Auto-Tuning on the Macro Scale: High Level Algorithmic Auto-Tuning for Scientific Applications

by

Cy P. Chan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2012

Certified by ....
Alan Edelman
Professor of Applied Mathematics
Thesis Supervisor

Accepted by ....
Leslie Kolodziejski
Chairman, Department Committee on Graduate Students

Auto-Tuning on the Macro Scale: High Level Algorithmic Auto-Tuning for Scientific Applications

by

Cy P. Chan

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2012, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

In this thesis, we describe a new classification of auto-tuning methodologies spanning from low-level optimizations to high-level algorithmic tuning. This classification spectrum of auto-tuning methods encompasses the space of tuning parameters from low-level optimizations (such as block sizes, iteration ordering, vectorization, etc.) to high-level algorithmic choices (such as whether to use an iterative solver or a direct solver).

We present and analyze four novel auto-tuning systems that incorporate several techniques that fall along a spectrum from the low-level to the high-level: i) a multi-platform, auto-tuning parallel code generation framework for generalized stencil loops, ii) an auto-tunable algorithm for solving dense triangular systems, iii) an auto-tunable multigrid solver for sparse linear systems, and iv) tuned statistical regression techniques for fine-tuning wind forecasts and resource estimations to assist in the integration of wind resources into the electrical grid. We also include a project assessment report for a wind turbine installation for the City of Cambridge to highlight an area of application (wind prediction and resource assessment) where these computational auto-tuning techniques could prove useful in the future.

Thesis Supervisor: Alan Edelman
Title: Professor of Applied Mathematics

Acknowledgments

To Alan Edelman, thank you for your invaluable advice during my graduate career. I especially want to thank you for your continuous encouragement and enthusiasm for my interests as they varied through my time at MIT. I consider myself very lucky to have had the benefit of your support, your experience, and your guidance.

To Saman Amarasinghe, Steve Connors, Martin Rinard, John Shalf, and James Stalker, thank you for your time, feedback, advice, and mentorship. The combined lessons learned from all of you helped shape not only the research conducted under your supervision, but also the goals and directions of my future career.

To Jason Ansel, Shoaib Kamil, Plamen Koev, Lenny Oliker, Sam Williams, and Yee Lok Wong, thank you for helping make my time at MIT and LBL both enjoyable and fruitful. It has been an immense pleasure and honor working with such incredibly talented people, and I very much look forward to future collaboration.

To Akamai, Lincoln Laboratory, and the Martin family, thank you for your generous financial support. The work presented in this thesis was also partially supported by Microsoft (Award #024263), Intel (Award #024894), U.C. Discovery (Award #DIG07-10227), the ASCR Office in the DOE Office of Science (Award #DE-AC02-05CH11231), NSF (Award #CCF-0832997), and an award from the Gigascale Systems Research Center.

To my parents, Sriwan and Stanley, thank you for the love and values you have instilled in me. I would not be where I am today if not for the sacrifices you made to help me get here. To Ceida and Nick, thank you for your love and support. It's been so great having you both close by to share in life's journey and to see Ian and Ryan take their first steps. Finally, to my wife, Jess, this thesis is dedicated to you. Your presence in my life has brought me so much happiness. Thank you for your continual patience, love, and companionship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High performance parallel computing is well on its way to becoming a mainstream computational resource, breaking free from the confines of server rooms and data centers. Since clock frequencies have largely flattened out since the mid 2000s due to heat and power limitations (see Figure 1-1), the number of parallel cores being integrated into computational hardware have been increasing to maintain the exponential growth of computing power. One of the major challenges in transitioning computing platforms to highly parallel architectures will be developing systems to productively program these increasingly parallel machines to achieve high performance.

High performance parallel computers can achieve their peak performance only through effective utilization of their many computational units. Unfortunately, obtaining efficient utilization on these architectures usually requires an intensive optimization process. The conventional approach is to hand-tune individual algorithms for each specific hardware platform in use. Under this development paradigm, when software is migrated to a new hardware platform, it must be re-tuned to match the characteristics of the new architecture. Instead of writing a new hand-tuned implementation of code for every hardware target, a smarter approach to achieving high performance across multiple architectures is through the use of auto-tuning: searching for the best algorithmic parameters by empirically measuring the performance of the program while varying the parameters.

Auto-tuning is increasingly being recognized as an essential tool for extracting the

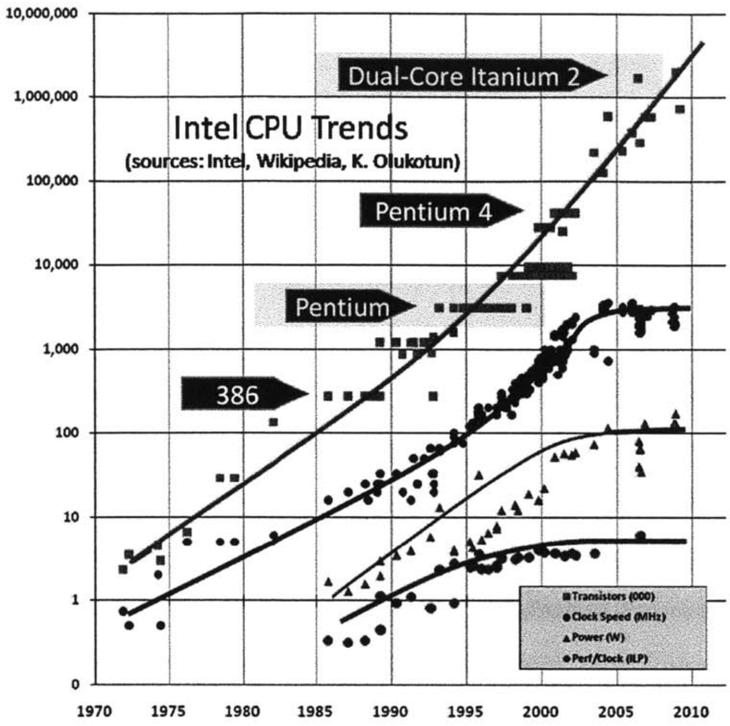


Figure 1-1: Intel CPU Trends. Transistor count (000), clock speed (MHz), power (W), and performance per clock (ILP) over time. As CPU clock frequencies have leveled off since the mid 2000s, parallelism has increased to maintain the exponential growth of computational power. Source: Herb Sutter. [38]

best performance from high performance parallel systems. Existing auto-tuning systems such as FFTW and ATLAS have already demonstrated the power of automatic parameter search to increase both performance and portability of an algorithm. In the circumstance that one has already chosen a particular algorithm to solve a problem, it is natural to focus on tweaking various low-level parameters of that algorithm (e.g. prefetch distance, block sizes, loop unrolling, etc.) to achieve the best performance. This is roughly what auto-tuners such as ATLAS, OSKI, and many others focus on.

Knowing which algorithm to use and when is just as important as tuning the parameters of each individual algorithm. This thesis pushes auto-tuning into newer, harder domains by exploring the idea of High-Level Algorithmic Tuning, which moves algorithm selection to the forefront as a primary tuning "variable" in the auto-tuning

#	First Author	Year	Nickname	Domain	Hardware Platform
1	Ali	2007	UHFFT	FFT	shared memory
2	Bondhugula	2008	PLuTO	loops	shared memory
3	Chen	2005	ECO	GEMM/stencil	shared memory
4	Choi	2010		SpMV	shared memory
5	Datta	2008		stencil	shared memory/Cell/GPU
6	Frigo	2005	FFTW	FFT	shared memory
7	Hartono	2009		LA/Stencil	shared memory
8	Im	2004	SPARSITY	SpMV	shared memory
9	Kuroda	2010		GMRES	cluster
10	Kurzak	2011		GEMM	GPU
11	Puschel	2005	SPIRAL	Signal Proc.	shared memory
12	Shin	2009		small GEMM	serial
13	Tiwari	2009	Active Harmony	general	shared memory
14	Vuduc	2005	OSKI	SpMV	shared memory
15	Whaley	2000	ATLAS	GEMM/LA	shared memory
16	Williams	2007		SpMV	shared memory/Cell
17	Williams	2011		LBMHD	cluster

Table 1.1: **Seventeen recent auto-tuning systems listed by first author and year of publication. Most auto-tuning systems are domain specific and explore performance on a particular type of hardware platform. Some abbreviations used: FFT = fast fourier transform, GEMM = general matrix multiply, SpMV = sparse matrix vector multiply, LA = linear algebra, GMRES = generalized minimum residual, LBMHD = lattice Boltzmann magnetohydrodynamics.**

search space. The techniques introduced here demonstrate the benefits of combining an auto-tuning approach with the flexibility of higher-level algorithmic search and selection.

Tables 1.1, 1.2, and 1.3 list a sample of seventeen recently developed auto-tuning systems along with their optimization search space and their tuning method. The optimization types have been roughly categorized from low-level to high-level. Under our categorization, low-level optimizations are those that are most closely tied to optimizing for specific hardware characteristics, such as loop unrolling and reordering to explore floating point pipeline depth, explicit prefetching to hide memory latency, or loop tiling to increase temporal locality in registers or cache. Medium-level optimizations are those that explore organizational variants of an algorithm, such as when to switch between recursive and serial versions of a code. High-level optimizations are those that explore completely different algorithms to solve a problem, such as using a direct algorithm versus an iterative algorithm to solve a linear system.

#	Low-level	Medium-level	High-level	Search method
1	access pattern, register block, instruction schedule, data structure	mathematical formulae		expert constrained brute-force for codelets; dynamic programming to assemble transforms from codelets
2	loop reorder, domain decomposition			cost function minimization
3	loop reorder, register and cache block, scalar replacement, array copy and pad, explicit prefetch			model-constrained brute-force search
4	data structure, block sizes			expert-constrained, model-assisted search
5	NUMA-aware, loop reorder, register and cache block, scalar replacement, array pad, explicit prefetch explicit vectorize local store DMA			expert constrained brute-force search
6	array copy and permute, operation order, explicit vectorize	radix choice, order of recursive decomposition	Direct, Cooley-Tukey, Rader, Bluestein	dynamic programming
7	array align and copy, loop reorder and tile, scalar replacement, explicit vectorize OpenMP parallelize			expert and heuristic constrained, brute-force, Nelder-Mead, simulated annealing
8	register and cache block, data storage, multiple vector,			model-assisted, expert-constrained brute-force search
9	loop unroll, data storage, explicit prefetch, communication model,			expert-constrained brute-force search

Table 1.2: **Optimizations explored by each of the auto-tuning systems listed in Table 1.1. Each optimization has been classified as either low-level, medium-level, or high-level. Low-level optimizations are those that are most closely tied to the hardware, such as loop unrolling, loop tiling, and explicit prefetching. Medium-level optimizations are those that explore organizational variants of an algorithm, such as when to switch between recursive and serial versions of a code. High-level optimizations are those that explore different algorithms to solve a problem, such as using a direct solver versus an iterative solver.**

Though the boundaries between these categories are far from black-and-white, this classification serves a useful purpose in demonstrating that most of the auto-tuning

#	Low-level	Medium-level	High-level	Search method
10	register and shmem block, domain decomposition, thread block shape, texture cache usage			expert and hardware-constrained brute-force search
11	loop unroll, code reorder, explicit code optimization	set of rules for formula expansion, rule application order		dynamic programming, evolutionary search
12	loop reorder and unroll explicit vectorization			expert-guided brute-force search
13	loop tiling, loop unroll-and-jam, scalar replacement, array copy			parameter space exploration via simplex transformation
14	register and cache block, data storage, multiple vector, variable block size, array splitting, array reordering			expert-constrained brute-force search
15	register and cache block, loop reorder and unroll, loop skew, array copy and reorder			probe hardware properties, expert-constrained brute-force search
16	register and cache block, TLB block, data storage, software pipelining, explicit prefetch, explicit vectorize, domain decomposition			heuristics for most optimizations brute-force search for blocking strategy and prefetch distance
17	loop unroll, instruction reorder, virtual vectors, explicit prefetch, explicit vectorize, domain decomposition, parallelization model			dual stage, expert-constrained brute-force search

Table 1.3: **Continuation of Table 1.2.**

systems in the literature currently focus on low-level optimizations of algorithms to make them run more efficiently. A few of them within the domain of signal processing explore breaking down problems in different ways by varying the ways in which problems are recursively decomposed, and one of them (FFTW) even has a family of different algorithmic types from which to choose.

The goal of this thesis is to push auto-tuning from the sphere of low-level optimizations "up the stack" into the higher-level optimization space, where one deliberately expresses different algorithmic ways to approach a problem and uses that additional flexibility to attain greater performance in different contexts. The work described applies this methodology to new application areas that may benefit from the additional performance that increased algorithmic flexibility may provide.

In some cases, tuning algorithmic choice could simply mean choosing the appropriate top-level technique during the initial function invocation; however, for many problems it is better to be able to utilize multiple techniques within a single function call or solve. For example, in the C++ Standard Template Library's sort routine, the algorithm switches from using the divide-and-conquer $O(n \log n)$ merge sort to $O(n^2)$ insertion sort once the working array size falls below a set cutoff. In multigrid, an analogous strategy might switch from recursive multigrid calls to a direct method such as Cholesky factorization and triangular solve once the problem size falls below a threshold.

In this thesis, several auto-tuning application domains are explored, each of which can be placed on the spectrum that ranges from low-level to high-level algorithmic tuning. In Chapter 2, we explore the low-end: auto-tuning blocking sizes and other low-level parameters for a generalized class of stencil computations on shared memory and GPU architectures. In Chapter 3, we examine the middle of the spectrum: the use of hybridized algorithmic variants (e.g. lazy vs. greedy, blocked vs. recursive) in conjunction with lower-level optimizations for dense triangular solves. In Chapter 4, we investigate the high level: the combined use of direct, iterative, and recursive algorithms in the implementation of a tuned multigrid linear solver.

1.1 Original contributions

This thesis makes the following original contributions:

- A novel auto-tuning framework for stencil computations:

  - First domain-specific, yet kernel independent stencil auto-tuner
  - Multi-path code generator that outputs optimized, parallel Fortran, C, and CUDA code from a serial Fortran stencil loop input

- An algorithmic auto-tuner for triangular solve
  - Combines algorithmic and low-level optimizations into a single tunable algorithm
  - Auto-tunes over a rich search space:
    * Provides choices between different computation and memory access variants (greedy and lazy)
    * Utilizes three individually tunable algorithmic variants: serial, blocked, and recursive
  - Hybridizes algorithms across different input sizes – the algorithm can switch algorithmic variants as the problem is decomposed into smaller chunks

- A novel high-level algorithmic auto-tuner for multigrid solvers
  - An auto-tuner that tunes multigrid *cycle shapes*
  - An accuracy metric that can be used to make comparisons between direct, iterative, and recursive algorithms in a multigrid setting for the purposes of auto-tuning
  - A dynamic programming methodology for efficiently building tuned multigrid algorithms that combine methods with varying levels of accuracy while providing that a final target accuracy is met
  - An accuracy-aware auto-tuner that produces families of optimized multigrid algorithms for different input sizes and accuracy targets
  - A demonstration that the performance of tuned multigrid algorithms is superior to more basic reference approaches and that their structure is dependent on platform architecture

Chapter 2

Auto-tuning generalized stencil codes

2.1 Introduction

This chapter presents a stencil auto-tuning framework that takes a stencil kernel as an input and produces tuned parallel implementations across a variety of hardware targets. We investigate the production of auto-tuned code by manipulating an abstract syntax tree representation of the input stencil kernel to implement various low-level optimizations. The framework automatically parallelizes the stencil computation in addition to auto-tuning the size and shape of the domain decomposition, cache blocks, register blocks, and varying array indexing strategies. This approach allows new stencil kernels to be automatically parallelized and tuned with low programmer effort. The code generator is capable of producing Fortran, C, and CUDA code for use on shared memory multi-core machines in addition to NVIDIA GPGPUs.

Petascale systems are becoming available to the computational science community with an increasing diversity of architectural models. These high-end systems, like all computing platforms, will increasingly rely on software-controlled on-chip parallelism to manage the trade-offs between performance, energy-efficiency and reliability [4]. This results in a daunting problem for performance-oriented programmers. Scientific progress will be substantially slowed without productive programming models and tools that allow programmers to efficiently utilize massive on-chip concurrency. The challenge is to create new programming models and tools that enable concise program expression while exposing fine-grained, explicit parallelism to the hardware across a diversity of chip multiprocessors (CMPs). Exotic programming models and domain-specific languages have been proposed to meet this challenge but run counter to the desire to preserve the enormous investment in existing software infrastructure.

This work presents a novel approach for addressing these conflicting requirements for stencil-based computations using a generalized auto-tuning framework. Our framework takes as input a straightforward sequential Fortran 95 stencil expression and automatically generates tuned parallel implementations in Fortran, C, or CUDA, thus providing performance portability across diverse architectures that range from conventional multicore processors to some of the latest graphics processing units (GPUs). This approach enables a viable migration path from existing applications to codes that provide scalable intra-socket parallelism across a diversity of emerging chip multiprocessors — preserving portability and allowing for productive code design and evolution.

Our work addresses the performance and portability of stencil (nearest-neighbor) computations, a class of algorithms at the heart of many calculations involving structured (rectangular) grids, including both implicit and explicit partial differential equation (PDE) solvers. These solvers constitute a large fraction of scientific applications in such diverse areas as heat transfer, climate science, electromagnetics, and engineering fluid dynamics. Previous efforts have successfully developed stencil-specific auto-tuners [10, 27, 35], which search over a set of optimizations and their parameters to minimize runtime and provide performance portability across a variety of architectural designs. Unfortunately, the stencil auto-tuning work to date has been limited to static kernel instantiations with pre-specified data structures. As such, they are not suitable for encapsulation into libraries usable by most real-world applications.

The focus of our study is to examine the potential of a generalized stencil auto-parallelization and auto-tuning framework, which can effectively optimize a broad range of stencil computations with varying data structures, dimensionalities, and topologies. Our novel methodology first builds an abstract representation from a straightforward user-provided Fortran stencil problem specification. We then use this intermediate representation to explore numerous auto-tuning transformations. Finally, our infrastructure is capable of generating a variety of shared-memory parallel (SMP) backend code instantiations, allowing optimized performance across a broad range of architectures.

To demonstrate the flexibility of our framework, we examine three stencil computations with a variety of different computational characteristics, arising from the 3D Laplacian, Divergence, and Gradient differential operators. Auto-parallelized and auto-tuned performance is then shown on several leading multicore platforms, including the AMD Barcelona, Sun Victoria Falls, NVIDIA GTX280, and the recently released Intel Nehalem. Results show that our generalized methodology can deliver significant performance gains of up to 22× speedup compared with the reference serial version, while allowing portability across diverse CMP technologies. Furthermore, while our framework only requires a few minutes of human effort to instrument each stencil code, the resulting code achieves performance comparable to previous hand-optimized code that required several months of tedious work to produce. Overall we show that such domain-specific auto-tuners hold enormous promise for architectural efficiency, programmer productivity, performance portability, and algorithmic adaptability on existing and future multicore systems.

This chapter describes joint work with Shoaib Kamil, Leonid Oliker, John Shalf, and Samuel Williams. My primary contribution to this original research was the design and implementation of the auto-parallelizing code generator for the various hardware targets and the auto-tuning strategy engines that determine the space of optimizations over which to search. I have included a complete description of the auto-tuning system here to give appropriate context for the work.

2.2 Related Work

Auto-tuning has been applied to a number of scientific kernels, most successfully to dense and sparse linear algebra. ATLAS [42] is a system that implements BLAS (basic linear algebra subroutines) and some LAPACK [29] kernels using compile-time auto-tuning. Similarly, OSKI [41] applies auto-tuning techniques to sparse linear algebra kernels, using a combination of compile-time and run-time tuning. FFTW [16] is a similar system for producing auto-tuned efficient signal processing kernels.

Unlike the systems above, SPIRAL [32] is a recent auto-tuning framework that implements a compiler for a specific class of kernels, producing high-performance tuned signal processing kernels. While previous auto-tuners relied on simple string manipulation, SPIRAL's designers defined an algebra suitable for describing a class of kernels, and built a tuning system for that class. Our work's goal is to create a similar system for stencil auto-tuning.

Optimizing stencil calculations have primarily focused on on tiling optimizations [35, 33, 30] that attempt to exploit locality by performing operations on cache-sized blocks of data before moving on to the next block. A study of stencil optimization [9] on (single-core) cache-based platforms found that tiling optimizations were primarily effective when the problem size exceeded the on-chip cache's ability to exploit temporal recurrences. Previous work in stencil auto-tuning for multicore and GPUs [10, 9] demonstrated the potential for greatly speeding up stencil kernels, but concentrated on a single kernel (the 7-pt Laplacian in 3D). Because the tuning system was hand-coded for that particular kernel, it cannot easily be ported to other stencils instantiations. In addition, it does not allow easy composition of different optimizations, or the integration of different search strategies such as hill-climbing.

Compiler optimizations for stencils concentrate on the polyhedral model [5] for improving performance by altering traversals to minimize (modeled) memory overhead. Auto-parallelization of stencil kernels is also possible using the polyhedral model [14]. Future work will combine/compare the polyhedral model with our auto-tuning system to explore the tradeoffs of simple models and comprehensive auto-tuning. Some planned optimizations (particularly those that alter the data structures of the grid) are currently not handled by the polyhedral model.

The goal of our framework is not just to automatically generate parallel stencil codes and tune the associated parallelization parameters, but also to tune the parallelization parameters in tandem with lower-level serial optimizations. Doing so will find the globally optimal combination for a given parallel machine.

Related work on optimizing stencils without the use of auto-tuning includes the ParAgent [31] tool, which uses static analysis to help minimize communication between compute nodes in a distributed memory system. The PLuTo system [14], strives to simultaneously optimize parameters for both data locality and parallelization, utilizing a polyhedral model and a unified cost function that incorporates aspects of both intra-tile locality and inter-tile communication. The recent Pochoir stencil compiler [39] utilizes a cache-oblivious decomposition based on earlier work [18] to automatically parallelize and optimize the locality of memory access for user-specified stencils. Our work differs from these methods primarily in our choice of optimization space and the fact that we leverage auto-tuning to find optimal parameters. Auto-tuning provides better results in cases where machine characteristics are difficult to model effectively using static analysis or where heuristically chosen parameters are sub-optimal.

This work presents a novel advancement in auto-tuning stencil kernels by building a framework that incorporates experience gained from building kernel-specific tuners. In particular, the framework has the applicability of a general domain-specific compiler like SPIRAL, while supporting multiple backend architectures. In addition, modularity allows the framework to, in the future, support data structure transformations and additional front- and backends using a simple plugin architecture. A preliminary overview of our methodology was presented at a recent Cray User's Group Workshop [26]; our current work extends this framework for a wider spectrum of optimizations and architectures including Victoria Falls and GPUs, as well as incorporating performance model expectations and analysis.

Stencil	Cache References (doubles)	Flops per Stencil	Compulsory Read Traffic	Writeback Traffic	Write Allocate Traffic	Capacity Miss Traffic	Naïve Arithmetic Intensity	Tuned Arithmetic Intensity	Expected Auto-tuning Benefit
Laplacian	8	8	8 Bytes	8 Bytes	8 Bytes	16 Bytes	0.20	0.33	1.66×
Divergence	7	8	24 Bytes	8 Bytes	8 Bytes	16 Bytes	0.14	0.20	1.40×
Gradient	9	6	8 Bytes	24 Bytes	24 Bytes	16 Bytes	0.08	0.11	1.28×

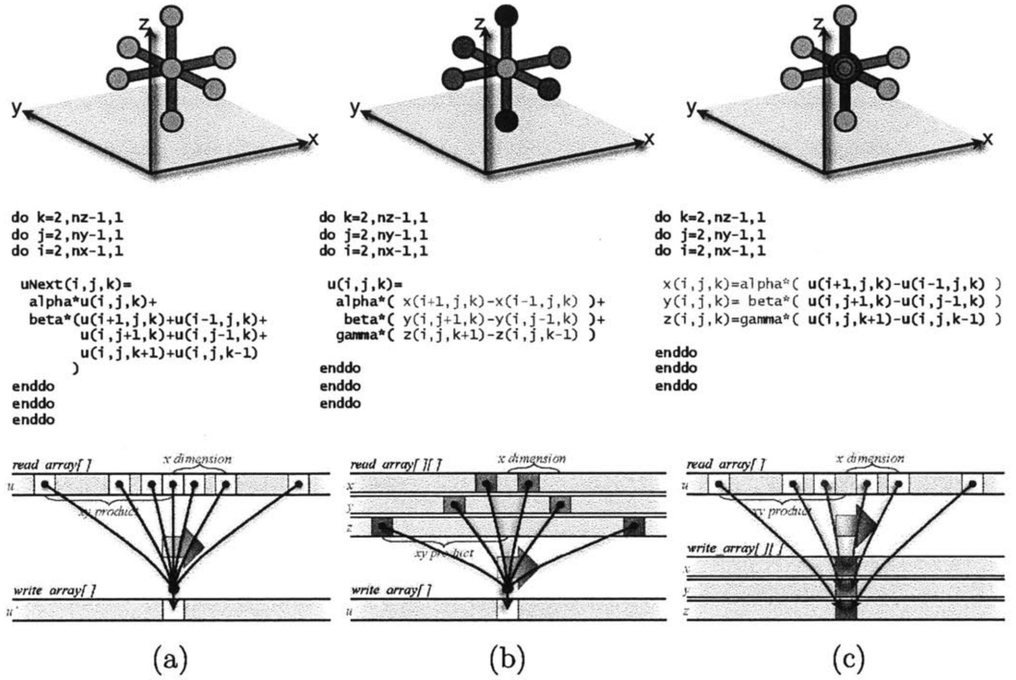
Table 2.1: **Average stencil characteristics. Arithmetic Intensity is defined as the Total Flops / Total DRAM bytes. Capacity misses represent a reasonable estimate for cache-based superscalar processors. Auto-tuning benefit is a reasonable estimate based on the improvement in arithmetic intensity assuming a memory bound kernel without conflict misses.**

2.3 Stencils & Architectures

Stencil computations on regular grids are at the core of a wide range of scientific codes. These applications are often implemented using iterative finite-difference techniques that sweep over a spatial grid, performing nearest neighbor computations called *stencils*. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors within a fixed distance in both time and space, locally solving a discretized version of the PDE for that data element. These operations are then used to build solvers that range from simple Jacobi iterations to complex multigrid and adaptive mesh refinement methods.

Stencil calculations perform repeated sweeps through data structures that are typically much larger than the data caches of modern microprocessors. As a result, these computations generally produce high memory traffic for relatively little computation, causing performance to be bound by memory throughput rather than floating-point operations. Reorganizing these stencil calculations to take full advantage of memory hierarchies has therefore been the subject of much investigation over the years.

Although these recent studies have successfully shown auto-tuning's ability to achieve performance portability across the breadth of existing multicore processors, they have been constrained to a single stencil instantiation, thus failing to provide broad applicability to general stencil kernels due to the immense effort required to hand-write auto-tuners. In this work, we rectify this limitation by evolving the auto-tuning methodology into a generalized code generation framework, allowing significant flexibility compared to previous approaches that use prepackaged sets of limited-



```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 uNext(i,j,k)=
  alpha*u(i,j,k)+
  beta*(u(i+1,j,k)+u(i-1,j,k)+
        u(i,j+1,k)+u(i,j-1,k)+
        u(i,j,k+1)+u(i,j,k-1)
       )
enddo
enddo
enddo
```

```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 u(i,j,k)=
  alpha*( x(i+1,j,k)-x(i-1,j,k) )+
   beta*( y(i,j+1,k)-y(i,j-1,k) )+
  gamma*( z(i,j,k+1)-z(i,j,k-1) )

enddo
enddo
enddo
```

```
do k=2,nz-1,1
do j=2,ny-1,1
do i=2,nx-1,1

 x(i,j,k)=alpha*( u(i+1,j,k)-u(i-1,j,k) )
 y(i,j,k)= beta*( u(i,j+1,k)-u(i,j-1,k) )
 z(i,j,k)=gamma*( u(i,j,k+1)-u(i,j,k-1) )

enddo
enddo
enddo
```

(a) (b) (c)

Figure 2-1: **(a) Laplacian, (b) Divergence, and (c) Gradient stencils. Top: 3D visualization of the nearest neighbor stencil operator. Middle: code as passed to the parser. Bottom: memory access pattern as the stencil sweeps from left to right. Note: the color represents cartesian component of the vector fields (scalar fields are gray).**

functionality library routines. Our approach complements existing compiler technology and accommodates new architecture-specific languages such as CUDA. Implementation of these kernels using existing languages and compilers destroys domain-specific knowledge. As such, compilers have difficulty proving that code transformations are safe, and even more difficulty transforming data layout in memory. The framework side-steps the complex task of analysis and presents a simple, uniform, and familiar interface for expressing stencil kernels as a conventional Fortran expression — while presenting a proof-of-concept for other potential classes of domain-specific generalized auto-tuners.

2.3.1 Benchmark Kernels

To show the broad utility of our framework, we select three conceptually easy-to-understand, yet deceptively difficult to optimize stencil kernels arising from the ap-

Core Architecture	AMD Barcelona	Intel Nehalem	Sun Niagara2	NVIDIA GT200 SM
Type	superscalar out of order	superscalar out of order	HW multithread dual issue	HW multithread SIMD
Clock (GHz)	2.30	2.66	1.16	1.3
DP GFlop/s	9.2	10.7	1.16	2.6
Local-Store	—	—	—	16KB**
L1 Data Cache	64KB	32KB	8KB	—
private L2 cache	512KB	256KB	—	—

System Architecture	Opteron 2356 (Barcelona)	Xeon X5550 (Gainestown)	UltraSparc T5140 (Victoria Falls)	GeForce GTX280
# Sockets	2	2	2	1
Cores per Socket	4	4	8	30
Threads per Socket‡	4	8	64	240
primary memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading	Multithreading with coalescing
shared L3 cache	2×2MB (shared by 4 cores)	2×8MB (shared by 4 cores)	2×4MB (shared by 8 cores)	—
DRAM Capacity	16GB	12GB	32GB	1GB (device) 4GB (host)

DRAM Pin Bandwidth (GB/s)	21.33	51.2	42.66(read)
			21.33(write)
				141 (device)
				4 (PCIe)

DP GFlop/s	73.6	85.3	18.7	78
DP Flop:Byte Ratio	3.45	1.66	0.29	0.55
Threading	Pthreads	Pthreads	Pthreads	CUDA 2.0
Compiler	gcc 4.1.2	gcc 4.3.2	gcc 4.2.0	nvcc 0.2.1221

Table 2.2: **Architectural summary of evaluated platforms. †Each of 2 thread groups may issue up to 1 instruction. ‡A *CUDA thread block* is considered 1 thread, and 8 may execute concurrently on a SM. **16 KB local-store shared by all concurrent *CUDA thread blocks* on the SM.**

plication of the finite difference method to the Laplacian ($u_{next} \leftarrow \nabla^2 u$), Divergence ($u \leftarrow \nabla \cdot \mathbf{F}$) and Gradient ($\mathbf{F} \leftarrow \nabla u$) differential operators. Details of these kernels are shown in Figure 2-1 and Table 2.1. All three operators are implemented using central-difference on a 3D rectahedral block-structured grid via Jacobi's method (out-of-place), and benchmarked on a 256^3 grid. The Laplacian operator uses a single input and a single output grid, while the Divergence operator utilizes multiple input grids (structure of arrays for Cartesian grids) and the Gradient operator uses multiple output grids. Note that although the code generator has no restrictions on data structure, for brevity, we only explore the use of structure of arrays for vector fields. As described below, these kernels have such low arithmetic intensity that they are expected to be memory-bandwidth bound, and thus deliver performance approximately equal to the product of their arithmetic intensity — defined as the ratio of arithmetic operations to memory traffic — and the system stream bandwidth.

Table 2.1 presents the characteristics of the three stencil operators and sets performance expectations. Like the 3C's cache model [24], we break memory traffic into compulsory read, write back, write allocate, and capacity misses. A naïve implementation will produce memory traffic equal to the sum of these components, and will therefore result in the shown arithmetic intensity ($\frac{totalflops}{totalbytes}$), ranging from 0.20–0.08. The auto-tuning effort explored in this work attempts to improve performance by eliminating capacity misses; thus it is possible to bound the resultant arithmetic intensity based only on compulsory read, write back, and write allocate memory traffic. For the three examined kernels, capacity misses account for dramatically different fractions of the total memory traffic. Thus, we can also bound the resultant potential performance boost from auto-tuning per kernel — 1.66×, 1.40×, and 1.28× for the Laplacian, Divergence, and Gradient respectively. Moreover, note that the kernel's auto-tuned arithmetic intensity will vary substantially from each other, ranging from 0.33–0.11. As such, performance is expected to vary proportionally, as predicted by the Roofline model [45].

2.3.2 Experimental Platforms

To evaluate our stencil auto-tuning framework, we examine a broad range of leading multicore designs: AMD Barcelona, Intel Nehalem, Sun Victoria Falls, and NVIDIA GTX 280. A summary of key architectural features of the evaluated systems appears in Table 2.2; space limitations restrict detailed descriptions of the systems. As all architectures have Flop:DRAM byte ratios significantly greater than the arithmetic intensities described in Section 2.3.1, we expect all architectures to be memory bound. Note that the sustained system power data was obtained using an in-line digital power meter while the node was under a full computational load, while chip and GPU card power is based on the maximum Thermal Design Power (TDP), extrapolated

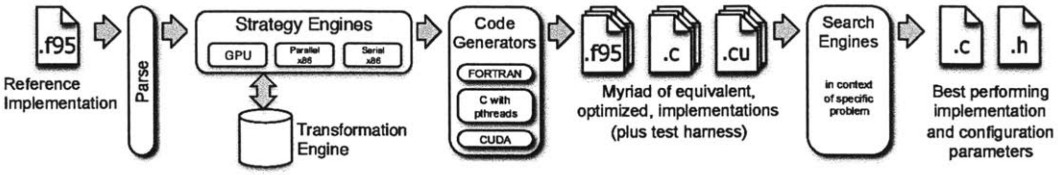


Figure 2-2: **Stencil auto-tuning framework flow. Readable domain-specific code is parsed into an abstract representation, transformations are applied, code is generated using specific target backends, and the optimal auto-tuned implementation is determined via search.**

from manufacturer datasheets. Although the node architectures are diverse, most accurately represent building-blocks of current and future ultra-scale supercomputing systems.

2.4 Auto-tuning Framework

Stencil applications use a wide variety of data structures in their implementations, representing grids of multiple dimensionalities and topologies. Furthermore, the details of the underlying stencil applications call for a myriad of numerical kernel operations. Thus, building a static auto-tuning library in the spirit of ATLAS [42] or OSKI [41] to implement the many different stencil kernels is infeasible.

This work presents a proof-of-concept of a generalized auto-tuning approach, which uses a domain-specific transformation and code-generation framework combined with a fully-automated search to replace stencil kernels with their optimized versions. The interaction with the application program begins with simple annotation of the loops targeted for optimization. The search system then extracts each designated loop and builds a test harness for that particular kernel instantiation; the test harness simply calls the kernel with random data populating the grids and measures performance. Next, the search system uses the transformation and generation framework to apply our suite of auto-tuning optimizations, running the test harness for each candidate implementation to determine its optimal performance. After the search is complete, the optimized implementation is built into an application-specific library that is called in place of the original. The overall flow through the auto-tuning system is shown in Figure 2-2.

2.4.1 Front-End Parsing

The front-end to the tranformation engine parses a description of the stencil in a domain-specific language. For simplicity, we use a subset of Fortran 95, since many stencil applications are already written in some flavor of Fortran. Due to the modularity of the transformation engine, a variety of front-end implementations are possible. The result of parsing in our preliminary implementation is an *Abstract Syntax Tree* (AST) representation of the stencil, on which subsequent transformations are performed.

2.4.2 Stencil Kernel Breadth

Currently, the auto-tuning system handles a specific class of stencil kernels of certain dimensionality and code structure. In particular, the auto-tuning system assumes a 2D or 3D rectahedral grid, and a stencil based on arithmetic operations and table lookups (array accesses). Future work will further extend the generality to allow grids of arbitrary dimensionality. Although this proof-of-concept framework does auto-tune serial kernels with imperfect loop nests, the parallel tuning relies on perfect nesting in order to determine legal domain decompositions and NUMA (non-uniform memory access) page mapping initialization — future framework extensions will incorporate imperfectly nested loops. Additionally, we currently treat boundary calculations as a separate stencil, although future versions may integrate stencils with overlapping traversals into a single stencil. Overall, our auto-tuning system can target and accelerate a large group of stencil kernels currently in use, while active research continues to extend the generality of the framework.

2.5 Optimization & Codegen

The heart of the auto-tuning framework is the transformation engine and the backend code generation for both serial and parallel implementations. The transformation engine is in many respects similar to a source-to-source translator, but it exploits domain-specific knowledge of the problem space to implement transformations that would otherwise be difficult to implement as a fully generalized loop optimization within a conventional compiler. Serial backend targets generate portable C and Fortran code, while parallel targets include pthreads C code designed to run on a variety of cache-based multicore processor nodes as well as CUDA versions specifically for the massively parallel NVIDIA GPGPUs.

Once the intermediate form is created from the front-end description, it is manipulated by the transformation engine across our spectrum of auto-tuned optimizations. The intermediate form and transformations are expressed in Common Lisp using the portable and lightweight ECL compiler [13], making it simple to interface with the parsing front-ends (written in Flex and YACC) and preserving portability across a wide variety of architectures. Potential future alternatives include implemention of affine scaling transformations or more complex AST representations, such as the one used by LLVM [7], or more sophisticated transformation engines such as the one provided by the Sketch [37] compiler.

Because optimizations are expressed as transformations on the AST, it is possible to combine them in ways that would otherwise be difficult using simple string substitution. For example, it is straightforward to apply register blocking either before or after cache-blocking the loop, allowing for a comprehensive exploration of optimization configurations. In the rest of this section, we discuss serial transformations and code generation; auto-parallelization and parallel-specific transformations and generators are explored in Section 2.5.2.

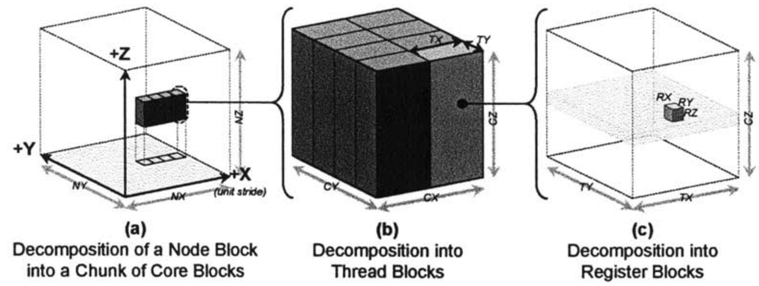
2.5.1 Serial Optimizations

Several common optimizations have been implemented in the framework as AST transformations, including loop unrolling/register blocking (to improve innermost loop efficiency), cache blocking (to expose temporal locality and increase cache reuse), and arithmetic simplification/constant propagation. These optimizations are implemented to take advantage of the specific domain of interest: Jacobi-like stencil kernels of arbitrary dimensionality. Future transformations will include those shown in previous work [10]: better utilization of SIMD instructions and common subexpression elimination (to improve arithmetic efficiency), cache bypass (to eliminate cache fills), and explicit software prefetching. Additionally, future work will support aggressive memory and code structure transformations.

We also note that, although the current set of optimizations may seem identical to existing compiler optimizations, future strategies such as memory structure transformations will be beyond the scope of compilers, since such optimizations are specific to stencil-based computations. Our restricted domain allows us to make certain assumptions about aliasing and dependencies. Additionally, the fact that our framework's transformations yield code that outperforms compiler-only optimized versions shows compiler algorithms cannot always prove that these (safe) optimizations are allowed. Thus, a domain-specific code generator run by the user has the freedom to implement transformations that a compiler may not.

2.5.2 Parallelization & Code Generation

Given the stencil transformation framework, we now present parallelization optimizations, as well as cache- and GPU-specific optimizations. The shared-memory parallel code generators leverage the serial code generation routines to produce the version run by each individual thread. Because the parallelization mechanisms are specific to each architecture, both the strategy engines and code generators must be tailored to the desired targets. For the cache-based systems (Intel, AMD, Sun) we use pthreads for lightweight parallelization; on the NVIDIA GPU, the only parallelization option is



(a)
Decomposition of a Node Block
into a Chunk of Core Blocks

(b)
Decomposition into
Thread Blocks

(c)
Decomposition into
Register Blocks

Figure 2-3: **Four-level problem decomposition: In (a), a *node block* (the full grid) is broken into smaller *chunks*. All *core blocks* in a chunk are processed by the same subset of threads. One core block from the chunk in (a) is magnified in (b). A single *thread block* from the core block in (b) is then magnified in (c). A thread block should exploit common resources among threads. Finally, the magnified thread block in (c) is decomposed into *register blocks*, which exploit data level parallelism.**

CUDA thread blocks that execute in a SPMD (single program multiple data) fashion.

Since the parallelization strategy influences code structure, the AST — which represents code run on each individual thread — must be modified to reflect the chosen parallelization strategy. The parallel code generators make the necessary modifications to the AST before passing it to the serial code generator.

Multicore-specific Optimizations and Code Generation

Following the effective blocking strategy presented in previous studies[10], we decompose the problem space into *core blocks*, as shown in Figure 2-3. The size of these core blocks can be tuned to avoid capacity misses in the last level cache. Each core block is further divided into *thread blocks* such that threads sharing a common cache can cooperate on a core block. Though our code generator is capable of using variable-sized thread blocks, we set the size of the thread blocks equal to the size of the core blocks to help reduce the size of the auto-tuning search space. The threads of a thread block are then assigned *chunks* of contiguous core blocks in a round robin fashion until the entire problem space has been accounted for. Finally each thread's stencil loop is *register blocked* to best utilize registers and functional units. The core block size, thread block size, chunk size, and register block size are all tunable by the framework.

The code generator creates a new set of loops for each thread to iterate over its assigned set of thread blocks. Register blocking is accomplished through strip mining and loop unrolling via the serial code generator.

NUMA-aware memory allocation is implemented by pinning threads to the hardware and taking advantage of first-touch page mapping policy during data initialization. The code generator analyzes the decomposition and has the appropriate processor touch the memory during initialization.

CUDA-specific Optimizations and Code Generation

CUDA programming is oriented around *CUDA thread blocks*, which differ from the thread blocks used in the previous section. CUDA thread blocks are vector elements mapped to the scalar cores (lanes) of a streaming multiprocessor. The vector conceptualization facilitates debugging of performance issues on GPUs. Moreover, CUDA thread blocks are analogous to threads running SIMD code on superscalar processors. Thus, parallelization on the GTX280 is a straightforward SPMD domain decomposition among CUDA thread blocks; within each CUDA thread block, work is parallelized in a SIMD manner.

To effectively exploit cache-based systems, code optimizations attempt to employ unit-stride memory access patterns and maintain small cache working sets through cache blocking — thereby leveraging spatial and temporal locality. In contrast, the GPGPU model forces programmers to write a program for each *CUDA thread.* Thus, spatial locality may only be achieved by ensuring that memory accesses of adjacent threads (in a CUDA thread block) reference contiguous segments of memory to exploit hardware coalescing. Consequently, our GPU implementation ensures spatial locality for each stencil point by tasking adjacent threads of a CUDA thread block to perform stencil operations on adjacent grid locations. Some performance will be lost as not all coalesced memory references are aligned to 128-byte boundaries.

The CUDA code generator is capable of exploring the myriad different ways of dividing the problem among CUDA thread blocks, as well as tuning both the number of threads in a CUDA thread block and the access pattern of the threads. For example, in a single time step, a CUDA thread block of 256 CUDA threads may access a tile of 32 x 4 x 2 contiguous data elements; the thread block would then iterate this tile shape over its assigned core block. In many ways, this exploration is analogous to register blocking within each core block on cache-based architectures.

Our code generator currently only supports the use of global "device" memory, and so does not take advantage of the low-latency local-store style "shared" memory present on the GPU. As such, the generated code does not take advantage of the temporal locality of memory accesses that the use of GPU shared memory provides. Future work will incorporate support for exploitation of CUDA shared memory.

2.6 Auto-Tuning Strategy Engine

In this section, we describe how the auto-tuner searches the enormous parameter space of serial and parallel optimizations described in previous sections. Because the combined parameter space of the preceding optimizations is so large, it is clearly infeasible to try all possible strategies. In order to reduce the number of code instantiations the auto-tuner must compile and evaluate, we used strategy engines to enumerate an appropriate subset of the parameter space for each platform.

The strategy engines enumerate only those parameter combinations (strategies) in the subregion of the full search space that best utilize the underlying architecture. For example, cache blocking in the unit stride dimension could be practical on the Victoria Falls architecture, while on Barcelona or Nehalem, the presence of hardware prefetchers makes such a transformation non-beneficial [9].

Further, the strategy engines keep track of parameter interactions to ensure that only legal strategies are enumerated. For example, since the parallel decomposition changes the size and shape of the data block assigned to each thread, the space of legal serial optimization parameters dependends on the values of the parallel parameters. The strategy engines ensure all such constraints (in addition to other hardware restrictions such as maximum number of threads per processor) are satisfied during

	Optimization		Parameter Tuning Range by Architecture		
Category	Parameter	Name	Barcelona/Nehalem	Victoria Falls	GTX280
Data Allocation	NUMA Aware		✓	✓	N/A
Domain Decomposition	Core Block Size	*CX*	NX	{8...NX}	$\{16^{\dagger}..NX\}$
		CY	{8...NY}	{8...NY}	$\{16^{\dagger}..NY\}$
		CZ	{128...NZ}	{128...NZ}	$\{16^{\dagger}..NZ\}$
	Thread Block Size	*TX*	CX	CX	$\{1..\frac{CX}{16}\}^{\ddagger}$
		TY	CY	CY	$\{\frac{CY}{16}..CY\}^{\ddagger}$
		TZ	CZ	CZ	$\{\frac{CZ}{16}..CZ\}^{\ddagger}$
	Chunk Size		$\{1...\frac{NX \times NY \times NZ}{CX \times CY \times CZ \times NThreads}\}$		N/A
Low Level	Array Indexing		✓	✓	✓
	Register Block Size	*RX*	{1...8}	{1...8}	1
		RY	{1...2}	{1...2}	1*
		RZ	{1...2}	{1...2}	1*

Table 2.3: **Attempted optimizations and the associated parameter spaces explored by the auto-tuner for a 256^3 stencil problem ($NX, NY, NZ = 256$). All numbers are in terms of doubles. † Actual values for minimum core block dimensions for GTX280 dependent on problem size. ‡ Thread block size constrained by a maximum of 256 threads in a CUDA thread block with at least 16 threads coalescing memory accesses in the unit-stride dimension. *The CUDA code generator is capable of register blocking the Y and Z dimensions, but due to a confirmed bug in the NVIDIA nvcc compiler, register blocking was not explored in our auto-tuned results.**

enumeration.

For each parameter combination enumerated by the strategy engine, the auto-tuner's search engine then directs the parallel and serial code generator components to produce the code instantiation corresponding to that strategy. The auto-tuner runs each instantiation and records the time taken on the target machine. After all enumerated strategies have been timed, the fastest parameter combination is reported to the user, who can then link the optimized version of the stencil into their existing code.

Table 2.3 shows the attempted optimizations and the associated parameter subspace explored by the strategy engines corresponding to each of our tested platforms. While the search engine currently does a comprehensive search over the parameter subspace dictated by the strategy engine, future work will include more intelligent search mechanisms such as hill-climbing or machine learning techniques [19], where the search engine can use timing feedback to dynamically direct the search.

2.7 Performance Evaluation

In this section, we examine the performance quality and expectations of our auto-parallelizing and auto-tuning framework across the four evaluated architectural platforms. The impact of our framework on each of the three kernels is compared in Figure 2-4, showing performance of: the original serial kernel (gray), auto-parallelization (blue), auto-parallelization with NUMA-aware initialization (purple), and auto-tuning (red). The GTX280 reference performance (blue) is based on a straightforward implementation that maximizes CUDA thread parallelism. We do not consider the impact of host transfer overhead; previous work [10] examined this potentially significant bottleneck in detail. Overall, results are ordered such that threads first exploit multithreading within a core, then multiple cores on a socket, and finally multiple sockets. Thus, on Nehalem, the two thread case represents one fully-packed core; similarly, the GTX280 requires at least 30 *CUDA thread blocks* to utilize the 30 cores (streaming multiprocessors).

2.7.1 Auto-Parallelization Performance

The auto-parallelization scheme specifies a straightforward domain decomposition over threads in the least unit-stride dimension, with no core, thread, or register blocking. To examine the quality of the framework's auto-parallelization capabilities, we compare performance with a parallelized version using OpenMP [15], which ensures proper NUMA memory decomposition via first-touch pinning policy. Results, shown as yellow diamonds in Figure 2-4, show that performance is well correlated with our framework's NUMA-aware auto-parallelization. Furthermore, our approach slightly improves Barcelona's performance, while Nehalem and Victoria Falls see up to a 17% and 25% speedup (respectively) compared to the OpenMP version, indicating the effectiveness of our auto-parallelization methodology even before auto-tuning.

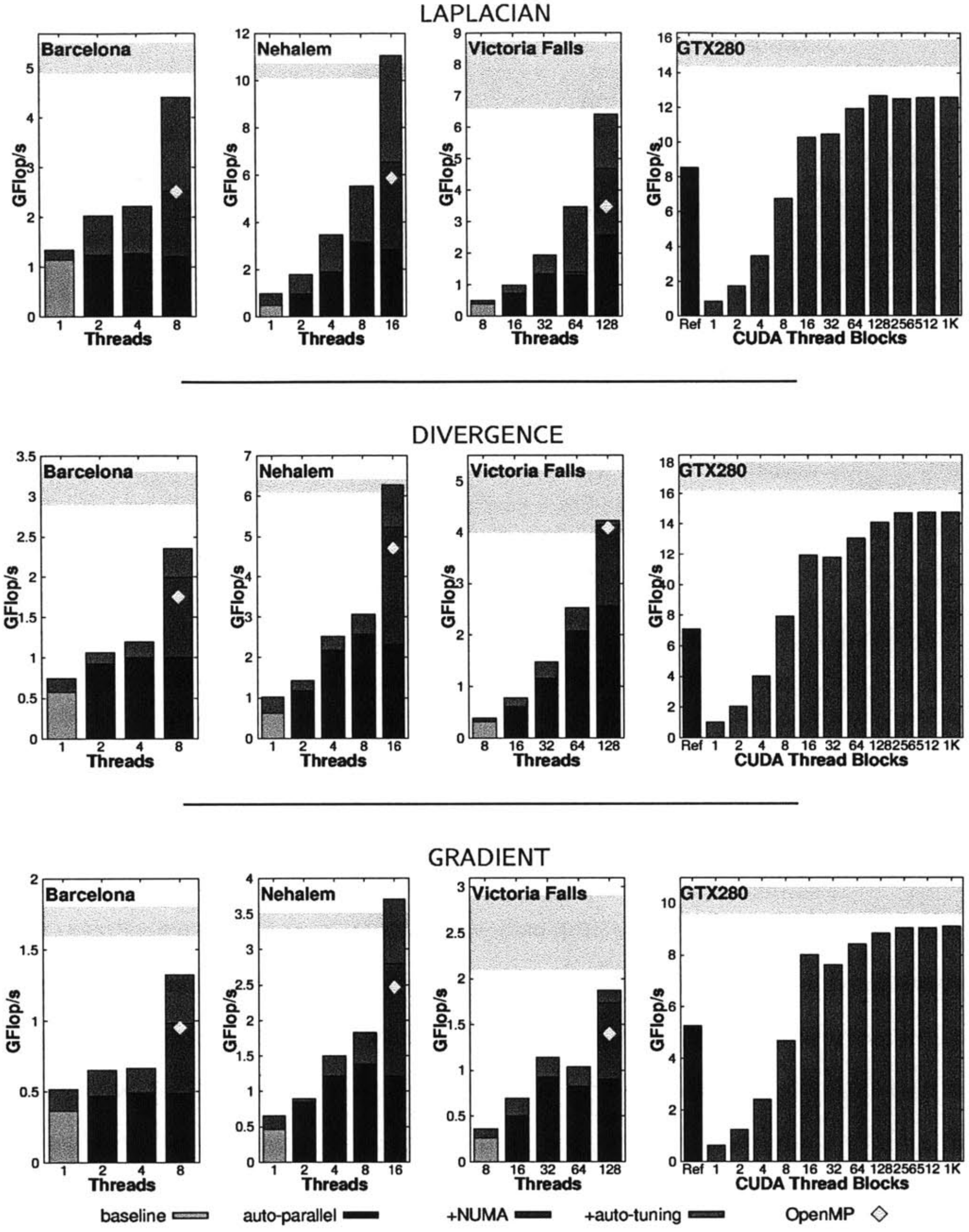


Figure 2-4: **Laplacian (top row), Divergence (middle row), and Gradient (bottom row) performance as a function of auto-parallelization and auto-tuning — on the four evaluated platforms. Note: the green region marks performance extrapolated from Stream bandwidth. For comparison, the yellow diamond shows performance achieved using the original stencil kernel with OpenMP pragmas and NUMA-aware initialization.**

2.7.2 Performance Expectations

When tuning any application, it is important to know when you have reached the architectural peak performance, and have little to gain from continued optimization. We make use of a simple empirical performance model to establish this point of diminishing returns and use it to evaluate how close our automated approach can come to machine limits. We now examine achieved performance in the context of this simple model based on the hardware's characteristics. Assuming all kernels are memory bound and do not suffer from an abundance of capacity misses, we approximate the performance bound as the product of streaming bandwidth and each stencil's arithmetic intensity (0.33, 0.20 and 0.11 — as shown in Table 2.1). Using an optimized version of the Stream benchmark [8], which we modified to reflect the number of read and write streams for each kernel, we obtain expected peak performance based on memory bandwidth for the CPUs. For the GPU, we use two versions of Stream: one that consists of exclusively read traffic, and another that is half read and half write.

Our model's expected performance range is represented as a green line (for the CPUs) and a green region (for the GPUs) in Figure 2-4. For Barcelona and Nehalem, our optimized kernels obtain performance essentially equivalent to peak memory bandwidth. For Victoria Falls, the obtained bandwidth is around 20% less than peak for each of the kernels, because our framework does not currently implement software prefetching and array padding, which are critical for performance on this architecture. Finally, the GTX280 results were also below our performance model bound, likely due to no array padding [10]. Overall, our fully tuned performance closely matches our model's expectations, while highlighting areas which could benefit from additional optimizations.

2.7.3 Performance Portability

The auto-tuning framework takes a serial specification of the stencil kernel and achieves a substantial performance improvement, due to both auto-parallelization and auto-tuning. Overall, Barcelona and Nehalem see between 1.7× to 4× improve-

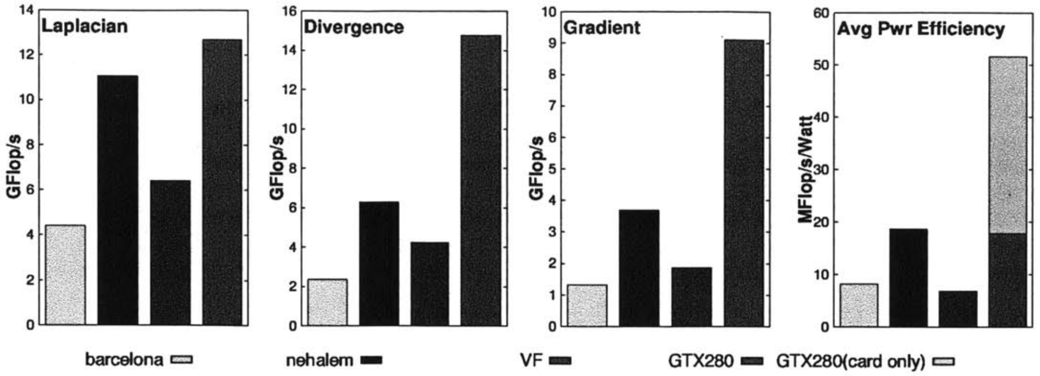


Figure 2-5: **Peak performance and power efficiency after auto-tuning and parallelization. GTX280 power efficiency is shown based on system power as well as the card alone.**

ment for both the one and two socket cases over the conventional parallelized case, and up to 10 times improvement over the serial code. The results also show that auto-tuning is essential on Victoria Falls, enabling much better scalability and increasing performance by 2.5× and 1.4× on 64 and 128 threads respectively in comparison to the conventional parallelized case, but a full 22× improvement over an unparallelized example. Finally, auto-tuning on the GTX280 boosted performance by 1.5× to 2× across the full range of kernels — a substantial improvement over the baseline CUDA code, which is implicitly parallel. This clearly demonstrates the performance portability of this framework across the sample kernels.

Overall, we achieve substantial performance improvements across a diversity of architectures – from GPU's to multi-socket multicore x86 systems. The auto-tuner is able to achieve results that are extremely close to the architectural peak performance of the system, which is limited ultimately by memory bandwidth. This level of performance portability using a common specification of kernel requirements is unprecedented for stencil codes, and speaks to the robustness of the generalized framework.

2.7.4 Programmer Productivity Benefits

We now compare our framework's performance in the context of programming productivity. Our previous work [10] presented the results of Laplacian kernel optimization using a hand-written auto-tuning code generator, which required months of Perl script implementation, and was inherently limited to a single kernel instantiation. In contrast, utilizing our framework across a broad range of possible stencils only requires a few minutes to annotate a given kernel region, and pass it through our auto-parallelization and auto-tuning infrastructure, thus tremendously improving productivity as well as kernel extensibility.

Currently our framework does not implement several hand-tuned optimizations [10], including SIMDization, padding, or the employment of cache bypass (*movntpd*). However, comparing results over the same set of optimizations, we find that our framework attains excellent performance that is comparable to the hand-written version. We obtain near identical results on the Barcelona and even higher results on the Victoria Falls platform (6 GFlop/s versus 5.3 GFlop/s). A significant disparity is seen on the GTX280, where previous hand-tuned Laplacian results attained 36 GFlop/s, compared with our framework's 13 GFlop/s. For the CUDA implementations, our automated version only utilizes optimizations and code structures applicable to general stencils, while the hand-tuned version explicitly discovered and exploited the temporal locality specific to the Laplacian kernel — thus maximizing performance, but limiting the method's applicability. Future work will continue incorporating additional optimization schemes into our automated framework.

2.7.5 Architectural Comparison

Figure 2-5 shows a comparative summary of the fully tuned performance on each architecture. The GTX280 consistently attains the highest performance, due to its massive parallelism at high clock rates, but transfer times from system DRAM to board memory through the PCI Express bus are not included and could significantly impact performance [10]. The recently-released Intel Nehalem system offers a substantial improvement over the previous generation Intel Clovertown by eliminating the front-side bus in favor of on-chip memory controllers. The Nehalem obtains the best overall performance of the cache-based systems, due to the combination of high memory bandwidth per socket and hardware multithreading to fully utilize the available bandwidth. Additionally, Victoria Falls obtains high performance, especially given its low clock speed, thanks to massive parallelism combined with an aggregate memory bandwidth of 64 GB/s.

Power efficiency, measured in (average stencil) MFlop/s/Watt, is also shown in Figure 2-5. For the GTX280 measurements we show the power efficiencies both with (red) and without the host system (pink). The GTX280 shows impressive gains over the cache-based architecture if considered as a standalone device, but if system power is included, the GTX280's advantage is diminished and the Nehalem becomes the most power efficient architecture evaluated in this study.

2.8 Summary and Conclusions

Performance programmers are faced with the enormous challenge of productively designing applications that effectively leverage the computational resources of leading multicore designs, while allowing for performance portability across the myriad of current and future CMP instantiations. In this work, we introduce a fully generalized framework for stencil auto-tuning that takes the first steps towards making complex chip multiprocessors and GPUs accessible to domain-scientists, in a productive and performance portable fashion — demonstrating up to 22× speedup compared with the default serial version.

Overall we make a number of important contributions that include the (*i*) introduction of a high performance, multi-target framework for auto-parallelizing and auto-tuning multidimensional stencil loops; (*ii*) presentation of a novel tool chain based on an abstract syntax tree (AST) for processing, transforming, and generating stencil loops; (*iii*) description of an automated parallelization process for targeting multidimensional stencil codes on both cache-based multicore architectures as well as GPGPUs; (*iv*) achievement of excellent performance on our evaluation suite using three important stencil access patterns; (*v*) utilization of simple performance model that effectively predicts the expected performance range for a given kernel and architecture; and (*vi*) demonstration that automated frameworks such as these can enable greater programmer productivity by reducing the need for individual, hand-coded auto-tuners.

The modular architecture of our framework enables it to be extended through the development of additional parser, strategy engine, and code generator modules. Future work will concentrate on extending the scope of optimizations (see Section 2.7.4), including cache bypass, padding, and prefetching. Additionally, we plan to extend the CUDA backend for a general local store implementation, thus leveraging temporal locality to improve performance and allowing extensibility to other local-store architectures such as the Cell processor. We also plan to expand our framework to broaden the range of allowable stencil computation classes (see Section 2.4.2), including in-place and multigrid methods. Finally, we plan to demonstrate our framework's applicability by investigating its impact on large-scale scientific applications, including a forthcoming optimization study of an icosahedral atmospheric climate simulation [23, 20].

Chapter 3

Auto-tuning triangular solve

Moving up the scale from low-level to medium-level optimizations, this chapter examines auto-tuning solvers for triangular, linear system of equations with multiple right-hand sides. This problem appears frequently on its own and as a subroutine in various dense linear algebra functions, such as dense (non-triangular) linear solves or the Cholesky and LU matrix factorizations, and is thus an important computational kernel. The tunable algorithm presented here explores the flexibility inherent in the structure of the algorithm, varying the ways in which the input matrix is decomposed and iterated over during the computation.

3.1 Description of Triangular Solve

A triangular solve consists of solving the equation $AX = B$ for the matrix X given triangular matrix A and matrix B. Common variations include A being lower or upper triangular, A being unit or non-unit on the diagonal, or solving the alternate equation $XA = B$. Though this chapter only examines the $AX = B$, non-unit, lower and upper triangular versions, these variations should have similar computational characteristics on most platforms, and the optimizations explored in this chapter can be applied in all of these cases.

A naïve implementation of lower-triangular solve is given in Algorithm 3.1. This is an "in-place" version of the algorithm: the array B is an input/output argument.

Algorithm 3.1 TRISOLVE-NAIVE(A, B)

```
for $i = 1 : M$ do
    for $j = 1 : N$ do
        $B(i,j) = B(i,j)/A(i,i)$
    end for
    for $j = 1 : N$ do
        for $k = i + 1 : M$ do
            $B(k,j) = B(k,j) - A(k,i) * B(i,j)$
        end for
    end for
end for
```

On entry, the array B contains the values of the matrix B. On exit, the array B contains the values of the solution matrix X.

The algorithm requires $O(n^3)$ floating-point operations over $O(n^2)$ data, yielding a high arithmetic intensity of computation. Therefore, the algorithm can be expected to have good performance if the utilization of the computational units and memory bandwidth can be effectively managed through optimization. Of particular consequence are the reuse of data in cache (temporal locality) and the order in which data are accessed (spatial locality) so that any bottlenecks due to the memory subsystem are minimized. Also important is the parallelization strategy, which affects the algorithm's ability to scale effectively on machines with many computational units. There are many optimizations (discussed in the next section), both algorithmic and low-level, that attempt to increase locality and parallelism in different ways to increase performance.

3.1.1 Algorithmic Variants and Optimizations

The following sections describe the set of optimizations explored by the auto-tunable triangular solve algorithm presented in this chapter. They include recursive, blocked, and non-blocked algorithmic variants; greedy versus lazy computation and memory access patterns; low-level array and loop reordering; and finally, parallelization strategy.

Let the input to the problem be an $M \times M$ triangular matrix A and a $M \times N$ right-hand side matrix B. The algorithm solves for the matrix X, such that $AX = B$. The problem formulation is equivalent to N independent $M \times M$ times $M \times 1$ matrix-vector triangular solves. Let NB denote the blocking dimension for any blocked algorithm.

Blocked and recursive versions

Blocking the triangular solve involves rearranging the computation to solve for multiple rows in groups. By breaking up the problem into multiple smaller triangular solves, this improves the temporal locality of the computation by reducing the reuse distance between memory accesses. Blocking also allows the expression of parts of the computation in terms of dense matrix multiplications, which in turn enables leveraging the increased efficiency of an optimized matrix multiply (e.g. Level 3 BLAS).

Algorithm 3.2 TRISOLVE-BLOCKED(A, B)

```
1: for $ib = 1 : NB : M$ do
2:     Let $A_1$ be the $NB \times NB$ diagonal block of $A$ starting at index $ib$
3:     Let $B_1$ be the $NB \times N$ row block of $B$ starting at row $ib$
4:     TRISOLVE($A_1, B_1$) // call non-blocked code on row block
5:     Let $K \leftarrow (M - (ib + NB))$
6:     Let $A_2$ be the $K \times NB$ panel below $A_1$
7:     Let $B_2$ be the $K \times N$ trailing sub-matrix below $B_1$
8:     $B_2 \leftarrow B_2 - A_2 B_1$ // matrix multiply to update trailing sub-matrix
9: end for
```

Algorithm 3.2 shows a greedy blocked version of triangular solve. In this algorithm, A_1 and B_1 correspond to the medium gray regions in Figure 3-1, while A_2 and B_2 correspond to the light gray regions. Line 4 computes the solution for the current row block, while Line 8 does the greedy update by computing a matrix-multiply-and-add calculation on the trailing sub-matrix of B. The iteration proceeds for each row block of B until the entire matrix has been solved.

The recursive variant of triangular solve (shown in Algorithm 3.3) is similar to the blocked version with a block size equal to half the input matrix. However, instead of calling a non-blocked triangular solve for each half, the algorithm recursively calls itself until some tunable cut-off size is reached, after which the algorithm switches to some non-recursive variant.

Algorithm 3.3 TRISOLVE-RECURSIVE(A, B)

```
if $M \leq$ cutoff-value then
    TRISOLVE($A, B$) // call non-recursive code
else
    Let $A_{11}$, $A_{21}$, and $A_{22}$ be the upper-left, lower-left, and lower-right quadrants of $A$, respectively
    Let $B_1$ and $B_2$ be the upper and lower halves of $B$, respectively
    TRISOLVE-RECURSIVE($A_{11}, B_1$) // recursive call
    $B_2 \leftarrow B_2 - A_{21}B_1$ // matrix multiply to update lower half of B
    TRISOLVE-RECURSIVE($A_{22}, B_2$) // recursive call
end if
```

One advantage of having a recursive decomposition as opposed to just using the blocked version is that it allows for the simultaneous blocking of multiple levels of the memory hierarchy, whereas a blocked algorithm is typically optimized for a single level. The tunable hybridized version presented in this chapter takes this flexibility one step further and allows mixing of the recursive version with blocked versions of different blocking sizes for different input sizes.

Greedy versus Lazy

There are two broad classes of triangular solve algorithms characterized by their computation and memory access patterns: the *greedy* and *lazy* variants described in [25].

In the greedy variant, immediately after a row (or block row) of B is solved, the region of B that depends on those values is updated. This computational pattern results in a rank-NB update to the trailing $K \times N$ sub-matrix during each iteration ($K \times NB$ times $NB \times N$). The height of the trailing sub-matrix K varies from $M - NB$ to NB as the algorithm sweeps over B. Figure 3-1 shows the memory access pattern of the greedy variant.

In the lazy variant, the algorithm updates the values of a row (or block row) using previously computed values immediately before the final values of that row (or block row) are solved. This computational pattern results in variable-rank updates to the current $NB \times N$ block row during each iteration ($NB \times K$ times $K \times N$). The rank

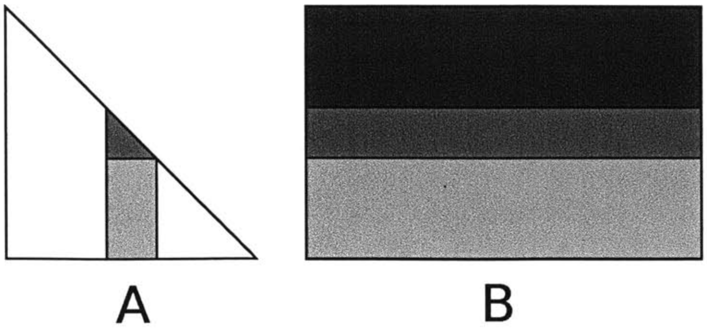


Figure 3-1: Greedy update pattern to solve triangular system $AX = B$. The array B is in an intermediate state in transition to containing the solution matrix X. The dark gray represents the region of B that has already been solved. The medium gray represents the regions of A and B accessed during current the block solve phase. Finally, the light gray represents the regions of A and B accessed during the greedy update.

K of the updates increases from NB to $M - NB$ as the algorithm sweeps over B. Figure 3-1 shows the memory access pattern of the lazy variant.

Depending on the input and blocking sizes, as well as the cache sizes and implementation of matrix multiply, the two different access patterns could have very different performance characteristics. For example, consider a machine where only one $NB \times N$ panel of B fits in cache at a time. The greedy variant requires that the current panel as well as the trailing sub-matrix of B be both read and written during each iteration. While the lazy variant also requires that the current panel be read and written, the leading sub-matrix needs only be read. Thus, under some circumstances, the lazy variant could be favored since fewer dirty cache lines have to be flushed to main memory compared to the greedy variant, reducing write traffic in the memory subsystem.

On the other hand, if the implementation uses a parallel matrix multiply where the threading domain decomposition relies on a large output matrix size for increased parallelism, then the greedy variant would be favored since the output is $K \times N$, where K varies between NB and $M - NB$, rather than being fixed at $NB \times N$ for each iteration. Having the flexibility to choose between these algorithmic variants

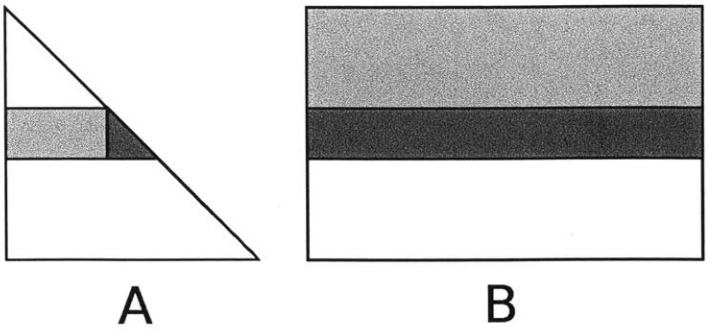


Figure 3-2: Lazy update pattern to solve triangular system $AX = B$. The array B is in an intermediate state in transition to containing the solution matrix X. The light gray represents the regions of A and B accessed during the lazy update. The medium gray represents the region of A that is accessed and the region of B that is solved during the current stage.

depending on the circumstances allows the user to achieve better performance by exploring different combinations of optimizations.

Low-level optimizations

In addition to the medium-level algorithmic variations described above, there are many low-level optimizations that can be implemented. This chapter focus on two such optimizations: loop reordering and array copying/reordering.

Consider the loop traversal shown in Algorithm3.1. The outermost loop iterates over the rows of the right-hand side, while the innermost two loops compute the update to the trailing sub-matrix. The order of traversal (row-wise versus column-wise) during the update of the trailing sub-matrix can significantly affect performance, since the traversal order dictates the memory access pattern. Depending on the layout of the matrix in memory (explored by the next optimization) and the hardware characteristics (e.g. presence of hardware prefetchers), performance can vary significantly.

The implementation of array reordering used here focuses on the storage format (column-major versus row-major) of the right-hand side matrix B. Since a triangular solve involves multiple traversals of this matrix, using a storage format that favors spatial locality of access can help tremendously. On the other hand, changing the storage format at run-time incurs an $O(n^2)$ performance penalty, so the decision to do this optimization should depend on the particular system and input problem size. The combination of tuning the storage format and the traversal via loop reordering allows finding the optimal combination for the best performance. While the algorithm presented here uses a column-major storage for the triangular matrix input A, this parameter could be tuned in future work.

There are many other low-level optimizations that could be explored to increase computation throughput, including loop unrolling, register tiling, explicit prefetching, explicit vectorization, software pipelining, and more. The focus of this work is mainly on auto-tuning the algorithmic optimizations, though adding in additional low-level optimizations should make the performance of the algorithm even better.

Parallelization

There are two ways the algorithm extracts parallelism from the triangular solve computation with multiple right-hand sides. The first is by partitioning B column-wise into sub-matrices and solving for each sub-matrix in parallel. Since the N right-hand side vectors can be solved independently from one another, this step is "embarrassingly parallel", requiring no communication between threads and synchronization only at the end of the computation.

The second way the algorithm extracts parallelism from the computation is during the update phase of the algorithm, immediately before or after a row block is solved, depending on whether the lazy or greedy variant is being used. Since the update involves a matrix-matrix multiply, this update can be parallelized through a recursive or blocked domain decomposition.

Generally speaking, once the problem has been tiled, there are many valid schedules to execute the computation (as in [22]). The algorithm searches through a subset of possible schedules consisting of the bulk-synchronous schedules that result from the recursive and blocked decompositions described earlier in this section.

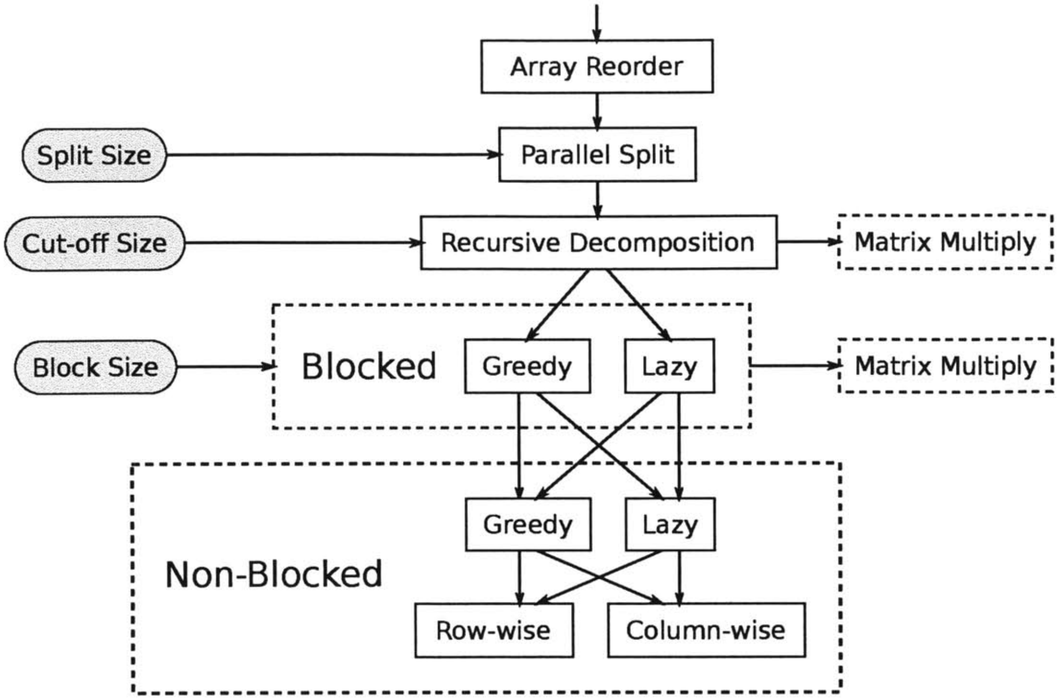


Figure 3-3: Hierarchical organization of the triangular solve algorithm. The autotuner explores all of the algorithmic choices presented in the figure. The values in the gray ovals represent tunable parameters to control the behavior of the algorithm. The algorithm has the option of skipping over intermediate stages given the appropriate tunable parameter values. The recursive and blocked algorithms call a tunable matrix multiply algorithm described in Section 3.2.2 and shown in Figure 3-4.

3.2 Tunable Algorithm Description

3.2.1 Hierarchical Structure

The tunable algorithm presented here leverages all of the optimizations described in Section 3.1.1: parallelization strategy; recursive versus blocked versus non-blocked; greedy versus lazy; and low-level optimizations such as loop and array reordering.

The code is organized as shown in Figure 3-3. At the top-most level, a choice is given to reorder the right-hand side matrix B to change from column-major to row-major storage. The result is then passed to a function that splits B in the N dimension for parallelization of the solve. The matrix is recursively split until a tunable cut-off value is reached.

The recursive split in the N dimension is followed by a recursive decomposition of both A and B matrices in the M dimension until a tunable cut-off value (separate from the previous value) is reached. Once the cut-off is reached, a row-blocked algorithm is then utilized, which can choose between a greedy and a lazy variant in addition to tuning the row block size.

The blocked algorithm uses a non-blocked algorithm as a subroutine to solve for each row block of the output. Finally, the non-blocked algorithm chooses between four base cases: each of the four combinations of greedy versus lazy and row-wise versus column-wise traversal.

The tunable triangular solve algorithm presented here is quite descriptive. Not only is it able to express complex algorithmic variants, simple variants are also expressible using the appropriate parameters and algorithm choices. For example, if recursive decomposition is not desired, the tunable cut-off value can be set to the size of the input. Similarly, if blocking is not desired, the block size can be set to the input size, skipping that stage of the algorithmic hierarchy.

3.2.2 Matrix Multiply

Since matrix multiply is a subroutine used in the recursive and blocked versions of the triangular solve, it can also be tuned to achieve greater performance. Additional performance may also be gained through the parallelism of a blocked matrix multiply. The matrix multiply used by the tunable triangular solve algorithm presented in this chapter utilizes a hierarchical structure similar to the one developed for the triangular solve. Figure 3-4 shows how the code is organized.

At the top-most level, a choice is given to reorder either input matrices to change from column-major to row-major storage. The output is then passed to a function that recursively decomposes the matrix until a tunable cut-off value is reached. The resulting matrix chunks are then passed to a blocked algorithm that does an IJK block traversal (the innermost loop loops over the innermost matrix dimension).

The blocked algorithm calls a non-blocked algorithm as a subroutine to compute

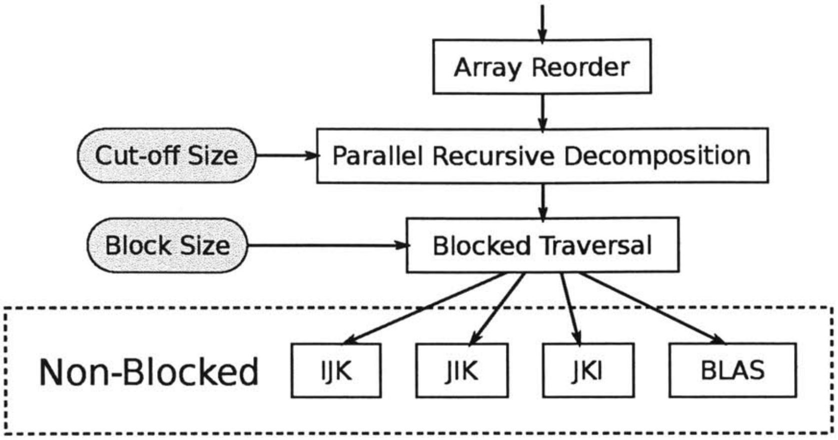


Figure 3-4: Organization of the matrix multiply sub-routine. The auto-tuner explores all of the algorithmic choices presented in the figure. The values in the gray ovals represent tunable parameters to control the behavior of the algorithm. As with the triangular solve, this algorithm has the option of skipping over intermediate stages given the appropriate tunable parameter values. The non-blocked choices: IJK, JIK, and JKI refer to the iteration order in which values are computed. Note that the external BLAS kernel includes many of its own optimizations independent of those shown here.

each block of the output. The non-blocked algorithm has three traversal order choices: a IJK traversal, a JIK traversal, and a JKI traversal., The IJK and JIK traversals both iterate over the innermost dimension in the innermost loop, but differ in their traversal of the output matrix values (row-wise versus column-wise, respectively). The JKI traversal does multiple column-wise rank-1 updates over the output matrix. This traversal is the one used by the generic BLAS, which was found to typically have very good performance for matrices in column-major storage on architectures with hardware prefetching.

Finally, the non-blocked code may call an optimized BLAS DGEMM kernel, which sometimes provides better matrix multiply performance than the other non-blocked choices, likely due to enhanced low-level optimizations not present in this version (e.g. register blocking, explicit prefetching, and explicit vectorization).

3.3 PetaBricks Language

The PetaBricks programming language [2] and compiler were leveraged to implement the tunable algorithm presented in this chapter. PetaBricks is an implicitly parallel programming language in which algorithmic choice is a first class language construct. PetaBricks allows the user to describe many ways to solve a problem and how they fit together. The PetaBricks compiler and runtime use these choices to auto-tune the program in order to find an optimal hybrid algorithm. The triangular solve algorithm presented in this chapter was written in the PetaBricks language and uses the PetaBricks genetic auto-tuner to conduct the search. For more information about the PetaBricks language, compiler, and auto-tuner see [2] and [3]; the following summary is included for background.

3.3.1 PetaBricks Language Design

The main goal of the PetaBricks language is to expose algorithmic choice to the compiler in order to empower the compiler to perform auto-tuning over aspects of the program not normally available to it. PetaBricks is an implicitly parallel language, where the compiler automatically parallelizes PetaBricks programs.

The PetaBricks language is built around two major constructs, *transforms* and *rules*. The *transform*, analogous to a function, defines an algorithm that can be called from other transforms or invoked from the command line. The header for a transform defines *to*, *from*, and *through* arguments, which represent inputs, outputs, and intermediate data used within the transform. The size in each dimension of these arguments is expressed symbolically in terms of free variables, the values of which must be determined by the PetaBricks runtime.

The user encodes choice by defining multiple *rules* in each transform. Each rule computes a region of data in order to make progress towards a final goal state. Rules can have different granularities and intermediate state. The compiler is required to find a sequence of rule applications that will compute all outputs of the program. Rules have explicit dependencies, allowing automatic parallelization and automatic detection and handling of corner cases by the compiler. The rule header references *to* and *from* regions which are the inputs and outputs for the rule. Free variables in these regions can be set by the compiler allowing a rule to be applied repeatedly in order to compute a larger data region. The body of a rule consists of C++-like code to perform the actual work.

3.3.2 PetaBricks Implementation

The PetaBricks implementation consists of three components: a source-to-source compiler from the PetaBricks language to C++, an auto-tuning system and choice framework to find optimal choices and set parameters, and a runtime library used by the generated code.

PetaBricks Compiler

The PetaBricks compiler works using three main phases. In the first phase, *applicable regions* (regions where each rule can legally be applied) are calculated for each possible choice using an inference system. Next, the applicable regions are aggregated together into *choice grids.* The choice grid divides each matrix into rectilinear regions where uniform sets of rules may legally be applied. Finally, a *choice dependency graph* is constructed and analyzed. The choice dependency graph consists of edges between symbolic regions in the choice grids. Each edge is annotated with the set of choices that require that edge, a direction of the data dependency, and an offset between rule centers for that dependency. The output code is generated from this choice dependency graph.

PetaBricks code generation has two modes. In the default mode, choices and information for auto-tuning are embedded in the output code. This binary can then be dynamically tuned, generating an optimized configuration file; subsequent runs can then use the saved configuration file. In the second mode, a previously tuned configuration file is applied statically during code generation. The second mode is included since the C++ compiler can make the final code incrementally more efficient when the choices are fixed.

Auto-tuning System and Choice Framework

The auto-tuner uses the *choice dependency graph* encoded in the compiled application. This choice dependency graph is also used by the parallel scheduler. This choice dependency graph contains the choices for computing each region and also encodes the implications of different choices on dependencies.

The intuition of the auto-tuning algorithm is to take a bottom-up approach to tuning. To simplify auto-tuning, it is assumed that the optimal solution to smaller sub-problems is independent of the larger problem. In this way algorithms are built incrementally, starting on small inputs and working up to larger inputs.

The auto-tuner builds a multi-level algorithm. Each level consists of a range of input sizes and a corresponding algorithm and set of parameters. Rules that recursively invoke themselves result in algorithmic compositions. In the spirit of a genetic tuner, a population of candidate algorithms is maintained. This population is seeded with all single-algorithm implementations. The auto-tuner starts with a small training input and on each iteration doubles the size of the input. At each step, each algorithm in the population is tested. New algorithm candidates are generated by adding levels to the fastest members of the population. Finally, slower candidates in the population are dropped until the population is below a maximum size threshold. Since the best algorithms from the previous input size are used to generate candidates for the next input size, optimal algorithms are iteratively built from the bottom up.

In addition to tuning algorithm selection, PetaBricks uses an n-ary search tuning algorithm to optimize additional parameters such as parallel-sequential cutoff points for individual algorithms, iteration orders, block sizes (for data parallel rules), data layout, as well as user specified tunable parameters.

All choices are represented in a flat configuration space. Dependencies between these configurable parameters are exported to the auto-tuner so that the auto-tuner can choose a sensible order to tune different parameters. The auto-tuner starts by tuning the leaves of the graph and works its way up. If there are cycles in the dependency graph, it tunes all parameters in the cycle in parallel, with progressively larger input sizes. Finally, it repeats the entire training process, using the previous iteration as a starting point, a small number of times to better optimize the result.

Runtime Library

The runtime library is primarily responsible for managing parallelism, data, and configuration. It includes a runtime scheduler as well as code responsible for reading, writing, and managing inputs, outputs, and configurations. The runtime scheduler dynamically schedules tasks (that have their input dependencies satisfied) across processors to distribute work. The scheduler attempts to maximize locality using a greedy algorithm that schedules tasks in a depth-first search order. Following the approach taken by Cilk [17], work is distributed with thread-private deques and a task stealing protocol.

3.4 Performance Analysis and Discussion

To demonstrate the effectiveness of the algorithmic auto-tuning approach presented here, two triangular solve algorithms (upper triangular solve and lower triangular solve) were tuned on three Intel hardware platforms spanning three micro-architecture generations (Core, Nehalem, and Sandy Bridge), with each platform having different amounts of available parallelism. The algorithmic choices and parameters chosen by the auto-tuner and the performances of the resulting algorithms vary across platforms.

3.4.1 Hardware Platforms

The oldest platform benchmarked is the server-class, dual-socket, quad-core Intel Harpertown X5460 based on the Core micro-architecture. Its cores are clocked at 3.16 GHz, have 32 KB private L1 data cache, and share 6 MB of L2 cache between pairs of cores. Both processors are connected through a common memory controller to quad-channel DDR2-667 RAM which provides a maximum theoretical shared memory bandwidth of 21 GB/s.

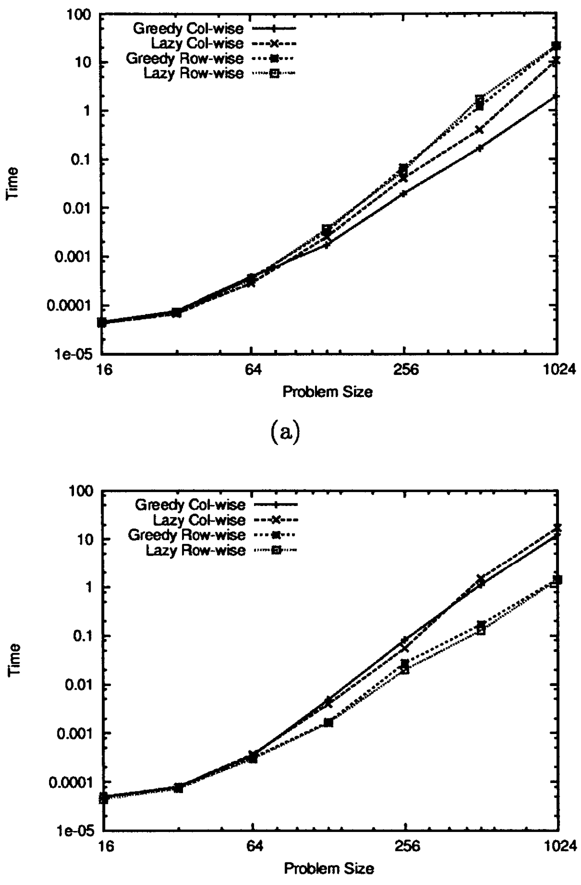
The second platform benchmarked is the server-class, dual-socket, hexa-core Intel Westmere-EP X5650 based on the Nehalem micro-architecture. Compared to the Harpertown, this machine has a higher level of available parallelism but a lower clock at 2.67 GHz. Each processor has an integrated memory controller connected to triple-channel DDR3-1333 RAM, providing a theoretical 32 GB/s of bandwidth per socket, or 64 GB/s total aggregate bandwidth. Additionally, the two processors are directly connected to each other via a Quick Path interface. The Quick Path interface provides full-duplex 25.6 GB/s of bandwidth (12.8 GB/s each way).

Finally, the newest platform tested is a desktop-class, single socket, dual-core Intel Sandy Bridge i3-2100. Hyper-threading is enabled to present four logical cores from two hardware cores, which may increase throughput if the scheduler can more fully utilize the available computational units. The integrated memory controller provides a dual-channel interface to DDR3-1333 RAM, yielding a potential total memory bandwidth of 21 GB/s. Although this total bandwidth is the same as the Harpertown platform's total bandwidth, it is split between two physical cores instead of eight, so the optimal algorithmic parameters that manage the trade-off between computational performance and memory bandwidth usage could differ.

All platforms have hardare prefetching enabled, which detects strided memory access patterns and attempts to prefetch cache lines from main memory so that the waiting time for data to arrive in cache is reduced.

3.4.2 Serial optimizations

To get a sense of the benefit from the serial, non-blocked optimizations, the performance of each of the variants were tested on an $N \times N$ lower-triangular input A with an $N \times N$ right-hand side B, for N equal to powers of 2 between 16 and 1024. Below size 16, the differences between algorithms is so low as to fall below the noise threshold of the measurements. Figure 3-5 shows the execution time required to compute a lower-triangular solve on the Sandy Bridge platform for each of the four serial, non-blocked algorithms. Figure 3-5(a) shows the performance using no array reordering transformation (using column-major storage), while Figure 3-5(b) shows performance



(a)

(b)

Figure 3-5: Performance of serial, non-blocked reference algorithms for solving NxN lower triangular system on N right-hand side vectors on the Sandy Bridge platform. a) shows the performance with no array reordering applied (column-major storage), while b) shows the performance with array reordering applied (row-major storage).

after an array reordering transformation (using row-major storage). In each graph, the memory access pattern was varied using either greedy or lazy updates, and the array update was varied using either column-wise or row-wise traversal.

As can be seen from the graphs, the best serial performance is generally achieved when the traversal matches the array ordering in such a way that the inner-most loop accesses the arrays in a unit-stride fashion. This confirms expectations that accessing memory in such a fashion provides enhanced spatial locality, increasing both cache line utilization and the effectiveness of the hardware prefetchers.

(a) Lower-triangular solve

Size	Harpertown	Westmere-EP	Sandy Bridge
16	NR GC	R LR	R GR
32	NR GC	R LR	R GR
64	NR R16 → GC	R LR	NR S32 → LR
128	NR BG16 → GC	R S64 → R64 → LR	NR S32 → GC
256	NR S64 → R128 → BG16 → GC	R S64 → R64 → LR	NR S64 → GC
512	NR S64 → R128 → BG16 → GC	R S64 → R64 → LR	NR S32 → GC
1024	NR S128 → R128 → BG16 → GC	R S64 → R64 → LR	NR S16 → GC

(b) Upper-triangular solve

Size	Harpertown	Westmere-EP	Sandy Bridge
16	NR GC	NR LR	NR GC
32	NR GC	NR LR	NR LC
64	NR S32 → GC	NR GC	NR LC
128	NR S32 → GC	NR R32 → GC	NR GC
256	NR S32 → BL135 → GC	NR S32 → R64 → GC	NR S64 → GC
512	NR S64 → BL112 → GC	NR S64 → R64 → GC	NR S16 → GC
1024	NR S128 → BG112 → GC	NR S128 → R32 → GC	NR S16 → GC

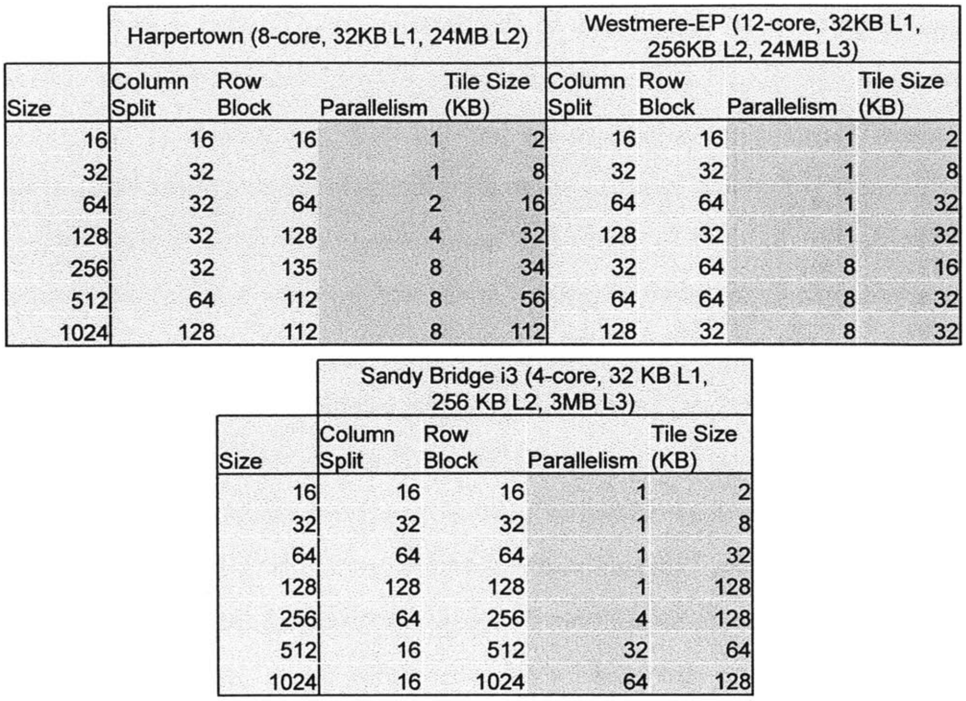
Table 3.1: **Auto-tuned algorithms found by PetaBricks genetic auto-tuner. The tuned algorithms vary across problem (lower vs upper triangular solve), problem size, and hardware platform. Abbreviations used: R/NR: array reorder/no reorder; Sx: parallel split using split size x; Rx: recursive decomposition using cut-off x; BGx/BLx: blocked greedy/lazy traversal using block size x; GR/GC/LR/LC: greedy row-wise/greedy column-wise/lazy row-wise/lazy column-wise traversal. In all cases where a matrix multiply was used as a subroutine, the auto-tuner chose to use a generic, ATLAS library DGEMM.**

3.4.3 Auto-tuned Algorithms

The auto-tuned algorithms discovered by the PetaBricks genetic auto-tuner are shown in Table 3.1. The optimal algorithmic choices discovered by the auto-tuner vary according to several variables, including problem type, input size, and hardware platform.

The notation used indicates which optimizations were applied, the algorithmic choices used for each optimization (depicted by the white rectangles in Figure 3-3), and the corresponding size parameters chosen (depicted by the grey ovals in Figure 3-3). For example, "NR GC" means that no array re-order optimization was applied (NR), and a greedy update with a column-wise traversal (GC) was used. In this case, no recursion or blocking was applied.

The more complex specification "NR S128 → R128 → BG16 → GC" indicates that no array re-ordering was applied (NR), a parallel split to blocks of size 128 (S128)



	Harpertown (8-core, 32KB L1, 24MB L2)				Westmere-EP (12-core, 32KB L1, 256KB L2, 24MB L3)			
Size	Column Split	Row Block	Parallelism	Tile Size (KB)	Column Split	Row Block	Parallelism	Tile Size (KB)
16	16	16	1	2	16	16	1	2
32	32	32	1	8	32	32	1	8
64	32	64	2	16	64	64	1	32
128	32	128	4	32	128	32	1	32
256	32	135	8	34	32	64	8	16
512	64	112	8	56	64	64	8	32
1024	128	112	8	112	128	32	8	32

	Sandy Bridge i3 (4-core, 32 KB L1, 256 KB L2, 3MB L3)			
Size	Column Split	Row Block	Parallelism	Tile Size (KB)
16	16	16	1	2
32	32	32	1	8
64	64	64	1	32
128	128	128	1	128
256	64	256	4	128
512	16	512	32	64
1024	16	1024	64	128

Figure 3-6: Summary of level of parallelism and tile shapes and sizes that result from the algorithmic configurations given in Table 3.1 for each hardware platform and input size. The values in the orange cells are computed from the configuration values given in the blue cells.

was applied to the N dimension, and a recursive decomposition down to blocks of size 128 (R128) was applied to the M dimension. To handle the resulting 128×128 chunks of the matrix, a blocked, greedy traversal utilizing blocks of size 16 (BG16) is used, along with a non-blocked base case using a greedy column-wise (GC) traversal.

Figure 3-6 summarizes the level of parallelism and tile shapes and sizes determined by the parallel split phase and recursive and blocked decomposition phases of the algorithm for each platform and for different input sizes.

Examination of Table 3.1 and Figure 3-6 yields a number of insights. There is a clear dependence of the optimal problem decomposition strategy on the hardware platform. On the Sandy Bridge platform, the only layer of complexity necessary when running on larger input sizes is the parallel split phase shown at the top of Figure 3-3. On the other hand, the tuned algorithms for the Westmere platform make use of the recursive decomposition in addition to the parallel split. Further, some of the tuned algorithms for the Harpertown platform make use of both recursive decomposition and blocked traversal in addition to the parallel split optimization.

The primary difference between these approaches is the size of the vertical dimension of the panels updated by the serial, non-blocked code. The Sandy Bridge architecture appears to favor taller, skinnier panels with an unbroken unit-stride memory access pattern, while the other architectures appear to favor the enhanced temporal locality exposed by shorter blocks. This preference could be a result of enhancements made to the hardware prefetchers in the Sandy Bridge architecture.

Another interesting feature in the table is how the parallel split size varies with problem size. For the Harpertown and Westmere architectures, the amount of parallelism (problem size divide by split size) increases towards about 8 as the problem size increases, which is to be expected for these machines given the amount of hardware parallelism available. On the other hand, for the Sandy Bridge platform, the amount of parallelism continues to increase up to 64 (with a split size of 16 on problems of size 1024). This indicates that the platform may require greater thread-level parallelism to get the best performance, possibly by utilizing the extra parallelism to hide memory latency.

3.4.4 Performance

To illustrate the performance of the auto-tuned triangular solve algorithm, it is compared to both a naïve PetaBricks algorithm and the parallel DTRSM function from the ATLAS 3.8.4 auto-tuning framework [44]. The naïve PetaBricks algorithm simply selects the best serial, non-blocked configuration and parallelizes by splitting B into n column blocks, where n is the maximum number of logical hardware threads (including hyper-threading, if enabled) available on the platform. For example, if there are 8 logical cores available on the machine, a 1024×1024 matrix is split into 1024×128 panels, where each panel is independently solved by a separate thread. For ATLAS, the framework was built and tuned from source on each tested hardware platform.

Figures 3-7 and 3-8 show the performance of lower and upper triangular solves, respectively, for each of the hardware platforms. The performance of the auto-tuned algorithm is shown alongside the best performing serial, non-blocked algorithm, the naïve parallel algorithm, as well as ATLAS. The problem size varies along the x-axis, while the y-axis shows the time to solve the system on a logarithmic scale.

The auto-tuned algorithms offer a significant performance improvement over the naïve strategies for nearly all configurations. On the 1024 input size, for the lower- and upper-triangular problem types, respectively, a 14x and 11x speedup is observed on the Harpertown platform, a 14x and 9x speedup on the Westmere platform, and a 3x speedup for both problem types on the Sandy Bridge platform. These performance improvements are a result of finding the best *combination* of algorithmic choices and low-level optimizations to leverage the available hardware resources on each particular platform.

The performance of the auto-tuned algorithm is quite good even when compared to the performance of ATLAS, except perhaps in the case of the Sandy Bridge platform. ATLAS performs 13% faster, 4% slower, and 322% faster than the auto-tuned PetaBricks algorithms on the Harpertown, Westmere-EP, and Sandy Bridge platforms, respectively.

Use of the PetaBricks language and runtime (based on C++) introduces some overhead to the computation when compared to pure C or Fortran as is used by ATLAS. Sources of this overhead include extra function calls made necessary by the manipulation of objects and a layer of abstraction around the data arrays. The effect of the overhead is apparent at the lower input sizes on all platforms. As the input size increases, both the auto-tuned algorithm and ATLAS show superior scaling behavior versus the naïve algorithms; however, ATLAS retains a performance advantage due not only to the optimizations available to it that are not currently supported by the auto-tuned PetaBricks algorithm, but also to this overhead.

In order to get a better sense of the effect of the overhead, Figure 3-9 shows the performance of four algorithms when solving a 1024 × 1024 lower triangular system. In addition to the auto-tuned PetaBricks algorithm and ATLAS, a naïve serial algorithm was implemented in both PetaBricks and C to measure the overhead that exists independent of any tuning. The larger performance gap between the two naïve implementations on the Sandy Bridge platform suggest that increased overhead could account for at least some of the performance differential between the auto-tuned PetaBricks algorithm and ATLAS on that platform.

Since the optimization space of this work is geared towards higher-level algorithmic optimizations, whereas ATLAS focuses exclusively on low-level optimizations, it can be hypothesized that combining the optimization search spaces of the two would yield even better results than either framework in isolation. Indeed, the types of algorithmic optimizations presented in this work do not preclude the use of additional low-level optimizations such as register blocking, loop unrolling, and explicit software prefetching and vectorization. The combination of algorithmic optimizations with a full set of low-level optimizations is an exciting future direction.

3.5 Related Work

There has been much prior investigation in tuning dense matrix computations. Much of the prior work has focused on matrix-matrix multiply (DGEMM), and matrix factorizations such as Cholesky, LU and QR. Some of the lessons learned from those auto-tuning programs can be applied to dense triangular solve. Indeed, many of them utilize dense triangular solve as a sub-routine and may benefit from auto-tuning of the triangular solve in addition to their existing optimizations.

Perhaps the most prominent dense linear algebra auto-tuner is the ATLAS project [44]. ATLAS leverages empirical timings to profile the hardware and identify the best optimizations for an L1 cache-contained dense matrix-matrix multiply. It then utilizes this kernel to build optimized Level 3 BLAS functions.

There have been numerous studies in tuning the blocking or recursion strategy for dense matrix computations. The PLASMA [22] [1] and MAGMA [40] projects allow great algorithmic flexibility in the matrix computations by representing them as graph of dependent computations, utilizing a pruned search for optimal tile and block sizes and a static schedule for execution. Whaley investigated auto-tuning of the NB blocking factor used in ATLAS's included LAPACK functions [43].

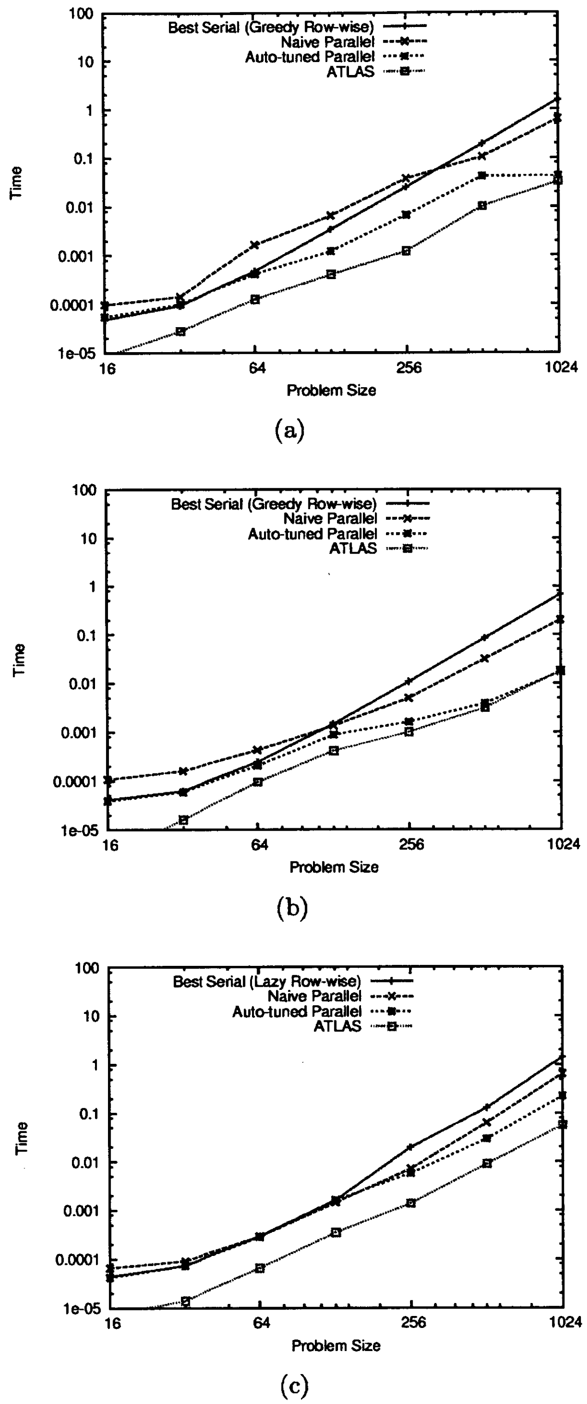
There has also been related work on automatic generation of valid dense linear algebra code variants without a focus on auto-tuning. For example, the FLAME system is a formal system for generating provably valid linear algebra algorithm variants [21]. Another example is work by Yi et. al. [46], which achieves automatic blocking of the LU and QR factorizations via dependence hoisting.

The primary distinction between the work presented here and these prior works is the higher-level algorithmic view of auto-tuning the computation. A hierarchy of algorithmic variants is used to increase the breadth of the optimization space available to the auto-tuner. Instead of focusing on a particular class of optimizations, such as blocking size, a structured holistic view of the problem as a set of hierarchical algorithmic variants (see Figure 3-3) is taken. Additionally, the algorithm is sensitive to the interactions between the algorithmic variants and the low-level optimizations and finds the best combination of these choices and parameters for optimal performance.

3.6 Conclusions and Future Work

Tuning the algorithmic aspects of triangular solve results in significant improvement in execution time compared to untuned or naïvely tuned code. Still, the raw performance of the algorithm in its current form may not achieve the peak potential possible on the hardware tested. There are many ways in which the performance of the auto-tuned code could be improved through further low-level optimizations. For example, manual register tiling and unrolling, instruction reordering, and explicit vectorization could potentially increase register reuse and increase instruction-level parallelism. Explicit prefetching could help decrease the number of stalled cycles spent waiting for data before the hardware prefetchers spin up. NUMA-aware data initialization and thread scheduling could also affect parallel performance on platforms with multiple sockets.

While the current algorithm does not incorporate these optimizations, the performance benefits of higher-level algorithmic auto-tuning should carry over to codes in which lower-level optimizations are performed. In other words, even optimized codes which have focused on low-level optimizations could stand to benefit from the types of higher-level algorithmic optimizations shown here. Since higher-level choices affect the context in which the low-level optimizations operate, they can change the structure of the computation in ways those optimizations cannot replicate. Because of this, combining algorithmic tuning with further low-level tuning should present more opportunities for synergistic optimization. These possibilities will be explored in future work.

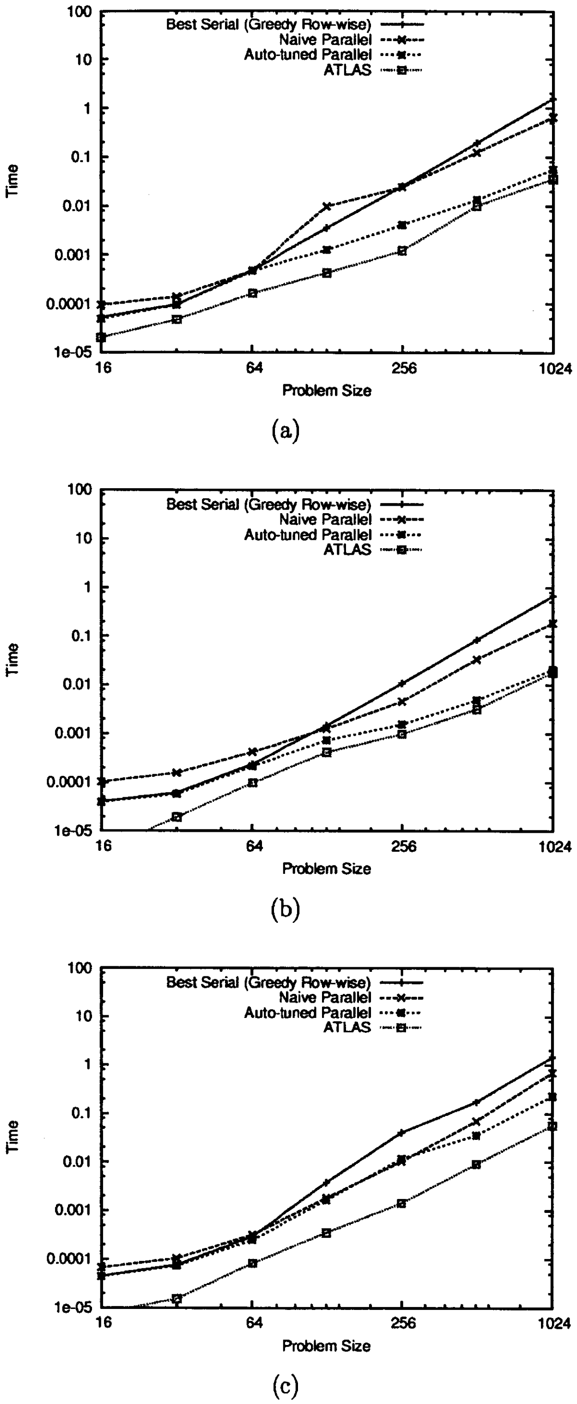


(a)

(b)

(c)

Figure 3-7: Performance of hybrid auto-tuned parallel algorithm versus best reference serial non-blocked algorithm, naïve parallel algorithm, and ATLAS for solving NxN lower triangular system on N right-hand side vectors on a) Intel Harpertown, b) Intel Westmere-EP, and c) Intel Sandy Bridge i3.



(a)

(b)

(c)

Figure 3-8: Performance of hybrid auto-tuned parallel algorithm versus best reference serial non-blocked algorithm, naïve parallel algorithm, and ATLAS for solving NxN upper triangular system on N right-hand side vectors on a) Intel Harpertown, b) Intel Westmere-EP, and c) Intel Sandy Bridge i3.

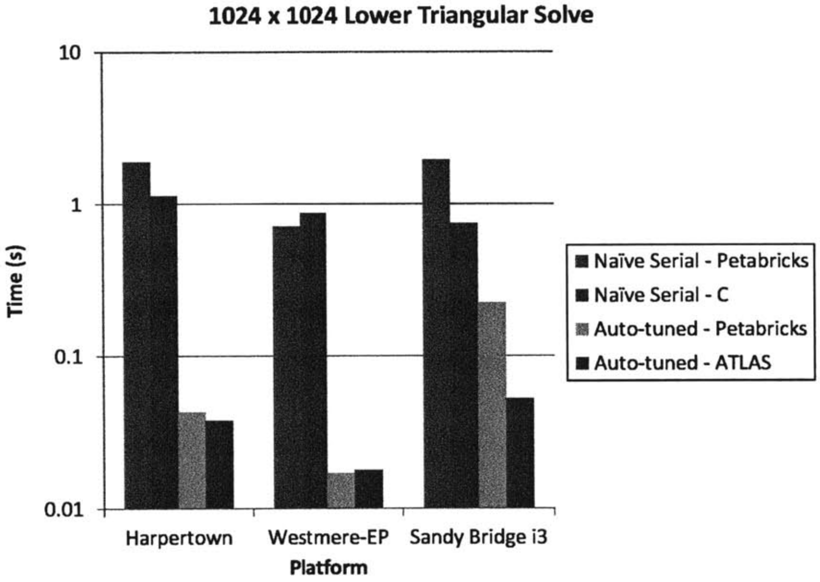


Figure 3-9: Performance comparison between the auto-tuned algorithm and ATLAS. Since there is some overhead introduced by the PetaBricks system outside the scope of this work, the performance of a simple naïve serial algorithm is given in both PetaBricks and C to help gauge the approximate effect of the overhead. Note that on the Sandy Bridge platform, where the difference in performance between the auto-tuned algorithm and ATLAS is the greatest, the difference between the two naïve serial algorithms is also the greatest.

Chapter 4

Auto-tuning the multigrid linear solver's cycle shapes

4.1 Introduction

Multigrid is a prime example of a technique where high-level algorithmic auto-tuning has the potential to enable greater performance. Not only is it possible to make algorithmic choices at the input grid resolution, but a program can switch techniques as the problem is recursively coarsened to smaller grid sizes to take advantage of algorithms with different scaling behaviors. The algorithm is further complicated by the fact that, being an iterative algorithm, it can produce outputs of varying accuracy depending on its parameters. Users with different convergence criteria must experiment with parameters to yield a tuned algorithm that meets their accuracy requirements. Finally, once an appropriately tuned algorithm has been found, users often have to start all over when migrating from one machine to another.

This chapter presents an algorithm and auto-tuning methodology that address all of these issues in an efficient manner. It describes a novel dynamic programming strategy that allows fair comparisons to be made between various iterative, recursive, and direct methods, resulting in an efficient, tuned algorithm for user-specified convergence criteria. The resulting algorithms can be visualized as tuned multigrid cycle shapes that apply targeted computational power to meet the accuracy requirements of the user.

These cycle shapes dictate the order in which grid coarsening and grid refinement are interleaved with both iterative methods, such as Jacobi or Successive Over-Relaxation, as well as direct methods, which tend to have superior performance for small problem sizes. The need to make choices between all of these methods brings the issue of variable accuracy to the forefront. Not only must the auto-tuning framework compare different possible multigrid cycle shapes against each other, but it also needs the ability to compare tuned cycles against both direct and (non-multigrid) iterative methods. This problem is addressed by defining an accuracy metric that measures how effectively each cycle shape reduces error, and then making comparisons between candidate cycle shapes based on this common yardstick. The results show that the flexibility to trade performance versus accuracy at all levels of recursive computation enables excellent performance on a variety of platforms compared to algorithmically static implementations of multigrid.

The methodology presented here does not tune cycle shapes by manipulating the shapes directly; it instead categorizes algorithms based on the accuracy of the results produced, allowing it to make high-level comparisons between *all types* of algorithms (direct, iterative, and recursive) and make tuning decisions based on that common yardstick. Additionally, the auto-tuner has the flexibility of utilizing different accuracy constraints for various components within a single algorithm, allowing the auto-tuner to independently trade performance and accuracy at each level of multigrid recursion.

This work on multigrid was developed using the PetaBricks programming language [2]. A summary of the PetaBricks language and compiler is given in Section 3.3 of Chapter 3. As stated in that section, PetaBricks is an implicitly parallel programming language where algorithmic choice is a first class construct, to help programmers express and tune algorithmic choices and cutoffs such as these to obtain the fastest combination of algorithms to solve a problem. While traditional compiler optimizations can be successful at optimizing a single algorithm, when an algorithmic change is required to boost performance the burden is put on the programmer to incorporate the new algorithm. Programs written in PetaBricks can naturally describe multiple algorithms for solving a problem and how they can fit together. This information is used by the PetaBricks compiler and runtime to create and auto-tune an optimized multigrid algorithm.

4.2 Auto-tuning multigrid

Although multigrid is a versatile technique that can be used to solve many different types of problems, the 2D Poisson's equation will be used as an example and benchmark to guide the discussion. The techniques presented here are generalizable to higher dimensions and the broader set of multigrid problems.

Poisson's equation is a partial differential equation that describes many processes in physics, electrostatics, fluid dynamics, and various other engineering disciplines. The continuous and discrete versions are

$$\nabla^2 \phi = f \quad \text{and} \quad Tx = b, \tag{4.1}$$

where T, x, and b are the finite difference discretizations of the Laplace operator, ϕ, and f, respectively.

Three basic algorithmic building blocks are used to build an auto-tuned multigrid solver for Poisson's equation: one direct (band Cholesky factorization through LAPACK's DPBSV routine), one iterative (Red-Black Successive Over Relaxation), and one recursive (multigrid). The table below shows the computational complexity of using any single algorithm to compute a solution.

Algorithm	Direct	SOR	Multigrid
Complexity	n^2 (N^4)	$n^{1.5}$ (N^3)	n (N^2)

From left to right, each of the methods has a larger overhead, but yields a better asymptotic serial complexity [12]. N is the size of the grid on a side, and $n = N^2$ is the number of cells in the grid.

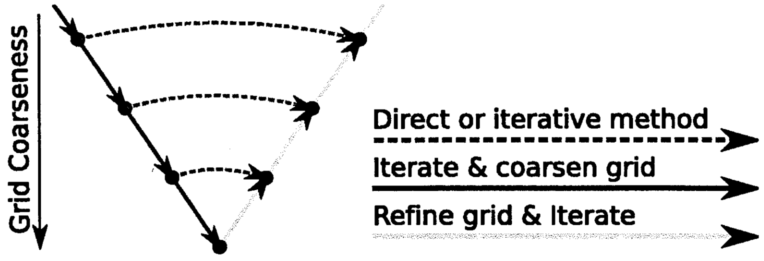


Figure 4-1: Simplified illustration of choices in the multigrid algorithm. The diagonal arrows represent the recursive case, while the dotted horizontal arrows represent the shortcut case where a direct or iterative solution may be substituted. Depending on the desired level of accuracy a different choice may be optimal at each decision point. This figure does not illustrate the auto-tuner's capability of using multiple iterations at different levels of recursion; it shows a single iteration at each level.

4.2.1 Algorithmic choice in multigrid

Multigrid is a recursive algorithm that uses the solution to a coarser grid resolution as part of the algorithm. This section addresses tuning symmetric "V-type" cycles. An extension to full multigrid will be presented in Section 4.2.4.

For simplicity, assume all inputs are of size $N = 2^k + 1$ for some positive integer k. Let x be the initial state of the grid, and b be the right hand side of Equation (4.1).

Algorithm 4.1 MULTIGRID-V-SIMPLE(x, b)

```
1: if $N \leq 3$ then
2:     Solve directly
3: else
4:     Relax using some iterative method
5:     Compute the residual and restrict to half resolution
6:     Recursively call MULTIGRID-V-SIMPLE on coarser grid
7:     Interpolate result and add correction term to current solution
8:     Relax using some iterative method
9: end if
```

It is at the recursive call on line 6 that the auto-tuning compiler can make a choice of whether to continue making recursive calls to multigrid or take a shortcut by using the direct solver or one of the iterative solvers at the current resolution. Figure 4-1 shows these possible paths of the multigrid algorithm.

The idea of choice can be implemented by defining a top level function MULTIGRID-

V, which makes calls to either the direct, iterative, or recursive solution. The function RECURSE implements the recursive solution.

Algorithm 4.2 MULTIGRID-V(x, b)

```
1: either
2:     Solve directly
3:     Use an iterative method
4:     Call RECURSE for some number of iterations
5: end either
```

Algorithm 4.3 RECURSE(x, b)

```
1: if $N \leq 3$ then
2:     Solve directly
3: else
4:     Relax using some iterative method
5:     Compute the residual and restrict to half resolution
6:     On the coarser grid, call MULTIGRID-V
7:     Interpolate result and add correction term to current solution
8:     Relax using some iterative method
9: end if
```

Making the choice in line 1 of MULTIGRID-V has two implications. First, the time to complete the algorithm is choice dependent. Second, the accuracy of the result is also dependent on choice since the various methods have different abilities to reduce error (depending on parameters such as number of iterations or weights). To make a fair comparison between choices, one must take both performance and accuracy of each choice into account. To this end, the auto-tuner keeps track of not just a single optimal algorithm at every recursion level, but a *set* of such optimal algorithms for varying levels of desired accuracy.

4.2.2 Full dynamic programming solution

This section describes a full dynamic programming solution to handling variable accuracy. Define an algorithm's *accuracy level* to be the ratio between the error norm of its input x_{in} versus the error norm of its output x_{out} compared to the optimal solution

x_{opt}:

$$\frac{||x_{in} - x_{opt}||_2}{||x_{out} - x_{opt}||_2}.$$

This ratio was chosen instead of its reciprocal so that a higher accuracy level is better, which is more intuitive. In order to assess the accuracy level of a potential tuned algorithm, it is assumed the user has access to representative training data so that the accuracy level of the algorithms during tuning closely reflects their accuracy level during use.

Let level k refer to an input size of $N = 2^k + 1$. Suppose that for level $k - 1$, the tuner has solved for some set A_{k-1} of optimal algorithms, where optimality is defined such that no optimal algorithm is dominated by any other algorithm in both accuracy and compute time.

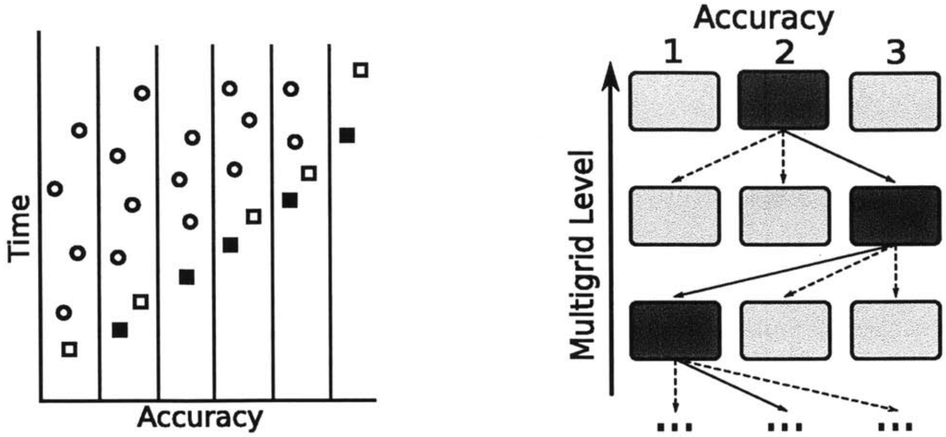
In order to construct the optimal set A_k, each of the algorithms in A_{k-1} are substituted for step 6 of RECURSE. Parameters in the other steps of the algorithm are also varied, including the choice of iterative methods and the number of iterations (possibly zero) in steps 4 and 8 of RECURSE and steps 3 and 4 of MULTIGRID-V.

Trying all of these possibilities will yield many algorithms that can be plotted as in Figure 4-2(a) according to their accuracy and compute time. The optimal algorithms added to A_k are the dominant ones designated by square markers.

The reason to remember algorithms of multiple accuracies for use in step 6 of RECURSE is that it may be better to use a less accurate, fast algorithm and then iterate multiple times, rather than use a more accurate, slow algorithm. Note that even if a direct solver is used in step 6, the interpolation in step 7 will invariably introduce error at the higher resolution.

4.2.3 Discrete dynamic programming solution

Since the optimal set of tuned algorithms can grow to be very large, the auto-tuner offers an approximate version of the above solution. Instead of remembering the full optimal set A_k, the compiler remembers the fastest algorithm yielding an accuracy of at least p_i for each p_i in some set $\{p_1, p_2, \ldots, p_m\}$. The vertical lines in Figure 4-2(a)



(a) (b)

Figure 4-2: (a) Possible algorithmic choices with optimal set designated by squares (both hollow and solid). The choices designated by solid squares are the ones remembered by the auto-tuner, being the fastest algorithms better than each accuracy cutoff line. (b) Choices across different accuracies in multigrid. At each level, the auto-tuner picks the best algorithm one level down to make a recursive call. The path highlighted in red is an example of a possible path for accuracy level p_2.

indicate the discrete accuracy levels p_i, and the optimal algorithms (designated by solid squares) are the ones remembered by the auto-tuner. Each highlighted algorithm is associated with a function MULTIGRID-V$_i$, which achieves accuracy p_i on all input sizes.

Due to restricted time and computational resources, to further narrow the search space, only SOR is used as the iteration function since it was found experimentally that it performed better than weighted Jacobi on the training data for similar computation cost per iteration. In MULTIGRID-V$_i$, the weight parameter of SOR is fixed to ω_{opt}, the optimal value for the 2D discrete Poisson's equation with fixed boundaries [12]. In RECURSE$_i$, SOR's weight parameter is fixed to 1.15 (chosen by experimentation to yield fast convergence). The number of iterations of SOR in steps 4 and 8 in RECURSE$_i$ is fixed to one. As more powerful computational resources become available over time, the restrictions on the algorithmic search space presented here may be relaxed to find a more optimal solution.

The resulting accuracy-aware Poisson solver is a family of functions, where i is the accuracy parameter:

Algorithm 4.4 MULTIGRID-V$_i(x, b)$

```
either
    Solve directly
    Iterate using SOR$_{\omega_{\text{opt}}}$ until accuracy $p_i$ is achieved
    For some $j$, iterate with RECURSE$_j$ until accuracy $p_i$ is achieved
end either
```

Algorithm 4.5 RECURSE$_i(x, b)$

```
if $N \leq 3$ then
    Solve directly
else
    Compute one iteration of SOR$_{1.15}$
    Compute the residual and restrict to half resolution
    On the coarser grid, call MULTIGRID-V$_i$
    Interpolate result and add correction term to current solution
    Compute one iteration of SOR$_{1.15}$
end if
```

The auto-tuning process determines what choices to make in MULTIGRID-V$_i$ for each i and for each input size. Since the optimal choice for any single accuracy for an input of size $2^k + 1$ depends on the optimal algorithms for *all* accuracies for inputs of size $2^{k-1} + 1$, the auto-tuner tunes all accuracies at a given level before moving to a higher level. In this way, the auto-tuner builds optimal algorithms for every specified accuracy level and for each input size up to a user specified maximum, making use of the tuned sub-algorithms as it goes.

The final set of multigrid algorithms produced by the auto-tuner can be visualized as in Figure 4-2(b). Each of the versions has the flexibility to choose any of the other versions during its recursive calls, and the optimal path may switch between accuracies many times as the algorithm recurses down towards either the base case or a shortcut case.

4.2.4 Extension to Auto-tuning Full Multigrid

Full multigrid methods have been shown to exhibit better convergence behavior than traditional symmetric cycle shapes such as the V and W cycles by utilizing an esti-

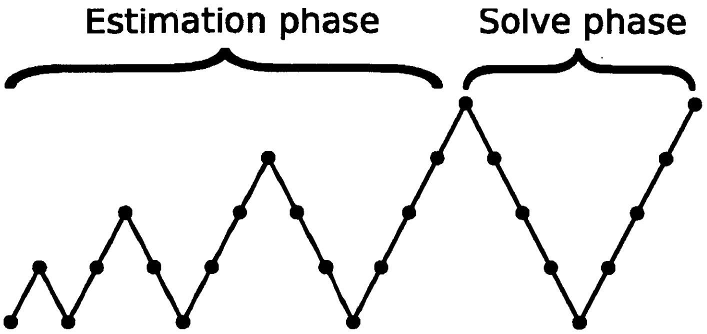


Figure 4-3: Conceptual breakdown of full multigrid into an estimation phase and a solve phase. The estimation phase can be thought of as just a recursive call to full multigrid up to a coarser grid resolution. This recursive structure, in addition to the auto-tuned "V-type" multigrid cycles, is used to build tuned full multigrid cycles.

mation phase before the solve phase (see Figure 4-3). The estimation phase of the full multigrid algorithm can be thought of as just a recursive call to itself at a coarser grid resolution. The auto-tuning ideas presented thus far are extended to leverage this structure and produce auto-tuned full multigrid cycles.

The following simplified code for ESTIMATE and FULL-MULTIGRID illustrates how to construct an auto-tuned full multigrid cycle.

Algorithm 4.6 ESTIMATE$_i(x, b)$

```
Compute residual and restrict to half resolution
Call FULL-MULTIGRID$_i$ on restricted problem
Interpolate result and add correction to $x$
```

Algorithm 4.7 FULL-MULTIGRID$_i(x, b)$

```
either
   Solve directly
   For some $j$, compute estimate by calling ESTIMATE$_j(x, b)$, then either:
     Iterate using SOR$_{\omega_{\text{opt}}}$ until accuracy $p_i$ is achieved
     For some $k$, iterate with RECURSE$_k$ until accuracy $p_i$ is achieved
end either
```

The discrete dynamic programming technique presented in Section 4.2.3 is used again here, where only small sets of optimized FULL-MULTIGRID$_j$ and MULTIGRID-V$_k$ functions are maintained for use in recursive calls. In FULL-MULTIGRID$_i$, there are three choices: the first is just a direct solve (line 2), while the latter two choices (lines 4 and 5) are similar to those given in MULTIGRID-V_i except an estimate is first calculated and then used as a starting point for iteration. Note that this structure is descriptive enough to include the standard full multigrid V or W cycle shapes, just as the MULTIGRID-V_i algorithm can produce standard regular V or W cycles.

The parameters j and k in FULL-MULTIGRID can be chosen independently, providing a great deal of flexibility in the construction of the optimized full multigrid cycle shape. In cases where the user does not require much accuracy in the final output, it may make sense to invest more heavily in the estimation phase, while in cases where very high precision is needed, a high precision estimate may not be as helpful as most of the computation would be done in relaxations at the highest resolution. Indeed, patterns of this type can be seen in the experimental results.

4.2.5 Limitations

It should be clear that the algorithms produced by the auto-tuner are not meant to be optimal in any theoretical sense. Because of the compromises made in the name of efficiency, the resulting auto-tuning algorithm merely strives to discover near-optimal algorithms from within the restricted space of cycle shapes reachable during the search. There are many cycle shapes that fall outside the space of searched algorithms; for example, this approach does not check algorithms that utilize different choices in succession at the same recursion depth instead of choosing a single choice and iterating. Future work may examine the extent to which this restriction impacts performance.

Additionally, the scalar accuracy metric is an imperfect measure of the effectiveness of a multigrid cycle. Each cycle may have different effects on the various error modes (frequencies) of the current guess, all of which would be impossible to capture in a single number. Future work may expand the notion of an "optimal" set of sub-algorithms to include separate classes of algorithms that work best to reduce different types of error. Though such an approach could lead to a better final tuned algorithm, this extension would obviously make the auto-tuning process more complex.

The results given in Section 4.3 demonstrate that although this methodology is not exhaustive, it can be quite descriptive, discovering cycle shapes that are both unconventional and efficient. That section will present actual cycle shapes produced by the multigrid auto-tuner and show their performance compared to less sophisticated heuristics.

4.3 Results

This section presents the resulting multigrid algorithm cycle shapes produced by the auto-tuner and their observed performance when optimized on three parallel architectures designed for a variety of purposes: Intel Xeon E7340 server processor, AMD Opteron 2356 Barcelona server processor, and the Sun Fire T200 Niagara low power, high throughput server processor. These machines provided architectural diversity, allowing the results to show not only how auto-tuned multigrid cycles outperform reference multigrid algorithms, but also how the shape of optimal auto-tuned cycles can be dependent on the underlying machine architecture.

To the best of the author's knowledge, there are no standard data distributions currently in wide use for benchmarking multigrid solvers. Since random uniform matrix inputs result in uniform spectral error frequency characteristics, they were chosen to test the solvers presented in this chapter. The matrices have entries drawn from two different random distributions: 1) uniform over $[-2^{32}, 2^{32}]$ (unbiased), and 2) the same distribution shifted in the positive direction by 2^{31} (biased). The random entries were used to generate right-hand sides (b in Equation 4.1) and boundary conditions (boundaries of x) for the problem. Experiments were done specifying a small number of random point sources/sinks in the right-hand side, but since the observed results were similar to those found with the unbiased random distribution, those results are not included in interest of space. If one wishes to obtain tuned multigrid cycles for a different input distribution, the training should be done using that data distribution.

4.3.1 Auto-tuned multigrid cycle shapes

During the tuning process for the MULTIGRID-V_i algorithm presented in Section 4.2.3, the auto-tuner first computes the number of iterations needed for the SOR and RECURSE$_j$ choices before determining which is the fastest option to attain accuracy p_i for each input size. Representative training data is required to make this determination. Once the number of required iterations of each choice is known, the auto-tuner times each choice and chooses the fastest option.

Figures 4-4(a) and 4-4(b) show the traces of calls to the tuned MULTIGRID-V_4 algorithms for unbiased and biased uniform random inputs of size $N = 4097$, on the Intel machine. As you can see, the algorithm utilizes multiple accuracy levels throughout the call stack. In general, whenever greater accuracy is required by the tuned algorithm, it is achieved through some repetition of optimal substructures determined by the dynamic programming method. This may be easier to visualize by examining the resulting tuned cycles corresponding to the auto-tuned multigrid calls.

Figures 4-5(a) and 4-5(b) show some tuned "V-type" cycles created by the auto-tuner for unbiased and biased uniform random inputs of size $N = 2049$ on the AMD Opteron machine. The cycles are shown using standard multigrid notation with some extensions: The path of the algorithm progresses from left to right through time. As the path moves down, it represents a restriction to a coarser resolution, while paths up represent interpolations. Dots represent red-black SOR relaxations, solid horizontal arrows represent calls to the direct solver, and dashed horizontal arrows represent calls to the iterative solver.

As seen in the figure, a different cycle shape is used depending on what level of accuracy is required by the user. Cycles shown are tuned to produce final accuracy levels of $10, 10^3, 10^5$, and 10^7. The leverage of optimal subproblems is clearly seen in the common patterns that appear across cycles. Note that in Figure 4-5(b), the call to the direct solver in cycle i) occurs at level 4, while for the other three cycles, the direct call occurs at level 5. This is an example of the auto-tuner trading accuracy

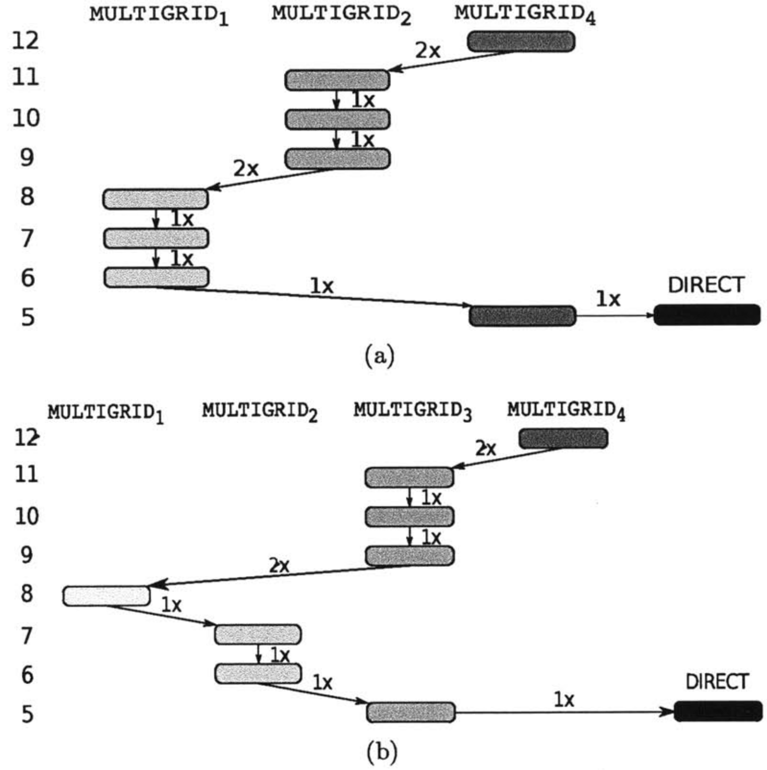
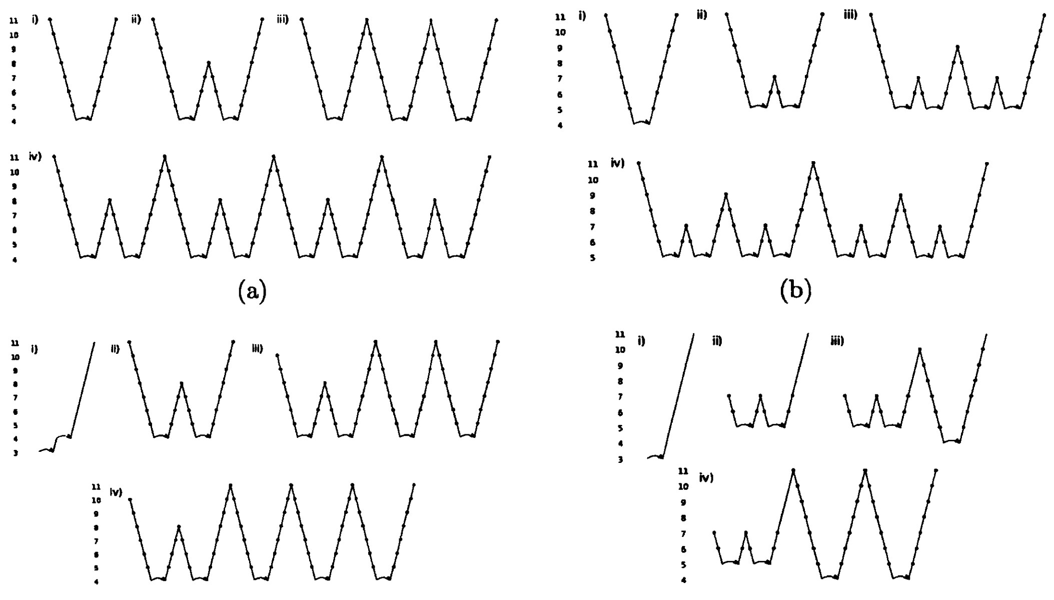


Figure 4-4: Call stacks generated by calls to auto-tuned MULTIGRID-V$_4$ for a) unbiased and b) biased random inputs of size $N = 4097$ on an Intel Xeon server. Discrete accuracies used during auto-tuning were $(p_i)_{i=1..5} = (10, 10^3, 10^5, 10^7, 10^9)$. The recursion level is displayed on the left, where the size of the grid at level k is 2^k+1. Note that each arrow connecting to a lower recursion level actually represents a call to RECURSE$_i$, which handles grid coarsening, followed by a call to MULTIGRID-V$_i$.

for performance while accounting for the accuracy requirements of the user.

Figures 4-5(c) and 4-5(d) show auto-tuned full multigrid cycles for unbiased and biased uniform random inputs of size $N = 2049$ on the AMD Opteron machine. Although similar substructures are shared between these cycles and the "V-type" cycles in 4-5(a) and 4-5(b), some of the expensive higher resolution relaxations are avoided by allowing work to occur at the coarser grids during the estimation phase of the full multigrid algorithm. The tuned full multigrid cycle in Figure 4-5(d)-iv) shows how the additional flexibility of using an estimation phase can dramatically alter the tuned cycle shape when compared to Figure 4-5(b)-iv).

It is important to realize that the call stacks in Figure 4-4 and the cycle shapes in



(a)

(b)

(c)

(d)

Figure 4-5: Optimized multigrid V (a and b) and full multigrid (c and d) cycles created by the auto-tuner for solving the 2D Poisson's equation on an input if size $N = 2049$. Subfigures a) and c) were trained on unbiased uniform random data, while b) and d) were trained on biased uniform random data. Cycles i), ii), iii), and iv), correspond to algorithms that yield accuracy levels of $10, 10^3, 10^5$, and 10^7, respectively. The solid arrows at the bottom of the cycles represent shortcut calls to the direct solver, while the dashed arrow in c)-i) represents an iterative solve using SOR. The dots present in the cycle represent single relaxations. Note that some paths in the full multigrid cycles skip relaxations while moving to a higher grid resolution. The recursion level is displayed on the left, where the size of the grid at level k is $2^k + 1$.

Figure 4-5 are all dependent on the specific situation at hand. They would all likely change were the auto-tuner run on other architectures, using different training data, or solving other multigrid problems. The flexibility to adapt to any of these changing variables by tuning over algorithmic choice is the auto-tuner's greatest strength.

4.3.2 Performance

This section will provide data showing the performance of the tuned multigrid Poisson's equation solver versus reference algorithms and heuristics. Test data was produced from the same distributions used for training described in Section 4.3.

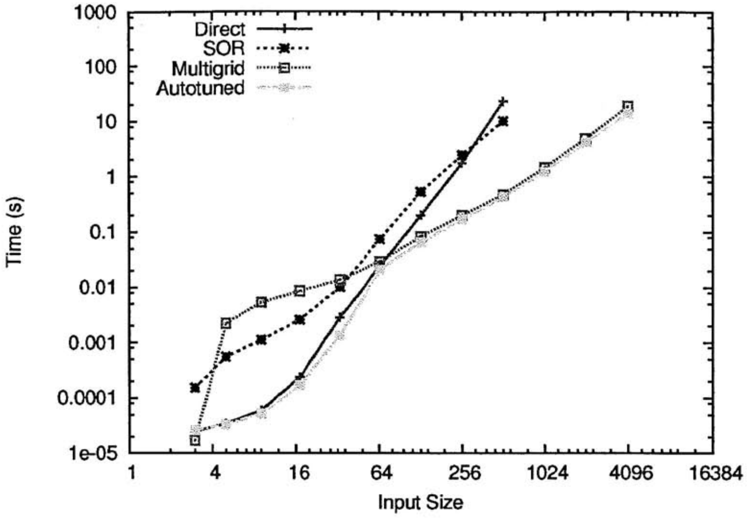


Figure 4-6: Performance for algorithms to solve Poisson's equation on unbiased uniform random data up to an accuracy of 10^9 using 8 cores. The basic direct and SOR algorithms as well as the standard V-cycle multigrid algorithm are all compared to the tuned multigrid algorithm. The iterated SOR algorithm uses the corresponding optimal weight ω_{opt} for each of the different input sizes.

Auto-tuned multigrid V algorithm

To demonstrate the effectiveness of the dynamic programming methodology, this section compares the auto-tuned MULTIGRID-V algorithm against more basic approaches to solving the 2D Poisson's equation to an accuracy of 10^9, including several multigrid variations. Results presented in the section were collected on the Intel Xeon server testbed machine.

Figure 4-6 shows the performance of the auto-tuned multigrid algorithm for accuracy 10^9 on unbiased uniform random inputs of different sizes. The auto-tuned algorithm uses internal accuracy levels of $\{10, 10^3, 10^5, 10^7, 10^9\}$ during its recursive calls. The figure compares the auto-tuned algorithm with the direct solver, iterated calls to SOR, and iterated calls to the reference V-cycle multigrid algorithm MULTIGRID-V-SIMPLE given in Section 4.2.1 (labeled Multigrid). Each of the iterative methods is run until an accuracy of at least 10^9 is achieved.

As to be expected, the auto-tuned algorithm outperforms all of the simple algorithms shown in Figure 4-6. At sizes greater than $N = 65$, the auto-tuned algorithm

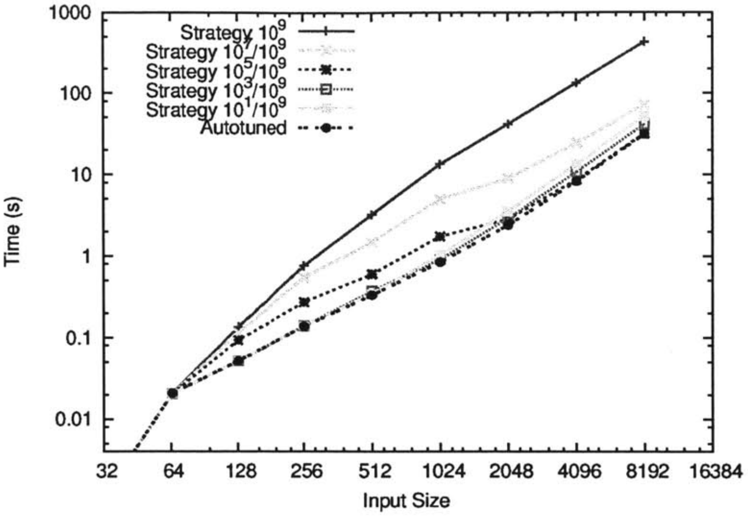


Figure 4-7: Performance for algorithms to solve Poisson's equation up to an accuracy of 10^9 using 8 cores. The auto-tuned multigrid algorithm is presented alongside various possible heuristics. The graph omits sizes less than $N = 65$ since all cases call the direct method for those inputs. To see the trends more clearly, Figure 4-8 shows the same data as this figure, but as ratios of times taken versus the auto-tuned algorithm.

performs slightly better than MULTIGRID-V-SIMPLE because it utilizes a more complex tuned strategy.

Figure 4-7 compares the tuned algorithm with various heuristics more complex than MULTIGRID-V-SIMPLE. The training data used in this graph was drawn from the biased uniform distribution. Strategy 10^9 refers to requiring an accuracy of 10^9 at each recursive level of multigrid until the base case direct method is called at $N = 65$. Strategies of the form $10^x/10^9$ refer to requiring an accuracy of 10^x at each recursive level below that of the input size, which requires an accuracy of 10^9. Thus, all strategies presented result in a final accuracy of 10^9; they differ only in what accuracies are required at lower recursion levels. All heuristic strategies call the direct method for smaller input sizes whenever it is more efficient to meet the accuracy requirement.

The lines in Figure 4-7 are somewhat close together and difficult to see on the logarithmic time scale, so Figure 4-8 presents the same data but showing the ratio of times taken versus the auto-tuned algorithm. One can more clearly see in this figure

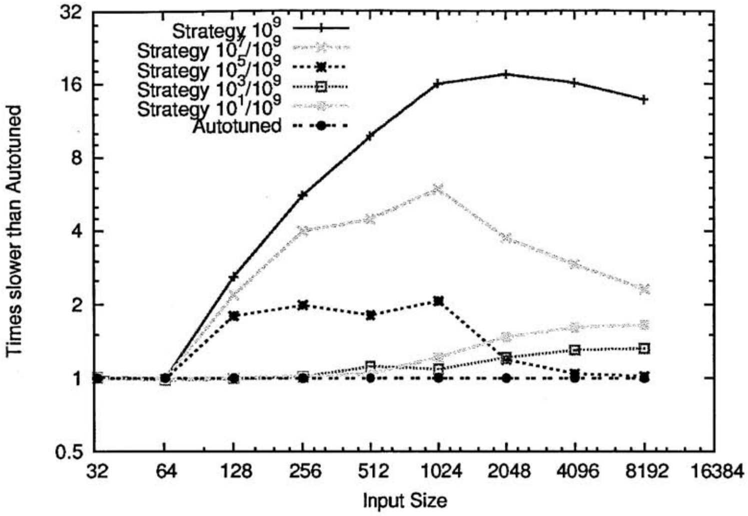


Figure 4-8: Speedup of tuned algorithm compared to various simple heuristics to solve Poisson's equation up to an accuracy of 10^9 using 8 cores. The data presented in this graph is the same as in Figure 4-7 except that the ratio of time taken versus the auto-tuned algorithm is plotted. Notice that as the problem size increases, the higher accuracy heuristics become more favored since they require fewer iterations at high resolution grid sizes.

that as the input size increases, the most efficient heuristic changes from Strategy $10^1/10^9$ to $10^3/10^9$ to $10^5/10^9$. The auto-tuner does better than just choosing the best from among these heuristics, since it can also tune the desired accuracy at each recursion level independently, allowing greater flexibility. This figure highlights the complexity of finding an optimal strategy and showcases the utility of an auto-tuner that can efficiently find this optimum.

Another big advantage this methodology provides is that it allows the user to easily produce optimized algorithms for both sequential performance and parallel performance. Figure 4-9 shows the speedup achieved by the tuned MULTIGRID-V algorithms on the Intel testbed machine, illustrating how the auto-tuner makes different choices depending on the number of cores available.

Auto-tuned full multigrid algorithm

In order to evaluate their performance on multiple architectures, the auto-tuned MULTIGRID-V and FULL-MULTIGRID algorithms were run on each platform for

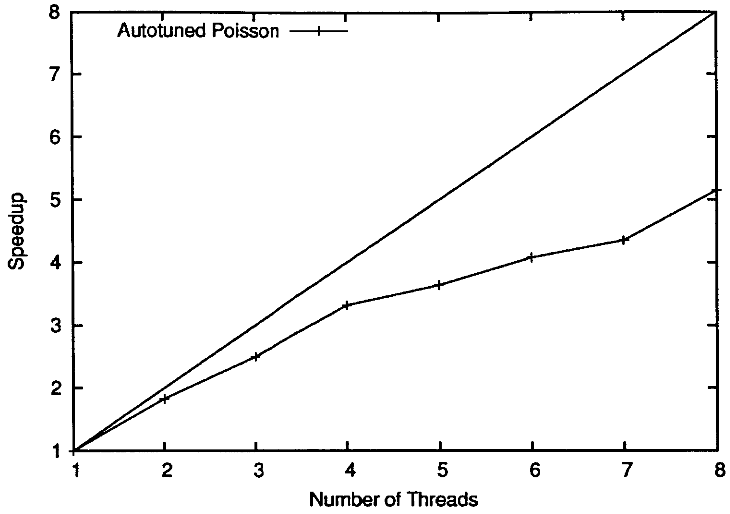


Figure 4-9: **Parallel scalability.** Speedup as more worker threads are added. Run on an 8 core (2 processor × 4 core) x86_64 Intel Xeon System.

problem sizes up to $N = 4097$ (up to 2049 on the Sun Niagara) for target accuracy levels of 10^5 and 10^9 alongside two reference algorithms: an iterated V cycle and a full multigrid algorithm. The reference V cycle algorithm runs standard V cycles until the accuracy target is reached, while the reference full multigrid algorithm runs a standard full multigrid cycle (as in Figure 4-3), then standard V cycles until the accuracy target is reached. These two reference algorithms were chosen since they are easy to understand and commonly implemented. From these starting points, performance tweaks are often manually applied to tailor the solver to each user's specific application domain.

Figure 4-10 shows the performance of both reference and auto-tuned multigrid algorithms for unbiased uniform random data relative to the reference iterated V-cycle algorithm on all three testbed machines. Figure 4-11 shows similar comparisons for biased uniform random data. The relative time (lower is better) to compute the solution up to an accuracy level of 10^5 is plotted against problem size.

On all three architectures, it can be seen that the auto-tuned algorithms provide an improvement over the reference algorithms' performances. There is an especially marked difference for small problem sizes due to the auto-tuned algorithms' use of the direct solve without incurring the overhead of recursion. Speedups relative to the reference full multigrid algorithm are also observed at higher problem sizes: e.g., for problem size $N = 2049$, speedups of 1.2x, 1.1x, and 1.8x are observed on the unbiased uniform test inputs, and 2.9x, 2.5x, and 1.8x on the biased uniform test inputs for the Intel, AMD, and Sun machines, respectively.

Figures 4-12 and 4-13 show similar performance comparisons, except to an accuracy level of 10^9. The auto-tuner had a more difficult time beating the reference full multigrid algorithm when training for both high accuracy and large size (greater than $N = 257$). For sizes greater than 257, auto-tuned performance is essentially tied with the reference full multigrid algorithm on the Intel and AMD machines, while improvements were still possible on the Sun machine. For input size $N = 2049$, a speedup of 1.9x relative to the reference full multigrid algorithm was observed on the Niagara for both input distributions. It is possible that performance gains are more difficult to achieve when solving for both high accuracy and size in some part due to a greater percentage of compute time being spent on unavoidable relaxations at the finest grid resolution.

Configuration Sensitivity

Figure 4-14 shows the sensitivity of the auto-tuned full multigrid algorithm's performance to the tuning configuration during a single stage of the auto-tuning search on the Harpertown platform. All configurations shown in the figure solve an input of size 1024×1024 up to an accuracy level of 10^7, but differ in the choices made during the recursive estimation and solve phases, which affect the resulting cycle shape. Measured performance is given relative to the best configuration found in a single stage of tuning.

The fastest configuration in the set is almost three times faster than the slowest configuration observed, and yields an approximate eight percent improvement in performance compared to the next best configuration. Slower configurations within the search space that either fall below a given threshold or are known to be inferior at particular input sizes are skipped by the auto-tuner to speed the tuning process and are not shown in the figure. Note that the search space is already pruned as described in Section 4.2.3, and the sensitivity of the performance to small changes in configuration is dependent on the granularity of those changes. However, it is clear from the figure that the algorithm's performance can vary substantially, even within a single stage of the auto-tuning search.

4.3.3 Effect of Architecture on Auto-tuning

Multicore architectures have drastically increased the processor design space resulting in a large variance in processors currently on the market. Such variance significantly hinders porting efforts of performance critical code.

Figure 4-15 shows the different optimized cycles chosen by the auto-tuner on the three testbed architectures. Though all cycles were tuned to yield the same accuracy level of 10^5, the auto-tuner found a different optimized cycle shape on each architecture. These differences take advantage of the specific characteristics of each machine. For example, the AMD and Sun machines recurse down to a coarse grid level of 2^4 versus 2^5 on the Intel machine. The AMD and Sun's cycles appear to make up for the reduced accuracy of the coarser direct solve by doing more relaxations at medium grid resolutions (levels 9 and 10).

The performance of tuned multigrid cycles appears to be quite sensitive to where the auto-tuning is performed in some cases. For example, the use of the auto-tuned full multigrid cycle for unbiased uniform inputs of size $N = 2049$ trained on the Sun Niagara but run on the Intel Xeon results in a 29% slowdown compared to the natively trained algorithm. Likewise, using the cycle trained on the Xeon results in a 79% slowdown compared to using the natively trained cycle on the Niagara.

4.4 Related Work

Some multigrid solvers using algorithmic choice have been presented in the past. SuperSolvers [6] is not an auto-tuner but rather a system for designing composite algorithms that leverage multiple algorithmic choices to solve sparse linear systems reliably. The approach given here differs in the use of tuning algorithmic choice at different levels of the multigrid hierarchy and the use of tuned subproblems during recursion. Unfortunately, no direct performance comparison was possible here due to the lack of availability of source code.

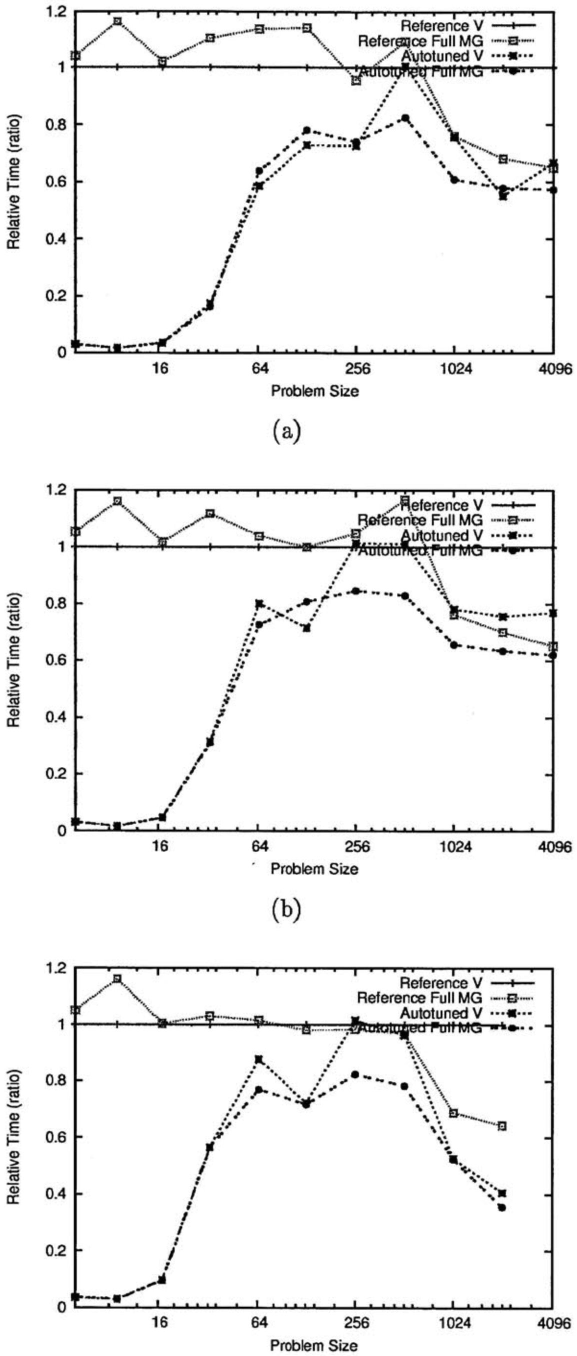
Cache-aware implementations of multigrid have also been developed. In [36], [34], and [28] optimizations improve cache utilization by reducing capacity and conflict misses during linear relaxation and inter-grid transfers. An auto-tuner was presented in [11] to automatically search the space of cache and memory optimizations for the relaxation step over a variety of hardware architectures. The optimizations presented in these related works are for the most part orthogonal to the approach taken here. There is no reason lower-level optimizations cannot be combined with algorithmic tuning at the level of cycle shape.

4.5 Future Work

An interesting direction this work could be taken in is toward the domain of tuning multi-level algorithms across distributed memory systems. The problem of discovering the best data layout and communications pattern for such a solver is very complex.

One specific problem this framework may help address is when to migrate data between machines. For example, one might want to use a smaller subset of machines once the problem is sufficiently small to reduce the surface area to volume ratio of each machine's working set. Doing so reduces the communications overhead of relaxations, but incurs the cost of the data transfer. One could extend the ideas presented here to produce "optimal" algorithms parameterized not just on size and accuracy, but also on data layout. The dynamic programming search could then take data transfers into account when comparing the costs of utilizing various "optimal" sub-algorithms, each with their own associated layouts.

Another direction to explore is the use of dynamic tuning where an algorithm has the ability to adapt during execution based on some features of the intermediate state. Such flexibility would allow the auto-tuned algorithm to classify inputs and intermediate states into different distribution classes and then switch between tuned versions of itself, providing better performance across a broader range of inputs. For example, it may be desirable to switch between cycle shapes during execution depending on the dominant error frequencies observed in the residual.

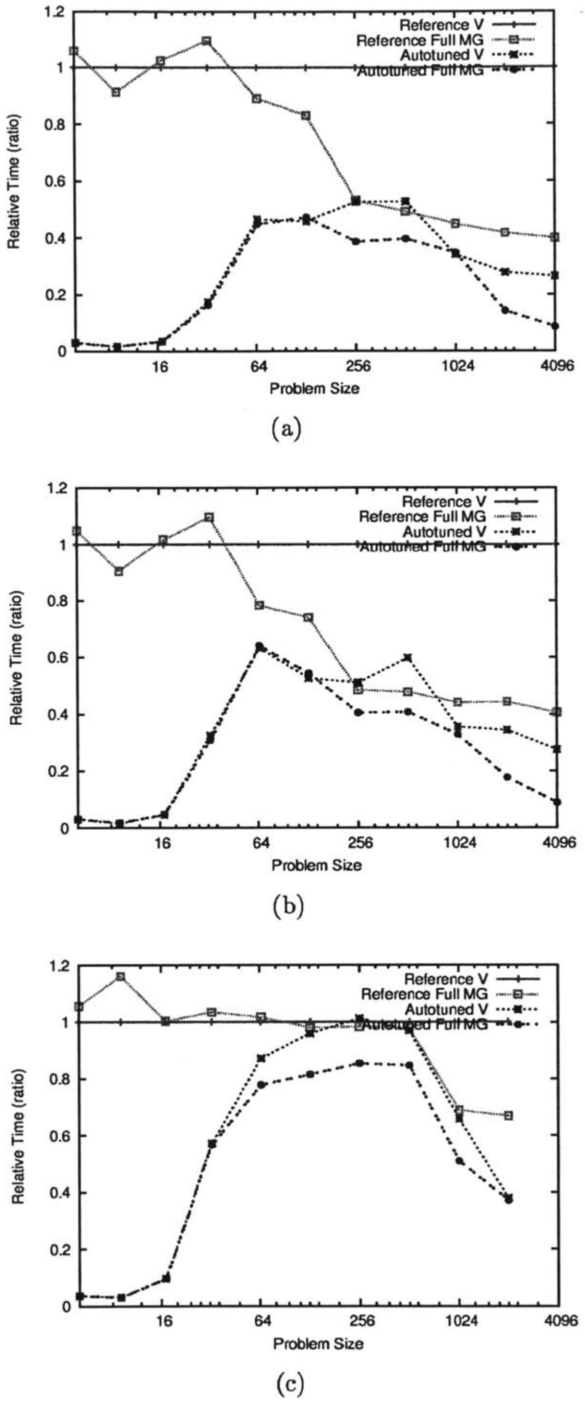


(a)

(b)

(c)

Figure 4-10: Relative performance of auto-tuned multigrid algorithms versus reference V cycle and full multigrid algorithms for solving the 2D Poisson's equation on unbiased uniform random data to an accuracy level of 10^5 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

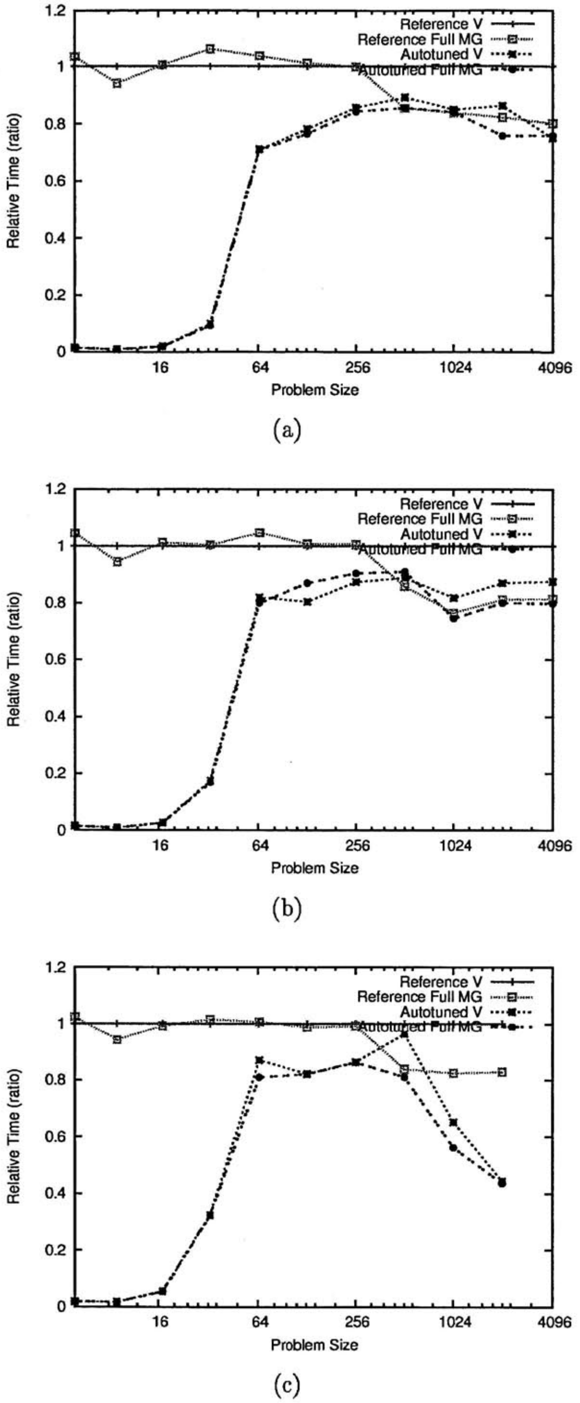


(a)

(b)

(c)

Figure 4-11: Relative performance of auto-tuned multigrid algorithms versus reference V cycle and full multigrid algorithms for solving the 2D Poisson's equation on biased uniform random data to an accuracy level of 10^5 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

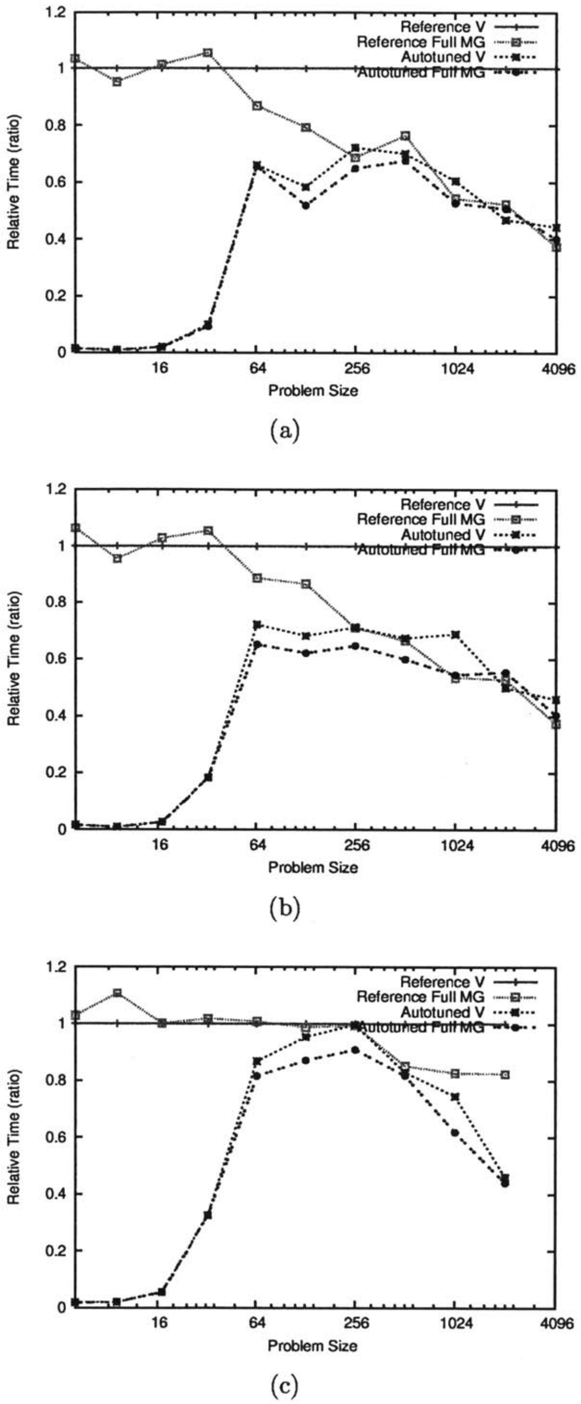


(a)

(b)

(c)

Figure 4-12: Relative performance of auto-tuned multigrid algorithms versus reference V cycle and full multigrid algorithms for solving the 2D Poisson's equation on unbiased uniform random data to an accuracy level of 10^9 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.



(a)

(b)

(c)

Figure 4-13: Relative performance of auto-tuned multigrid algorithms versus reference V cycle and full multigrid algorithms for solving the 2D Poisson's equation on biased uniform random data to an accuracy level of 10^9 on a) Intel Harpertown, b) AMD Barcelona, and c) Sun Niagara.

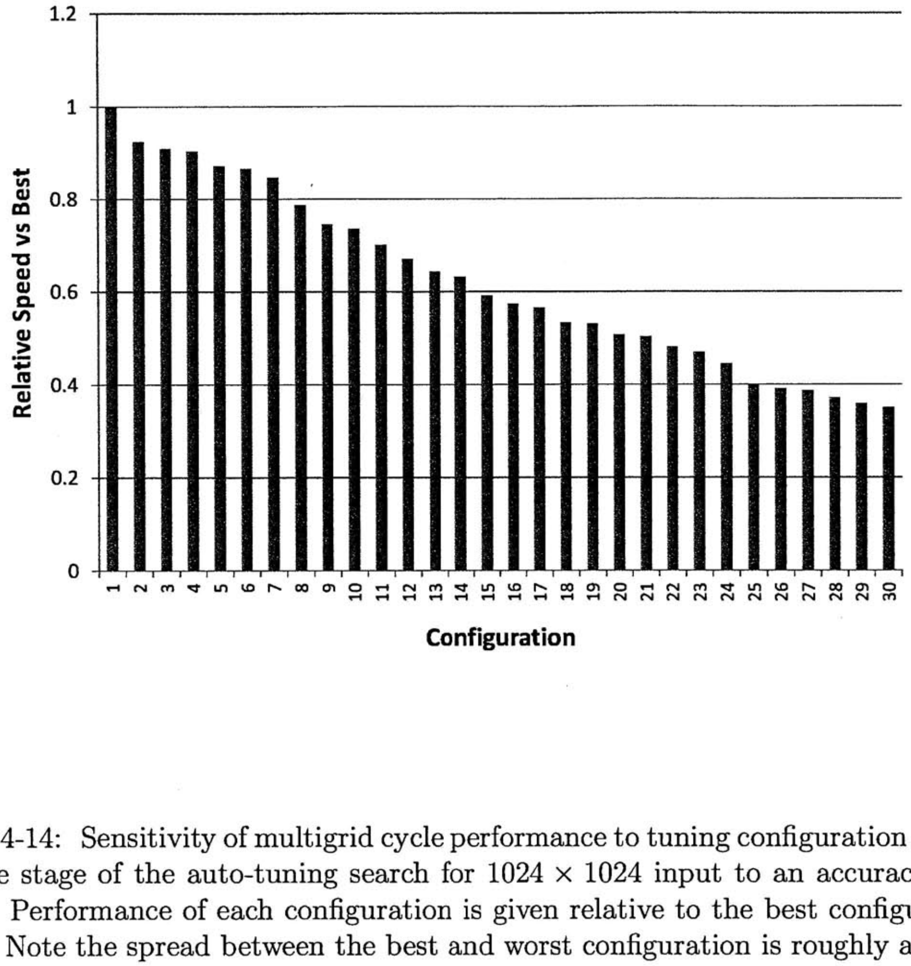


Figure 4-14: Sensitivity of multigrid cycle performance to tuning configuration during a single stage of the auto-tuning search for 1024 × 1024 input to an accuracy level of 10^7. Performance of each configuration is given relative to the best configuration found. Note the spread between the best and worst configuration is roughly a factor of three, showing diversity in the search space even within a single stage.

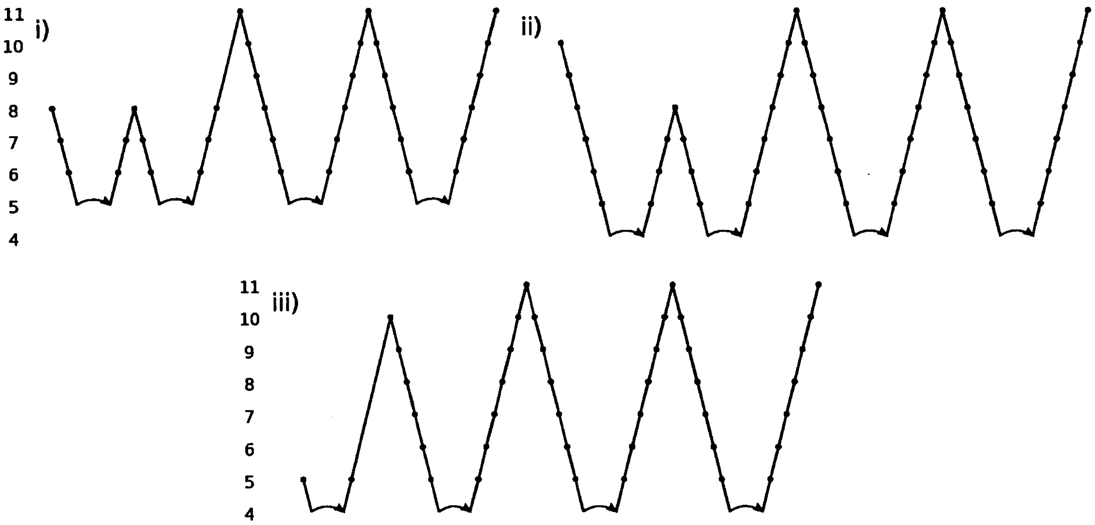


Figure 4-15: Comparison of tuned full multigrid cycles across machine architectures: i) Intel Harpertown, ii) AMD Barcelona, iii) Sun Niagara. All cycles solve the 2D Poisson's equation on unbiased uniform random input to an accuracy of 10^5 for an initial grid size of 2^{11}.

Bibliography

[1] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 20:1–20:12, New York, NY, USA, 2009. ACM.

[2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *PLDI '09: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[3] Jason Ansel, Maciej Pacula, Saman Amarasinghe, and Una-May O'Reilly. An efficient evolutionary algorithm for solving incrementally structured problems. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1699–1706, New York, NY, USA, 2011. ACM.

[4] K. Asanovic, R. Bodik, B. Catanzaro, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS, University of California, Berkeley, 2006.

[5] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04:Parallel Architectures and Compilation Techniques*, Washington, DC, 2004.

[6] Sanjukta Bhowmick, Padma Raghavan, and Keita Teranishi. A combinatorial scheme for developing efficient composite solvers. In *ICCS '02: Proceedings of*

the International Conference on Computational Science-Part II, pages 325–334, London, UK, 2002. Springer-Verlag.

[7] Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[8] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms.* PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[9] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

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

[11] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[12] James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, August 1997.

[13] Embeddable Common Lisp. http://ecls.sourceforge.net/.

[14] Bondhugula et al. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[15] OpenMP API Specification for Parallel Programming. http://openmp.org.

[16] Matteo Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 34(5):169–180, 1999.

[17] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, Jun 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[18] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[19] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *Workshop on Hot Topics in Parallelism*, March 2009.

[20] GreenFlash. http://www.lbl.gov/CS/html/greenflash.html.

[21] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[22] A Haidar, H Ltaief, A YarKhan, and J Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.

[23] R. Heikes and D.A. Randall. Numerical integration of the shallow-water equations of a twisted icosahedral grid. part i: basic design and results of tests. *Mon. Wea. Rev.*, 123:1862–1880, 1995.

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

[25] Thierry Joffrain. Parallel implementation of triangular solve, 1998.

[26] S. Kamil, C. Chan, S. Williams, et al. A generalized framework for auto-tuning stencil computations. In *Cray User Group*, 2009.

[27] S. Kamil, K. Datta, S. Williams, et al. Implicit and explicit optimizations for stencil computations. In *Workshop Memory Systems Performance and Correctness*, San Jose, CA, 2006.

[28] Markus Kowarschik and Christian Weiss. Dimepack – a cache-optimized multigrid library. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), volume I*, pages 425–430. CSREA, CSREA Press, 2001.

[29] LAPACK: Linear Algebra PACKage. http://www.netlib.org/lapack/.

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

[31] S. Mitra, S. C. Kothari, J. Cho, and A. Krishnaswamy. *ParAgent: A Domain-Specific Semi-automatic Parallelization Tool*, pages 141–148. Springer, 2000.

[32] M. Püschel, J. Moura, J. Johnson, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

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

[34] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[35] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[36] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):115–133, 2004.

[37] Armando Solar-Lezama, Gilad Arnold, Liviu, et al. Sketching stencils. In *International Conference on Programming Languages Design and Implementation (PLDI)*, June 2007.

[38] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[39] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[40] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *IPDPS Workshops'10*, pages 1–8, 2010.

[41] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[42] 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.

[43] R. Clint Whaley. Empirically tuning lapacks blocking factor for increased performance. In *Proceedings of the International Multiconference on Computer Science and Information Technology*, page 303310, 2008.

[44] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

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

[46] Qing Yi, Ken Kennedy, Haihang You, Keith Seymour, and Jack Dongarra. Automatic blocking of qr and lu factorizations for locality. In *Proceedings of the 2004 workshop on Memory system performance*, MSP '04, pages 12–22, New York, NY, USA, 2004. ACM.