limit assumes no cache misses associated with the vectors. For most matrices, the write traffic is much, much smaller than the read traffic. Putting it together, SpMV has a compulsory arithmetic intensity less than $\frac{2 \times NNZ}{12 \times NNZ}$, or about 0.166.

The register blocking used in OSKI encodes only one column index per register block. As register blocks can grow quite large, it is possible to amortize this one integer among dozens of nonzeros. Asymptotically, register blocked SpMV can reach a FLOP:compulsory byte ratio of 0.25 — a 50% improvement. This upper bound presumes no explicit zeros were added.

8.2.3 Mapping SpMV onto the Roofline model

Figure 8.4 on the next page maps SpMV's FLOP:compulsory byte ratio onto the Roofline performance model discussed in Chapter 4. From this figure, we should be able to predict performance, and which optimizations should be important across architectures. We continue to refer to this Roofline throughout this chapter.

The dashed red lines on the left in Figure 8.4 on the following page denote the SpMV FLOP:compulsory byte ratio for the standard CSR implementation, while the green dashed lines to the right mark the FLOP:compulsory byte ratio for the best possible register blocked SpMV implementation. The purple shaded region highlights the performance range corresponding to a range in potential arithmetic intensities arising from varying degrees of success when register blocking. The lowermost diagonal denotes unit-stride performance without any optimization — a likely access pattern for the value and index arrays. Out-of-the box SpMV performance is expected to fall on or to the left of the red dashed vertical line due to the potential for vector capacity misses or conflict misses on any access reducing the arithmetic intensity. Additionally, performance should fall below the 'without ILP' ceiling. Finally, it is likely that non-floating-point operations will constitute the bulk (greater than 80%) of the dynamic instruction mix. However, this overhead is only an issue on Victoria Falls.

8.2.4 Performance Expectations

Figure 8.4 on the next page suggests the Clovertown will be heavily memory-bound. Not only will register blocking help by reducing memory traffic, but for sufficiently small matrices, a super linear benefit will occur. This benefit arises because the snoop filter will eliminate more snoop traffic on a small matrix than a large one. We generally expect performance to be between 1 and 2 GFLOP/s without register blocking, and perhaps up to 3 GFLOP/s with perfect register blocking.

The Roofline for the Santa Rosa Opteron shows significant variation in attainable memory bandwidth before and after optimization. As a result, we expect to see profound improvements in performance with NUMA, register blocking and prefetching: from 0.7 GFLOP/s to 4.0 GFLOP/s. At the highest possible performance, SIMD or ILP may need to be exploited.

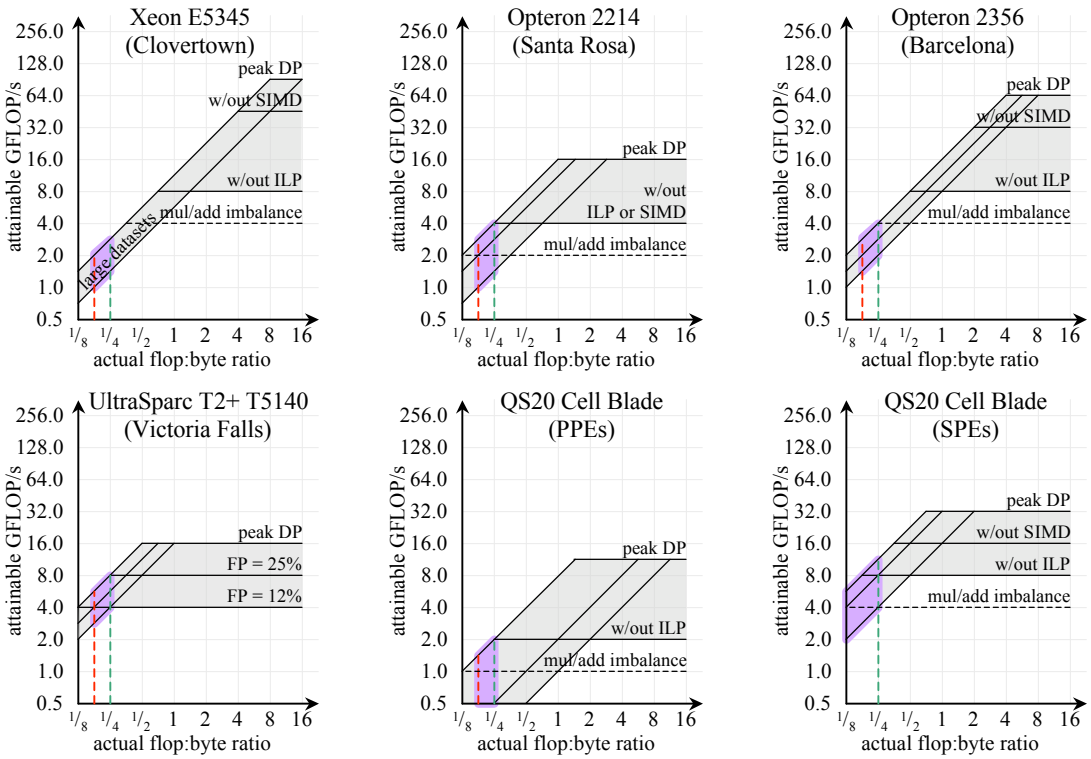


Figure 8.4: Expected range of SpMV performance imposed over a Roofline model of SpMV. Note the lowest bandwidth diagonal assumes a unit-stride access pattern. Note the log-log scale.

Although the Barcelona Opteron has the same raw pin bandwidth, the Roofline suggests multicore may effectively obviate some of the need for NUMA optimizations, and ILP and SIMDization will be of lesser value. Nevertheless, we expect performance between 1.4 and 2.8 GFLOP/s without register blocking, and perhaps over 4 GFLOP/s with perfect register blocking. SIMD and ILP will be of much less value on Barcelona.

On Victoria Falls, the limited instruction issue bandwidth suggests that the large number of non-floating-point operations may limit performance. The Roofline suggests that NUMA, prefetching, and register blocking will be essential in delivering up to 8 GFLOP/s. Without these optimizations, performance will fall to about 6, 4, or 2 GFLOP/s respectively.

Without optimization, the Roofline suggests Cell PPE performance will be abysmal. Even after full opimization — heavily dependent on prefetching — performance is still expected to be less than 2 GFLOP/s. In order for Cell SPE performance to hit a 12 GFLOP/s peak, it will require NUMA, effective DMA double buffering, and register blocking. In addition, local store blocking is required to avoid copious redundant source vector accesses.

On all computers, memory bandwidth optimizations must be coupled with memory traffic reduction techniques. SIMDization or other exotic vectorization techniques are not likely to be beneficial as the architectures are primarily bound by memory bandwidth rather than a lack of instruction or data-level parallelism.

Large bandwidth matrices may generate large numbers of capacity misses. As such, their arithmetic intensities will be less than ideal. As a result, SpMV performance will likely be poor on those matrices. Cache blocking techniques may be applicable.

8.3 Matrices for SpMV

Unlike the lattice Boltzmann method discussed in Chapters 5 and 6, the problem dimensions alone do not specify a sparse matrix. As discussed in Chapter 7, sparse matrices are characterized by both dimension and sparsity — which matrix elements are nonzero. To that end, we have created a dataset of 14 sparse matrices extracted from the SPARSITY [72] matrix suite. The criteria for selection were:

- **Size** — the matrices should not fit in any chip's cache.
- **Sparsity** — they should contain a range of sparsity patterns.
- **Relevance** — they should be representative of a wide range of actual applications.

Figure 8.5 on the following page shows the characteristics of each of our selected matrices. Note they have been grouped into four categories: one matrix is dense, nine are well structured, three are poorly structured, and one is an extreme aspect ratio matrix. Matrices are further sorted by the number of nonzeros per row (NNZ) — a parameter correlated to loop overhead amortization. We use this ordering of the matrices throughout the rest of this chapter.

A few observations of the matrices will aid in the analysis later in this chapter. In Figure 8.5 on the next page, the last two rows estimate the each shared cache's requisite capacity for both the matrix stored in CSR as well as the vectors as the number of such caches increases. When the either of these sizes exceeds the actual cache sizes, capacity misses will certainly occur between SpMVs. For poorly structured matrices, capacity misses may occur within each SpMV. Note, the last four matrices have cache capacity requirements $\approx$1MB or greater for the vectors. Thus, we expect low performance on most architectures. Furthermore, those matrices will require significant data movement between sockets across successive SpMVs. Conversely, we generally expect the Dense matrix stored in sparse format to provide an upper bound to performance. Additionally, matrices such as QCD, Economics, and Circuit are significantly smaller than Clovertown's snoop filter. As such, they will see significantly better bandwidth through the snoop filter eliminating unnecessary coherency traffic. Finally, as the number of nonzeros per row decreases, the loop overhead will tend to dominate kernel time. As a result, we expect CSR performance to continually decrease from the Protein matrix to the Epidemiology matrix.

8.4 Auto-tuning SpMV

Section 8.1.3 discussed the myriad of matrix formats and optimizations designed to maximize SpMV performance. In this work, we chose to restrict ourselves to only changing the matrix data structure and the SpMV kernel. The vectors are always the standard dense

	Dense	Protein	Spheres	Cantilever	Wind Tunnel	Harbor	QCD	Ship	Economics	Epidemiology	Accelerator	Circuit	webbase	LP
Spyplot														
Rows	2K	36K	83K	62K	218K	47K	49K	141K	207K	526K	121K	171K	1M	4K
Cols	2K	36K	83K	62K	218K	47K	49K	141K	207K	526K	121K	171K	1M	1M
NNZ	4.0M	4.3M	6.0M	4.0M	11.6M	2.4M	1.9M	4.0M	1.3M	2.1M	2.6M	0.9M	3.1M	11.3M
average NNZ/Row	2000	119	72	65	53	50	39	28	6	4	22	6	3	2825
Symmetric	-	✓	✓	✓	✓	-	-	✓	-	-	✓	-	-	-
Matrix Footprint	48 / N	52 / N	72 / N	48 / N	140 / N	29 / N	23 / N	49 / N	16 / N	27 / N	32 / N	12 / N	41 / N	136 / N
Vector Footprint	0.03	0.6 / N	1.3 / N	1.0 / N	3.3 / N	0.7 / N	0.8 / N	2.3 / N	3.3 / N	8.0 / N	0.9 - 1.8	1.3 - 2.6	7.6 - 15.3	7.6

Figure 8.5: Matrix suite used during auto-tuning and evaluation sorted by category, then by the number of nonzeros per row. Footprint is measured in MB. Partitioning implies the requisite cache capacity drops with the number (N) of shared caches exploited.

vectors. Furthermore, we explore only two of the matrix formats, but add several other optimization spaces. Within each of these new optimizations is a large parameter space. To efficiently explore this optimization space, we employed an auto-tuning methodology similar to that seen in libraries such as ATLAS [138], OSKI [135], and SPIRAL [97]. A large number of kernel 'variations' are produced and individually timed. Performance determines the winner.

Once again, the first step in auto-tuning is the creation of a code generator. We wrote a Perl script that generates all the variations of SpMV for all matrix formats explored. For the SIMD architectures, we implemented an additional Perl script to generate SSE intrinsic-laden C code variants. Note that the code generation and auto-tuning process on Cell is a significantly restricted and simplified approach. Future work will expand the auto-tuning approach on Cell.

The code generator can generate hundreds of variations for the SpMV operation. They are all placed into a pointer to function table indexed by the optimizations and matrix format. To determine the best configuration for a given matrix and thread concurrency, we run a tuning benchmark to search the space of possible code and data structure optimizations. In some cases the search space may be heuristically pruned of optimizations and storage formats unlikely to improve performance. In a production environment, the time required for this one time tuning is amortized by the time required to load the matrix from disk and number of sparse matrix-vector multiplications in the full code. As the auto-tuner searches through the optimization space, we measure the per multiplication performance averaged over a ten time step trial and report the best.

In the following sections, we add optimizations to our code generation and auto-

tuning framework. At each step we benchmark the performance of all architectures exploiting the full capability of the auto-tuner implemented to that point. Thus, at each stage we can make an inter-architecture performance comparison at equivalent productivity, allowing for commentary on the relative performance of each architecture with a productive subset of the optimizations implemented. We have ordered the optimizations from those that are easiest to implement — simple outer loop parallelization — to the most complex — register, cache and TLB blocking.

8.4.1 Maximizing In-core Performance

One generally expects SpMV to be memory-bound. However, on architectures with limited numbers of cores or weak double-precision implementations, in-core performance can limit SpMV performance. To that end, we designed our code generators to produce code superior to the standard BCSR and BCOO implementations.

First, SIMD optimized code was generated on the relevant architectures. Given the simplicity of the SpMV inner loop, this was very easy. Second, computers without branch prediction like Cell will suffer greatly when on average there are few nonzeros per row, even when using BCOO. To ameliorate this, we experimented with branchless implementations (software predication) on both Cell and the x86 architectures. The technique had value on Cell, but not on the x86 architectures. Thus, it was not subsequently used on the x86 architectures. Finally, we included a software pipelined implementation on Cell — once again, there was no gain on the x86 architectures. This optimization is designed to hide the longer instruction and local store latency.

8.4.2 Parallelization, Load Balancing, and Array Padding

In order to provide a baseline for our multicore auto-tuning, we first run the standard serial CSR SpMV implementation on our multicore SMPs. The lowest bars in Figure 8.8 on page 165 show Clovertown, Santa Rosa, and Barcelona all deliver comparable out-of-the-box single-thread performance. Not only is this performance an abysmal fraction of peak FLOPs, it is also only 14% of peak DRAM bandwidth. Despite this poor performance, the x86 cores are an order of magnitude faster than a single thread on Victoria Falls or the Cell PPE. Clearly, we must explicitly exploit thread-level parallelism to achieve good performance on those computers.

As discussed in Chapter 7, in a matrix-vector multiplication, there is no dependency between rows. As such, parallelization by rows ensures there are no data dependencies or reductions of private vectors. Rather than employing the standard loop parallelization techniques, Figure 8.6(b) on page 163 exemplifies how we partition each matrix by rows into disjoint thread blocks. There is one matrix thread block per thread of execution. Each thread will perform its own submatrix-vector multiplication, writing into the shared destination vector without data hazards. The granularity of parallelization is a cache line. We load balance SpMV by attempting to balance the total number of nonzeros in each thread block. In a CSR implementation, each thread block has its own *Value*, *ColumnIndex*, and *RowPointer* arrays. Thus, the thread blocked sparse matrix is implemented as a structure of sparse matrix structures. A `malloc()` call is performed for each array of each sub-matrix.

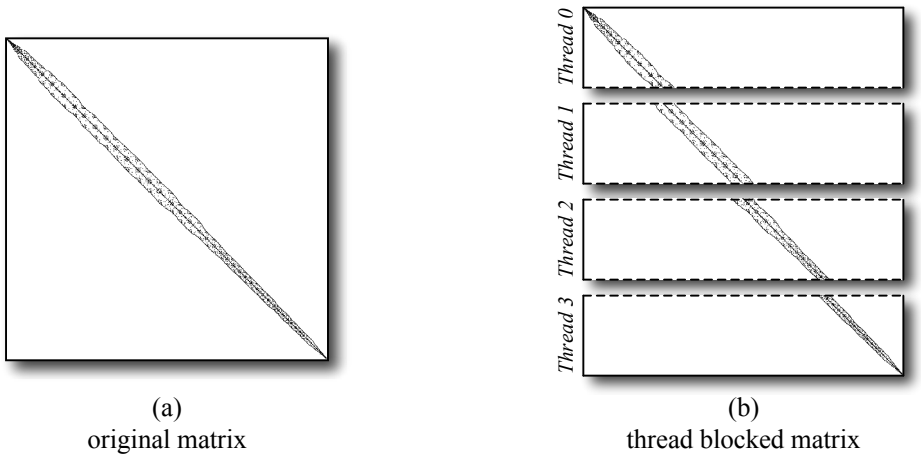


Figure 8.6: Matrix Parallelization. For load balancing, all sub-matrices have about the same number of nonzeros, but are stored separately to exploit NUMA architectures.

Most architectures with shared caches have more associativity than threads. As such, inter-thread conflicts are unlikely. However, architectures such Victoria Falls have far more threads sharing the L1 than L1 associativity — 8 and 4 respectively. Moreover, L2 conflicts can be more hazardous due to the increased probability — 64 threads into a 16-way cache — as well as the increased miss penalty required to fetch from DRAM. Furthermore, the L2 only has 8 banks shared among 8 cores. As such, it is important to ensure that not only are there no L1 or L2 conflicts, but there are no bank conflicts either. To that end, we align array `malloc()` to a 256 KB boundary — the way size. Next, we pad it to ensure each group of eight threads within a core so that it is uniformly spread throughout the L2. We then pad it to ensure that threads within a core are uniformly spread within the L1. Finally, we pad it to ensure that threads are uniformly spread across L2 banks. Figure 8.7 on the next page shows this three level padding, which ultimately improved peak performance by 20% after subsequent optimizations were included. This approach is identical to the lattice-aware padding described in Section 6.3.3 with the provision that instead of padding to spread the points of a stencil, we pad to spread the threads' arrays.

Figure 8.8 on page 165 shows SpMV performance at full concurrency. Note, the horizontal axis represents the 14 matrices in our suite plus a median performance number. The order of matrices in Figure 8.5 on page 161 has been preserved. Generally speaking, the x86 multicores see little benefit from 4- to 8-way parallelism, but the multithreaded Victoria Falls and Cell PPE see dramatic improvements. In fact, we see a reversal of fortune on Victoria Falls. It is now 50% faster than the closest x86 multicore SMP. Table 8.2 notes the highest sustained floating-point and bandwidth performance for the dense matrix in sparse format, as well as percentage of machine peak, for each architecture. Note, bandwidth is calculated based on the FLOP:compulsory byte ratio assuming all compulsory cache misses arise from matrix accesses.

When examining Clovertown performance, we see 8-way multicore parallelism only doubled performance. Nevertheless, performance is remarkably constant — around

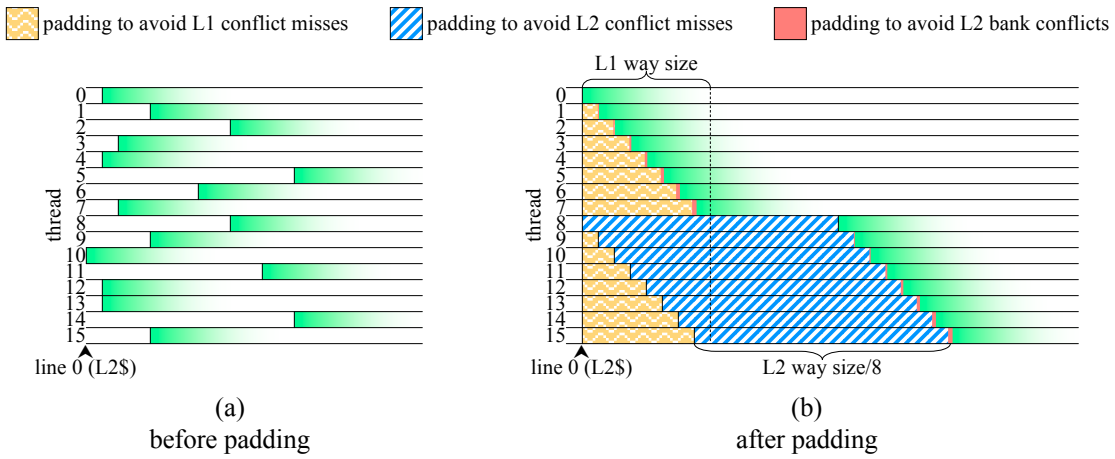


Figure 8.7: Array Padding (a) Arrays are individually allocated resulting in numerous bank and cache conflicts. (b) Each array for each thread block is padded such that for a given array all threads' elements map to a different L1 and L2 set, and the number of threads per bank is balanced.

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	(SPE)
GFLOP/s (% peak)	1.02 (1.4%)	0.84 (4.8%)	1.50 (2.0%)	2.50 (13.4%)	0.34 (2.7%)	—
GB/s (% peak)	6.12 (28.7%)	5.04 (47.3%)	9.00 (84.4%)	15.00 (70.3%)	2.04 (8.0%)	—

Table 8.2: Initial SpMV peak floating-point and memory bandwidth performance for the dense matrix stored in sparse format. Note, on NUMA architectures, percentage of peak bandwidth is defined as percentage of single socket bandwidth. The Cell SPE version cannot be run without further essential optimizations.

1 GFLOP/s or 6 GB/s. Clearly, the snoop filter can eliminate snoop traffic on the dual independent bus for the smallest matrices — Harbor, QCD, and Economics. As a result, they see bandwidths up to 10.5 GB/s. Nevertheless, this is far below either the aggregate FSB or total DRAM bandwidth. As expected, capacity misses on the source vectors of the largest two matrices impairs performance on Clovertown.

Both Santa Rosa and Barcelona Opteron performance is quite similar. Santa Rosa typically delivers better than 0.8 GFLOP/s or about 5 GB/s, where Barcelona typically delivers nearly 1.5 GFLOP/s or nearly 9 GB/s. Although the sustained bandwidth on Santa Rosa is far below a single socket's 10.66 GB/s, sustained Barcelona bandwidth is approaching the limits of a single socket. Thus, when bandwidth-limited, multicore scaling is expected to be poor. In fact, 4-way parallelism only doubled Santa Rosa performance, where 8-way parallelism only improved Barcelona performance by a factor of 2.5×. Unlike Clovertown with its giant caches, capacity misses strike Barcelona and Santa Rosa on the six most challenging matrices.

At this point — simple parallelization — Victoria Falls is clearly the best per-

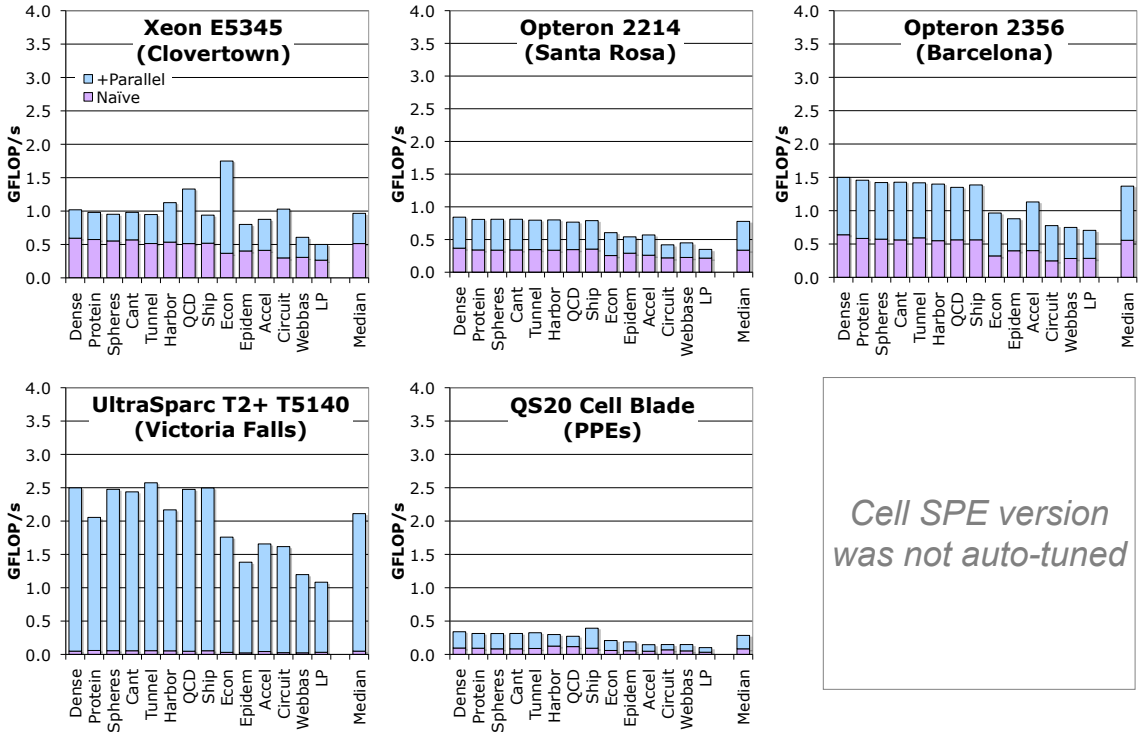


Figure 8.8: Naïve serial and parallel SpMV performance.

forming architecture delivering 2.5 GFLOP/s or 15 GB/s. This represents about 70% of a socket's bandwidth. Performance is higher than any other architecture because the machine has double the bandwidth per socket and nearly comparable efficiency. Like the Opterons, the limited cache cannot contain the largest vectors. Additionally, the small number of rows in the extreme aspect ratio Linear Programming matrix does not lend itself to massive parallelization by rows.

As the Cell SPE version cannot be run without significant further requisite optimizations, we initially only examine the Cell PPE performance. It is clear that two-way in-order multithreading on each core is wholly insufficient in satisfying the approximately 5 KB of concurrency required per socket by Little's Law (200 ns $\times$ 25 GB/s). As a result, the Cell PPE version delivers pathetic performance even when compared to the other architectures — delivering less than 0.4 GFLOP/s. Nevertheless, it is clear that together multithreading and multicore delivered at 3.4$\times$ speedup over naïve serial.

8.4.3 Exploiting NUMA

At this point, without further optimization, it is reasonable to conclude that Clovertown is FSB-bound, the Opterons and Victoria Falls are likely single-socket memory bandwidth-bound, and the Cell PPEs are latency-bound — they do not satisfy Little's Law. As such, fully engaging the memory controllers on the second socket is essential in increasing memory bandwidth on Santa Rosa, Barcelona, and Victoria Falls. (Doubling

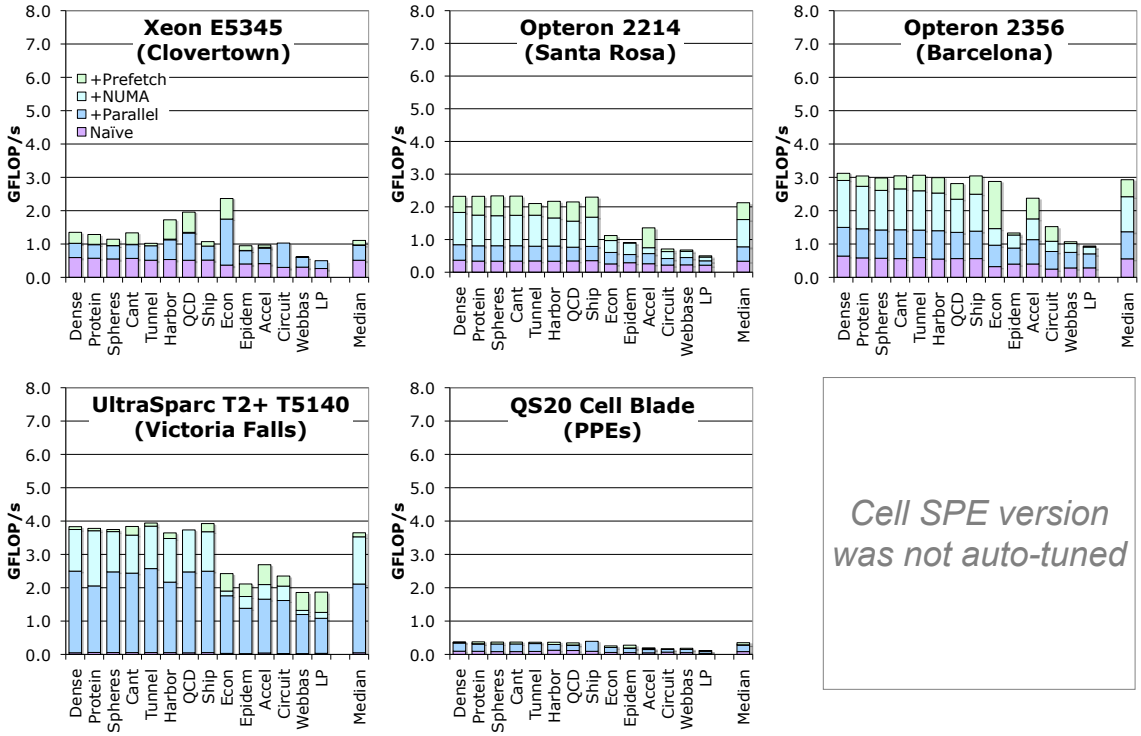


Figure 8.9: SpMV performance after exploitation of NUMA and auto-tuned software prefetching.

the available bandwidth will not help the Cell PPEs) Thus, we must update our matrix initialization routines to exploit NUMA. As each thread block described in Section 8.4.2 is individually `malloc()`'d, it is possible to modify the `malloc()` routines to allocate the thread block on the same socket so that the thread tasked to process it will be assigned to it. On Victoria Falls, we implement our own simplified, NUMA-aware, heap management routines. We maintain one heap per socket. The first vector is placed on the first socket and the second vector is placed on the second socket. Remember, we alternate between $y = Ax$ and $x = Ay$. Figure 8.9 shows performance after the auto-tuner exploits NUMA optimizations. We discuss the results in conjunction with those of the following section.

8.4.4 Software Prefetching

Previous work [81, 140] has shown that software prefetching can significantly improve streaming bandwidth. We reproduce that optimization here. We perform an exhaustive search for the optimal prefetch distance for both the value and column index arrays. Note no prefetching is not the same as prefetch by zero, because a completely different code variant is generated. With two prefetched arrays, four variants are generated. We search the prefetch distances from 0 to 1024 in increments of the cache line size.

As implemented, one prefetch is inserted per inner loop iteration — that is, one per nonzero. Although simple, this is inefficient as only one prefetch is needed per eight or

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	(SPE)
GFLOP/s (% peak)	1.35 (1.8%)	2.32 (13.2%)	3.12 (4.2%)	3.83 (20.5%)	0.37 (2.9%)	—
GB/s (% peak)	8.10 (38.0%)	13.92 (65.3%)	18.72 (87.8%)	22.98 (53.9%)	2.22 (4.3%)	—
Speedup from Optimization	+32%	+176%	+108%	+53%	+9%	—

Table 8.3: SpMV peak floating-point and memory bandwidth performance for the dense matrix stored in sparse format after auto-tuning for NUMA and software prefetching. The Cell SPE version cannot be run without further essential optimizations. Speedup from optimization is the incremental benefit from NUMA and software prefetching.

sixteen nonzeros. Thus, we don't expect optimal bandwidth on architectures that don't coalesce redundant prefetches. Section 8.4.5 will show how this can be significantly improved.

Figure 8.9 on the previous page shows performance after the inclusion of NUMA and software prefetching in the auto-tuning framework. There is a significant benefit in correctly exploiting NUMA on the memory-bound NUMA architectures. Additionally, there is a moderate benefit from software prefetching on all architectures. For reference, Table 8.3 shows the floating-point and bandwidth performance for the dense matrix in sparse format.

The two Clovertown sockets interface with DRAM through a common external memory controller hub. Just as in Chapter 6 for LBMHD, NUMA will have no benefit on a uniform memory access architecture. Despite instantiating five hardware prefetchers on each Clovertown chip, software prefetching is surprisingly beneficial — delivering approximately roughly a 25% boost for many matrices. Once again, the snoop filter delivers better bandwidth for the smaller matrices — Harbor, QCD, Economics. In fact, the Economics matrix achieves better than 14 GB/s. However, the larger dense matrix only attains about 8 GB/s.

The NUMA optimization is immensely beneficial on both the Santa Rosa and the Barcelona Opterons: it doubles performance. On top of this, software prefetching further improves the performance of the well structured matrices. Santa Rosa and Barcelona are about 1.7× and 2.3× faster than Clovertown on the dense matrix. This performance implies Santa Rosa and Barcelona achieve 13.9 and 18.7 GB/s, respectively. These bandwidths are approximately 65% and 88% of peak bandwidth. Clearly, AMD dramatically improved the memory subsystem on Barcelona through the addition of a DRAM prefetcher. There remains a performance bifurcation on the Opteron. Matrices for which the vectors do not fit in cache continue to run slowly — NUMA and software prefetching don't address this deficiency.

The Victoria Falls results are very counterintuitive. First, the NUMA optimization only improved performance by about 50%. This suggests that the relatively low frequency, dual-issue cores are likely becoming instruction issue-limited rather than bandwidth-limited. Second, unlike the other architectures, software prefetching helped on the challenging matrices rather than on the well structured ones. It is likely that large numbers of capacity misses to the source vector will generate many long latency misses that normally stall the in-order cores. The inclusion of software prefetching injects more concurrency into the memory subsystem than multithreading alone is capable of. Nevertheless, SpMV performance is

typically about 4 GFLOP/s, or nearly 24 GB/s. However, despite the performance boost, Victoria Falls is now only 20% faster than the Barcelona Opteron in the median case.

The NUMA and software prefetching optimizations are designed to improve memory bandwidth. As the Cell PPE doesn't satisfy Little's Law, it will see little benefit. In fact, although it sees a 23% boost in the median case, these optimizations only improve performance on the dense case by 8%. The inefficient implementation of software prefetching within a 1×1 CSR is obvious on this architecture as there is no other latency hiding paradigm to fall back upon. Victoria Falls is about 10× faster than the Cell PPEs.

8.4.5 Matrix Compression

Tables 8.2 and 8.3 clearly showed SpMV performance is nearly memory or FSB-bound after parallelization, exploitation of NUMA, and auto-tuning the software prefetch distance. Moreover, performance is primarily limited by compulsory cache misses. The only hope for improved performance is the elimination of compulsory misses associated with the loading of the matrix. There are two potential solutions: changing the algorithm as exemplified by A^k methods [40], or compressing the matrix. We explore the latter. SpMV is dominated by reads, not writes. Thus, cache bypass is not applicable as it eliminates the write allocate behavior.

Strategies

Our first strategy attempts to eliminate sparsely spaced row indices or redundant row pointers. Associated with each nonzero, there is some meta data used to calculate the appropriate row and column index. In coordinate (COO), coupled with each nonzero is a row and column index. One may sort the nonzeros by rows then by columns. As such, many adjacent nonzeros have the same row index. CSR eliminates these redundant indices in favor of a row pointer for each row. As the sparsity pattern for a thread (or eventually cache) block may have many empty rows, COO may result in a smaller footprint on some matrices, while CSR may be smaller on others. Selection of the appropriate format may eliminate redundant meta data, and result in higher SpMV performance. In practice, we didn't observe substantial selection of COO over CSR with the row start/end (RSE) optimization [99] on cache-based architectures. Thus, the benefit from COO was miniscule.

Our second strategy attempts to keep the column indices as small as possible. The columns spanned by a thread block ($last\ column - first\ column$) may be significantly less than 2^{32}. As such, many of the high bits in every 32-bit column index will be zero — why load zeros over and over? Thus, we allow any block spanning less than 2^{16} columns to use 16-bit indices rather than the standard 32-bit indices used in CSR. In general, this approach could be extended to select the minimum number of requisite bytes or ultimately the minimum number of bits. For simplicity, we restrict compression of indices to either none or by 16 bits.

The third strategy we employ attempts to eliminate neighboring column and row indices all together. OSKI exploits register blocking as a technique designed to improve peak performance. It encodes matrices in a format known as Blocked Compressed Sparse Row (BCSR). Often, on single core machines of five years ago, the performacne gains come from

expression of instruction and data-level parallelism in the number of rows in a register block. Additionally, the loop overhead per nonzero is amortized. As a result, it was not uncommon for a serendipitous increase in total matrix size to result in an increase in performance.

Fast-forward five years, and architecture has changed dramatically. Multicore provides abundant and untapped parallelism but little additional memory bandwidth. As a result, bandwidth is the resource that must be conserved. We do not employ register blocking (BCSR) to increase the expression of parallelism. Rather, we exploit register blocking to eliminate redundant meta data. Thus, for every register block, we select a potentially unique register blocking that minimizes that thread block's memory footprint. For simplicity we only explore the 16 power of 2 register blockings between 1×1 and 8×8 inclusive. Asymptotically, register blocking can eliminate 33% of the total memory traffic, resulting in a 50% increase in the FLOP:Byte ratio. When prefetching, we insert at least one prefetch per register block, asymptotically one per cache line. Thus, small register blocks — less than a cache line — have more prefetches than necessary.

We combine the previous five techniques into a one pass data structure optimization routine. For each thread block, we scan through its columns eight rows at a time. We copy the nonzeros we detect into a single 8×8 register block. When we have covered eight adjacent columns, column scanning stops, and we analyze the resultant 8×8 register block to calculate how many R×C tiles it contains for all R and C. A running tally of the total number of R×C tiles is incremented. Thus, when the thread block has been completely scanned, the total number of requisite tiles associated with any power of 2 register blocking are known exactly. We then examine all valid combinations of format register blocking and index size, and select the combination that minimizes the total memory footprint of the thread block. Note, that each thread block is individually compressed. Thus, it is possible for a matrix to include hundreds of different blockings.

Results

Figure 8.10 on the following page shows performance after matrix compression is added to the auto-tuning framework. Interestingly, acorss architectures, some matrices see significant benefits, while others see none. Thus, we move away from the previous results where performance was dictated by memory bandwidth and cache capacity misses. Victoria Falls clearly delivers the highest peak and median performances. However, the Opteron Barcelona delivers comparable median performance. Although both 1.166 GHz Victoria Falls and the 2.3 GHz Barcelona are considered low frequency parts, Victoria Falls is far from the bandwidth limit. Consequently, moderate frequency improvements will translate into moderate increases in performance. As Barcelona is near the bandwidth limit, higher frequency will show little benefit. Table 8.4 shows the floating-point and bandwidth performance for the dense matrix in sparse format.

On Clovertown, matrix compression doubled the SpMV performance on the dense matrix in sparse format using better than half of the raw DRAM bandwidth. This was quite surprising, as register blocking nominally would improve performance by 50%. There are two reasons for this boost to performance. First, in 1×1 CSR software prefetching is implemented by inserting one software prefetch intrinsic per nonzero. Clearly, this generates eight to sixteen times more prefetches than are required depending on the line size. When

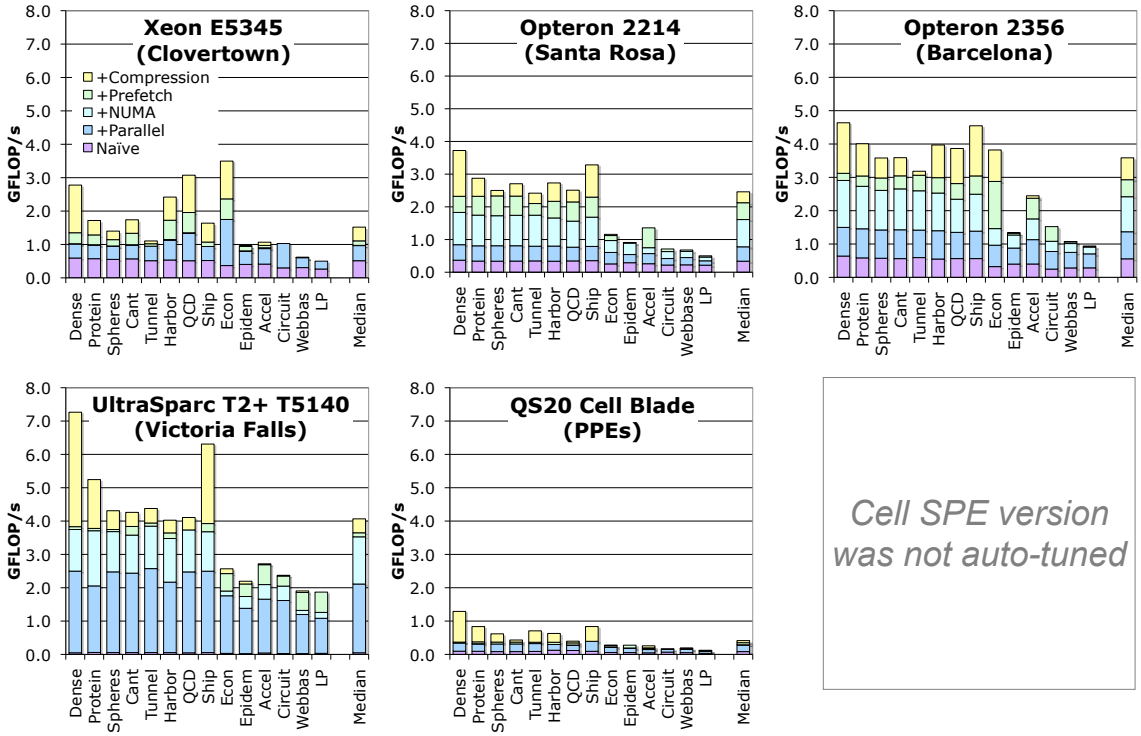


Figure 8.10: SpMV performance after matrix compression.

the number of elements in a register block is a perfect multiple of the cache line size, only one prefetch is inserted per cache line. This is clearly the optimal case. As such, bandwidth is improved. Finally, matrix compression reduces the memory footprint of a matrix. As a result, the snoop filter becomes more effective, and snoops are generated less frequently. Many other matrices amenable to register blocking saw significant boosts to performance.

The Santa Rosa and Barcelona Opterons continued to deliver very good memory bandwidths with Santa Rosa showing a 7% increase in bandwidth on the dense matrix. Of course, register blocking on the dense matrix reduces memory traffic by 33%. Thus performance improved by 1.6× and 1.5×, respectively. Barcelona delivered 1.7× better performance than Clovertown for both the dense and median cases. Surprisingly, on the most challenging matrix, Barcelona was 1.85× faster. This disparity arises from the fact that vectors are swapped between successive SpMVs. Thus, for poorly structured matrices, this swap acts much like an all-to-all broadcast — clearly a memory intensive operation. Although the quad-core Barcelona consistently outperformed the dual-core Santa Rosa, the lack of additional memory bandwidth severely limits scalability. In fact, Barcelona shows little scalability beyond two cores per socket on SpMV. Much of the benefit is derived from the vastly improved DRAM prefetcher.

The performance gains on Victoria Falls are enigmatic. Dense performance increased by 90% while median performance only improved by 11%. The 1.16 GHz Victoria Falls is a rather low frequency part with vast amounts of bandwidth. As a result, our as-

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	(SPE)
GFLOP/s (% peak)	2.78 (3.7%)	3.72 (21.1%)	4.64 (6.3%)	7.27 (38.9%)	1.29 (10.1%)	—
GB/s (% peak)	11.12 (52.1%)	14.88 (69.8%)	18.56 (87.0%)	29.08 (68.2%)	5.16 (10.1%)	—
Speedup from Optimization	+106%	+60%	+49%	+90%	+249%	—

Table 8.4: SpMV peak floating-point and memory bandwidth performance for the dense matrix stored in sparse format after the addition of the matrix compression optimization. The Cell SPE version cannot be run without further essential optimizations. Speedup from optimization is the incremental benefit from matrix compression.

sumption that with enough threads any code should be memory-bound is certainly not true on this architecture. Hence, the matrix compression heuristic — minimize matrix footprint — may not yield optimal or even superior results.

The dense matrix likely sees dramatic performance gains for several reasons. First, effective prefetching injects more parallelism into the memory subsystem than threading alone. This results in a 20% performance boost over register blocking without prefetching. Second, there is a low fraction of floating-point instructions in the 1×1 CSR SpMV implementation. On architectures that are instruction bandwidth-limited, increasing the faction of floating-point instructions in the instruction mix through elimination of non-floating-point instructions will improve performance. This improvement is clearly shown in the Victoria Falls Roofline model in Figure 8.4 on page 159. Register blocking will asymptotically improve the floating-point fraction from around 12% to better than 50%. Additional experiments have shown that the 1.4 GHz Victoria Falls delivers proportionally better performance, lending credence to our hypothesis that SpMV performance is limited by in-core performance. A bandwidth-only heuristic is inappropriate for architectures with extremely low FLOP:Byte ratios.

Aside from minimalistic multithreading, software prefetching is the only latency hiding technique possessed by the Cell PPEs. As such, any implementation that efficiently exploits it will significantly improve memory bandwidth. Register blocking in conjunction with efficient software prefetching improved dense performance by 3.5× but median performance by only 1.2×. As a result, sustained memory bandwidth on the dense matrix improved by 2.3×. Clearly, register blocking not only reduces memory traffic, but also increases memory bandwidth. Like Victoria Falls, the heuristic used on the Cell PPE should be augmented. However, for the PPE, the variation in memory bandwidth induced by effective software prefetching necessitates a heuristic which incorporates traffic, bandwidth, and execution time profiles for each and every register blocking.

8.4.6 Cache, Local Store, and TLB blocking

With regard to the matrix, we have discussed strategies designed to minimize conflict and compulsory misses: padding and compression. Capacity misses are not an issue as there is no temporal reuse of the nonzeros. However, nearly half of the matrices will show large numbers of capacity misses associated with the source vectors. These can be

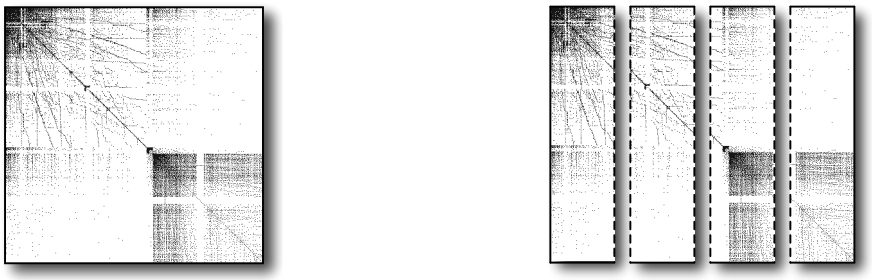


Figure 8.11: Conventional Cache Blocking: (a) The original matrix stored in CSR, (b) The matrix is blocked so that each cache block spans the same fixed number of columns. Each cache block is individually stored in CSR. In practice, each cache block should span between 10K and 100K columns.

divided into two categories: those that generate capacity misses within a SpMV, and those that cannot hold the vectors in cache between SpMV's. The latter is the class of matrices whose bandwidths are smaller than the cache sizes, but whose vectors do not fit in cache — consider a billion row tri-diagonal matrix. In this section, we focus on the former for cache-based machines, and both types for the Cell processor.

Figure 8.11(b) on page 172 illustrates the naïve attempts [99, 141] to simply partition the matrix into blocked columns whose corresponding source vector elements can fit in cache. Hence, there are $N/CacheBlockSize$ blocked columns in the resultant matrix. Each cache blocked column would then be stored in CSR with its own value, column, and row pointer arrays. This optimization was nothing more than applying the standard cache blocking techniques found in structured grids and dense linear algebra to SpMV. This was extended to the parallel case for the Cell processor by assigning rows to each SPE [141]. The local store on Cell restricted cache blocks to be less than 16K elements wide, as the equivalent of a capacity miss must be handled in software. Thus, the largest matrices could not be run as they would require as many as 100 blocked columns stored in CSR each with a 1M element row pointer array. This would exceed the main memory capacity of a Cell blade. As a result, the matrices previously capable of running the Cell blade were very small. One final point is that this approach does not partition the data structure for NUMA or multicore. Combining these inefficiencies make this approach very unattractive.

Strategy

In this work, we extend the standard cache blocking techniques from dense linear algebra to the sparse motif. In a dense matrix-vector multiplication, every source vector element will be used repeatedly. In the sparse case, this may not be true. Due to sparsity, some elements will never be used in the current cache block. Thus no cache or local store capacity should be reserved to store them. The goal is to only load the cache lines containing one or more source vector elements that will be used in the current cache block. Although

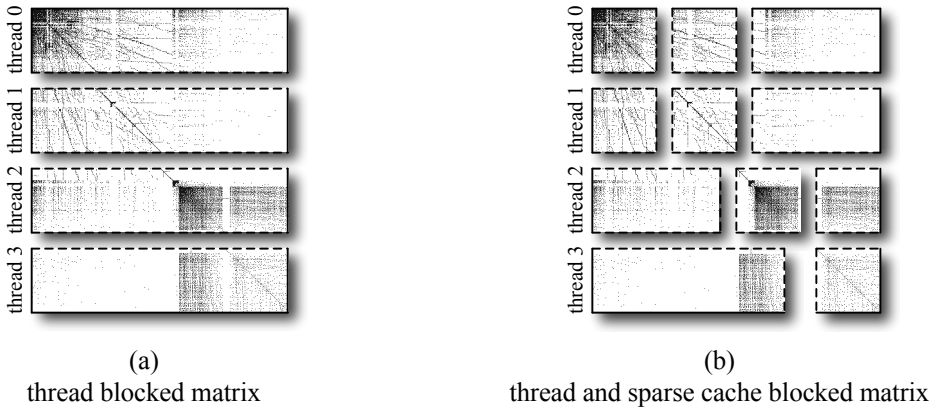


Figure 8.12: Thread and Sparse Cache Blocking: (a) a thread blocked matrix. Each thread block is individually stored in CSR, (b) A thread and sparse cache blocked matrix. Each cache block is individually stored in CSR. In addition, although they may span vastly different numbers of columns, each cache block should touch the same number of source vector cache lines or source vector TLB pages.

cache blocks may span vastly different numbers of columns, they should all touch the same number of cache lines. Figure 8.12(b) on page 173 illustrates the sparse cache blocking technique when applied on a previously thread blocked matrix. Clearly, the number of columns spanned can vary greatly. However, the number of requisite source vector cache lines per cache block is roughly constant.

Traditional cache blocking is sparsity agnostic. To effectively block for a sparse matrix, the matrix must be examined. In our approach, each architecture has a specified cache capacity broken into a number of cache lines and the number of doubles per line. We allocate 40% of that to caching source vectors, reserve another 20% for caching row pointers, and the remaining 40% for caching destination vector elements. Thus, we have specified the number cache blocked rows to partition each thread block into: 40% of the cache capacity. During the data structure optimization phase, we scan through the resultant cache blocked row and mark which source vector cache lines are referenced. We then convert this bitmask into a cumulative distibution. By columns, we scan through the cache blocked row a second time. When the fraction of the cumulative distribution reaches the cache capacity we stop scanning and create a new cache block. Column indices are DRAM relative, but are stored relative to the first column within the cache block. This allows for more effective index compression (discussed in Section 8.4.5) even on matrices that don't suffer from vector capacity misses. In addition, for CSR, we store the indices of the first and last non-empty row to avoid data transfers and computation on empty rows; the row start/end (RSE) optimization [99].

This technique can be trivially extended to blocking for Cell's local store. The difference is that instead of encoding a DRAM relative column indices relative to the first column and relying on the cache to handle misses, a DMA gather list must be created and

column indices are stored relative to their local store (gathered) addresses. This approach is not difficult. As we scan through the cumulative distribution we simultaneously create a DMA list element for each contiguous stanza of referenced cache lines. During execution of each cache block, its DMA list must be loaded. Note, each DMA list item contains the base address relative to the source vector and the number of bytes in the transfer. However, the DMA list command treats the addresses as absolute DRAM addresses rather than relative to some base. Thus, after the list itself is read into the local store, every address must be incremented by the address of the first element of the vector.

The resultant Cell code is orchestrated with clockwork precision:

1. The cache block after next header is loaded via a double buffered DMA. The headers contain all the relevant parameters, and pointers for the given cache block.

2. Concurrently, the next cache block's list of DMAs is loaded while simultaneously executing the DMA gather for the current cache block packing the result into the local store.

3. At the same time, the working copy of the destination vector can be zeroed out. Once that is complete, the flow control takes over and streams through blocks of nonzeros tiles.

4. Each tile is decoded and processed accessing the appropriate element of the source vector copy. The row's running sum is always written to the local copy of the destination vector. When done with all buffers of nonzeros, the copy of the destination vector is copied back to DRAM via a DMA.

5. Neither copy of the source and destination vectors is double buffered to maximize available local store capacity.

Observe that the same sparse cache blocking technique can be applied to TLB blocking. The only difference is in the granularity: 32 entries of 512 doubles instead of 8K entries of 8 doubles. Thus, we may reuse the same code but block for the TLB rather than the cache. The code still scans through the matrix marking touched blocks — pages in this case. Most architectures use 4K pages (512 doubles), but Solaris uses 4 MB pages. Hence, we don't explore TLB blocking on Victoria Falls.

Results

Figure 8.14 on page 176 shows performance after cache and TLB blocking are added to the auto-tuning framework. Note that with this last set of optimizations, we can start including results for the Cell SPEs. Although the blocking parameters are based on heuristics, we search on three global strategies: no blocking, only cache blocking, cache and TLB blocking. In the figure, the latter two strategies are combined into a single optimization. Blocking should only be expected to be beneficial on matrices with very large bandwidths. Remember, matrix bandwidth is the bandwidth for which $A_{ij} = 0$ when $|i - j| > BW$. Only two matrices fall into this class: Webbase and Linear Programming. Webbase suffers from very short rows — averaging less than three nonzeros per row, but

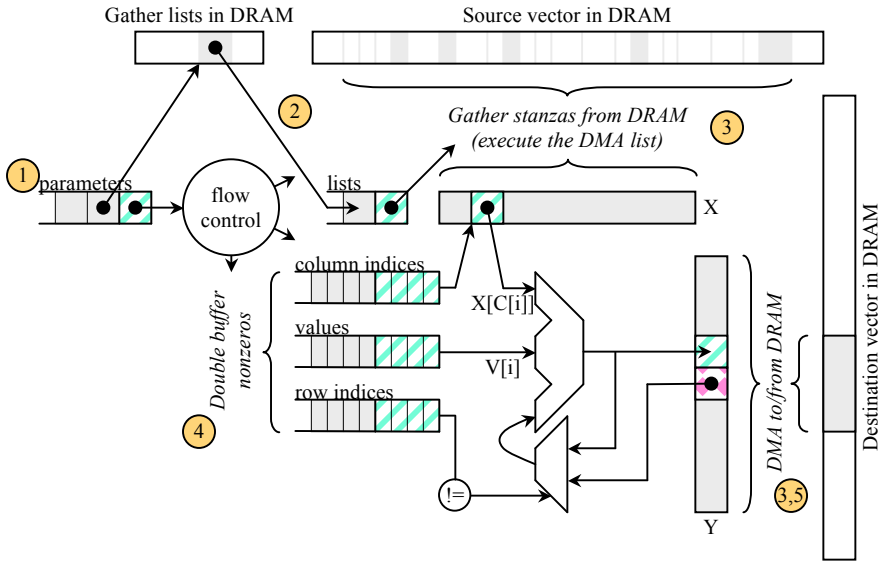


Figure 8.13: Orchestration of DMAs and double buffering on the Cell SpMV implementation.

the linear programming matrix has very long rows and very high bandwidths. Across most architectures, the linear programming matrix is consistently the only matrix to show speedups. Moreover, it was typically the conjunction of cache and TLB blocking that showed the larger speedup. Table 8.5 shows the floating-point and bandwidth performance for the dense matrix in sparse format. Note that only the Cell SPE data is different in this table as cache blocking is not beneficial on a small matrix but is essential for a local store implementation.

Despite Clovertown's 16 MB of L2 cache, cache and TLB blocking improved performance on the linear programming matrix by 1.8×. The matrix bandwidth is sufficiently large that it cannot fit in any chip's 4 MB of L2 cache. Barcelona also saw a 1.8× speedup from cache and TLB blocking despite each chip also having only 4 MB. In the end, Barcelona's significantly greater bandwidth resulted in it being 1.85× faster than Clovertown. Interestingly, Victoria Falls, also with 4 MB of cache per chip, only saw a 1.1× increase in performance. Perhaps its untapped memory bandwidth softened the capacity cache miss penalty. Alternatively, as TLB blocking was often more valuable on the x86 machines, Solaris' use of 4 MB pages eliminated the benefit on Victoria Falls. The benefit of cache and TLB blocking was even greater on the machines with small caches or TLBs. The Santa Rosa Operton and the Cell PPE saw a 2.5× and 2.6× improvement, respectively.

Due to the weak double-precision implementation and lack of scalar instructions like those available in x86 SSE, the minimum register blocking implemented on the Cell SPEs was 2×1. This is trivially SIMDized, making the kernels easy to implement. However, the downside is that 1×1 register blocking is the optimal blocking on many matrices. As a result, Cell will often require more memory traffic for the matrix than the other architectures. Furthermore, to further expedite implementation on Cell, only a sorted blocked

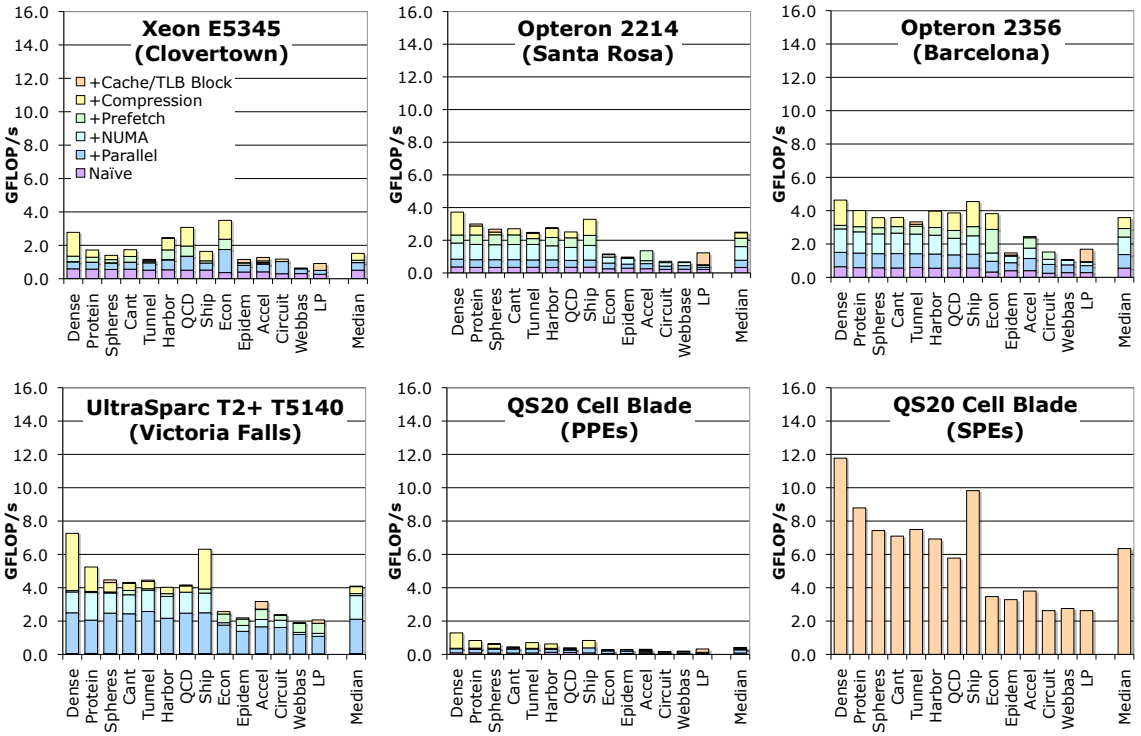


Figure 8.14: SpMV performance after cache, TLB, and local store blocking were implemented. Note local store blocking is required for correctness when using the Cell SPEs.

coordinate format (BCOO) was used. Blocked compress sparse row is significantly more difficult to implement on a double buffered DMA architecture and would provide relatively little benefit. As mentioned, the local store is sufficiently small to allow 16-bit indices to always be used in conjunction with cache blocking for DMA. As a result, every nonzero tile is at least a 2×1 block with a 16-bit column coordinate, and a 16-bit row coordinate. Thus in the worst case, arithmetic intensity will be less than 0.10. Nevertheless, in the best case, the best arithmetic intensity will be nearly the same as any other machine: 0.25. Hence, one would expect Cell to win on the easy matrices as it has more bandwidth and the same arithmetic intensity. However, it will be significantly challenged on the complex matrices as all spatial and temporal locality will have to be found and encoded into the matrix, and the combination of register blocking and format will hurt the arithmetic intensity.

The Cell SPEs and PPE share the same memory controllers and thus have access to the same memory raw bandwidth. Despite collectively the SPEs having the same bandwidth and little more than double the peak double-precision FLOPs, the SPE version of the code delivers more than a 15× speedup in the median case. When examining memory bandwidth on the dense case it becomes blatantly obvious that DMA is extremely effective in utilizing the available memory bandwidth — achieving about 92% of the raw bandwidth. On the same matrix, Cell nearly doubles Victoria Falls' performance, tripples Opteron performance, and quadruples Clovertown performance.

Machine	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	QS20 Cell Blade (PPE)	QS20 Cell Blade (SPE)
GFLOP/s (% peak)	2.78 (3.7%)	3.72 (21.1%)	4.64 (6.3%)	7.27 (38.9%)	1.29 (10.1%)	11.78 (40.2%)
GB/s (% peak)	11.12 (52.1%)	14.88 (69.8%)	18.56 (87.0%)	29.08 (68.2%)	5.16 (10.1%)	47.1 (92.0%)

Table 8.5: SpMV floating-point and memory bandwidth performance for the dense matrix stored in sparse format after the addition of cache, local store, and TLB blocking.

Although peak Cell performance is very good, qualitatively the performance distribution is lacking the consistent behavior seen on the Opteron. This arises because Cell's use of 2×1 BCOO for productivity can severely hamper performance when 1×1 BCSR is optimal. We believe that 1×1 BCSR can be implemented on the new eDP (enhanced double-precision) QS22 blades without fear of making the code computationally-bound. Nevertheless, sparse cache blocking to encode spatial and temporal locality when the matrix is created was effective despite the tiny 256 KB local store capacity.

8.5 Summary

Table 8.6 on the next page details the optimizations used by each architecture and grouped by the Roofline-oriented optimization goal: maximizing memory bandwidth, minimizing total memory traffic, and maximizing in-core performance. In addition, for each optimization, we list how the auto-tuner found the relevant parameters for each matrix for each architecture. Note that the Cell SPE implementation used every optimization as the cache implementation except it never selects CSR as the format, and always chooses a register blocking greater than 2×1.

Bandwidth Optimization	Auto-tuning approach	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
NUMA Allocation	model	N/A	✓	✓	✓	✓	✓
Prefetch/DMA (matrix)	search[5]	✓	✓	✓	✓	✓	✓
Prefetch/DMA (vectors)	heuristic	—	—	—	—	—	✓

Traffic Optimization	Auto-tuning approach	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
Array Padding	model	✓	✓	✓	✓	✓	✓
Register Blocking	heuristic	✓[1]	✓[1]	✓[1]	✓[1]	✓[1]	✓[2]
Format (BCSR)	heuristic	✓	✓	✓	✓	✓	—
Format (BCOO)	heuristic	✓	✓	✓	✓	✓	✓
Cache / TLB Blocking	heuristic[6]	✓	✓	✓	✓	✓	✓[3]

In-core Optimization	Auto-tuning approach	Xeon E5345 (Clovertown)	Opteron 2214 (Santa Rosa)	Opteron 2356 (Barcelona)	T2+ T5140 (Victoria Falls)	Cell Blade (PPE)	Cell Blade (SPE)
SIMDized	N/A	✓	✓	✓	N/A	N/A	✓
Branchless	N/A	✓[4]	✓[4]	✓[4]	—	—	✓
Software Pipelined	N/A	—	—	—	—	—	✓

Table 8.6: Auto-tuned SpMV optimizations employed by architecture and grouped by Roofline optimization category: maximizing memory bandwidth, minimizing total memory traffic, and maximizing in-core performance. [1]powers of two from 1×1 through 8×8, [2]powers of two from 2×1 through to 8×8, [3]sparse blocking for local store using DMA, [4]implementation resulted in no observed speedup, [5]Cell used only heuristics. [6]search was used to decide whether to block, but heuristics were used to determine the blocking parameters. Note optimization parameters may vary from one matrix to the next. In such cases, a ✓notes that parameters were chosen.

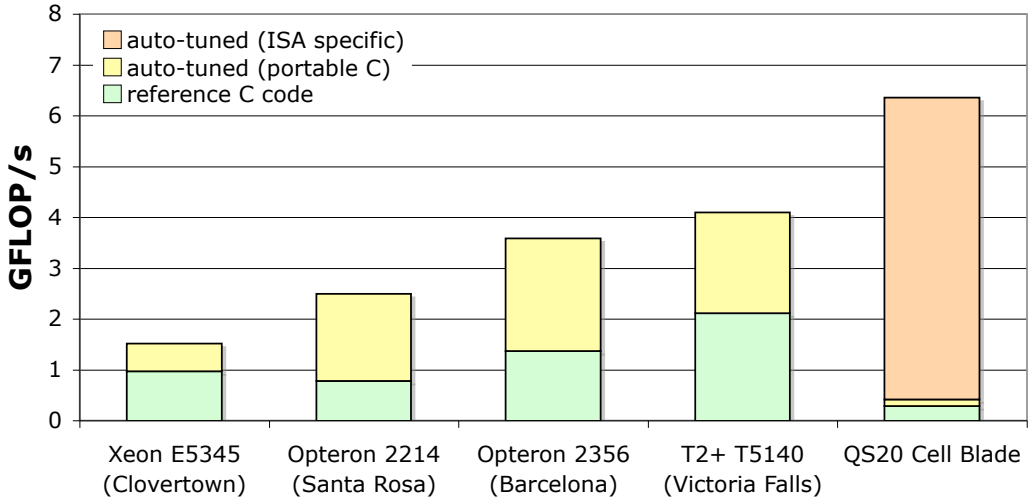


Figure 8.15: Median SpMV performance before and after tuning.

8.5.1 Initial Performance

Auto-tuners are not completely automatic. Although the search process has been automated, the optimization conceptualization process is still tied to a programmer. As such, out-of-the box performance remains an interesting metric used to compare different architectures. Figure 8.15 shows both naïvely parallelized and fully auto-tuned parallel median SpMV performance across all five SMPs. The three x86 architectures deliver comparable out-of-the-box performance. Although Victoria Falls delivers more than twice the performance of the Santa Rosa Opteron, the PPE's on the Cell blade deliver less than half the performance despite having more available DRAM read bandwidth than Victoria Falls. We see that there was no benefit for ISA-specific auto-tuning on any architecture except Cell.

Figure 8.16 on the next page shows SpMV performance on the dense matrix stored in sparse format before and after auto-tuning. Most architectures are near their respective streaming bandwidth limits. This should be no surprise as the diagonals in the chart refer to kernels with unit-stride memory accesses — typical of SpMV on a dense matrix in sparse format.

8.5.2 Speedup via Auto-Tuning

The benefits of auto-tuning SpMV varied wildly from one architecture to the next but more significantly from one matrix to another. Clovertown is no exception. Although some matrices saw speedups of 2.7×, the speedup in median performance was only 1.6×. This result in itself is surprising as matrix compression can only reduce memory traffic by 33%. Clearly software prefetching played a small benefit, and there is the possibility that some matrices were compressed sufficiently to fit either within the snoop filter or within the caches. Thus, on some matrices we see a jump from the lower bandwidth diagonal in

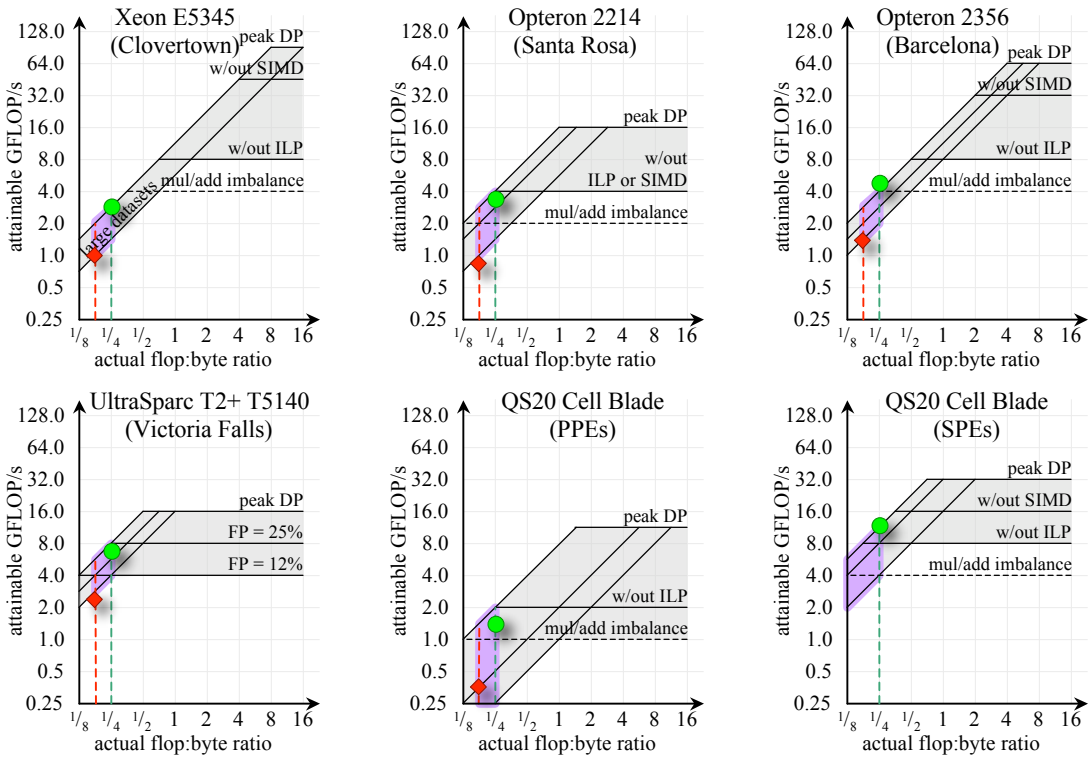


Figure 8.16: Actual SpMV performance for the dense matrix in sparse format imposed over a Roofline model of SpMV. Note the lowest bandwidth diagonal assumes a unit-stride access pattern. Red diamonds denote untuned performance, where green circles mark fully tuned performance. Note the log-log scale.

Figure 8.16 to the upper one. When examining scalability, it is clear that dual core and dual socket are of value. However, quad core provides no additional benefit — a testament to the bandwidth-limited nature of SpMV.

The Opterons are known to be capable of sustaining far greater bandwidth than Clovertown. Moreover, sustained Opteron stream bandwidth is tied to whether or not NUMA and software prefetching are effectively exploited. This observation is most readily true for the case of the dense matrix in sparse format. Here, auto-tuning increased Santa Rosa performance by 4.4× and Barcelona performance by 3.1× despite Barcelona's higher core count and improved hardware prefetching but identical raw bandwidth. The values of these optimizations in conjunction with register blocking are clearly visible in Figure 8.16. Although the dense matrix is an upper bound to performance and speedup, median performance still improved by 3.2× and 2.6× respectively. When examining scalability, we see that once again, quad core is of little value on SpMV for both the dense and the median case.

On Victoria Falls, auto-tuning dramatically improved performance on some matrices, but showed only modest benefits on others. Aside from parallelization, the only optimization that showed any significant benefit to median performance was NUMA aware

matrix allocation, which provided 1.67× of the 1.9× total speedup. The Roofline in Figure 8.16 on the previous page clearly shows that the expected range of SpMV performance straddles a dramatic and critical range in the fraction of the dynamic instruction mix that is floating-point. Naïve SpMV may have a floating-point dynamic instruction fraction as little as 10%. Register blocking has a triple advantage: increasing ILP, increasing arithmetic intensity, and effectively exploiting software prefetching. Thus, the auto-tuned SpMV on a dense matrix exploits all three, insuring the in-core ceilings are not a limiting factor in performance. Thus, we believe median SpMV performance was limited by the fact that simply minimizing memory traffic is insufficient on architectures with a low FLOP:Byte balance. Multicore scalability across Victoria Falls' 16 cores was 13× in the median case, but dropped to 9.8× in the dense case. As discussed, the dense matrix can nearly saturate a socket's memory bandwidth. Thus, one only expects scaling until bandwidth saturation.

Cell required two implementations. The first, a portable auto-tuned implementation, only ran on the PPEs. Auto-tuning provided them nearly a 1.5× increase in performance, but the raw performance remained much lower than the other cache-based architectures. When the SPE implementation was included, we saw a phenomenal 15× further increase in performance over the auto-tuned 4-thread PPE implementation. Collectively, auto-tuning the SPEs provided a 22× increase in performance over the parallelized standard CSR implementation. Figure 8.16 on the preceding page demonstrates that although the Cell PPE came close to the bandwidth Roofline for the dense matrix in sparse format, the SPE implementation was completely limited by DRAM bandwidth.

8.5.3 Performance Comparison

When comparing auto-tuned performance, we see that the Santa Rosa Opteron is 1.3× and 1.6× faster than Intel's Clovertown for the dense and median cases, respectively. When moving to the quad-core Barcelona, we see it provides 1.25× and 1.4× the performance of the Santa Rosa Opteron, and achieves better than 87% of its memory bandwidth. This makes AMD's quad-core nearly 1.7× and 2.4× as fast as Intel's current quad-core offering. In many ways, SpMV is an ideal match for Victoria Falls. Nevertheless, we see Victoria Falls only achieves about 1.14× Barcelona's performance despite having a double the raw bandwidth, a testament to the low per-core performance. In the end, Cell's lack of a cache was not critical as DMAs could be used to orchestrate the irregular source vector accesses. However, Cell's weak double-precision implementation forced a suboptimal, SIMD-friendly implementation that can often significantly increase the compulsory traffic. Nevertheless, Cell delivered nearly 1.8× better performance than Barcelona. We believe a future faster double-precision implementaion (eDP Cell) will allow for a more efficient and general implementation that will improve median SpMV performance. Overall, with a handicapped, but auto-tuned implementation, Cell delivers 1.6× better median performance than Victoria Falls, 1.8× the Barcelona Opteron, 2.5× the Santa Rosa Opteron, and almost 4.2× the median performance of the Intel Clovertown. Cell's SpMV performance for a dense matrix was 1.6× better than Victoria Falls, 2.5× the Barcelona Opteron, 3.2× the Santa Rosa Opteron, and 4.2× the median performance of the Intel Clovertown.

8.6 Future Work

Although an extensive number optimizations have been implemented here, significant SpMV-specific work remains to be explored. This research is divided into three categories: improving access to the vectors, minimizing the memory traffic associated with the matrix, and better heuristics. We save the discussion of broadening this effort to other kernels within the sparse motif for Chapter 9.

8.6.1 Minimizing Traffic and Hiding Latency (Vectors)

For matrices limited by access to the source and destination vectors, there are two basic principals we focus on: hiding memory latency and minimizing total memory traffic associated with the source vectors.

Unlike accessing the matrix, accessing the source vectors can result in non-unit stride access patterns. As a result, hardware prefetchers can be confused, resulting in memory latency stalls. Although massive thread level parallelism or DMA are solutions adept at hiding memory latency associated with non-stride memory access patterns, there is additional potential value in using software prefetching. Currently, four parameterized prefetch code variants are produced for each blocking and format combination: prefetch neither the value or index arrays, prefetch just the value array, prefetch just the index arrays, or prefetch both arrays. They are parameterized by the prefetch distance which is automatically tuned for — a rather time consuming operation. This process could be extended by providing a complimentary set of variants in which the source vector elements are prefetched. In this case, they would be parameterized by how many nonzeros should be prefetched. *i.e.* prefetching `X[col[i+number]]` prefetches "number" elements ahead in an attempt to hide the latency of a miss. Although some initial experimentation showed modest speedups, an exhaustive search would increase auto-tuning time by more than an order of magnitude.

We could attempt to address both memory traffic and latency by reordering the matrix [34]. We performed a few initial experiments on the reordered webbase matrix and saw less than a 10% speedup. This was much in line with the results from Im *et al.* [78] perhaps due to the limited scope of our matrix suite. It might be more appropriate not to strive for a perfectly diagonal matrix, but rather a matrix with good spatial locality on cache line granularities. In addition, it is possible to reorder the matrix to facilitate register blocking [108].

Finally, some vectors are so large that they will not fit in cache. When evaluating $y \leftarrow Ax$ on such a matrix there is no point in even attempting to cache y. As such, it is possible to generate another set of variants that use the x86 cache bypass instruction `movntpd`. Thus, a cache line fill will not occur on the write miss to y. Heuristically, this optimization should be beneficial on matrices with very large numbers of rows and relatively few nonzeros per row.

	Explicit Storage of all Nonzero Values	**Elimination of Redundant Nonzero Values**
Explicit Storage of all Nonzero Coordinates	16·N + 16·NNZ	16·N + (8...16)·NNZ
Elimination of Redundant Nonzero Coordinates	16·N + (8...16)·NNZ	16·N + (0...16)·NNZ

Table 8.7: Memory traffic as a function of storage.

8.6.2 Minimizing Memory Traffic (Matrix)

As demonstrated, the existing auto-tuning approach can achieve a high fraction of memory bandwidth. Given SpMV's bounded arithmetic intensity, higher performance can only be achieved by minimizing the compulsory memory traffic associated with the matrix. As Table 8.7 suggests, one should consider storage of nonzero values and coordinates as orthogonal optimizations. On can attempt to implicitly encode values or coordinates or both. Elimination of one might double COO performance. However, elimination of both could asymptotically improve performance by $1 + NNZ/Row$. Of course, this is just a bound; the asymptotic realization of this is the structured grid motif. Ultimately, we desire the performance of a structured grid code with the flexibility of a sparse method.

Alternate BCSR Representations

In Section 8.4.5 we discussed register blocking and other optimizations designed to compress away as much redundant meta data as possible. When register blocking, our heuristic only examined power of two blockings. It is possible to heuristically or exhaustively extend this to all possible register blockings. Although this may quadruple the optimization routine running time, it may increase performance by more than 50% as 3×3 register blocking can be common, and has a substantially higher arithmetic intensity than 1×1. Furthermore, the Cell version should be expanded to implement sub-SIMD 1×C register blocks.

CSR implementations on local-store architectures are difficult to implement as the loop structure must be changed from rows and nonzeros to buffers and nonzeros within a buffer. As such, we did not allow the selection of the CSR format because of the perceived weak Cell double-precision implementation. To further facilitate CSR implementations on all architectures, branchless or segmented scan implementations are desirable. However, branchless and segmented implementations do not work correctly when there are empty rows. Thus, we believe that using the empty-row CSR implementation discussed in Chapter 7 will greatly facilitate a common implementation. Architectures with few cores and high branch overheads operating on matrix blocks with few nonzeros per row will see a significant benefit. However, the reduction in total matrix memory traffic will be small.

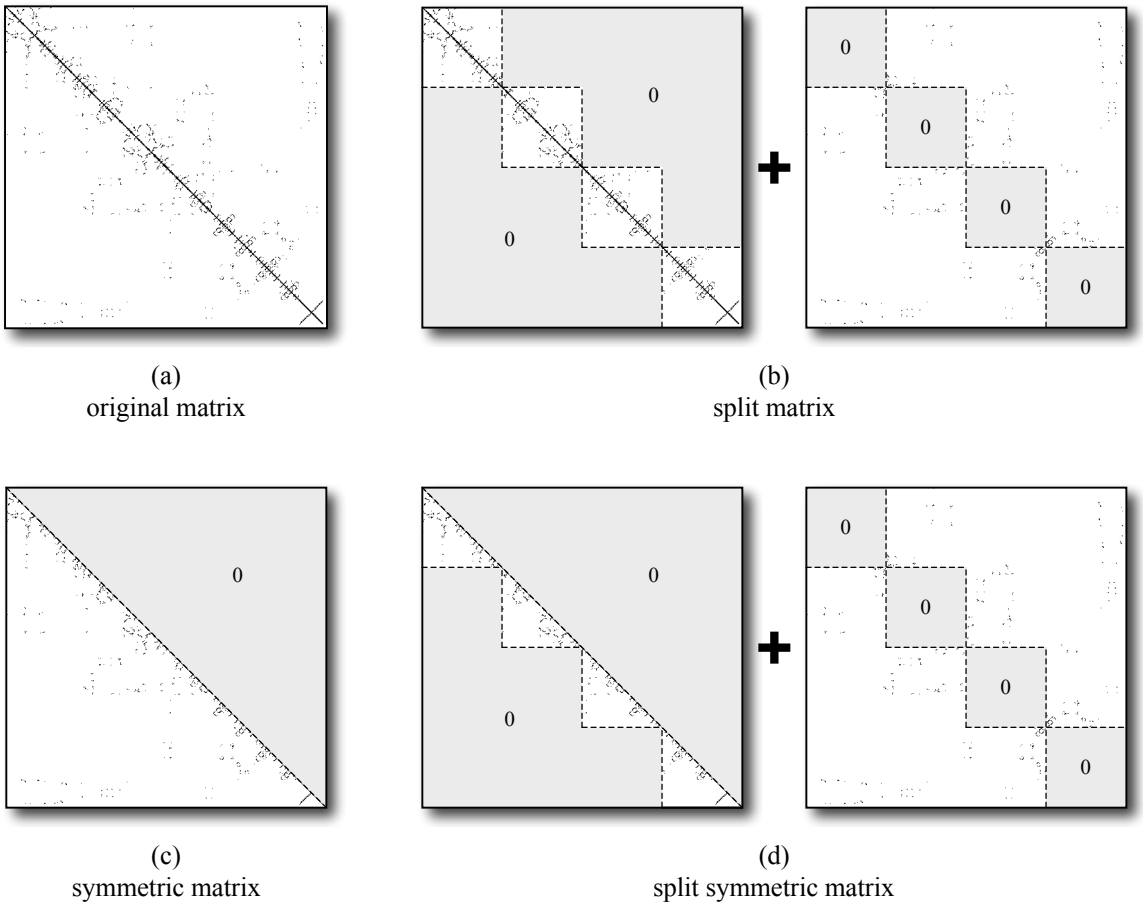


Figure 8.17: Exploiting Symmetry and Matrix Splitting. From the original matrix (a), one may split the matrix (b) or exploit symmetry (c). Exploiting both is shown in (d).

Symmetric Storage

Figure 8.5 on page 161 shows nearly half of the matrices in our evaluation suite are symmetric; that is, $A = A^T$ or $A_{ij} = A_{ji}$. Our matrix loader recognizes this, and converts any symmetric matrix to non-symmetric by duplicating nonzeros. By doing so, a common set of optimizations and SpMV routines may be executed. Although the total number of floating-point multiplies remains unchanged, the clear downside is that twice the storage and memory traffic are required — thereby potentially cutting performance in half. Unfortunately, symmetric storage is not easily parallelized. We examine several ideas applicable to multicore computers.

For symmetric matrices with the bulk of nonzeros near the diagonal, like `Protein`, `Spheres`, `Cant`, `Tunnel`, and `Ship`, it is possible to split the matrix into a low-bandwidth symmetric matrix and another symmetric matrix containing the remaining nonzeros far from the diagonal. The symmetric storage optimization would be applied to the low bandwidth

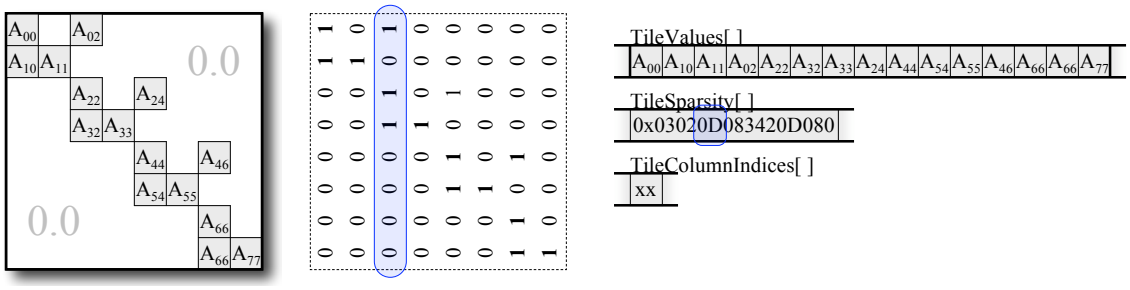


Figure 8.18: Avoiding zero fill through bit masks. One sparse register block is shown. A column-major bit mask marks the nonzero elements per register block.

matrix, where the matrix of the remaining nonzeros would be stored in a non-symmetric format. This could double performance on the heavily bandwidth-limited x86 architectures and quite possibly all future versions of Cell. However, Victoria Falls' limited FLOP rate and instruction bandwidth would dramatically limit performance gains. This approach is also applicable to the case of very special complex (in the mathematical sense) matrices. In a Hermitian matrix, $A = A^{\dagger}$ or $A_{ij} = A^{*}_{ji}$. Thus, with nothing more than one `XOR` instruction to negate a floating-point number, we may change $Real_{ij} + Imaginary_{ij}$ into $Real_{ji} - Imaginary_{ji}$ and then apply the same optimizations used in symmetric matrices.

Figure 8.17 on the previous page demonstrates thus approach. Given the original symmetric matrix stored in a non-symmetric form (a), it is possible to eliminate the upper triangle of nonzeros as their values are duplicated in the lower triangle (c). However, this approach doesn't parallelize well and would be difficult to effectively implement on Cell. Alternately, one could split the matrix into two matrices: one containing blocks near the diagonal and one containing the off diagonal blocks (b). Although this approach would parallelize well, it hasn't reduced the memory traffic. However, one could now exploit symmetry along the diagonal blocks (d). If they constitute the bulk of the nonzeros, then not only is this easy to parallelize, but it will possess a significantly higher arithmetic intensity — well worth the data structure transformation and more complex SpMV.

Sparse BCSR

The traditional register blocking fills in explicit zeros to create nicely sized dense blocks. In an era where FLOPs are perceived as being free, or at least are becoming 15% cheaper every year, we must call into question whether or not explicit fill is required. Figure 8.18 shows it is possible to store only the nonzeros of a register block, but add an additional array representing the sparsity pattern within the register block using a bit mask — a 8×8 unfilled register block would require an additional 64-bit bit mask. In this example, one register block is shown. Moreover, the bit mask represents the column major sparsity of the register block. `popcount` can be used to calculate the actual storage requirement for a sparse register block.

There are now three options for processing these compressed register blocks. One could decompress blocks of them into dense register blocks stored in the cache or local

store and then use existing BCSR routines, or alternately, one could decompress them on the fly within the inner loop without explicit storage. Although the latter is slightly more complicated, there is a tremendous opportunity to reduce the number of wasted floating-point operations. The third option is to create special instructions that could efficiently process these tiny sparse matrix-vector multiplications. This storage format may ultimately provide up to a 50% performance benefit on a much broader set of matrices, but at a significant productivity cost.

Elimination of Redundant Values

There is one final potential future work item geared at reducing matrix memory traffic that we discuss here. It is possible that many double-precision matrices can be represented exactly in single-precision. For example, if the 29 least significant bits of the mantissa of the double-precision representation of a floating-point number are zero, and the exponent has an absolute magnitude less than 2^{127}, a floating-point number can be represented exactly in single-precision. The storage of the matrix values in single-precision could reduce memory traffic by 1.5× to 2×. Of course, the vectors must always be stored in double-precision, and all computation would still be performed in double-precision. During execution of a single-precision block, as each nonzero is loaded from DRAM, it could be converted to double-precision without loss of accuracy. All subsequent computation would be performed in double-precision, and the resultant vector would be stored in double-precision without loss of accuracy. The auto-tuner could be expanded to inspect the matrix at run-time, determine which cache blocks can be represented in single-precision without a loss of accuracy, then convert them. One could even contemplate a 16-bit half-precision or even bit representations of the values of a matrix.

Alternately, just as one observes that adjacent rows and columns have the same coordinates and can thus create register blocks, one could observe that there may be a finite number of unique floating point values within a matrix. Consider a matrix with 10M nonzeros of which there are only 16K unique values. Rather than storing one double-precision floating-point value, and at least one 32-bit integer per nonzero, we could replace the 64-bit float with a 16-bit index to an array of the unique floating-point values as motivated by [83]. As such, the SpMV arithmetic intensity would improve from 0.166 to 0.333 — a 2× improvement. In addition, if one could perfectly apply register blocking, the arithmetic intensity could be improved to nearly 1.0 — a 6× improvement. As the number of unique values decreases, the better the performance.

8.6.3 Better Heuristics

The third arena for future work is the development and application of better heuristics. Better heuristics may improve performance and reduce tuning time.

As previously discussed in Section 8.4.5, not all architectures are heavily memory-bound. In fact, some may be computationally-limited. As such, the matrix compression heuristic could be augmented with a more traditional OSKI style approach to register blocking selection. As time passes, the FLOP:Byte ratio of more and more architectures will significantly exceed that of SpMV. We are currently in a transition period where most,

but not all architectures have done so. As such, in the long term, our existing heuristic will be acceptable. Thus, better heuristics are a minor concern.

Second, when prefetching, we perform an exhaustive search over both read streams in cache line granularities up to 1 KB. This search is extremely expensive. Although the optimal choice can vary greatly from one architecture to another, it does not vary significantly across matrices. As such, a one time offline tuning could find a reasonable prefetch distance for each architecture.

Most matrices on most architectures were well load balanced. It is typically very easy to balance 4 threads — the parallelism per thread is high, and the probability that one thread used a spectacularly bad register blocking is low. However, the extreme multithreading on Victoria Falls ensures the parallelism per thread is low and there is a chance that one of the 128 threads may select 1×1 register blocking when all other threads chose something larger. Thus there were several matrices for which Victoria Falls was 25% unbalanced. Although 25% sounds small, when architectures are within a comparable performance-bound it can significantly skew one's conclusions. Future work should strive to ensure effective, perhaps dynamic, load balancing as thread counts may tend to 1,000.

8.7 Conclusions

In this chapter, we examined the applicability of auto-tuning to sparse kernels on multicore architectures. As sparse linear algebra is an immensely broad motif, we chose sparse matrix-vector multiplication (SpMV) — as an interesting and common example kernel. Despite the fact that the standard CSR implementation has existed for decades, we see auto-tuning provided substantial speedups on all architectures.

Although, as suggested by the Rooflines, SIMDization is ineffective, we see that some of the largest benefits came from minimization of memory traffic and maximizing memory bandwidth through NUMA allocation and software prefetching. Before auto-tuning, we see most of the cache-based architectures providing similar performance. Unfortunately, the out-of-the box Cell PPE performance was extremely poor. As we trade productivity to expand the capability of the auto-tuner, all cache based machines quickly reach good performance and high fractions of their respective attainable memory bandwidths. However, the data structure transformation associated with register blocking provides the most substantial impact. The Cell implementation, although heuristically tuned, required slightly more work than the fully optimized cache implementations. Unlike LBMHD, Cell's extremely weak double-precision implementation is not a significant performance bottleneck for SpMV.

Without radical technological or algorithmic advantages, the trends in computing suggest SpMV will become increasingly memory-bound as core counts increase. This doesn't obviate the need for auto-tuning. Rather, it increases its value. First, achieving peak memory bandwidth is challenging and only achieved through selection of the optimal data layouts and prefetching. Moreover, the superfluous computational capability will allow one to realize complex data structures and compression techniques aimed at minimizing the total memory traffic. Ultimately, algorithms with superior arithmetic intensities and at least comparable computational complexity are required.

Chapter 9

Insights and Future Directions in Auto-tuning

In this chapter, we take a high-level integrative view of the auto-tuning efforts of Chapters 6 and 8 and discuss several directions this work could take in the future. To that end, Section 9.1 makes several observations of LBMHD and SpMV and discusses their implications for auto-tuning, architecture, and algorithms. In Section 9.2 we discuss how one could apply the auto-tuning techniques and methodology employed for LBMHD and SpMV to specific kernels from other motifs. Unfortunately, existing auto-tuning efforts still lack the desired productivity for efficiency-layer programmers. To that end, Sections 9.3, 9.4, and 9.5 discuss how auto-tuning might reach its full potential. Finally, Sections 9.6 discusses some high-level conclusions.

9.1 Insights from Auto-tuning Experiments

Consider the work from Chapter 6 and 8. In both chapters, we restricted ourselves to one particular "representative" kernel and extensively auto-tuned it. In this section, we attempt to capture the high-level cross-motif insights and discuss their implications for auto-tuning, algorithms, and architecture. Although not presented in this work, we include the insights acquired from our auto-tuning of another structured grid kernel — a 7-point Laplacian operator applied to a 3D scalar grid [37]. We believe these insights will facilitate the application of auto-tuning to other kernels in the structured grid and sparse linear algebra motifs as well as applying auto-tuning to the newer, non-scientific computing motifs.

Broadly speaking, we observe the timescale for auto-tuning adaptation and innovation must be much less than the timescale for architectural innovation. That is, we must adapt to and optimize for an architecture long before its successor is released. The timescale for a single generation is perhaps two years. As such, we demand auto-tuning adaptation and optimization in perhaps one month. Similarly, we observe that algorithmic innovation and acceptance is typically measured in decades. Thus, we have the time to implement algorithmic acceleration features in hardware. In the following subsections, we discuss the implications and future directions in auto-tuning, architecture, and algorithms.

9.1.1 Observations and Insights

In our examination of the application of auto-tuning to SpMV, LBMHD, and the 7-point stencil, we have noted several trends and insights. These can be categorized into four domains: bandwidth, in-core, parallelism, and search. We discuss them here.

Bandwidth and Traffic

When examining these three kernels, we observe they all have constant, or more appropriately, bounded compulsory arithmetic intensity with respect to problem size. Moreover, these arithmetic intensities are so low, they are less than the ridge point for virtually any machine today. As the Roofline model suggests, performance can be bounded by the product of arithmetic intensity and Stream bandwidth. As such, these three kernels are often labeled as *memory-intensive* kernels. Thus, regardless of problem size, we naïvely expect roughly the same performance. Naïvely, one might believe that hardware prefetching and out-of-order execution are sufficient to guarantee performance at the arithmetic intensity–Stream bandwidth product. Thus, one might erroneously conclude that auto-tuning is unnecessary. As borne out in the data in Figures 6.16 on page 131 and 8.15 on page 179, it is clear that significant increases in performance can be achieved via auto-tuning memory-intensive kernels.

One must ponder the reasons for poor performance on memory-intensive kernels. There are only two real possibilities: poor memory bandwidth or memory traffic in excess of the compulsory memory traffic. We believe that by using both software and hardware prefetching as well as multithreading, we can achieve a high fraction of Streaming bandwidth, although only performance counters could verify that belief. Thus, our primary task is to ensure the total memory traffic is not significantly greater than the compulsory memory traffic. Unfortunately, minimization of memory traffic can be a daunting and poorly understood task for novice programmers as they do not understand the intricate and complex behavior of caches with finite capacities and associativity. Not all memory requests will hit in the cache. As such, in addition to the compulsory misses, it is possible for caches to generate conflict and capacity misses. Moreover, write allocate caches will fill the cache line in question on a write miss. Thus, we classify memory traffic as compulsory, capacity, conflict, or allocate.

Consider our three kernels. When we consider the primitive operations, a lattice update, a 7-point stencil on a scalar grid, or a nonzero multiply-accumulate (MACC), we observe that they will show different degrees of reuse and thus different working set sizes. There is no inter-lattice update reuse, but there is reuse within an update. As such, in the absence of bandwidth considerations, we can choose any traversal of the data, but to maximize bandwidth and efficiently vectorize the lattice updates, we choose a conventional traversal and thus control the working set size. When examining the 7-point stencil, we see there is partial reuse among three stencils in any direction. As such, we must select an optimal traversal that maintains a sufficiently large working set in the cache. Failure to do so will result in extra capacity misses. Finally, if we consider a MACC on a nonzero, we see that the source vector element will be reused by all other nonzero MACC's in its column, and the running sum will be reused by all nonzero MACC's in its row. As such, to minimize

capacity misses, one should select a traversal of the nonzeros that maintains a sufficiently large working set in the cache.

Limited cache associativity can give rise to conflict misses before the capacity of the cache is exhausted by the working set. Currently, we believe this is primarily a problem on structured grid codes with their rigid stencil structure. Nevertheless, as discussed in Section 8.4.2, it is possible for sparse methods to be hampered by conflict misses. We only addressed nonzero conflicts between threads.

Although structured grid codes typically read as much data as they write, sparse codes typically read much more data than they write. As a result, write allocation traffic predominately affects only the structured grid kernels here. Moreover, elimination of this traffic can reduce the total memory traffic by up to 33% — a huge boon on memory-intensive kernels.

Finally, we observe that unlike structured grid codes, both the addressing and edge weights are explicit in sparse codes. As such, vector data is the minority of the "compulsory" traffic. By eliminating some redundancy in the indices through register blocking, we can cut the compulsory traffic by perhaps 33%.

Figures 6.15 and 8.14 visualize the potential performance gains from these cache and memory observations. We observe these memory bandwidth and traffic-oriented optimizations deliver the bulk of the performance gains.

In-Core Optimizations

When we look at the requisite optimizations, we see nearly a complete lack of in-core optimizations. In fact, reordering and unrolling delivered small increases in performance. Essentially, the SSE implementation was only implemented to facilitate the cache bypass optimization of avoiding the write allocate traffic. Unfortunately, most compilers were incapable of optimizing intrinsic-laden code. Thus, reordering and unrolling were beneficial only because compilers couldn't exploit cache bypass.

Such a trend should come as no surprise given the Roofline model. It clearly shows that most architectures will be heavily memory-bound. Moreover, there is sufficient instruction-level parallelism in the structured grid codes to ensure a lack of multiply-add balance or SIMD doesn't impede performance.

Parallelism

For better or worse, the thread-level parallelism in these kernels was static, abundant, and trivially discovered. As discussed in Chapters 5 and 7, such characteristics are not omnipresent.

Both the LBMHD and 7-point stencil structured grid codes use Jacobi's method. As such, every point may be updated in parallel, and thus load balancing is easily achieved. The primary challenge is how to divide up the grid into pieces. This task has two major challenges: dealing with shared caches, and handling NUMA allocation with potentially large pages.

If we were to have examined an upwinding stencil, the parallelism would vary substantially as only a diagonal plane in a cube might be executed in parallel. A diagonal

plane at a corner has no parallelism. As the plane sweeps through the cube, the parallelism quickly increases to N^2, then quickly drops back to none. Clearly, this is far more challenging to parallelize on multicore SMPs with as many as 128 threads. Nevertheless, although the parallelism is variable, the parallelism at a step can be statically predicted as soon as the problem is specified. At the extreme are AMR codes where the parallelism is not only variable, but cannot be predicted, and is in fact dynamic at any time step. Thus, barrier, shared queue, and load balancer performance become critical.

When it comes to SpMV, once again, we can use the high-level knowledge that all destination vector elements may be independently calculated. Although all rows are independent, the computation required for each can vary substantially. As such, load balancing is the challenge. Only when the matrix is specified, is it possible to load balance the computation. If one were to consider the other sparse kernel discussed in Chapter 7, sparse triangular solve (SpTS), we see that DAG inspection is required to determine the characteristics of parallelism. In SpTS, the parallelism is not static, may not be abundant, and can be difficult to discover.

In the end, no parallelism-oriented auto-tuning was performed for LBMHD or SpMV. That is, we heuristically selected one parallelization strategy. However, it was clear that extensive parallelism-oriented auto-tuning was performed in [37] through chunking, core blocking, and thread blocking. We believe that in the future, as kernel complexity increases, so too will the parallelism-oriented auto-tuning effort.

Search Methodology

The goal of an auto-tuner's search methodology is to efficiently explore the optimization parameter space; that is, minimize the exploration while maximizing the performance. When we examine the search methodologies employed by these three auto-tuning attempts across perhaps 10 different optimizations, we see a variety of different search methodologies with no clear winner. Often exhaustive search or greedy search is used as a crutch when we don't understand the intricacies of an architectural paradigm. Typically, cache conflicts and hardware prefetching are the most challenging architectural features to understand. On Cell, without caches and where DMA functionality is easily understood, simple heuristics are efficiently employed as the search methodology. However, on machines with low cache associativities or complex and opaque hardware prefetching mechanisms, search has proven to be easier than attempting to understand the interacting forces. When examining the raw LBMHD data, we observe that the optimal vector length is often well correlated with cache sizes, and the optimal reordering and unrolling for SIMDized code was well correlated with cache line sizes. Thus, a heuristic could have been developed to accelerate the exhaustive search. Depending on architecture and optimizations, an auto-tuner may need a combination of search techniques.

9.1.2 Implications for Auto-tuning

Given the trends in computing and the slowly evolving algorithms, we believe that most computers will become increasingly bandwidth-limited on kernels with ample parallelism. For example, if the number of cores grows faster than bandwidth, then the ridge

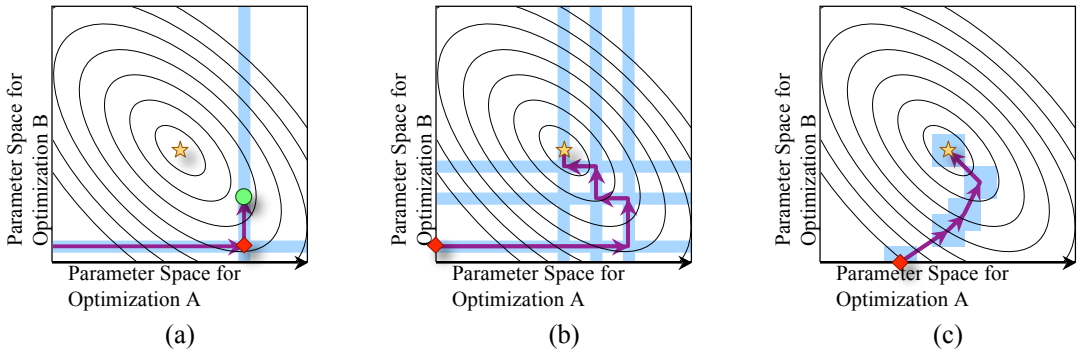


Figure 9.1: Visualization of three different strategies for exploring the optimization space: (a) Hill-climbing search, (b) Iterative hill-climbing, (c) Gradient descent. Note, curves denote combinations of constant performance. The gold star represents the best possible performance.

points in the Roofline model will move to the right. As a result, more and more kernels will be bandwidth-limited. Thus, the demands on in-core performance and compiler capabilities will be reduced. Moreover, we believe that auto-tuning would be greatly simplified as one could employ bound-and-bottleneck-based heuristics. That is, if we can identify a communication bottleneck in a computer's execution of a kernel, then the auto-tuning effort should be geared to minimizing traffic. To facilitate such approaches on cache-based architectures, one should utilize algorithms that eliminate cache effects by copying into fixed buffers stored in cache like the circular queue [37, 81].

For kernels that do not lend themselves to any obvious high-level heuristic, there are several possible directions auto-tuning might take aside from a naïve exhaustive search. First, as discussed in Chapter 4 one could iterate on constraint identification, optimization, and removal of said constraint. The challenge is the selection of the appropriate optimization and parameters. In all likelihood, the programmer would be required to annotate or analyze the optimizations and pass that information to the auto-tuner.

Second, one could employ an iterative form of the hill climbing algorithm discussed in Section 2.4 of Chapter 2 and shown in Figure 9.1(a). That is, if the performance hyper-surface is sufficiently smooth, then one iterates over and over through the optimization space examining the performance for each optimization as a function of parameter. For each optimization, the best known solution is the starting point for the next optimization search. Figure 9.1(b) clearly shows that the number of trials may be significantly larger than those required in Figure 9.1(a), and there is no guarantee an optimal solution will be found. With some local sampling of the performance hyper-surface, one might be able to employ a method similar to gradient descent [120]. Figure 9.1(c) shows that such an approach might dramatically improve the search time.

Finally, a completely novel technique would be to apply machine learning. As suggested by Ganapathi *et al.*, one could randomly sample the optimization parameter space and apply machine learning techniques to look for correlations between optimization

parameters and performance metrics [56, 55] via KCCA [8]. Clearly, this model is completely oblivious of the architectural details. Given the highest performing input parameter, one can look for its neighbors in projected space and back project them into the original parameter space. One can then try different permutations of these key parameters. Results have shown this can achieve very good performance on the problems for which exhaustive search is simply not tractable. This is an area of continuing research for which KCCA is but one of several possible approaches.

9.1.3 Implications for Architectures

Given our observations and insights into auto-tuning, we discuss their implications for architecture. Primarily, these can be categorized into the effects on core microarchitecture, cache sizes, and off-chip bandwidth.

Given our breadth of architectures running auto-tuned SpMV and structured grid codes, we observe that the product of high frequency, superscalar, and out-of-order execution is overkill. Two or possibly only one of the three are sufficient. Moreover, for these codes multiply-add balance is atypical and irrelevant on memory-bound kernels. Perhaps in the future algorithms may express more locality and thus computation will become more important. If such a day were to come, we could imagine rebalancing the floating-point datapaths away from the LINPACK-centric balance between multiplies and adds. Stencil codes often exhibit more than an order of magnitude more floating-point adds than multiplies. As such, we could envision asymmetric SIMD units to ensure efficient utilization of silicon. That is, fully pumped 128-bit floating-point adders, but half-pumped 64-bit floating-point multipliers. Thus the throughput for SIMD adds would be one per cycle, but SIMD multiplies would be executed at one every other cycle.

Although untuned codes generally perform better on computers with large caches, auto-tuned codes will restructure themselves to adapt to the smaller working set sizes. We observe that processors like Cell with only 256 KB per core can run auto-tuned sparse and structured grid codes extremely well. Thus, there is little need to design computers with per core cache capacities in excess of 1 MB. Although auto-tuning is particularly adept at adapting to differing cache capacities, architectures and auto-tuners tend to struggle with low per thread associativities. We suggest that caches be designed with significantly more associativity than threads sharing them. That is, caches shared by 8 threads should be at least 8-way associative. Moreover, local store architectures implicitly eliminate the possibility of cache conflict misses and both conflict and capacity TLB misses. Thus, local store functionality is desirable. Either a custom local store can be added to the memory hierarchy, or part of the cache can be reconfigured as a local store.

Numerical methods for which the arithmetic intensities are constant with respect to problem size invalidate the simple multicore version of Moore's Law. That is, there is no justification for doubling the number of cores every two years if bandwidth has only increased by 40% in that same time frame. Rather, two more plausible, conventional solutions arise. First, manufacturers could increase both the number of cores and the bandwidth by 40% every two years. This would imply chips get smaller and smaller, but performance would only increase at 20% per year. Alternatively, one could double the number of cores every two years, but reduce the per core performance by 40% in that same time period. As a

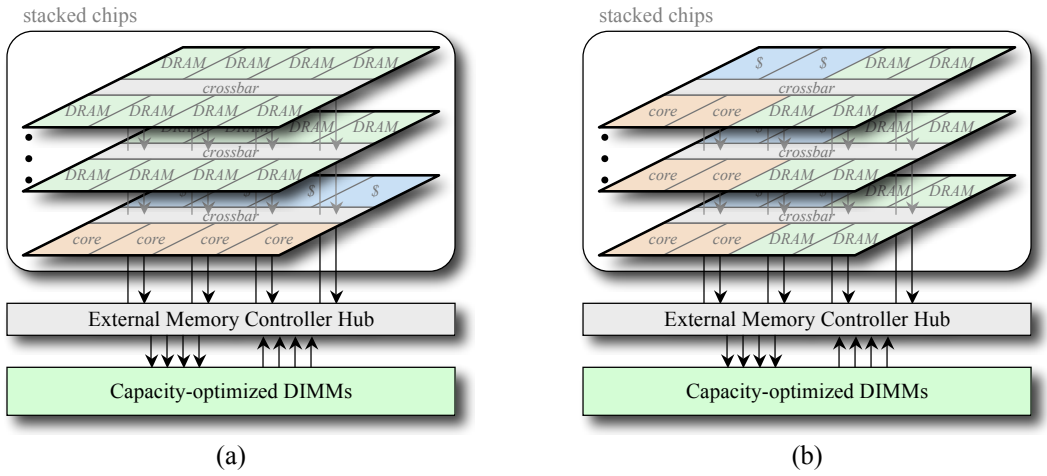


Figure 9.2: Potential stacked chip processor architectures. (a) DRAM chips stacked on a multicore chip, (b) identical chips with multiple cores and embedded DRAM.

result, the cores would continue to deliver sufficient performance to use all the available memory bandwidth. The principal upside of such an approach is the power consumed by the chips would likely decrease by much more than 20% per year — a clear advantage in a mobile or green market.

What is truly required is novel and performance scalable means of attaching large capacities of main memory to processors. We assume we need at least 256 MB of main memory per core and several GB per SMP. Three reasonable possibilities have been proposed.

First, optically connect processors to DIMMs. Current research suggests that a tremendous benefit is possible [13, 132]. Of course, there is no need to utilize photonic's potential in a single step. Instead, the performance boost could be doled out over the course of a decade to maintain a constant FLOP:byte ratio.

Second, Figure 9.2(a) shows that it is possible to stack DRAMs on top of multicore chips in a system-in-package (SIP) using *through-silicon vias* to interconnect the chips. Doing so could greatly increase the bandwidth to part or all of main memory. Bleeding edge DRAM modules are nearly 256 MB/cm^2, and current quad-core superscalar and eight-core Cell chips are about 1 cm^2. Thus, to achieve 256 MB/core, one must stack 4 to 8 DRAM chips on top of a multicore chip. In doing so, one sacrifices expandability and sheer capacity afforded by multiple DIMMs for performance. To ameliorate this, one could still employ a conventional external memory controller hub attached to conventional DIMMs. Thus, one could partition the physical address space into a relatively small, but fast on-chip main memory, and a large, but slower, off-chip main memory. A specialized version of `malloc()` could be used to allocate on-chip memory.

Finally, Figure 9.2(b) shows that one could integrate a couple cores and perhaps 256 MB of DRAM onto a single chip. One could then stack multiple identical chips to create a multicore socket. Clearly such an approach is both a *non-uniform on-chip memory access* and *non-uniform off-chip memory access* architecture. Just as current external memory

controller hubs can integrate two or more chips, the same approach could be reused here. In many ways, this is a stacked and multicore reincarnation of IRAM [84, 57]

9.1.4 Implications for Algorithms

Perennially, the timescale for algorithmic innovation is measured in decades. As such, we in the computing industry have relied on technological and software advances to deliver superior time to solutions for various problems. Assuming current technological scaling trends will continue into the future, to achieve performance that will scale better than bandwidth, we need algorithms whose arithmetic intensity increases over time or at the very least is not constant with respect to problem size. Unfortunately, this means we require a relaxation on implementation or at the very least, a high-level conceptualization of the problem definition within which we are free to choose the appropriate implementation.

9.2 Broadening Auto-tuning: Motif Kernels

In this thesis, we applied the auto-tuning optimization technique to one kernel in each of two computational motifs. We believe auto-tuning can be applied to any well defined kernel in virtually any motif. Note, this doesn't imply the same optimizations are required or the same improvements will be seen. In this section, we discuss how one might auto-tune various kernels from a subset of the computational motifs.

9.2.1 Structured Grids

When the auto-tuning work of Chapter 6 is examined in conjunction with the optimizations presented in [37], we believe the bulk of the structured grid-related optimizations have been enumerated. However, there are a few other kernels that demand additional optimizations.

As previously discussed, Gauss-Seidel or upwinding stencils demand dramatically different approaches to parallelization when compared with Jacobi's method. Here the performance of barrier collectives dictates the maximum parallelization for a particular problem size. Consider Figure 9.3 on the next page. There are two orthogonal optimizations: data layout and parallelization. Figure 9.3(a) shows one could choose to parallelize within the execution of each diagonal. Clearly, barrier or synchronization time is critical. Figure 9.3(b) shows one could choose to block the grid and execute some blocks in parallel. Barriers are less frequent, but there is also less parallelism. When it comes to data structure, one might choose to lay the data out either by rows for simplicity or by diagonals to match the execution's access pattern. Moreover, if the form of parallelism is blocking, one might choose to apply a hierarchical storage format. That is, by rows or diagonals within a block. Thus, one must also tune to find the optimal parallelization style, block sizes, and concurrency.

9.2.2 Sparse Linear Algebra

Although we extensively auto-tuned SpMV in Chapter 8, there are dozens of other important sparse kernels that could be auto-tuned for multicore architectures.

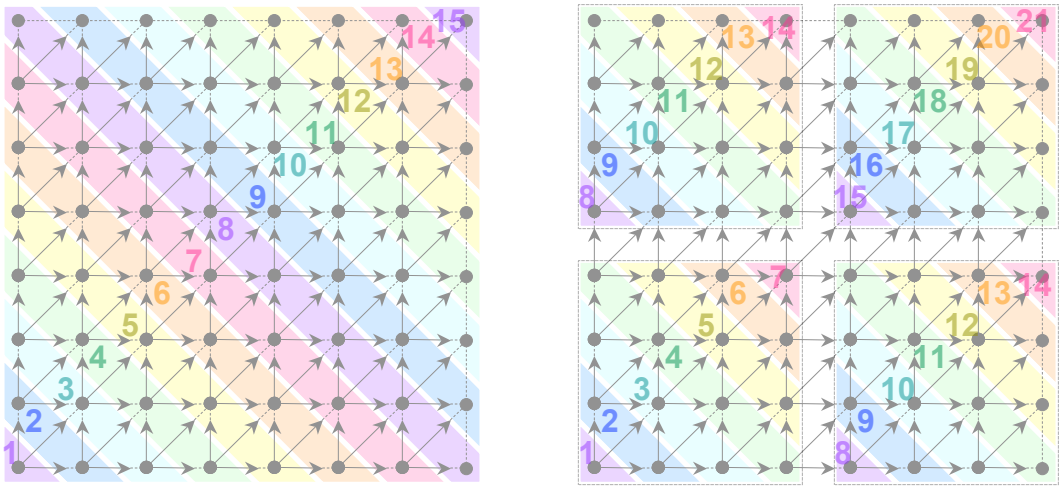


Figure 9.3: Execution strategies for upwinding stencils: (a) unblocked, execute entire diagonals, (b) blocked, execute by diagonals within a block.

The most obvious next kernel, as suggested by [134], is sparse matrix-matrix multiplication (SpMM). That is, a sparse matrix times a tall, skinny dense matrix (a dense matrix with far more rows than columns). The simplest approach would be to use the existing SpMV framework with small changes to the inner kernel. In essence, one should calculate multiple destination vectors simultaneously by simply loading multiple elements from the same row of the source vector. The principle tuning parameter is how many of these destination vectors should be simultaneously calculated. Clearly, the matrix is still parallelized among threads. One could instead exploit the multiple cores to individually calculate different destination vectors. The first core would load nonzeros from main memory, calculate its vector elements, and then pass the nonzeros to the next core. That core would in turn load its own source vector elements and update its destination vector. Such a technique is applicable on architectures with good inter-core bandwidth, cores that can individually saturate main memory bandwidth, and SpMM operations with dense matrices larger than the sparse ones.

Chapter 7 clearly motivated the value of both SpMV and sparse triangular solve (SpTS). Vuduc performed some preliminary auto-tuning of SpTS on single-threaded architectures using both register blocking and large dense blocks [134]. Multicore demands a new form of parallelism be discovered. Clearly, some DAG analysis is required to enumerate the nodes that can be computed in parallel. The desire for good cache locality counterbalances this parallelism. As such, one must pass up potential parallelism to avoid cache misses — a challenging analysis or tuning problem. Optimization is further complicated by the non-pipelined nature of floating-point divides. Thus, parallelization and SIMDization of such divides is critical.

Finally, given the bandwidth intensive nature of SpMV, the performance impact of

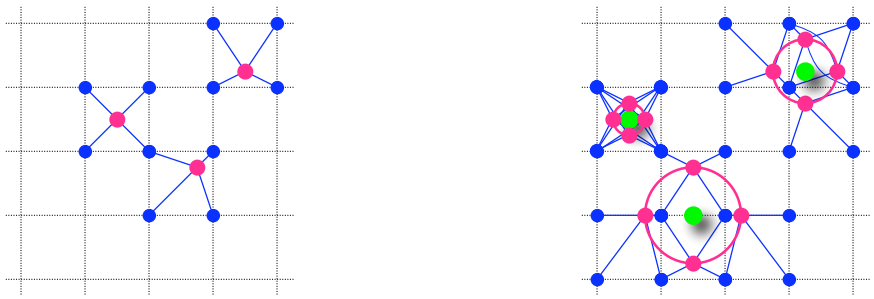


Figure 9.4: Charge Deposition operation in PIC codes: (a) deposition when particles are points, (b) deposition for codes like GTC where particles are rings.

software implementations of complex or double-double SpMV may only be a factor of two. The potentially improved application level numerical accuracy and stability may justify its use.

9.2.3 N-body

One interesting area for future auto-tuning research is auto-tuning particle-mesh or particle-in-cell (PIC) codes. Algorithmically, these codes deliver O(N) complexity rather than the normal $O(N^2)$ at the expense of poor locality and throughput. The most challenging step of PIC codes is the charge or mass deposition, where in preparation for solving Poisson's equation the grid must mimic a continuous distribution of mass or charge.

Figure 9.4 visualizes two PIC codes. Each particle is a point in Figure 9.4(a). Thus, it is always boxed in by four points. In the charge deposition phase, each particle's charge (red) is distributed among the four grid points (blue) via a scatter increment. For any given particle, those four updates are independent and thus may be executed in parallel. A problem arises. Any two particles may attempt to update the same point. So how can this code be parallelized across a large multicore SMP? Moreover, Figure 9.4(b) approximates the behavior of Gyrokinetic Toroidal Code (GTC) [88]. Here, ions and electrons gyrate around a line perpendicular to the plane (green). Thus, they are approximated by rings (red). As the radius of the rings are constrained by their finite energies, the rings can in turn be approximated by four points (red). Each of these four points must deposit one quarter of the particle's charge to its neighboring grid points. Thus, not only is it possible for there to be intra-particle data hazards, but also inter-particle data hazards. How can this be efficiently executed on a multicore SMP?

Let's examine multi-thread parallelism as it is likely to require discovery of the greatest degree of parallelism. The simplest solution, inspired by GTC's vectorization technique on the SX-6 [102], is to replicate the grid P times: one per thread. All threads update their own private grid, then at the end of the deposition phase, the P grids are reduced to one. Such techniques are viable when the number of threads is less than the average number of particles per grid point. The second major parallelization approach is to spatially bin

(cheaper than sort) the particles. The grid (and thus the particles) can then be partitioned among threads. If the particles move slowly, then the grids can be padded, and the binning can be performed infrequently.

As seen in Chapter 6, achieving parallel efficiency is far easier than architectural efficiency. We believe a similar situation will arise on PIC codes. Within a thread, the grid updates are essentially random gather-scatters on grids exceeding a few megabytes. As such, we expect hardware prefetchers to be ineffective and we will thus expose main memory latency. Perhaps software prefetching, multithreading or, DMA may ameliorate this, but ultimately we require cache locality. If binning particles is not possible, we might be able to bin the updates into a quad-tree of lists. We could tune for the fan-out and depth to balance the desire for cache locality with the multiple streaming bin operations.

Aside from parallel efficiency, we may tune for single-thread performance. Observe that similar to the approach of allocating a copy of the grid for every thread, we could allocate a copy of the grid for each of the four points on the ring. Within a thread, this increases the parallelism from 4-way to 16-way. Moreover, given the bounds on gyration radius, one could visualize the update as a dense tile of updates. As suggested by Figure 9.4(b), there are only a finite number of different, tetris-style blocks that can arise. As such, one could bin particles based on their pattern. Within each block, one could explicitly avoid data hazards by reading each point once, and performing multiple increments in registers.

9.2.4 Circuits

The circuits motif encompasses both combinational logic and sequential logic. Unlike other motifs where the typical operands are floating-point or integer numbers and the typical operators are add, subtract, multiply, and divide, the typical operand in the circuits motif is a bit or a bus, and the typical operators are logical bitwise operators like `AND`, `OR`, `NOT`, `XOR`, and `MUX`.

The kernels of the circuits motif include cyclic redundancy checks (CRC), error checking and correcting (ECC), encryption, and hashing. Unfortunately, such kernels typically operate on data (bit) streams and express relatively little parallelism. For example, CRC32 only defines 32 independent operations. This presents an enormous problem on multicore SMPs. Consider a Clovertown SMP performing bitwise `XOR` operations. Each core may execute three 128-bit SIMD `XOR` instructions per cycle. With eight cores, that's 3K independent `XOR` bit-operations per cycle. Clearly, CRC operations will make poor use of such compute capability. On the upside, Clovertown has so little main memory bandwidth (sustaining less than 32 bits per cycle) that it may be bandwidth-limited using only one core. Moreover, such processors may be tasked with performing several independent circuits kernels that could be efficiently parceled out to multiple cores. The importance of this motif cannot be understimated as evidenced by SSE's recent inclusion of CRC and AES instructions.

If simply finding parallelism wasn't a big enough problem, existing SIMD instruction sets are designed for integer and floating-point operations. As such, they are typically oriented around packed loads and stores. They often implement gather and scatter operations through insertion and extraction of 64-bit, 32-bit, 16-bit, or very recently, 8-bit

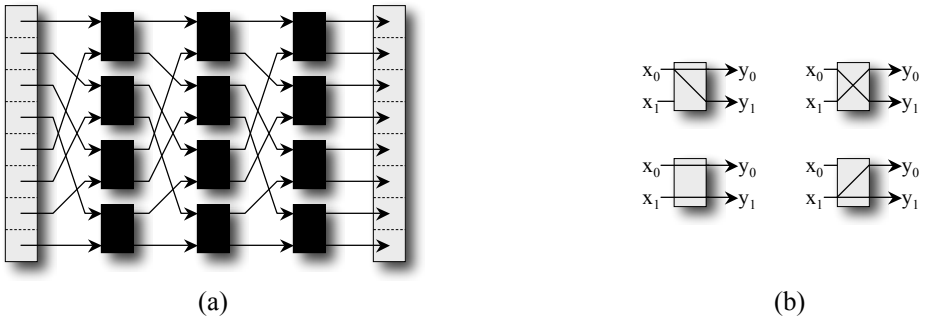


Figure 9.5: Using an Omega network for bit permutations: (a) 8 node network implemented in 3 stages, (b) modified functionality per "switch."

elements. Consider the vector analogy. Vector processors without scatter or gather instructions are relegated to executing only the simplest kernels.

Given a combinational logic circuit, one could visualize it as a series of stages in which there are some number of `AND`'s, some number of `OR`'s, some number of `NOT`'s, and some number of `XOR`'s that must be performed. The total number of such stages is the depth of the DAG. All `XOR`'s within one stage could be strip mined into a series of SIMD instructions. Thus, one could attempt to map this DAG onto a grid of operations (stages $\times$ operation $\times$ bit). The ultimate challenge of this motif is the implementation of an register transfer language (RTL) compiler that can place gates onto this grid so that the number of inter-stage bit permutation instructions is minimized. To that end, operations may be executed late or multiple times. Moreover, space may be wasted within each stage to minimize the number of bit permutation instructions. From a tuning perspective, one must decide how much waste or duplication is acceptable. For example, each bit signal could be stored in a 32-bit register. Four of these could be efficiently packed into a SIMD register. However, this uses a small fraction of a SIMD instruction's inherent bit-level parallelism. Alternately, one could use anywhere from eight 16-bit elements to store eight 1-bit signals to 128 1-bit elements. The ISA's facilities for bit permutation dictate the optimal implementation.

Parallel code implementations demand the circuit exhibit tremendous bit-level parallelism so that the circuit can be partitioned. Even if such parallelism is present, high performance may demand cores perform redundant work.

Software tuning may be insufficient in facilitating the exploration and development of kernels within the combinational logic motif. As such, manufacturers should attempt to facilitate development within this motif through the addition of general instructions. Consider an omega network [68] retasked to permute the bits of a 128-bit register. Although it is impractical to encode such functionality in a single instruction, we believe that the functionality of each stage could be encapsulated in one instruction. Thus, execution of the same instruction seven times would allow an arbitrary 128-bit permutation. Figure 9.5 shows a dimunitive version that shuffles bits within a byte. Each "switch" must be modified

to also broadcast either input. Thus, 2 control bits are required for each of the $\frac{b}{2}$ switches per stage, and the entire control word per stage (instruction) can be stored in another 128-bit register.

At the high-level, discrete event simulators have often been used to simulate combinational logic circuits. These clearly have the advantage of not executing sub-circuits for which none of the inputs have changed at the expense of maintaining an event wheel of trivial operations. When combined with efficient SIMD executions of combinational logic circuits, one could consider blocking combinational logic circuits into sub-circuits. Sub-circuit execution is triggered by discrete events, but the execution proper is an auto-tuned SIMD kernel. Thus, one must tune each discrete event circuit simulator for the underlying architecture.

9.2.5 Graph Traversal and Manipulation

The graph traversal and manipulation motif is extremely broad and lacks any substantial auto-tuning effort. There are three major concepts: graph attributes and characteristics, graph kernels, and graph representation.

Broadly speaking, graphs are defined by a set of vertices and a set of edges, each of which connect exactly two vertices. Vertices can encapsulate a wide range of data. Edges are often individually weighted and can be directed. Graphs can vary from the simplest linked lists and trees to the most complex DAGs.

Graph kernels can be broadly subdivided into graph traversal and graph manipulation. In graph traversal algorithms like breadth- or depth-first search, the graph is static and read-only. This greatly facilitates parallelization but does not guarantee efficient parallelization. Graph manipulation algorithms can change not only the values stored at the vertices but may also change the structure of the graph through the insertion of deletion of nodes. Such characteristics make parallelization far more challenging and far less efficient.

Typically, graph kernels don't reuse vertex data sufficiently to be computationally-limited. Moreover, poor data structures and placement in memory will result in latency-limited performance. Multithreaded architectures attempt to solve this problem in hardware, but we believe that choosing data representations cognizant of architecture will improve performance. We assume the kernel will be run enough times to amortize the overhead involved in changing storage representation. When selecting a data representation, one should integrate both graph and kernel characteristics into the decision making process.

For example, if common traversals only access one value of the record stored at each vertex, then the records should be stored as a structure-of-arrays to maximize spatial locality. Similarly, depth-first traversals should attempt to lay out data accordingly so that hardware prefetchers are effectively utilized [60]. Simply calling `malloc()` for each node will probably result in poor performance as typical node traversals will not exhibit spatial locality exceeding a few bytes. A similar suggestion could be made for breadth-first kernels. For tree traversals, one might consider grouping subtrees into dense blocks allocated together. One could tune to determine the optimal balance between block size, prefetcher effectiveness, and cache usage ($\frac{log(b)}{b}$), where b is the subtree block size.

We assume discovery of parallelism is explicit in the algorithm. However, efficient exploitation of such parallelism demands efficient implementations of parallel stacks, queues,

and barriers. We discuss these in Section 9.3.

9.3 Broadening Auto-tuning: Primitives

As discussed in Chapter 2, the Berkeley View set forth a pattern language for parallel programming. The lowest layers of this stack require efficiency layer programmers implement a number of parallel structures, collectives and routines. Rajesh Nishtala *et al.* investigated the auto-tuning of barriers and collectives on SMPs [20]. Typically, this involved exploration of various information dissemination trees. We believe that auto-tuning can and should be extended to parallel structures like shared stacks and queues.

When it comes to parallel queue management, one could implement it with locks or a dedicated thread for queue management. However, for many parallel algorithms, items are queued in bulk, threads synchronize, and then items are dequeued in bulk. As such, between synchronization points, any queue ordering is acceptable. Thus, threads could maintain their own queues and at the synchronization point, the private queues are interleaved or concatenated, perhaps by only changing queue bounds.

9.4 Broadening Auto-tuning: Motif Frameworks

We can continue with the existing auto-tuning strategy in which every time a new "key" kernel is written, we write a kernel-specific auto-tuner for it. Although an individual's ability to write an auto-tuner may improve over time, it will always remain a significant effort. We must take a step towards motif-wide auto-tuning. To that end, at the very least, we must define a motif-specific pattern language that describes the characteristics of any kernel with said motif. Then, we may build an auto-tuner that, based on the kernel description, may apply any motif-specific optimizations and parallelization strategies. Just as PhiPAC [16] is viewed as the progenitor of auto-tuned kernels, we believe SPIRAL project [97, 124] should be viewed as the progenitor of auto-tuned motifs.

At a high-level, we consider the construction of an auto-tuned structured grid framework. Chapter 5 introduced a structured grid pattern language. First, the grid is characterized by several parameters including node valence, topological dimensionality and periodicity, and data type. Second, computation within the structured grid motif is limited to stencil operations for which there are several styles of parallelism. One could specify the stencil, computation (a code snippet on local data), and style of parallelism. One could encapsulate these into a configurable family of data structures chosen at tuning- or runtime. An auto-tuner could then explore a preset list of optimizations depending on the specified parameters. In time, as motifs become cleanly defined, we believe SPIRAL-style auto-tuning frameworks could be created for each.

Ultimately, the kernels of many motifs can be viewed as static DAGs operating on double-precision floating-point inputs. At the high level, what allows us to perform certain optimizations, for example register blocking in sparse linear algebra? We believe each motif specifies the functionality of each node in the DAG and a series of rewrite rules that allow transformations of the DAG. For example, the nodes in SpMV are floating-point multiplication and addition. As such, floating-point commutativity allows the nodes in a

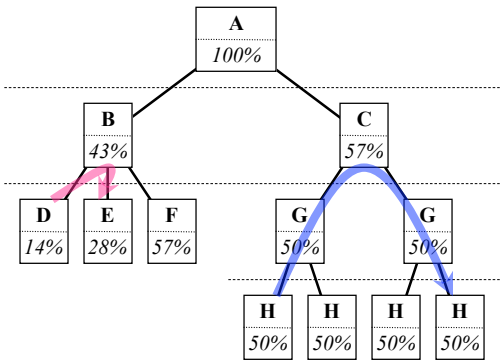


Figure 9.6: Composition of parallel motifs. Note: Rather than specifying the exact number of cores, the programmers of each level specify the fraction (%'s) of that level's computational resources. Motifs may only communicate through their parent frameworks.

row to be interchanged. Moreover, addition with zero doesn't change the result. As a result, additional nodes can be inserted so long as they only add zero to the running sum. We believe many motifs could be generalized to an arbitrary data type so long as the algebra on said data type doesn't change the underlying assumptions of the motif in question. That is, one could reuse the optimizations employed in auto-tuning SpMV on floating-point numbers for SpMV on arbitrary objects so long as "addition" and "multiplication" on these arbitrary objects behaves the same as floating-point addition and multiplication.

9.5 Composition of Motifs

Composition of multiple motifs into an application is a difficult, yet poorly defined problem. We believe it will become the preeminent problem once the challenges from the previous two sections have been a least partially solved. There are two principal problems with composition: efficient parallelization in hierarchical compositions, and interoperability between motifs.

For efficient and portable composition of motifs into frameworks or applications, we believe it is inappropriate and likely detrimental to hardcode the desired number of cores. Figure 9.6 shows the structure of an application written by as many as eight programmers. The programmer writing the parallel section "A" wants to execute "B" and "C" in parallel. However, the sections' computational requirements demand that "C" be given 33% more cores. Rather than hardcoding the number of cores, he simply specifies that "B" receives 43% of "A"s resources, but "C" gets the remaining 57%. A second programmer independently writing "B" is oblivious of the fact that "C" will be run concurrently. As such, she might naïvely attempt to use all of a computer's cores rather than those that have been allocated by its parent. Ultimately, the auto-tuning of the leaf kernels must be performed for all possible concurrencies.

The second major issue is communication between motifs. Some motifs, like sparse linear algebra, already interface well with the dense motif through the common dense vec-

tors. Given M motifs we don't have to define M^2 routines to exchange data, but rather define one universally portable representation for each motif. The programmer is then required to implement the necessary transformations from these common types. Moreover, within each motif, we believe that within each motif there are very few values of interest. Thus, the motif writers must provide the mechanisms to extract those values.

The arrows in Figure 9.6 show how communication and composition might mesh. Motifs "D" and "E" each provide an external representation and access to their respective internal conceptual variables. The writer of "B" must either implement a representation transformation or extract and insert the relevant variables. Similarly, for two instantiations of "H" to communicate, both "G" and "C" must transport data or access to data, but no transformations are required.

Consider codes where physics from two different domains must be composed. One could image a multi-phase SPMD implementation that alternates between different physics codes; temporal partitioning of hardware. However, in a multicore world, it may be simpler to spatially partition hardware, and allow domain experts code their part oblivious of the number of codes they will be allocated.

9.6 Conclusions

In this chapter, we discussed our observations and insights derived from auto-tuning LBMHD and SpMV. Clearly, the drive for in-core performance over the last decade has driven many kernels into the bandwidth-bound region. This shift has dramatically changed the nature of auto-tuning from instruction scheduling into a quest to maximize memory bandwidth and minimize memory traffic. The combination of multicore and shared caches actually complicates the quest for the latter as threads can contend for both capacity and associativity. All too often, the result is a flood of capacity and conflict misses. Auto-tuning can eliminate these by adapting the kernels to the architecture.

We may either accept our fate and tune memory-limited kernels or demand architectural or algorithmic changes. New architectures must deliver bandwidth that will scale with the number of cores. Moreover, shared caches must have both associativity and capacity that scales with the number of cores sharing them. When it comes to algorithms, we must discover algorithms that deliver a superior balance of computational complexity and arithmetic efficiency. Moreover, to deliver superior throughput on future architectures, the arithmetic efficiency must scale with the number of cores. That is, algorithms with constant arithmetic intensity will scale slowly with bandwidth.

We believe that the auto-tuning process can be applied kernel by kernel, motif by motif. In Section 9.2 we discuss how one might apply auto-tuning to kernels within a number of motifs. The only discussed motif for which in-core performance may be critical is the circuits motif due to the lack of instruction support for bit-level manipulation. To that end, we discussed both software and new instructions to facilitate efficient implementation of combinational logic kernels on multicore processors.

Ultimately, without a productive means of auto-tuning an arbitrary kernel, auto-tuning will be relegated to a few key kernels auto-tuned by experts. To that end, we discuss how motif-wide auto-tuning might be realized. This clearly requires each motif be well

defined including the creation of a motif-language to describe any kernel within that motif.

Finally, we put forth some ideas as to how multiple motifs could efficiently interoperate within an application. To that end, each level of the composition must be oblivious of the layers above or below. Moreover, it must be oblivious of any other subtree within the application. We believe that programmers should allocate fractions of a computer's computational capability rather than programming for a fixed number of cores.

Chapter 10

Conclusions

Architectures developed over the last decade have squandered the exponential promises of Moore's Law by expending transistors to increase instruction- and data-level parallelism — not to mention cache sizes — in an effort that only moderately increased single-thread performance. We embrace multicore as the better solution to deliver scalable performance in the future. In doing so, we must accept parallel programming as the new preeminent challenge in computing. The computational motifs as set forth in the Berkeley View allow domain expert programmers to encapsulate key numerical methods into libraries or frameworks so that productivity-level programmers may view them as black boxes. In this thesis, we take on the role of domain experts and apply automated tuning or *auto-tuning* as a technique to provide performance portability across both the breadth and evolution of multicore computers when running two specific kernels from these motifs.

Thesis Contributions

- In Chapter 4, we created the Roofline model, a visually-intuitive graphical representation of a machine's performance characteristics. For the computers used in this thesis, we created Roofline models that combined floating-point performance and main memory bandwidth. These Roofline models were useful in identifying performance bottlenecks, and identifying them as either inherent in the architecture, or an artifact of a program's implementation. Moreover, when used in the context of performance bounds, they were useful in noting when to stop tuning. We discussed how the Roofline model could be extended to other communication or computation metrics and even detailed how one could use performance counters to generate a runtime-specific Roofline model.

- We expanded auto-tuning to the structured grid motif. To that end, we selected one of the more challenging structured grid kernels — Lattice-Boltzmann Magnetohydrodynamics (LBMHD) and created an auto-tuner for it. Chapter 6 showed that the complexity of LBMHD's data structure demands blocking in the higher-dimensional lattice velocity space to mitigate TLB capacity misses.

- As all future computers will be built from multicore processors, we extended auto-tuning single-thread performance to auto-tuning multicore architectures. Note, this is

a fundamentally different approach from tuning single-thread performance and then running the resultant code on a multicore machine. Heuristics are an effective means of tackling the search space explosion problem for kernels with limited locality. We implemented this multicore auto-tuning approach for both the LBMHD and sparse matrix-vector multiplication (SpMV) kernels on six multicore architectures. We showed that despite the naïve implementation's exhibition of deceptively good parallel efficiency, multicore-aware auto-tuning can provide significant performance enhancements.

- We concretely address the aspects auto-tuners should focus on. Throughout Chapters 6 and 8, we showed how imperative it was for auto-tuners to explore data structure and even serial optimizations with the goal of minimizing memory traffic. Moreover, we show the importance of NUMA and cache optimizations in the context of multithreaded codes.

- Finally, throughout Chapters 6 and 8, we analyze the breadth of multicore architectures using these two auto-tuned kernels. Processing power has quickly outpaced bandwidth. As such many so-called compute-intensive kernels have become bandwidth-limited in the multicore era. Moreover, although multithreading solves many performance optimization problems with a single programming paradigm, it greatly exacerbates the requisite cache optimizations to avoid conflict and capacity misses.

Future Work

Although we made significant advances in the areas of auto-tuning LBMHD and SpMV on a wide range of multicore computers, much work remains within both the structured grid and sparse linear algebra motifs. Moreover, may other motifs have been untouched by auto-tuning endeavors. We summarize potential auto-tuning efforts:

- **Auto-tuning individual kernels:** Inspired and guided by our efforts in Chapters 6 and 8, we believe any well-defined kernel could be auto-tuned. In many respects, the benefits of auto-tuning are limited only by the flexibility allowed in the selection of data representation and algorithm.

- **Auto-tuning primitives:** The performance of collectives and primitives like barriers, shared queues, and locks is key to parallel performance for many strong scaling applications. That is, as the number of threads increase, the time each thread spends in isolation is inversely proportional to the total number of threads. However, the time each thread spends in collectives or primitives must increase. As such, for a quanta of work, there is a maximum number of threads that can efficiently process it. Improving collective performance pushes out the point of diminishing returns.

- **Auto-tuning motif frameworks:** Ultimately, we should consider the definition of a motif-description language for each motif. Any kernel within said motif could be precisely described by this motif language. A per-motif language will greatly facilitate per-motif auto-tuning as it simply constrains the auto-tuner.

- **Motif composition:** The ultimate potential of auto-tuning motifs will not be realized until productivity-level programmers can easily integrate multiple auto-tuned motifs into a single application or routine. We believe these programmers should be oblivious of the actual number of cores any routine will utilize but should have the ability to assign fractions of the number of cores to each sub-task. Composition is also complicated by the opaque and flexible nature of an auto-tuned motif's internal data representation. An agreed upon format for exporting or importing of data may help, but the burden ultimately falls upon the integrating programmer.

The transition to multicore hardware from single-core processors is widely viewed as the necessary step to continued exponential performance gains. The hardware community is firmly ensconced in the transition to on-chip software-managed parallelism, whether it be in the form of ever wider SIMD units, ever more superscalar cores, or ever more of the simple, lightweight cores seen in graphics or game processors today. Regardless of the solution, we are faced with a software crisis as few applications are written to take advantage of these features. Moreover, existing tools cannot automatically exploit these technologies using existing serial software. We believe that auto-tuning, in part, offers the solution to this software crisis by providing a productive and performance portable approach to building libraries and frameworks. The results in this thesis demonstrate that auto-tuning is an effective approach to all three forms of hardware parallelism. It can exploit superscalar multicore, it can exploit lightweight manycore, and it can utilize small ISA augmentations such as software prefetching and SIMD instructions without full compiler support. We have identified a roadmap for future breadth, depth, and evolution of auto-tuning work regimented around the structure of the Berkeley View "motifs."

Bibliography

[1] A. Agarwal. The Why, Where and How of Multicore. In *In Workshop on EDGE Computing Using New Commodity Architectures (EDGE)* *http://www.cs.unc.edu/~geom/EDGE/SLIDES/agarwal.pdf*, May 2006.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, USA, 1977.

[3] Software Optimization Guide for AMD Family 10h Processors. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf`, May 2007.

[4] AMD64 Architecture Programmers Manual Volume 2: System Programming. `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf`, September 2007.

[5] C. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comput.*, 13(4):923–947, 1992.

[6] 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. *(submitted to) Communications of the ACM*, May 2008.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[8] F. Bach and M. Jordan. Kernel independent component analysis. Technical Report UCB CSD-01-1166, University of California, Berkeley, 2001.

[9] D.A. Bader, V. Agarwal, and K. Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2007)*, Long Beach, CA, USA, 2007.

[10] D. Bailey. Little's Law and High Performance Computing. In *RNR Technical Report*, 1997.

[11] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202, 1997.

[12] J. Barnes and P. Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature*, 324(6096):446–449, December 1986.

[13] C. Batten, A. Joshi, J. Orcutt, A. Khilo, B. Moss, C. Holzwarth, M. Popovic, H. Li, H. Smith, J. Hoyt, F. Kartner, R. Ram, V. Stojanovic, and K. Asanovic. Building manycore processor-to-dram networks with monolithic silicon photonics. *High-Performance Interconnects, Symposium on*, 0:21–30, 2008.

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

[15] The Berkeley UPC Compiler. `http://upc.lbl.gov`, 2002.

[16] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC.

[17] D. Biskamp. *Magnetohydrodynamic Turbulence*. Cambridge University Press, 2003.

[18] David Thomas Blackston. *Pbody: a parallel n-body library*. PhD thesis, University of California, Berkeley, 2000. Chair-James Demmel.

[19] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented Operations for Sparse Matrix Computations on Vector Multiprocessors. Technical Report CMU-CS-93-173, Department of Computer Science, CMU, 1993.

[20] D. Bonachea, R. Nishtala, P. Hargrove, M. Welcome, and K. Yelick. Optimized Collectives for PGAS Languages with One-Sided Communication . Poster Session, Supercomputing, November 2006.

[21] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, Jul-Aug, 1999.

[22] A. Brandt. Multi-level adaptive solutions to boundary value problems. *Math. Comp.*, 31:333–390, 1977.

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

[24] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[25] Cactus homepage. `http://www.cactuscode.org`.

[26] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Machines. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[27] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-point Operations in Loops. *ACM Transactions on Programming Languages and Systems*, 16:1768–1810, 1994.

[28] Laura C. Carrington, Xiaofeng Gao, Nicole Wolter, Allan Snavely, and Roy L. Jr. Campbell. Performance Sensitivity Studies for Strategic Applications. In *DOD UGC '05: Proceedings of the 2005 Users Group Conference on 2005 Users Group Conference*, page 400, Washington, DC, USA, 2005. IEEE Computer Society.

[29] J. Carter, M. Soe, L. Oliker, Y. Tsuda, G. Vahala, L. Vahala, and A. Macnab. Magnetohydrodynamic Turbulence Simulations on the Earth Simulator Using the Lattice Boltzmann Method. In *Proc. SC2005: High performance computing, networking, and storage conference*, 2005.

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

[31] P. Colella. Defining Software Requirements for Scientific Computing (presentation), 2004.

[32] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[33] R. Courant, K. Friedrichs, and H. Lewy. On the Partial Difference Equations of Mathematical Physics. *IBM Journal of Research and Development*, 11:215–234, 1967.

[34] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the ACM National Conference*, 1969.

[35] K. Datta. private communication, 2005.

[36] 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) (to appear)*, 2008.

[37] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, J. Shalf D. Patterson, 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.

[38] PC2-3200/PC2-4200/PC2-5300/PC2-6400 DDR2 SDRAM Unbuffered DIMM Design Specification. http://www.jedec.org/download/search/4_20_13R15.pdf, January 2005.

[39] P.J. Dellar. Lattice Kinetic Schemes for Magnetohydrodynamics. *J. Comput. Phys.*, 79, 2002.

[40] James Demmel, Mark Frederick Hoemmen, Marghoob Mohiyuddin, and Katherine A. Yelick. Avoiding Communication in Computing Krylov Subspaces. Technical Report UCB/EECS-2007-123, EECS Department, University of California, Berkeley, Oct 2007.

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

[42] Jack J. Dongarra, Jack J. Dongarra, Jeremy Du Croz, Jeremy Du Croz, Sven Hammarling, Sven Hammarling, Richard J. Hanson, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[43] J. Doweck. Inside intel core microarchitecture. In *HotChips 18*, 2006.

[44] P. Dubey. A platform 2015 workload model: Recognition, mining and synthesis moves computers to the era of tera. Technical report, Intel Corporation, 2005.

[45] Iain S. Duff, Michele Marrone, and Carlo Vittoli. A set of Level 3 Basic Linear Algebra Subprograms for sparse matrices. *ACM Trans. Math. Softw*, 23:379–401, 1997.

[46] Energy Star Computer Specifications. http://www.energystar.gov/index.cfm?c=revisions.computer_spec.

[47] Technical Note FBDIMM Channel Utilization (Bandwidth and Power). http://download.micron.com/pdf/technotes/ddr2/tn4721.pdf, 2006.

[48] Solaris Memory Placement Optimization and Sun FireServers. http://www.sun.com/software/solaris/performance.jsp, March 2003.

[49] B. Flachs, S. Asano, S.H. Dhong, et al. A streaming processor unit for a cell processor. *ISSCC Dig. Tech. Papers*, pages 134–135, February 2005.

[50] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS05)*, 2005.

[51] M. Frigo and V. Strumpen. The Memory Behavior of Cache Oblivious Stencil Computations. *J. Supercomput.*, 39(2):93–112, 2007.

[52] 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.

[53] Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar Ramachandran, and Z W(l. Cache-oblivious algorithms. Extended abstract submitted for publication. In *In Proc. 40th Annual Symposium on Foundations of Computer Science*, pages 285–397. IEEE Computer Society Press, 1999.

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, USA, 1994.

[55] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. Using Machine Learning to Autotune a Stencil Code on a Multicore Architecture. In *(submitted to) Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, 2008.

[56] 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.

[57] J. Gebis, S. Williams, C. Kozyrakis, and D. Patterson. VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM. In *41st Design Automation Student Design Contenst*, 2004.

[58] P. P. Gelsinger. Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers. In *Proc. In International Solid State Circuits Conference, (ISSCC)*, San Francisco, CA, 2001.

[59] R. Geus and S. Röllin. Towards a Fast Parallel Sparse Matrix-Vector Multiplication. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Proceedings of the International Conference on Parallel Computing (ParCo)*, pages 308–315. Imperial College Press, 1999.

[60] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *In VLDB05*, pages 577–588. MIT, 2005.

[61] X. Gou, M. Liao, P. Peng, G. Wu, A. Ghuloum, and D. Carmean. Report on Sparse Matrix Performance Analysis. Intel report, Intel, United States, 2008.

[62] Green500 Supercomputer Site. `http://www.green500.org`.

[63] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[64] M. Gschwind. Chip Multiprocessing and the Cell Broadband Engine. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, New York, NY, USA, 2006.

[65] M. Gschwind, H. P. Hofstee, B. K. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[66] R. Heikes and D.A. Randall. Numerical integration of the shallow-water equations on a twisted icosahedral grid. Part I: basic design and results of tests. *Monthly Weather Review*, 123:1862, 1995.

[67] R. Heikes and D.A. Randall. Numerical integration of the shallow-water equations on a twisted icosahedral grid. Part II. A detailed description of the grid and analysis of numerical accuracy. *Monthly Weather Review*, 123:1862, 1995.

[68] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach; fourth edition*. Morgan Kaufmann, San Francisco, 2007.

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

[70] Intel64 and IA-32 Architectures Optimization Reference Manual. `http://support.intel.com/design/processor/manuals/248966.pdf`, May 2007.

[71] Intel 64 and IA-32 Architectures Software Developers Manual. `http://download.intel.com/design/processor/manuals/253665.pdf`, September 2008.

[72] E. J. Im, K. Yelick, and R. Vuduc. SPARSITY: Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[73] Intel 5000X Chipset Memory Controller Hub (MCH) Datasheet. `http://www.intel.com/design/chipsets/datashts/313070.htm`, September 2006.

[74] Intel Advanced Vector Extensions Programming Reference. `http://software.intel.com/sites/avx/`, August 2008.

[75] Intel SSE4 Programming Reference. `http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.htm`, July 2007.

[76] C. Jablonowski. Test of the Dynamics of two global Weather Prediction Models of the German Weather Service: The Held-Suarez Test (diploma thesis), September 1998.

[77] Ankit Jain. pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures. Technical Report (pending), MS Report, EECS Department, University of California, Berkeley, 2008.

[78] Eun jin Im and Katherine Yelick. Optimizing sparse matrix vector multiplication on smps. In *In Proc. of the 9th SIAM Conf. on Parallel Processing for Sci. Comp*, 1999.

[79] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[80] S. Kamil, C. Chan, K. Datta, S. Williams, J. Shalf, L. Oliker, and K. Yelick. In-Place Auto-tuning of Structured Grid Kernels. `http://www.cs.berkeley.edu/~skamil/stencilautotunerposter.ppt`, December 2008.

[81] 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.

[82] 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.

[83] Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *Conf. Computing Frontiers*, pages 87–96, 2008.

[84] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM. In *HotChips 12*, 2000.

[85] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[86] B. C. Lee, R. Vuduc, J. Demmel, and K. Yelick. Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-Vector Multiply. In *Proceedings of the International Conference on Parallel Processing*, Montreal, Canada, August 2004.

[87] W. W. Lee. Gyrokinetic particle simulation model. *J. Comp. Phys.*, 72, 1987.

[88] Z. Lin, T.S. Hahm, W.W. Lee, W.M. Tang, and R.B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, September 1998.

[89] LINPACK Benchmark. `http://www.netlib.org/benchmark/hpl`.

[90] A. Macnab, G. Vahala, L. Vahala, and P. Pavlo. Lattice Boltzmann Model for Dissipative MHD. In *Proc. 29th EPS Conference on Controlled Fusion and Plasma Physics*, volume 26B, Montreux, Switzerland, June 17-21, 2002.

[91] D.O. Martinez, S. Chen, and W.H. Matthaeus. Lattice Boltzmann magnetohydrodynamics. *Phys. Plasmas*, 1, 1994.

[92] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, USA, 2004.

[93] 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.

[94] J. Mellor-Crummey and J. Garvin. Optimizing Sparse Matrix Vector Multiply Using Unroll-and-Jam. In *Proc. LACSI Symposium*, Santa Fe, NM, USA, October 2002.

[95] T.C. Meyerowitz. *Single and Multi-CPU Performance Modeling for Embedded Systems*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, April 2008.

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

[97] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, and Manuela Veloso. SPIRAL: Automatic Implementation of Signal Processing Algorithms. In *High Performance Embedded Computing (HPEC)*, 2000.

[98] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[99] R. Nishtala, R. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication, and Computing*, March 2007.

[100] NVIDIA CUDA programming guide 1.1. `http://www.nvidia.com/object/cuda_develop.html`, November 2007.

[101] L. Oliker, A. Canning, J. Carter, J. Shalf, D. Skinner, S. Ethier, et al. Performance evaluation of the SX-6 vector architecture for scientific computations. *Concurrency and Computation; Practice and Experience*, 17:1:69–93, 2005.

[102] L. Oliker, J. Carter, M. Wehner, A. Canning, S. Ethier, et al. Leading Computational Methods on Scalar and Vector HEC Platforms. In *Proc. SC2005: High performance computing, networking, and storage conference*, Seattle, WA, 2005.

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

[104] B. Palmer and J. Nieplocha. Efficient algorithms for ghost cell updates on two classes of MPP architectures. In *Proc. PDCS International Conference on Parallel and Distributed Computing Systems*, volume 192, 2002.

[105] David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[106] D. Pham, S. Asano, M. Bollier, et al. The design and implementation of a first-generation cell processor. *ISSCC Dig. Tech. Papers*, pages 184–185, February 2005.

[107] S. Phillips. Victoriafalls: Scaling highly-threaded processor cores. In *HotChips 19*, 2007.

[108] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. Supercomputing*, 1999.

[109] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[110] K. Remington and R. Pozo. NIST Sparse BLAS: Users Guide. `gams.nist.gov/spblas`, 1996.

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

[112] C. Ronchi, R. Iacono, and P.S. Paolucci. Finite Difference Approximation to the Shallow Water Equations on a Quasi-uniform Spherical Grid. In *Lecture Notes in Computer Science*, volume 919, pages 741–747, Berlin / Heidelberg, 1995. Springer.

[113] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. *Graph Theory and Computing*, pages 183–217, 1973.

[114] T. Ruge. Does Your Software Scale ? Multi-GPU Scaling for Large Data Visualization. In *NVISION*, 2008.

[115] D. Scarpazza, O. Villa, and F. Petrini. High-speed String Searching Against Large Dictionaries on the Cell/B.E. Processor. In *IPDPS*, pages 1–12. IEEE, 2008.

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

[117] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, March 1981.

[118] A. Snavely, N. Wolter, and L. Carrington. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 149–156, Washington, DC, USA, 2001. IEEE Computer Society.

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

[120] J.A. Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Springer, New York, 2005.

[121] 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.

[122] The SPARC Architecture Manual Version 9. `http://www.sparc.org/standards/SPARCV9.pdf`, 1994.

[123] Synergistic processor unit instruction set architecture, October 2006.

[124] SPIRAL Project. `http://www.spiral.net/`.

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

[126] S. Succi. The Lattice Boltzmann Equation For fluids and beyond. *Oxford Science Publ.*, 2001.

[127] D. Sylvester and K. Keutzer. Microarchitectures for Systems on a Chip in Small Process Geometries. In *Proceedings of the IEEE*, pages 467–489, Apr. 2001.

[128] O. Takahashi, C. Adams, D. Ault, E. Behnen, O. Chiang, S.R. Cottier, P. Coulman, J. Culp, G. Gervais, M.S. Gray, Y. Itaka, C.J. Johnson, F. Kono, L. Maurice, K.W. McCullen, L. Nguyen, Y. Nishino, H. Noro, J. Pille, M. Riley, M. Shen, C. Takano, S. Tokito, T. Wagner, and H. Yoshihara. Migration of Cell Broadband Engine from 65nm SOI to 45nm SOI. In *ISSCC*, 2008.

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

[130] S. Toledo. Improving Memory-System Performance of Sparse Matrix-Vector Multiplication. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[131] Top500 Supercomputer Site. `http://www.top500.org`.

[132] Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P. Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G. Beausoleil, and Jung Ho Ahn. Corona: System implications of emerging nanophotonic technology. *SIGARCH Comput. Archit. News*, 36(3):153–164, 2008.

[133] B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005.

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

[135] 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.

[136] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, USA, June 2002.

[137] G. Wellein, T. Zeiser, S. Donath, and G. Hager. On the single processor performance of simple lattice Boltzmann kernels. *Computers and Fluids*, 35(910), 2005.

[138] 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.

[139] 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.

[140] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. SC2007: High performance computing, networking, and storage conference*, 2007.

[141] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potential of the Cell Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[142] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific Computing Kernels on the Cell Processor. *International Journal of Parallel Programming*, 35(3):263–298, 2007.

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

[144] 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.

[145] XDR Memory Architecture. `http://www.rambus.com/us/products/xdr/index.html`.