Automatic Performance Tuning of Sparse Matrix Kernels

by

Richard Wilson Vuduc

B.S. (Cornell University) 1997

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor James W. Demmel, Chair
Professor Katherine A. Yelick
Professor Sanjay Govindjee

Fall 2003

The dissertation of Richard Wilson Vuduc is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2003

Automatic Performance Tuning of Sparse Matrix Kernels

Abstract

Automatic Performance Tuning of Sparse Matrix Kernels

by

Richard Wilson Vuduc

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor James W. Demmel, Chair

This dissertation presents an automated system to generate highly efficient, platform-adapted implementations of *sparse matrix kernels*. These computational kernels lie at the heart of diverse applications in scientific computing, engineering, economic modeling, and information retrieval, to name a few. Informally, sparse kernels are computational operations on matrices whose entries are mostly zero, so that operations with and storage of these zero elements may be eliminated. The challenge in developing high-performance implementations of such kernels is choosing the data structure and code that best exploits the structural properties of the matrix—generally unknown until application run-time—for high-performance on the underlying machine architecture (*e.g.*, memory hierarchy configuration and CPU pipeline structure). We show that conventional implementations of important sparse kernels like sparse matrix-vector multiply (SpMV) have historically run at 10% or less of peak machine speed on cache-based superscalar architectures. Our implementations of SpMV, automatically tuned using a methodology based on empirical-search, can by contrast achieve up to 31% of peak machine speed, and can be up to 4× faster.

Given a matrix, kernel, and machine, our approach to selecting a fast implementation consists of two steps: (1) we *identify* and *generate* a space of reasonable implementations, and then (2) *search* this space for the fastest one using a combination of heuristic models and actual experiments (*i.e.*, running and timing the code). We build on the SPARSITY system for generating highly-tuned implementations of the SpMV kernel $y \leftarrow y + Ax$, where A is a sparse matrix and x, y are dense vectors. We extend SPARSITY to support tuning for a variety of common non-zero patterns arising in practice, and for additional

kernels like sparse triangular solve (SpTS) and computation of $A^TA\cdot x$ (or $AA^T\cdot x$) and $A^\rho\cdot x$.

We develop new models to compute, for particular data structures and kernels, the best absolute performance (*e.g.*, Mflop/s) we might expect on a given matrix and machine. These *performance upper bounds* account for the cost of memory operations at all levels of the memory hierarchy, but assume ideal instruction scheduling and low-level tuning. We evaluate our performance with respect to such bounds, finding that the generated and tuned implementations of SpMV and SpTS achieve up to 75% of the performance bound. This finding places limits on the effectiveness of additional low-level tuning (*e.g.*, better instruction selection and scheduling). Instances in which we are further from the bounds (*e.g.*, for $A^TA\cdot x$) indicate new opportunities to close the gap by applying existing automatic low-level tuning technology. We also use these bounds to assess (partially) what architectures are good for kernels like SpMV. Among other conclusions, we find that performance improvements may be possible for SpMV (and other streaming applications) by ensuring strictly increasing cache line sizes in multi-level memory hierarchies.

The costs and steps of tuning imply changes to the design of sparse matrix libraries. We propose extensions to the recent standardized interface, the Sparse Basic Linear Algebra Subroutines (SpBLAS). We argue that such an extended interface complements existing approaches to sparse code generation, and furthermore is a suitable building block for widely-used higher-level scientific libraries and systems (*e.g.*, PETSc and MATLAB) to provide users with *high-performance* sparse kernels.

Looking toward future tuning systems, we consider an aspect of the tuning problem that is common to all current systems: the problem of *search*. Specifically, we pose two search-related problems. First, we consider the problem of stopping an exhaustive search while providing approximate bounds on the probability that an optimal implementation has been found. Second, we consider the problem of choosing at run-time one from among several possible implementations based on the run-time input. We formalize both problems in a manner amenable to attack by statistical modeling techniques. Our methods may potentially apply broadly to tuning systems for as yet unexplored domains.

Professor James W. Demmel
Dissertation Committee Chair

To my bringer of wing zings.

Contents

List of Figures

List of Tables

List of Symbols

BCSR	Block compressed sparse row storage format
BLAS	Basic Linear Algebra Subroutines
COO	Coordinate storage format
CSC	Compressed sparse columnw storage format
CSR	Compressed sparse row storage format
DIAG	Diagonal storage format
ELL	ELLPACK/ITPACK storage format
FEM	Finite element method
GEMM	Dense matrix-matrix multiply BLAS routine
GEMV	Dense matrix-vector multiply BLAS routine
JAD	Jagged diagonal storage format
MPI	Message Passing Interface
MSR	Modified sparse row storage format
RSDIAG	Row segmented diagonal format
SKY	Skyline storage format
TBCSR	Tiled blocked compressed sparse row format
TCSR	Tiled compressed sparse row format
VBR	Variable block row storage format
Sp$A\&A^T$	The sparse kernel $y \leftarrow y + Ax, z \leftarrow z + A^T w$
SpA^TA	The sparse kernel $y \leftarrow y + A^TA \cdot x$
SpAA^T	The sparse kernel $y \leftarrow y + AA^T \cdot x$
SpBLAS	Sparse Basic Linear Algebra Subroutines
SpMM	Sparse matrix-multiple vector multiply

SpMV	Sparse matrix-vector multiply
SpTS	Sparse triangular solve
SpTSM	Sparse triangular solve with multiple right-hand sides

Acknowledgments

First and foremost, I thank Jim Demmel for being a wonderfully supportive and engaging advisor. He is truly a model of what great scientists can ask, imagine, and achieve. I have yet to meet anyone with his patience, his attention to detail, or his seemingly infinite capacity for espresso-based drinks. I will miss our occasional chats when walking between Soda Hall and Brewed Awakening.

I also thank Kathy Yelick for being so very understanding, not to mention a good listener—she often understood my incoherent, ill-formed questions and statements long before I even finished them. I especially appreciate her guidance on and insights into the systems aspects of computer science.

Among other faculty at Berkeley, I thank Susan Graham, Ching-Hsui Cheng, and Sanjay Govindjee for taking the time to serve on my quals and dissertation committees. External to Cal, Zhaojun Bai has always made me feel as though my work were actually interesting. David Keyes has given me more of his time than I probably deserve, arranging in particular for access to Power4 and Ultra3 based machines (the latter due also in part to Barry Smith). For my first experience with research, I thank Bohdan Balko and Maile Fries at the Institute for Defense Analyses. I would have been prepared neither to experience the frequent joy nor the occasional pain of science without their early guidance.

Of my countless helpful colleagues, I especially admire Jeff Bilmes, Melody Ivory, and Remzi Arpaci-Dusseau for their amazing humor, intelligence, and hard work. They were my personal models of graduate student success. The inspiration for this dissertation specifically comes from early work by Jeff Bilmes and Krste Asanovic on PHiPAC, as well as pioneering work on SPARSITY by my friend and colleague, Eun-Jin Im.

For their pleasant distractions (*e.g.*, video games, Cowboy Bebop, snacks, dim sum, useless informationabout the 1980s, the occasional Birkball experience, and excursions in and around the greater Bay Area including the Lincoln Highway), I am especially grateful to Andy Begel, Jason Hong, (Honorary Brother) Benjamin Horowitz, Francis Li, and Jimmy Lin. Jason Riedy reminded me to eat lunch every now and again, introduced me to the historic Paramount Theater in Oakland, and made delicious cookies when I needed them most (namely, at my qualifying exam!). For being inquisitive colleagues and understanding companions, I offer additional thanks and best wishes to Mark Adams, Sharad Agarwal, David Bindel, Tzu-Yi Chen and Chris Umans, Inderjit Dhillon, Plamen Koev,

Osni Marques, Andrew Ng, David Oppenheimer, and Christof Vömel. I learned as much about academic life from this group as I did by experiencing it myself.

I was very fortunate to have worked alongside a number of incredibly talented undergraduate researchers. These individuals contributed enormously to the data collected in this dissertation: Michael deLorimier, Attila Gyulassy, Jen Hsu, Sriram Iyer, Shoaib Kamil, Ben Lee, Jin Moon, Rajesh Nishtala, and Tuyet-Linh Phan. I also owe much inspiration for learning and teaching to my first class of CS 61B students (Fall '97). I have a particularly deep respect for Sriram Iyer and Andy Atwal, from whom I learned much about the greeting card business and writing code in "the real world."

The Extended Buffalo Funk Family imparted much wisdom on me. I thank them for everything since our days on the Hill. And from way, way back in the day, I hope Sam Z., Ted S., and Kevin C. will forgive me for having not kept in better touch while I whittled away at this rather bloated tome.

Agata, you are my best friend and the love of my life. I could not have finished this "ridiculous thing" without your constant emotional support, confidence, understanding, tolerance, and faith in me. "I choo-choo-choose you."

Most of all, I owe an especially deep gratitude to Mom for her never-ending love, patience, and encouragement. She created the circumstances that allowed me to pursue my dreams when there should not have been a choice to do so. I will always remember the things she sacrificed to make a better life for me possible.

This research was supported in part by the National Science Foundation under NSF Cooperative Agreement No. ACI-9813362, NSF Cooperative Agreement No. ACI-9619020, the Department of Energy under DOE Grant No. DE-FC02-01ER25478, and a gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. Some experiments were performed at the High Performance Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory. Many experiments were completed thanks to the friendly staff at Caffé Strada, Brewed Awakening, the Free Speech Movement Café, the Hudson Bay Café, and the Coffee Bean. For help with collecting test matrices, I thank the following individuals: Mark Adams, Robert Taylor, Ali Pinar, Parry Husbands, Tzu-Yi Chen, Jason Riedy, Xiaoye Li, Pauletto Giorgio, Michel Juillard, Sanjay Govindjee, Z. Sroczynski, Osni Marques, Darren Wilkinson, Linda Garside, Ken Pearson, Roman Geus, David Bindel, and Florin Dobrian.

Chapter 1

Introduction

Contents

This dissertation presents an automated system to generate highly efficient, platform-adapted implementations of *sparse matrix kernels*. These kernels are frequently computational bottlenecks in diverse applications in scientific computing, engineering, economic modeling, and information retrieval (to name a few). However, the task of extracting near-peak performance from them on modern cache-based superscalar machines has proven to be extremely difficult. In practice, performance is a complicated function of many factors, including the underlying machine architecture, compiler technology, each kernel's instruction mix and memory access behavior, and the nature of the input data which might be known only at run-time. The gap in performance between tuned and untuned code can be severe. We show that one of the central sparse kernels, sparse matrix-vector multiply (SpMV), has historically run at 10% or less of machine peak. Our implementations, automatically tuned using a methodology based on empirical search, can by contrast achieve up to 31% of peak machine speed, and can be up to 4× faster.

A number of our findings run counter to what practitioners might reasonably expect, as the following examples suggest. Many sparse matrices from applications have a natural block structure that can be exploited by storing the matrix as a collection of blocks. For SpMV, doing so enhances spatial and temporal locality. However, Section 1.3 shows an example in which SpMV on a matrix with an "obvious" block structure nevertheless runs 2.6× faster using a different, non-obvious block structure. Furthermore, we show that if a matrix has no obvious block structure, SpMV can still go up to 2× faster by *imposing* block structure through explicitly stored zeros, even though doing so results in extra work (see Section 4.2). We can also create block structure by reordering rows and columns of the matrix in some cases, yielding 1.5× speedups (Section 5.3) for SpMV. Moreover, we can sometimes reorganize computations at the algorithmic level to improve temporal locality—for instance, by evaluating the composite operation $A^TA\cdot x$, where A is a sparse matrix and x is a dense vector, as a single operation instead of the usual 2 operations ($t \leftarrow A\cdot x$ followed by the transpose of A times t). When no natural blocking exists, this combined operation can go up to 1.6× faster, as we show in Chapter 7. We contribute automated techniques to decide when and how we can perform these kinds of optimizing transformations.

As the preceeding examples suggest, the key to achieving high-performance for sparse kernels is choosing appropriate data structure and code transformations that best exploit properties of both the underlying machine architecture and the structure of the *sparse matrix* (input data) which may be known only at run-time. Informally, a matrix is *sparse* if it consists of relatively few non-zeros. Storage of and computation with the zero entries can be eliminated by a judicious choice of data structure which stores just the non-zero entries, plus some additional indexing information to indicate which non-zeros have been stored. However, the price of a more compact representation in the sparse case, when compared to more familiar kernels like matrix multiply on dense matrices, is more computational overhead per non-zero entry—overheads in the form of extra instructions and, critically, extra memory accesses. In addition, memory references are often indirect and the memory access patterns irregular. The resulting performance behavior depends on the non-zero structure of a particular matrix, therefore making accurate static analysis or static performance modeling of sparse code difficult.

Indeed, we argue that algorithmic and low-level tuning are becoming more difficult over time, owing to the surprising performance behavior observed when running sparse kernels on modern machines (Sections 1.2–1.3 and Section 3.1.2). This difficulty is unfortu-

nate because the historical sparse kernel performance data which we present suggests that such tuning plays an effective and increasingly critical role in achieving high performance. Nevertheless, our thesis is that *we can ameliorate the difficulty of tuning by using a methodology based on automated empirical search* in which we automatically generate, model, and execute candidate implementations to find the one with the best performance.

The ultimate goal of our work is to generate sparse kernel implementations whose performance approaches that which might be achieved by the best *hand-tuned* code. Recent work on other computational kernels like matrix multiply and the fast Fourier transform (FFT), has shown that it is possible to build *automatic tuning systems* to generate implementations whose performance competes with, and even exceeds that of, the best hand-tuned code [46, 324, 123, 255, 225]. The lessons learned in building these systems have inspired our system. Moreover, they have motivated us to ask what the absolute limits of performance (Mflop/s) are for sparse kernels. Among other contributions, we develop theoretical models for a number of common sparse kernels that allow us to compute those limits and evaluate how closely we approach them.

Our system builds on an existing successful prototype, the SPARSITY system for generating highly tuned implementations of one important sparse kernel, SpMV [164]. We improve and extend the suite of existing SPARSITY optimization techniques, and furthermore apply these ideas to new sparse kernels. Inspired both by SPARSITY and the other automated tuning systems, our approach to choosing an efficient data structure and implementation, given a kernel, sparse matrix, and machine, consists of two steps. For each kernel, we

1. *identify* and *generate* spaces of reasonable implementations, and
2. *search* these spaces for the best implementation using a combination of heuristic models and experiments (*i.e.*, actually running and timing the code).

For a particular sparse kernel and matrix, the implementation space is a set of data structures and corresponding implementations (*i.e.*, code). Like the well-studied case of dense linear algebra, there are many reasonable ways to select and order machine language instructions statically. However, in contrast to the dense case, the number of possible non-zero structures (*sparsity patterns*)—and, therefore, the number of possible data structures to represent them—makes the implementation space much larger still. This dissertation addresses data structure selection by considering classes of data structures that capture the

most common kinds of non-zero structures; we then leverage the established ideas in code generation to consider highly efficient implementations.

We search the implementation space to choose the best implementation by evaluating heuristic models that combine benchmarking data with estimated properties of the matrix non-zero structure. The benchmarks, which consist primarily of executing each implementation (data structure and code) on synthetic matrices, need to be executed only once per machine. When the sparse matrix is known (in general, not until it is constructed at application run-time), we estimate certain performance-relevant structural properties of the matrix. The heuristic models combine these benchmarking and matrix-specific data to predict what data structure will yield the best performance. This approach uses a combination of modeling and experiments, as well as a mix of off-line and run-time techniques.

There are two aspects of the sparse kernel tuning problem which are beyond the scope of traditional compiler approaches. First, for a particular sparse matrix, we may choose a completely different data structure from the initial implementation; this new data structure may even alter the non-zero structure of the matrix by, for example, reordering the rows and columns of the matrix, or perhaps by choosing to store some zero values explicitly. These kinds of transformations, which we present in later chapters, depend on semantic kernel-specific information that cannot be justified using traditional static dependency analysis. Second, our approach to tuning identifies candidate implementations using models of both the kernel and run-time data. We would expect compilers built on current technology neither to identify such candidates automatically, nor posit the right models for choosing among these candidates. Third, searching has an associated cost which can be much longer than traditional compile-times. Knowing when such costs can be tolerated, particularly if they must be incurred at run-time, must be justified by expected application behavior.

The remainder of this chapter presents a summary of our contributions (Section 1.1) and more detailed support of our claim that algorithmic and low-level tuning are becoming increasingly important (Sections 1.2–1.3). We review the historical developments in both software and hardware leading up to our work, showing in particular that (1) "untuned" codes run at below 10% of machine peak and are steadily getting worse over time, but (2) conventional manual tuning significantly breaks the 10% barrier, highlighting the need for tuning to achieve better absolute performance, and furthermore (3) the gap between untuned and tuned codes is growing over time (Section 1.2). Moreover, we provide

the key intuition behind our approach by presenting the surprising quantitative results of an experiment in tuning SpMV (Section 1.3): we show instances on modern architectures in which observed performance behavior does not match what we would reasonably expect, and worse still, that performance behavior varies dramatically across platforms. These observations compose the central insight behind our claim that automatic performance tuning requires a platform-specific, search-based approach.

1.1 Contributions

Recall that the specific starting point of this dissertation is SPARSITY [167, 164, 166], which generates tuned implementations of the SpMV kernel, $y \leftarrow y + Ax$, where A is a sparse matrix and x, y are dense vectors. We improve and extend this work in the following ways:

- We consider an implementation space for SpMV that includes a variety of data structures beyond those originally proposed by SPARSITY (namely, splitting for multiple block substructure and diagonals, discussed in Chapter 5). We also present an improved heuristic for the tuning parameter selection for the so-called *register blocking* optimization (Chapter 3) [316].

- We apply these techniques to new sparse kernels, including

  - *sparse triangular solve (SpTS)* (Chapter 6): $y \leftarrow T^{-1}x$, where T is a sparse triangular matrix [319],
  - *multiplication by A^TA or AA^T* (Chapter 7): $y \leftarrow A^TAx$ or $y \leftarrow AA^Tx$ [317].
  - *applying powers of a matrix*, *i.e.*, computing $y \leftarrow A^kx$, where k is an positive integer.

- We develop new matrix- and architecture-specific *bounds* on performance, as a way to evaluate the quality of code being generated (Chapter 4). For example, sometimes these bounds show that our implementations are within, say, 75% of "optimal" in a sense to be made precise in Chapter 4. In short, the bounds guide us in understanding when we should expect the pay-offs from low-level tuning (*e.g.*, better instruction scheduling) to be significant [316]. Moreover, these bounds partially suggest what architectures are well-suited to sparse kernels. We also study architectural aspects

and implications, in particular, finding that strictly increasing line sizes could boost performance for SpMV, and streaming applications more generally.

- We examine the search problem as a problem in its own right (Chapter 9). We pose two problems that arise in the search process, and show how these problems are amenable to statistical modeling techniques [313]. Our techniques complement existing approaches to search, and will be broadly applicable to future tuning systems.

(Citations refer to earlier versions of this material; this dissertation provides additional details and updated results on several new architectures.)

1.2 Problem Context and History

Developments in automatic performance tuning have been driven both by trends in hardware architectures and the emergence of standardization in software libraries for computational kernels. Below, we provide a brief history of these technologies and trends that are central to our work. We explore connections to related work more deeply in subsequent chapters.

1.2.1 Hardware and software trends in sparse kernel performance

We begin by arguing that trends in SpMV performance suggest an increasing gap between what level of performance is possible when one relies solely on improvements in hardware and compiler technology compared to what is possible with software tuning. This gap motivates continued innovations in algorithmic and low-level tuning, in the spirit of automatic tuning systems like the one we are proposing for sparse kernels.

Although Moore's Law suggests that microprocessor transistor capacity—and hence performance—should double every 1.5–2 years,[1] the extent to which applications can realize the benefits of these improvements depends strongly on memory access patterns. Analysts have observed an exponentially increasing gap between the CPU cycle times and memory access latencies—this phenomenon is sometimes referred to as the *memory wall* [333], reflecting a lack of *balanced* machine designs [216, 65]. However, Ertl notes that simultaneous improvements in memory system design have for the time being still hidden this memory

[1] At least until physical (*e.g.*, thermal and atomic) barriers are encountered [229]: current projections suggest Moore's Law can be maintained at least until 2010 [188].

wall effect for at least some widely used applications [113]. Still, few argue with the idea that the gap exists and is worsening.

Figure 1.1 (*top*) shows where SpMV performance stands relative to Moore's Law. Specifically, we show SpMV speeds in Mflop/s over time based on studies conducted on a variety of architectures since 1987 [266, 51, 23, 88, 301, 326, 52, 129, 293, 197, 223, 167, 221, 323, 316]. (The tabulated data and remarks on methodology appear in Appendix A. Data points taken from the NAS CG benchmark [23] are handled specially, and marked in Figure 1.1 by an 'N'. See Appendix A.) We distinguish between vector processors (shown with solid red triangles) and microprocessors (shown with blue squares and green asterisks), since Moore's Law applies to microprocessors. Furthermore, for microprocessors we separate performance results into "reference" (or "untuned") implementations (shown by green asterisks), and "tuned" implementations (shown by hollow blue squares)—in most studies, authors report performance both before and after application of some proposed data structure or optimization technique.[2] Finally, through each set of points we show performance trend lines of the form $p(t) = p_0 2^{\frac{t}{\tau}}$, where t is time (in years since 1987), and p_0, τ are chosen by a linear regression fit of the data to $\log_2 p(t)$. In this model, τ is the *doubling-time*, *i.e.*, the period of time after which performance doubles. Below, we answer the question of how the doubling-time τ compares between untuned implementations, tuned implementations, and theoretical peak performance (Moore's Law).

First, observe that the untuned performance doubles approximately every two years (2.07), which is consistent with Moore's Law. Indeed, if one examines the doubling-time of the peak speeds for the machines shown in the plot, one finds that peak performance doubles every 1.94 years. SpMV is memory-bound since there are only two floating point operations (flops) worth of work for every floating point operand fetched from main memory. Thus, one possibly surprising aspect of the trend in untuned performance is that it scales according to Moore's Law. In fact, SpMV represents one type of workload, which we later show has a memory access pattern that is largely like streaming applications (Chapters 3–4). It may be that increasingly aggressive cache and memory system designs (*e.g.*, hardware prefetching, longer cache line sizes, support for larger numbers of outstanding misses) have helped to mask the effective latency of memory access for SpMV workloads, thereby helping SpMV scale with processor performance improvements [113].

[2] We do not separate by tuning in the vector processor case due to a lack of consistently separately reported performance results.

Second, observe that the doubling-time of the tuned implementations is slightly faster than the tuned doubling-time: 1.85 vs. 2.07. Indeed, the projected trend in 2003 suggests that tuned performance will be a factor of 2 higher than untuned performance, and that this gap will continue to grow over time.

The rate of improvement in the case of tuned codes is possible because SpMV performance is such a low fraction of absolute peak performance. In Figure 1.1 (*bottom*), we show the data points and trend lines of Figure 1.1 (*top*) normalized to machine peak. Untuned SpMV performance on microprocessors is typically below 10% of peak machine speed, and appears to be worsening gradually over time. Tuned SpMV codes can break the 10% barrier, and the gap in the trends between tuned and untuned implementations appears to be growing over time. Thus, tuning is becoming more important in better leveraging the improvements in hardware and compiler technology.

For comparison, we show the fraction of peak achieved by the Top 500 machines on the LINPACK benchmark, in which the performance of solving a *dense* system of linear equations is measured [100]. The median fraction (shown by a black horizontal line) is nearly 70% of peak, suggesting that the problem of implementing sparse kernels differs considerably from the problem of implementing dense codes dominated by matrix multiply.

1.2.2 Emergence of standard library interfaces

We view the emergence of standard library interfaces for computational kernels as a key development motivating work in automatic tuning. The following is a short history of a few of the ideas that have inspired our work.

One well-known example of a standard library interface is the Basic Linear Algebra Subroutines (BLAS) standard, which specifies an interface to common operations with dense matrices and vectors, such as computing dot-products, matrix multiply, triangular solve, among others [50, 101, 102, 203]. Highly-tuned implementations of the BLAS are available for nearly all major hardware platforms, courtesy of hardware vendors and dedicated implementors [158, 163, 169, 134, 156, 292]. Dense matrix multiply is among the most important of the BLAS kernels, both because 75% of more of peak speed can be achieved on most machines and because many of the BLAS routines can be formulated as calls to matrix multiply [182, 181]. In addition, higher-level computational kernels for dense linear algebra (*e.g.*, solving linear systems, computing eigenvectors and eigenvalues) have been

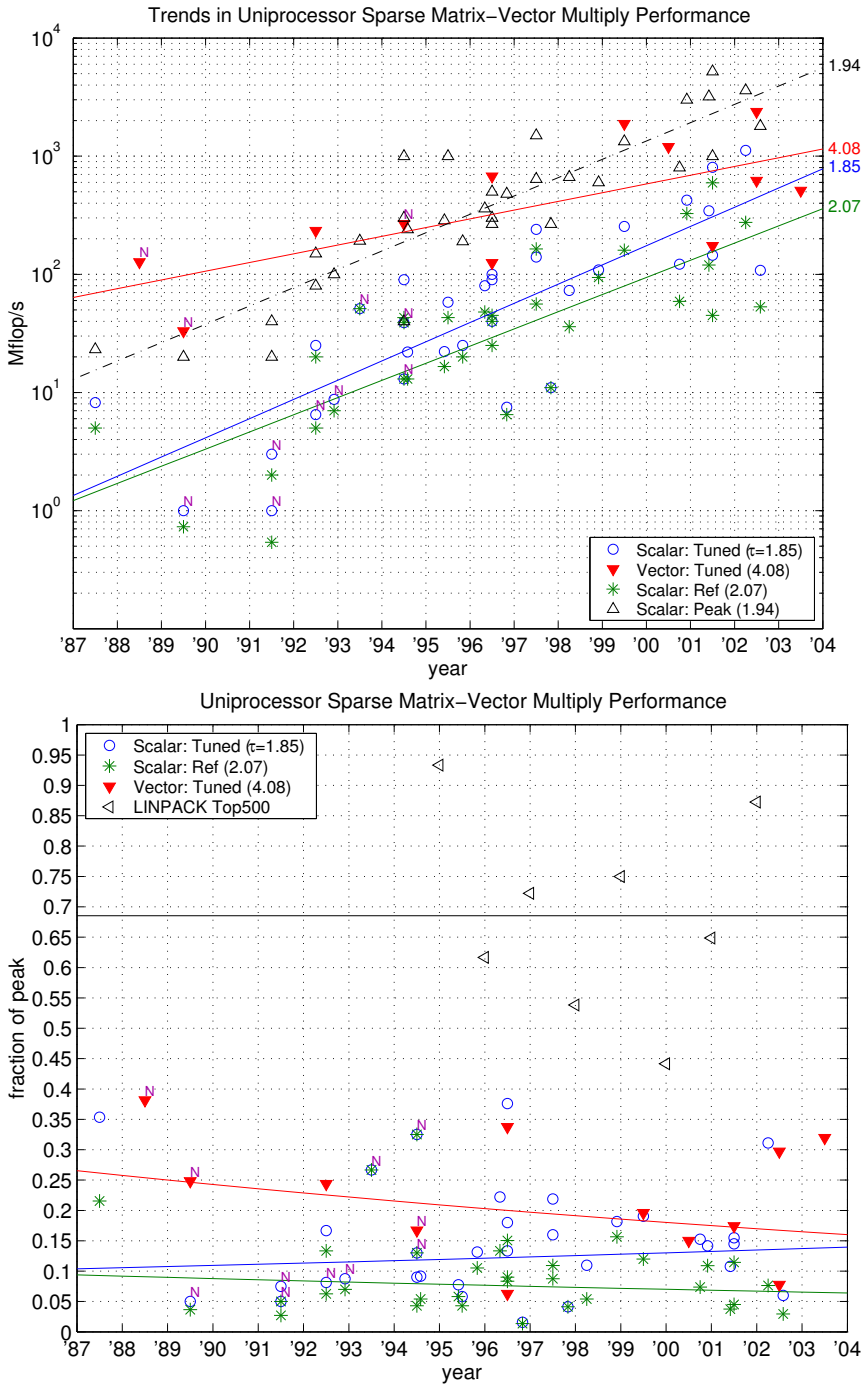

Figure 1.1: **SpMV performance trends across architectures and over time**. (*Top*) Reported single-processor absolute performance (Mflop/s) for SpMV since 1987. Tuned vector, and both tuned and untuned microprocessor data are shown, along with trend lines. The doubling-time (in years) is shown next to each trend line. (*Bottom*) Same data as above normalized as a fraction of processor peak speed. We show peak LINPACK (dense linear systems solve) performance for comparison. Untuned SpMV performance on microprocessors is largely clustered at 10% or less of uniprocessor peak speed. The gap between tuned (blue) and untuned (green) codes is growing over time.

developed on top of the BLAS in the widely-used LAPACK library [14]. Applications that can be expressed in terms of calls to the BLAS or LAPACK benefit both in performance, as well as in reduced costs of development and porting.

Motivated by the cost of vendor libraries and the increasing complexity of tuning even dense matrix multiply on a rapidly growing list of machine architectures, Bilmes, *et al.*, developed the PHiPAC system for automatically generating dense matrix multiply routines tuned to a given architecture [46]. PHiPAC originally proposed (1) a set of coding conventions, using C as a kind of high-level assembly language, to expose instruction-level parallelism and scheduling opportunities to the compiler, (2) various ways to write matrix multiply using these conventions, and (3) a prototype system to search over the space of these implementations. Whaley and Dongarra extended the scope of these ideas to the entire BLAS and to new architectures (notably, Intel x86 machines) in their ATLAS system [324], which is at present included in the commercial engineering package, MATLAB. Both systems report performance that is comparable, and often exceeding, that of hardware vendor-tuned libraries.

A number of libraries and interfaces have been developed for sparse kernels [267, 258, 116]. Indeed, the most recent revision of the BLAS standard specifies a Sparse Basic Linear Algebra Subroutines (SpBLAS) interface [108, 109]. Unlike prior proposals, SpBLAS hides the data structure from the user, thereby allowing the library implementation to make a decision about what data structure or particular implementation to use. The idea of automatic tuning as proposed in PHiPAC, as well as the then on-going development of SpBLAS provided some of the motivation for the Sparsity system. As mentioned before, the system proposed in this dissertation extends Sparsity. An important question which we later address is what implications the results of Sparsity and of this dissertation have for the design of sparse kernel library interfaces like the SpBLAS (Chapter 8).

Automatic tuning systems have been developed to support the growing list of performance-critical computational kernels. In the domain of signal and image processing, these systems include FFTW (developed at the same time as PHiPAC, and also presently distributed with MATLAB) [123, 122], SPIRAL [255], and UHFFT [224]. Automatic tuning is a sensible approach for signal processing due to the large number of embedded platforms—and, therefore, varied architectures—on which these kernels are likely to run. The Message Passing Interface (MPI) API [126] provides an interface for application level distributed parallel communications. As with the BLAS, MPI has been widely adopted and hardware

vendor implementations are not unusual. The MPI collective communication operations (*e.g.*, broadcast, scatter/gather) have been shown to be amenable to automatic tuning techniques [306]. In short, automatic tuning using empirical search has proven effective in a variety of domains, and has furthermore had considerable impact on commercial software. We review the scope and ideas of these systems in a subsequent chapter (Section 9.4), and also refer the interested reader to a recent surveys on the notions of *active libraries* [310] and *self-adapting numerical software* [103].

1.3 The Case for Search

One of the principal ideas of Figure 1.1 is that tuning has historically had an impact in sustaining performance trends; for SpMV performance in particular, tuning has helped maintain Moore's-Law-like behavior. In this section, we argue that the process of tuning can be surprisingly complex, thus motivating the need for *automated machine-specific search*. We provide just the necessary level of detail to understand the argument here, and defer a more extensive review of sparse matrix data structures to Chapter 2. (We return to the "case for searching" in Chapter 9, where we show that even for the well-studied kernel of dense matrix-matrix multiply, finding the best implementation can be like looking for a needle in a haystack.)

Figure 1.2 (*left*) shows an example of a sparse matrix that arises in a fluid flow simulation (Matrix `raefsky3`, from the University of Florida Sparse Matrix Collection [90]). This matrix has dimensions 21216×21216, but only contains 1.5 million non-zeros—only 0.33% of the total number of possible entries is non-zero. Roughly speaking, common sparse data structures store the matrix compactly by storing with each non-zero value an additional integer index to indicate which non-zero has been stored.

A closer inspection reveals that the matrix consists entirely of uniformly aligned, dense 8×8 blocks, as shown in Figure 1.2 (*right*). The classical technique of *register-level blocking* exploits this dense substructure by storing the matrix as a sequence of, say, 8×8 dense blocks, requiring only one index per block. This format reduces the index storage overhead by roughly a factor of 64. SpMV on a matrix stored in this blocked format proceeds block-by-block, where all reasonable implementations fully unroll the multiplication by each block, thereby exposing instruction-level parallelism and opportunities for register-level reuse. This implementation can be generalized straightforwardly to handle $r \times c$ blocks.

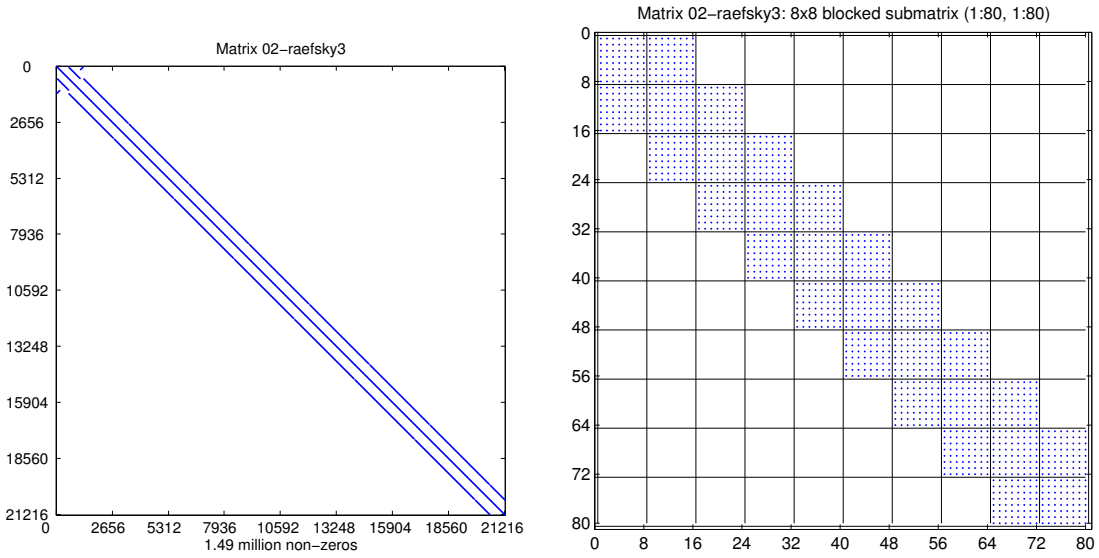

Figure 1.2: **Spy plot of sparse matrix `raefsky3`.** (*Left*) Sparse matrix `raefsky3`, arising from a finite element discretization of an object in a fluid flow simulation. This matrix has dimension 21216 and contains approximately 1.5 million non-zeros. (*Right*) Matrix `raefsky3` consists entirely of 8×8 dense blocks, uniformly aligned as shown in this 80×80 submatrix.

Most application developers expect this choice of storage format and corresponding SpMV implementation to be optimal for this kind of matrix.

In practice, performance behavior can be rather surprising. Consider an experiment in which we measure the performance in Mflop/s of the blocked SpMV implementation described, coded in C, for all $r \times c$ formats that would seem sensible for this matrix: $r, c \in \{1, 2, 4, 8\}$, for a total of 16 implementations in all. Figure 1.3 shows the observed performance on six different cache-based superscalar microprocessor platforms, where we have used the recent compilers and the most aggressive compilation options (the experimental setup is described in Appendix B). For each platform, each $r \times c$ implementation is both shaded by its performance in Mflop/s and labeled by its speedup relative to the conventional unblocked (or 1×1) case. We make the following observations.

- As we argue in more detail in Section 3.1, we might reasonably expect the 8×8 performance to be the best, with performance increasing smoothly as $r \times c$ increases. However, this behavior is only nearly exhibited on the Sun Ultra 2i platform [Figure 1.3 (*top-left*)], and, to a lesser extent, on the Pentium III-M [Figure 1.3 (*top-right*)]. Instead, 8×8 performance is roughly the same as or slower than 1×1 performance:

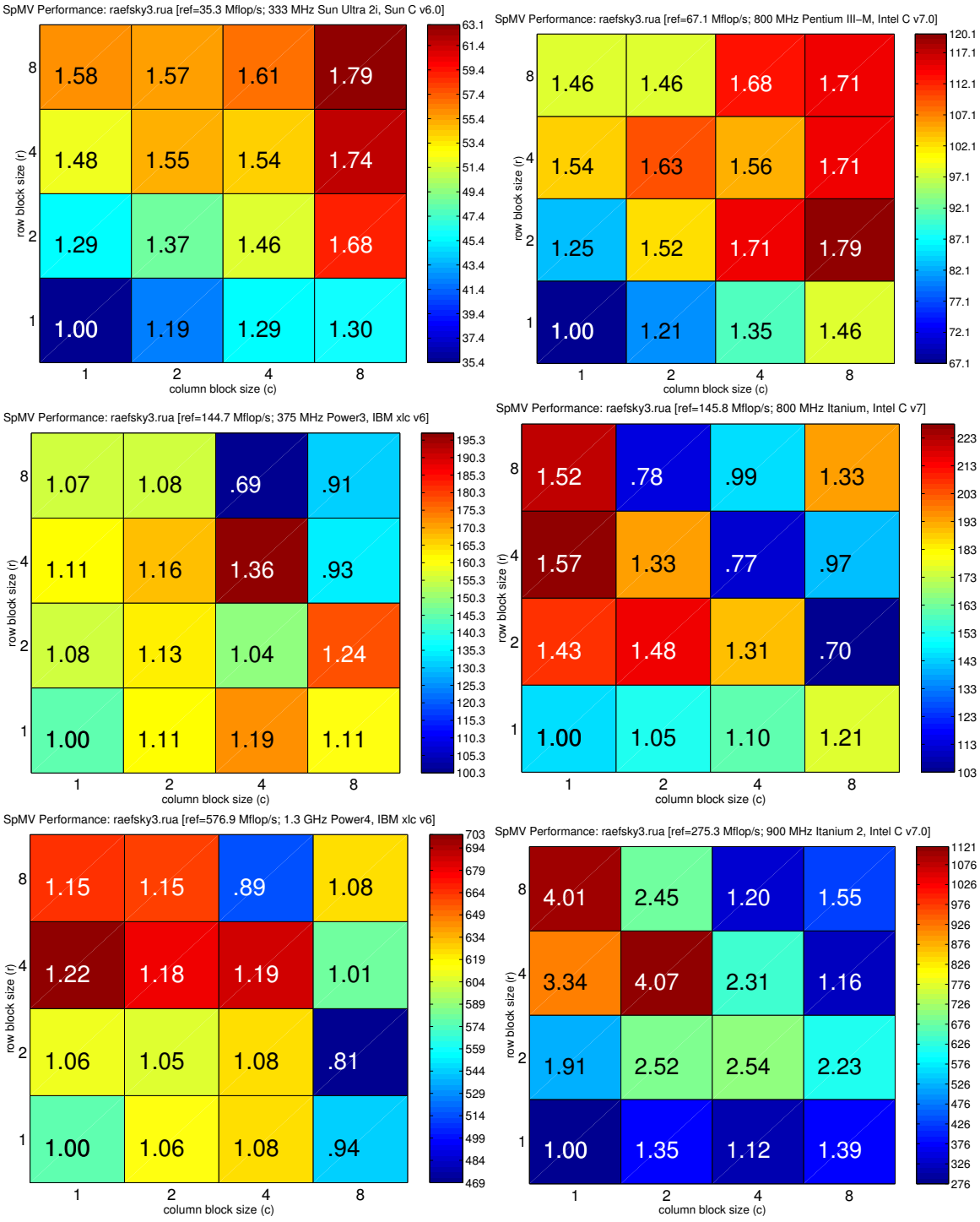

Figure 1.3: **The need for search: SpMV performance on `raefsky3` across six platforms**. Each $r \times c$ implementation is shaded by its performance in Mflop/s, and labeled by its speedup relative to the unblocked (1×1) code, where $r, c \in \{1, 2, 4, 8\}$. Although most users would expect 8×8 to be the fastest, this occurs on only one of the 6 platforms shown. See also Table 1.1. The platforms shown: (*top-left*) Sun Ultra 2i (*top-right*) Intel Pentium III-M (*middle-left*) IBM Power3 (*middle-right*) Intel Itanium (*bottom-left*) IBM Power4 (*bottom-right*) Intel Itanium 2

1.10× faster on the Power4 [Figure 1.3 (*bottom-left*)], but 9% worse on the Power3 [Figure 1.3 (*middle-left*)]. This behavior is not readily explained by register pressure issues: the Power3 and Power4 both have 32 floating point registers but the smallest 8×8 speedups, while the Pentium III-M and Ultra 2i have the fewest registers (8 and 16, respectively) but the best 8×8 speedups.

- Choosing a block size other than 8×8 can yield considerable performance improvements. For instance, 4×2 blocking on Itanium 2 is 2.6× faster than 8×8 blocking. Considerable gains over the 1×1 performance are possible by choosing just the right block size—here, from 1.22× up to 4.07×, or up to 31% of peak on the Itanium 2.

- Furthermore, the fraction of peak with just the right blocking can exceed the 5–10% of peak which is typical at 1×1.

- Performance can be a very irregular function of $r \times c$, and varies across platforms. It is not immediately obvious whether there is a simple analytical model that can capture this behavior. Furthermore, though not explicitly shown here, the performance depends on the structure of the matrix as well.

The characteristic irregularity appears to become worse over time, roughly speaking. The platforms in Figure 1.3 are arranged from left-to-right, top-to-bottom, by best SpMV performance achieved over all block sizes for this matrix. Furthermore, they happen to be arranged in nearly chronological order by year of release as shown in Table 1.1. Though we have argued that careful tuning is necessary to maintain performance growth similar to that of Moore's Law, the problem of tuning—even in a seemingly straightforward case—is a considerable and worsening challenge.

1.4 Summary, Scope, and Outline

Our central claim is that achieving and maintaining high performance over time for application-critical computational kernels, given the current trends in architecture and compiler development, requires a platform-specific, search-based approach. The idea of generating parameterized spaces of reasonable implementations, and then searching those spaces, is modeled on what practitioners do when hand-tuning code. Automating this process has proved enormously successful for dense linear algebra and signal processing. The intuition

	Release Year	Peak Mflop/s	1×1 Mflop/s	8×8 Mflop/s	Best Mflop/s	$r \times c$
Ultra 2i	1998	667	35	63	63	8×8
Pentium III-M	1999	800	67	115	120	2×8
Power3	1998	1500	145	132	196	4×4
Itanium	2001	3200	146	194	229	4×1
Power4	2001	5200	577	623	703	4×1
Itanium 2	2002	3600	275	427	1120	4×2

Table 1.1: **The need for search: Summary of SpMV performance**. This table summarizes the raw data shown in Figure 1.3. Achieving more than 5–10% of peak machine speed requires careful selection of the block size, which often does not match the expected optimal block size of 8×8 for this matrix.

that tuning is a challenging problem is captured by Figure 1.3, showing that performance behavior in a relatively simple example can be rather surprising.

The primary aim of this dissertation is to show why and how a search-based approach can be used to build an automatic tuning system for sparse matrix kernels, where a key factor is the choice of the right data structure to match both the matrix and the underlying machine architecture. We review commonly used ("classical") sparse matrix formats in Chapter 2, showing that on modern cache-based superscalar architectures, these formats do not perform well. In addition, we establish experimentally that the compressed sparse row (CSR) format is a reasonable default format.

Chapter 3 considers techniques for automatically choosing a good data structure for a given matrix. In particular, we present an improved heuristic for the *register blocking* optimization originally proposed for Sparsity. We refer to the original Sparsity heuristic as the Version 1 heuristic. Our new Version 2 heuristic replaces the previous version. We quantify the cost of this heuristic in order to understand how they can be integrated and used in a practical sparse kernel tuning system.

Tuning sparse matrix kernels requires careful consideration of both data structure and code generation issues. In Chapter 4, we present a detailed, theoretical performance analysis of SpMV that abstracts away issues of code generation and considers the data structure only. Specifically, we present a model of performance upper and lower bounds with two goals in mind. First, we use these bounds to identify data structure size as the primary performance bottleneck. Second, we compare these bounds to experimental results obtained using the Sparsity system to understand how well we can do in practice,

and identify where the opportunities for further performance enhancements lie. We show that SPARSITY-generated code can achieve 75% or more of the performance upper-bounds, placing a limit on low-level code tuning (*e.g.*, instruction selection and scheduling). A careful, detailed analysis of these results justifies the suite of techniques and ideas explored in the remainder of the dissertation. We further use these models to explore consequences for architectures. In particular, we show (1) the relationship between a measure of *machine balance* (ratio of peak flop rate to memory bandwidth) and achieved SpMV performance, and (2) the need for strictly increasing cache line sizes in multi-level memory hierarchies for SpMV and other streaming applications.

Chapter 5 considers some of the cases in which SPARSITY did not yield significant improvements, and proposes a variety of new techniques for SpMV. This chapter takes a "bottom-up" approach, presenting sample matrices that arise in practice, examining their non-zero structure, and showing how to attain high-performance by exploiting this structure. By exploiting multiple blocks and diagonals, we show that we can achieve speedups of up to 2× over a CSR implementation. We present a summary of our observations on additional techniques considered for inclusion in SPARSITY, providing a wealth of pointers to this work and commenting on current unresolved issues.

We show how the ideas of the previous chapters can be applied to SpTS in Chapter 6. By using a hybrid sparse/dense data structure, we show speedups of up to 1.8× on several current uniprocessor systems.

Recalling the limits placed on low-level tuning by the bounds of Chapter 4, Chapter 7 looks at higher-level sparse kernels which have more opportunities for reusing the elements of the sparse matrix. One such kernel is multiplication of a dense vector by A^TA or AA^T, where A is a sparse matrix. In principle, A can be brought through the memory hierarchy just once, in addition to being combined with the techniques of Chapter 5. This kernel arises in the inner-loop of methods for computing the singular value decomposition (SVD), and in interior point methods of linear programming and other optimization problems, and is thus of significant practical interest. We also present early results in tuning the application of powers of a sparse matrix ($A^{\rho} \cdot x$), based on a recent idea by Strout, *et al.* [288].

Collectively, these results have implications for the design and implementation of sparse matrix libraries. Recently, the BLAS standards committee revised the BLAS to include an interface to sparse matrix kernels (namely, SpMV, SpTS, and their multiple-

vector counterparts) [49]. In Chapter 8, we propose upwardly-compatible extensions to the standard to support tuning in the style this dissertation pursues. We argue that the SpBLAS standard, with our tuning extensions, is a suitable building block for integration with existing, widely-used libraries and systems that already have sparse kernel support, *e.g.*, PETSc [27, 26] and MATLAB [296].

Chapter 9 looks forward to future tuning systems, and considers an aspect of the tuning problem that is common to all systems: the problem of search. Specifically, we demonstrate techniques based on statistical modeling to tackle two search-related problems: (1) the problem of stopping an exhaustive search early with approximate bounds on the probability that an optimal implementation has been found, and (2) the problem of choosing one from among several possible implementations at run-time based on the run-time input. We pose these problems in very general terms to show how they can be applied in current and future tuning systems. We close this chapter with an extensive survey of related research on applying empirical-search techniques to a variety of kernels, compilers, and run-time systems.

Chapter 2

Basic Sparse Matrix Data Structures

Contents

The goal of this chapter is to provide the reader with some background in basic data structures (or storage formats) for storing sparse matrices, including experimental intuition about the performance of sparse matrix-vector multiply (SpMV) using these formats on modern machines built around cache-based superscalar microprocessors. In particular, the data we present in Section 2.2 lends support to our claim in Chapter 1 that untuned performance is typically 10% or less of machine peak. In addition, we use this data to show that the so-called compressed sparse row (CSR) format is a reasonable default data structure. Indeed, CSR is the default format in existing sparse matrix libraries like PETSc [27, 26] and

Saad's SPARSKIT library [267]. In subsequent chapters, performance in CSR format is the baseline against which we compare our tuned implementations. Readers familiar with the basic storage formats may wish to proceed directly to Section 2.2.

Throughout this chapter, we consider the SpMV operation $y \leftarrow y + Ax$, where A is an $m \times n$ sparse matrix with k non-zero elements, and x, y are dense vectors. We refer to x as the *source vector* and y as the *destination vector*. Algorithmically, SpMV can be defined as follows, where $a_{i,j}$ denotes the element of A at position (i, j):

$$\forall a_{i,j} \neq 0 : y_i \leftarrow y_i + a_{i,j} \cdot x_j \tag{2.1}$$

According to Equation (2.1), SpMV simply enumerates the non-zero elements of A, updating corresponding elements of y. Each $a_{i,j}$ is touched exactly once, and the only reuse possible on cache-based machines occurs in the accesses to the elements of x and y. If x and y—but not A—were completely contained in cache, then the time to execute SpMV would be dominated by the time to move the elements of A. Futhermore, there is very little work (only 2 flops per matrix element) to hide the cost of reading A. Thus, it becomes extremely important to reduce any overheads associated with storing the matrix.

Typical sparse matrix formats incur storage and instruction overheads per non-zero element, since information is needed to keep track of which non-zero values have been stored. One aim in selecting a data structure is to minimize these overheads. Roughly speaking, the way to reduce these overheads is to recognize patterns in the arrangement and numerical structure of non-zeros. Many of the basic formats surveyed in Section 2.1, as well as the aggressive structure-exploiting techniques of this dissertation, reduce the data structure overheads by making assumptions about the non-zero patterns.

The remainder of this chapter specifically discusses formats included in Saad's SPARSKIT library [267]. Our survey (Section 2.1) presents these formats, summarizing for each data structure both the total storage and the most natural way in which to enumerate the non-zeros for SpMV. In practice, users consider additional factors in choosing a sparse storage format, including the following:

- What operations need to be performed on the data structure? For example, sparse triangular solve (SpTS) has a particular value-dependency structure which requires enumerating non-zeros in a certain order. Sparse LU factorization requires a dynamic data structure that can support fast insertions of new non-zeros, to handle fill-in of elements created during the factorization.

- On what architecture will the application run? Several of the "basic" formats which we review are designed for vector architectures, in order to exploit the data parallelism inherent in Equation (2.1).

The interested reader can find discussion of these topics in presentations by Saad [267], in the "Templates" guide to solving linear systems [30], and in Duff's book on sparse LU factorization [107], among other sources [108, 266, 253].

2.1 Survey of Basic Data Structures

A "logical" (or "mathematical") matrix is represented by one or more "physical" one-dimensional arrays. For a variety of basic storage formats, we specify the size of these arrays and how logical elements $a_{i,j}$ map to array elements in Sections 2.1.2–2.1.4, below. Before these descriptions, we present a simple example of matrix-vector multiply in both dense and sparse storage formats, to show how sparse storage and instruction overheads manifest themselves. Moreover, this example contrasts a compiler approach to the problem of tuning sparse matrix-vector multiply (SpMV) performance to the "kernel-centric" approach adopted in this dissertation.

Although we take all physical arrays to be one-dimensional, for ease of notation we will sometimes also use two-dimensional arrays. We discuss the Fortran and C language conventions, known respectively as *column-major* and *row-major* formats, for mapping two-dimensional arrays to one-dimensional arrays in Section 2.1.2. When referring to a 2-D array, we will specify the assumed mapping if it is relevant. Otherwise, the mapping may be freely chosen by the implementation.

2.1.1 Example: Contrasting dense and sparse storage

When A is dense, a common storage scheme is *column-major* storage, in which A is represented by a physical array `val` of size $\texttt{stride} \cdot n$, where $\texttt{stride} \geq m$. The (i, j) element of A is stored at position $\texttt{val}[i + \texttt{stride} \cdot j]$, assuming the zero-based indexing convention: $0 \leq i < m$ and $0 \leq j < n$. This mapping allows random access to any matrix element. In addition, when $\texttt{stride} > m$, we can interpret A as a submatrix of some larger matrix. The conventional implementation of Equation (2.1) in pseudo-code is

```
1    for i = 0 to m − 1 do
2        for j = 0 to n − 1 do
3            y[i] ← y[i] + val[i + stride · j] · x[j]
```

All array accesses are affine functions of the iteration variables i and j, and standard compiler register- and cache-level tiling transformations may be applied statically [7].

In the case of sparse A, the simplest sparse format is coordinate (COO) format. We store A using three arrays, `val`, `rowind`, `colind`, each of length k. Array element `val`$[l]$ holds the value of the matrix element at row `rowind`$[l]$ and column `colind`$[l]$. The storage overhead is thus 2 integers per non-zero value. Matrix-vector multiply for COO matrices is

```
1    for l = 0 to k − 1 do
2            y[rowind[l]] ← y[rowind[l]] + val[l] · x[colind[l]]
```

It is much more difficult to tile this loop statically due to the indirect addressing through `rowind`, `colind`, shown in red. Furthermore, two additional load instructions are required per non-zero compared to the dense code.

From a compiler perspective, one possible way to eliminate these overheads is to inspect the indices at run-time, perhaps using inspector-executor and iteration reordering frameworks [270, 289], for instance. This dissertation approaches the problem in an alternative way. Based on run-time knowledge and estimation of the matrix pattern, and knowing that a particular sparse operation is being implemented, we allow ourselves to change the data structure completely, and even to change the *matrix* structure itself by, for instance, introducing explicit zeros.

2.1.2 Dense storage formats

The dense Basic Linear Algebra Subroutines (BLAS) standard supports a variety of schemes for mapping the 2-D structure of A into a 1-D linear sequence of memory addresses for the following classes of matrix structures:

- General, dense matrices: The number of non-zeros k is nearly or exactly equal to mn, and there is no assumed pattern in the non-zero values. For this class of structures, we describe *column-major*, *row-major*, *block-major*, and *recursive* storage formats below.
- Symmetric matrices: When A is dense but $A = A^T$, we only need to store approximately half of the matrix entries. We describe the *packed* storage format below. This

format is appropriate for other mathematical properties of A like skew symmetry ($A = -A^T$), or, when A is complex, Hermitian and skew Hermitian properties.

- Triangular matrices: When A is either lower or upper triangular, only half of the possible entries need to be stored. Like the symmetric case, we can use the packed storage format (Section 2.1.2).

- Band matrices: Only some number of consecutive diagonals above and below the main diagonal of A are non-zero. We describe a *band* storage format below.

Triangular and band matrices are structurally sparse (*i.e.*, typically consisting of mostly zero elements), but we include them in this discussion on "dense" storage formats because each of these formats allows efficient (constant time) random access to any non-zero element by simple indexing calculations.

General, dense storage

Column-major format is shown in Figure 2.1. A is represented by an array `val` of length $\texttt{stride} \cdot n$, where $\texttt{stride} \geq m$, and $a_{i,j}$ is stored in $\texttt{val}[i + \texttt{stride} \cdot j]$. Allowing `stride` to be greater than m allows A to be stored as a submatrix of some larger matrix. Column-major format is sometimes referred to as the Fortran language convention, since two-dimensional array declarations in Fortran are physically stored as one-dimensional arrays laid out as described above.

The C language convention, known as *row-major* format, is shown in Figure 2.2. In contrast to column-major format, consecutive elements within a *row* map to consecutive memory addresses: $a_{i,j}$ is stored in $\texttt{val}[i \cdot \texttt{stride} + j]$, where $\texttt{stride} \geq n$.

Wolf and Lam proposed a *copy optimization* in which the matrix storage is reorganized so that `R`×`C` submatrices of A are stored contiguously [201]. We show an example of such a copy-optimized, or *block-major* [325], format on a 6×6 matrix for $R = 3$ and $C = 2$ in Figure 2.3. Blocks within a block column are stored consecutively, and each `R`×`C` block may itself be stored in column- or row-major order. The rationale for the block-major format is to choose the block sizes so that blocks fit into cache, and then operate on blocks. Most implementations of the BLAS matrix multiply routine, GEMM, perform copying automatically for the user when there is sufficient storage and the cost of copying is small relative to the improvement in execution time [325].

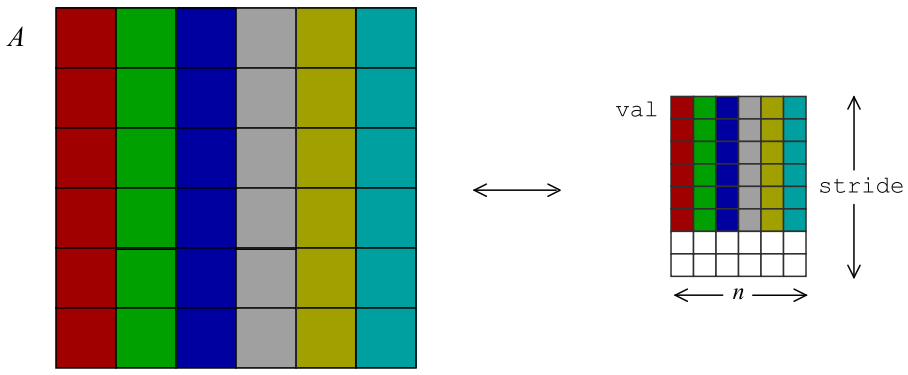

Figure 2.1: **Dense column-major format**. A is stored in an array `val` of size `stride`$\cdot n$, where elements of a given column are stored contiguously in `val`.

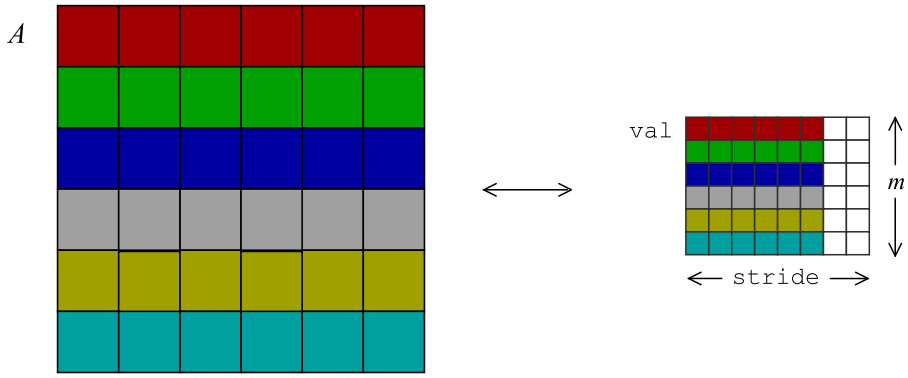

Figure 2.2: **Dense row-major format**. A is stored in an array `val` of size $m\times$`stride`, where elements of a given row are stored continguously in `val`.

All of the preceeding three storage formats allow fast random access to the matrix elements by relatively simple indexing calculations. Moreover, the column- and row-major formats permit random access to arbitrary contiguous submatrices, a property exploited in LAPACK. (If these properties are not essential to an application, a fourth class of *recursive* storage formats has been proposed for representing dense matrices. We defer a discussion of these formats to Section 2.3.)

Block-major format has been proposed specifically for cache-based architectures. Common dense linear algebra operations can be implemented efficiently on both superscalar and vector architectures, owing to the regularity of the indexing.

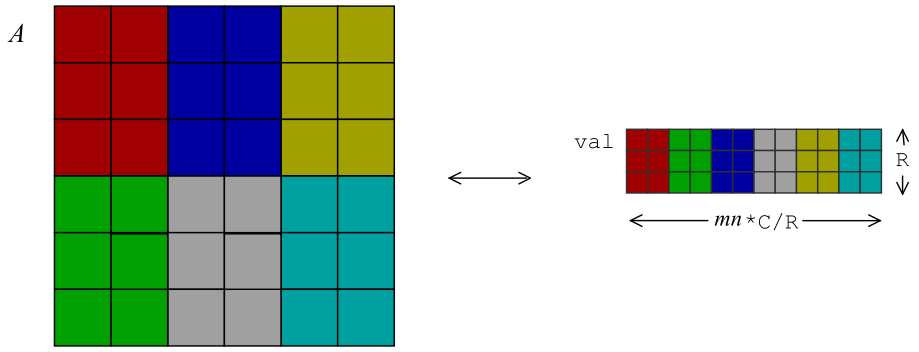

Figure 2.3: **Dense block-major format**. In block-major format, `R`×`C` submatrices are stored contiguously in blocks in `val`. Each block may furthermore be stored in any dense matrix format (*e.g.*, column major, row major, ...). Here, A is 6×6 and `R` = 3, `C` = 2.

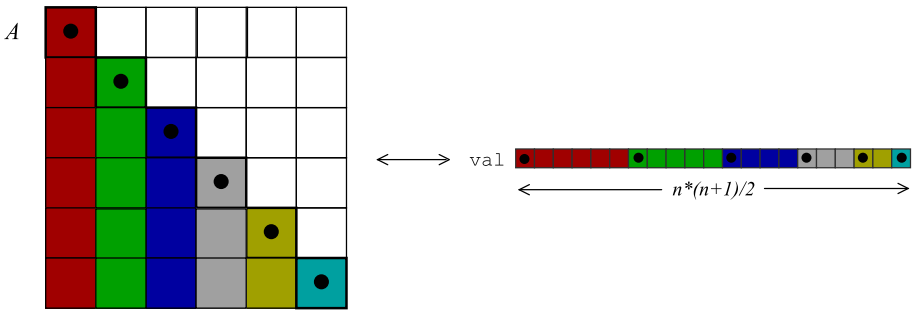

Figure 2.4: **Dense packed lower triangular (column major) format**. The packed storage format simply stores each column in sequence in a linear array `val`. Black dots indicate where diagonal elements of A map into `val`.

Packed triangular storage

If A is triangular, then we can eliminate storage of the zero part of the matrix using *packed* storage: columns of the triangle are stored contiguously. Figure 2.4 shows an example of a lower triangular $n \times n$ matrix A and its array representation `val`, where $a_{i,j}$ is stored in $\texttt{val}[i + nj - \frac{j(j-1)}{2}]$, for all $0 \leq j \leq i < n$. If A is upper triangular instead, then $a_{i,j}$ is stored in $\texttt{val}[i + \frac{j(j+1)}{2}]$, for all $0 \leq i \leq j < n$.

Computing the indices is more complex than for the general dense formats, but still allows random access at the cost of several integer multiplies and adds.

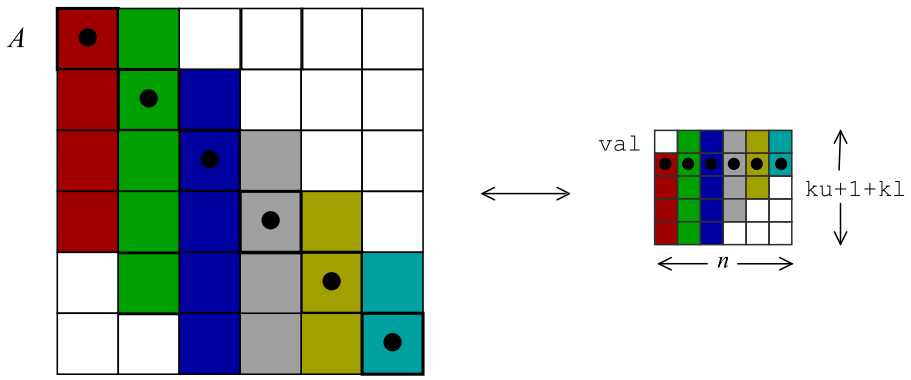

Figure 2.5: **Dense band format**. Here, a banded matrix A in dense band (column-major) format is stored in a $(\texttt{ku} + 1 + \texttt{kl}) \times n$ array `val`, where `ku` is the upper-bandwidth and `kl` is the lower-bandwidth. In this example, $\texttt{ku} = 1$ and $\texttt{kl} = 3$, and columns are stored contiguously in `val`. The main diagonal has been marked with solid black dots to show that diagonal elements lie in a row of `val`.

Band storage

Some matrices consist entirely of a dense region immediately above, below, and including the main diagonal. We refer to the number of full diagonals above the main diagonal as the *upper bandwidth*, and define the *lower bandwidth* similarly. A diagonal matrix would have both the upper and lower bandwidths equal to zero. We show an example of a band matrix in Figure 2.5, where the upper bandwidth $\texttt{ku} = 1$ and $\texttt{kl} = 3$.

In the BLAS, an $n \times n$ band matrix is represented by an array `val` containing $(\texttt{ku} + \texttt{kl} + 1) \cdot n$ elements. Each $a_{i,j}$ is stored in $\texttt{val}[\texttt{ku} + i - j + (\texttt{ku} + \texttt{kl} + 1) \cdot j]$, where $\max\{0, j - \texttt{ku}\} \leq i \leq \min\{j + \texttt{kl}, n - 1\}$. This format requires a storing a few unused elements, shown as empty (white) boxes in the example of Figure 2.5.

2.1.3 Sparse vector formats

Before discussing sparse matrix formats, we mention a common format for storing sparse vectors: the *compressed sparse vector* format, or simply *sparse vector* for short.[1]

An example of a sparse vector is shown in Figure 2.6. The non-zero elements of a

[1] A variety of other sparse 1-D formats are used in various contexts, mostly as temporary data structures in higher-level sparse algorithms such as LU factorization. We refer the interested reader elsewhere for details on these formats, which include sparse accumulators (SPA) [131] and alternate enumerators [254]. Both have been used to support dynamic changes to sparse matrix data structures.

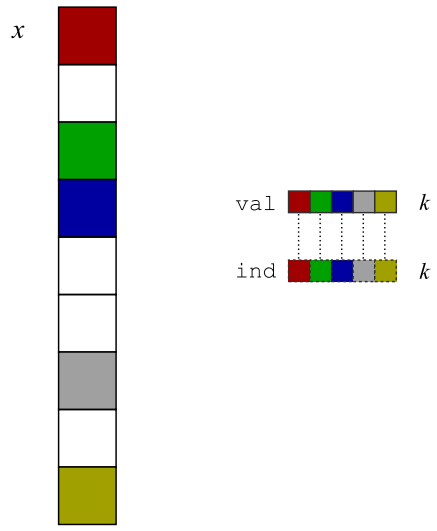

Figure 2.6: **Sparse vector example**. The sparse vector x (left) is represented by two packed arrays (right): a non-zero element $x_{\texttt{ind}[l]}$ of x is stored in $\texttt{val}[l]$, where $0 \leq l < k$. Here, the number of non-zeros k is 5.

vector x are stored packed continguously in an array `val`. An additional integer array `ind` stores the corresponding integer index for each non-zero value, *i.e.*, $\texttt{val}[l]$ is the non-zero value $x_{\texttt{ind}[l]}$. There is no explicit constraint on how non-zero elements are ordered in the physical representation. Therefore, random access to elements is not possible to implement more efficiently than by a linear scan of all stored non-zero elements.[2]

Some vector architectures include explicit support in the form of *gather* and *scatter* operations to help implement some basic kernels on sparse vectors.

2.1.4 Sparse matrix formats

A wide variety of sparse matrix formats are in use, each tailored to the particular application and matrix. In addition, several of the formats were created specifically with vector architectures in mind. Our discussion summarizes the formats supported by the public domain SPARSKIT library, which provides format conversion, SpMV, and sparse triangular solve (SpTS) support for many of these formats [267]. Specifically, we review the technical details of the following sparse matrix formats:

[2]Of course, binary search is possible if ordering is imposed.

- Coordinate (or triplet) format
- Compressed sparse stripe formats: compressed sparse row/column
- Diagonal format
- Modified sparse row format
- ELLPACK/ITPACK format
- Jagged diagonal format
- Skyline format
- Block compressed sparse stripe formats
- Variable block format

Coordinate storage

The *coordinate (COO) format* stores both the corresponding row and column index for each non-zero value. A typical implementation, uses three arrays `rowind`, `colind`, `val`, where `val`$[l]$ is the non-zero value at position (`rowind`$[l]$, `colind`$[l]$) of A. There are typically no ordering constraints imposed on the coordinates.

Compressed stripe storage

This class of formats includes the compressed sparse row (CSR) format and compressed sparse column (CSC) format. CSR can be viewed as a collection of sparse vectors (Section 2.1.3), allowing random access to entire rows (or, for CSC, columns) and efficient enumeration of non-zeros within each row (or column). Generally speaking, the compressed stripe formats are particularly well-suited to capturing general irregular structure, and tend to be poorly suited to vector architectures.

CSR is illustrated in Figure 2.7. The idea is to store each row (shown as elements having the same shading) as a sparse vector. A single value array `val` stores all sparse row vector values in order, and a corresponding array of integers `ind` stores the column indices. Each element `ptr`$[i]$ of a third array stores the offset within `val` and `ind` of row i. The array `ptr` has $m + 1$ elements, where the last element is equal to the number of non-zeros. This data structure allows random access to any row, and efficient enumeration of the elements

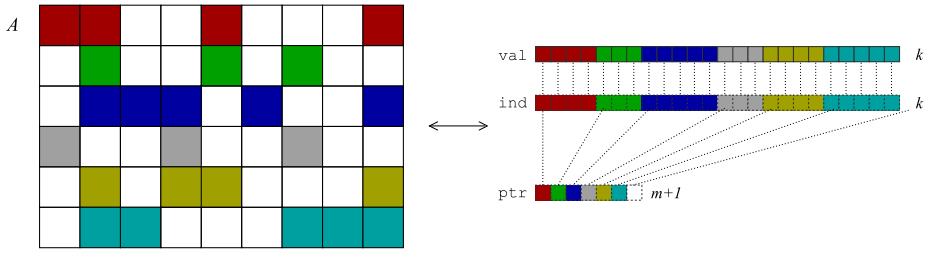

Figure 2.7: **Compressed sparse row (CSR) format**. The elements of each row of A are shaded using the same color. Each row of A is stored as a sparse vector, and all rows (*i.e.*, all sparse vectors) are stored contiguously in `val` and `ind`. The `ptr` array indicates where each sparse vector begins in `val` and `ind`.

of a given row. An implementation of sparse matrix-vector multiply (SpMV) using this format is as follows:

type `val` : **real**$[k]$
type `ind` : **int**$[k]$
type `ptr` : **int**$[m+1]$
1 **foreach** row i **do**
2 **for** $l =$ `ptr`$[i]$ **to** `ptr`$[i+1]-1$ **do**
3 `y`$[i] \leftarrow$ `y`$[i] +$ `val`$[l] \cdot$ `x`[`ind`$[l]$]

In the limit of $k \gg m$, there is only 1 integer index per non-zero instead of the 2 in the COO implementation.

This implementation exposes the potential reuse of elements of y, since `y`$[i]$ can be kept in a register during the execution of the inner-most loop. In addition, since `val` and `ind` are accessed with unit stride, it is possible to prefetch their values.[3] However, other loop-level transformations are more difficult to apply effectively since the loop bounds cannot be predicted statically. For instance, it is possible to unroll either loop, but the right unrolling depth will depend on the number of non-zeros `ptr`$[i+1]-$`ptr`$[i]$ in each row i. Moreover, to tile accesses to x for registers or caches requires knowledge of the run-time values of `ind`.

CSC is similar to CSR, except we store each column as a sparse vector as shown Figure 2.8. The corresponding SpMV code is as follows:

[3]Indeed, the IBM and Intel compilers listed in Appendix B insert prefetch instructions on elements of these arrays.

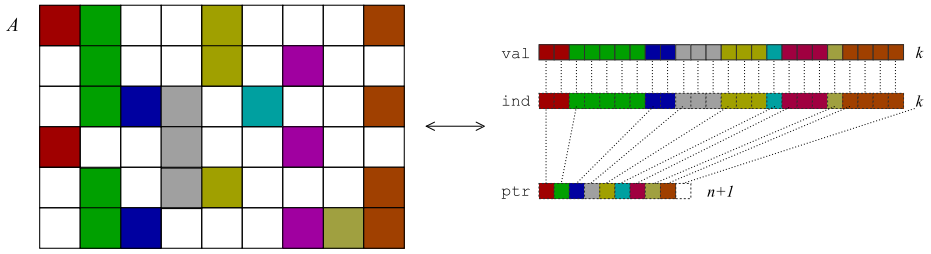

Figure 2.8: **Compressed sparse column (CSC) format**. The elements of each column of A are shaded using the same color. Each column of A is stored as a sparse vector, and all columns (*i.e.*, all sparse vectors) are stored contiguously in `val` and `ind`. The `ptr` array indicates where each sparse vector begins in `val` and `ind`.

```
    type val : real[k]
    type ind : int[k]
    type ptr : int[n + 1]
1   foreach column j do
2       for l = ptr[j] to ptr[j + 1] − 1 do
3           y[ind[l]] ← y[ind[l]] + val[l] · x[j]
```

The CSC implementation contains dependencies among accesses to `y`, which complicates static analyses to detect parallelism.

The compressed sparse stripe formats can be generalized to store sparse vectors along diagonals as well, but we do not know of any actual implementations in use.

Diagonal format

The diagonal (DIAG) format is designed for the important class of sparse matrices consisting of some number of full non-zero diagonals.[4] Since each diagonal is assumed to be full, we only need to store one index for each non-zero diagonal, and no indices for the individual non-zero elements. Furthermore, common operations with diagonals are amenable to efficient implementation on vector architectures, provided the diagonals stored are sufficiently long. DIAG generalizes the dense band format (Section 2.1.2) by allowing arbitrary diagonals to be specified, not just diagonals adjacent to the main diagonal.

We *number* diagonals according to the following convention. A non-zero element

[4] A common source of such matrices arise in stencil calculations.

at position (i, j) lies on diagonal number $j - i$. The main diagonal is numbered 0, upper-diagonals have positive numbers, and lower-diagonals have negative numbers.

We illustrate DIAG in Figure 2.9, where we show a square matrix A ($m = n = 7$) with five diagonals. Let s denote the number of diagonals; in this example, $s = 5$. In DIAG, all of the diagonals are stored in a 2-D array `val` of size $m \times s$, along with an additional 1-D array `diag_num` of length s to indicate the number of the diagonal stored in each column. Since upper- and lower-diagonals will have a length less than m, some elements of `val` will be unused. The usual convention for DIAG format is to store an upper-diagonal starting in row 0 of `val`, and to store a lower-diagonal d starting in row $-d$. The ordering of diagonals among the columns of `val` is arbitrary. The standard implementation of SpMV in DIAG format is as follows:

type `val` : **real**$[m \times s]$
type `diag_num` : **int**$[s]$
1 **for** $p = 0$ **to** $s - 1$ **do**
2 $d \leftarrow$ `diag_num`$[p]$
3 **for** $i = \max(0, -d)$ **to** $m - \max(d, 0) - 1$ **do**
4 `y`$[i] \leftarrow$ `y`$[i] +$ `val`$[i, p] \cdot$ `x`$[d + i]$

The inner-most loop can be vectorized.

Modified compressed sparse row format

The modified sparse row (MSR) format is a variation on CSR in which an additional array is used to store the main diagonal, which is typically full, therefore incurring no index overhead for the diagonal. Figure 2.10 shows an example of a matrix in MSR format. Despite the index overhead savings along the diagonal, the total storage is generally comparable between CSR and MSR formats unless the number of non-zeros per row is small.

ELLPACK/ITPACK format

The ELLPACK/ITPACK (ELL) format, originally used in the ELLPACK and ITPACK sparse iterative solver software libraries [138, 262], is best suited to matrices in which most rows of A have the same number of non-zeros. Efficient implementations of SpMV on vector architectures were the original motivation for this format [336]. ELL is the base format in IBM's sparse matrix library, ISSL [163].

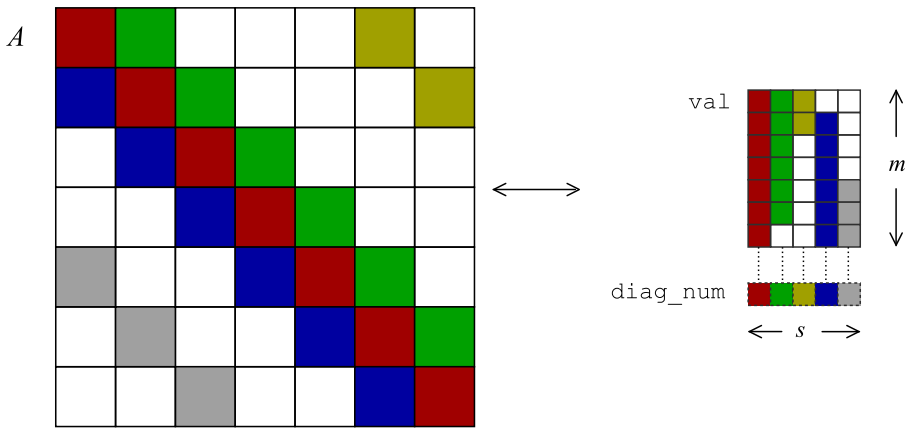

Figure 2.9: **Diagonal format (DIAG)**. Elements of each diagonal of A are stored continguously in a column of `val`. Upper-diagonals are stored beginning in the first row of `val`, and lower-diagonals are stored ending at in the last row of `val`. Each element `diag_num`$[l]$ of `diag_num` indicates which diagonal is stored in column l of `val`.

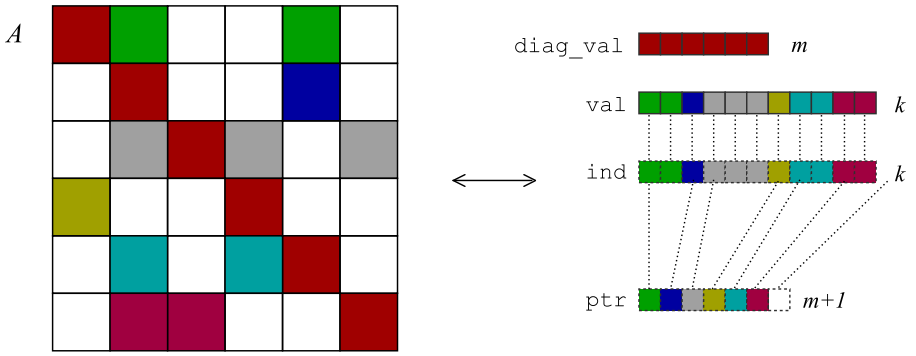

Figure 2.10: **Modified sparse row (MSR) format**. MSR is identifical to CSR, except that the diagonal elements are stored separately in a dense array (`diag_val`) where no indexing information need be stored. The off-diagonal elements are stored in CSR format.

Figure 2.11 shows an example of ELL. If the maximum number of non-zeros in any row is s, then ELL stores the non-zero values of A in an 2-D array `val` of size $m \times s$, and a corresponding 2-D array of indices `ind`. The elements of each row i are packed consecutively in row i of `val`. If a row i has fewer than s non-zeros in it, then the remaining elements of row i in both `val` and `ind` are padded with zero elements from the row. This convention implies that both extra storage of explicit zeros and extra load and floating point operations

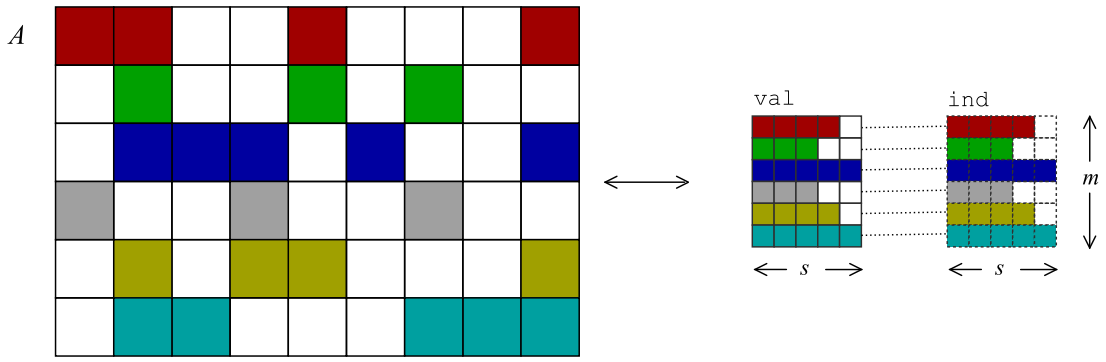

Figure 2.11: **ELLPACK/ITPACK format**. Non-zero values of A are stored by row in an $m \times s$ array `val`, where s is the maximum number of non-zeros in any row of A. For each $\texttt{val}[i,j]$, the corresponding column index is given by $\texttt{ind}[i,j]$.

on those zeros will be performed. Thus, this format best supports matrices in which the number of non-zeros in all rows is close to s. SpMV in this format is as follows:

type `val` : **real**$[m \times s]$
type `ind` : **int**$[m \times s]$
1 **foreach** row i **do**
2 **for** $p = 0$ **to** $s - 1$ **do**
3 $\texttt{y}[i] \leftarrow \texttt{y}[i] + \texttt{val}[i,p] \cdot \texttt{x}[\texttt{ind}[i,p]]$

The loops may be interchanged, and on vector architectures with explicit gather support, vectorization across either rows or columns is possible.

Jagged diagonal format

The jagged diagonal (JAD) format was designed to overcome the problem of variable length rows/columns in the CSR/CSC formats. The performance of SpMV can be especially poor on vector architectures in CSR format when the number of non-zeros per row is typically less than the machine's vector length. The main idea behind JAD format is to reorder rows of A so as to expose more opporunities to exploit data parallelism [266].

Storing A in JAD format consists of two steps, as illustrated in Figure 2.12. First, the rows of A are logically permuted in decreasing order of non-zeros per row by a permutation matrix P. In Figure 2.12 (*top*), the first element of every row i is labeled by i; in

Figure 2.12 (*bottom*), the rows have been permuted. P is stored in an integer array `perm`. (Depending on the precise interface between a SpMV routine and the user, the permutation may be needed to undo the logical permutation.)

Next, we define the *d-th jagged diagonal* to be the set of all of the d-th elements from all rows. The example in Figure 2.12 (*bottom*) shows 5 jagged diagonals: elements from the 0-jagged diagonal are shaded in red, from the 1-jagged diagonal are shaded green, and so on. Permuting A has ensured that as d increases, the length of the d-th jagged diagonal decreases. Furthermore, all of the elements in a given jagged diagonal will lie consecutively starting at the first row of the permuted A. Just as with CSR and CSC formats, we store each jagged diagonal as a sparse vector, and store all these vectors continguously in `val` and `ind` arrays. An array `ptr` holds the offset of the first element in each jagged diagonal. SpMV in JAD format is as follows, where s is the number of jagged diagonals:

```
    /* Note: val, ind, ptr store P · A */
    type val : real[k]
    type ind : int[k]
    type ptr : int[s + 1]
1   for d = 0 to s − 1 do /* for each jagged diagonal */
2       for l = 0 to ptr[d + 1] − ptr[d] do
3           z[l] ← z[l] + val[ptr[d] + l] · x[ind[ptr[d] + l]]
```

This code actually computes $z \leftarrow z + P \cdot A \cdot x$. Depending on the interface, the user may need to also compute $z \leftarrow P \cdot y$ on entry and $y \leftarrow P^{-1} \cdot z$ on exit. In typical iterative methods, the user only needs to perform these permutations at the beginning of the method, perform SpMV on z many times, and then unpermute at the end.

The JAD-based implementation of SpMV is similar to both CSR and CSC. Like CSR, JAD shares indirect accesses to x. Like CSC, JAD performs vector scaling in the inner-most loop. Thus, on cache-based superscalar machines, we would expect performance behavior comparable to CSR and CSC formats.

JAD format was revisited in an experimental study by White and Saddayapan [326], and is at present the base format used in the GeoFEM finite element library [237]. In keeping with the spirit of the original vector architecture-based work on the JAD format, GeoFEM's target architecture is the Earth Simulator.

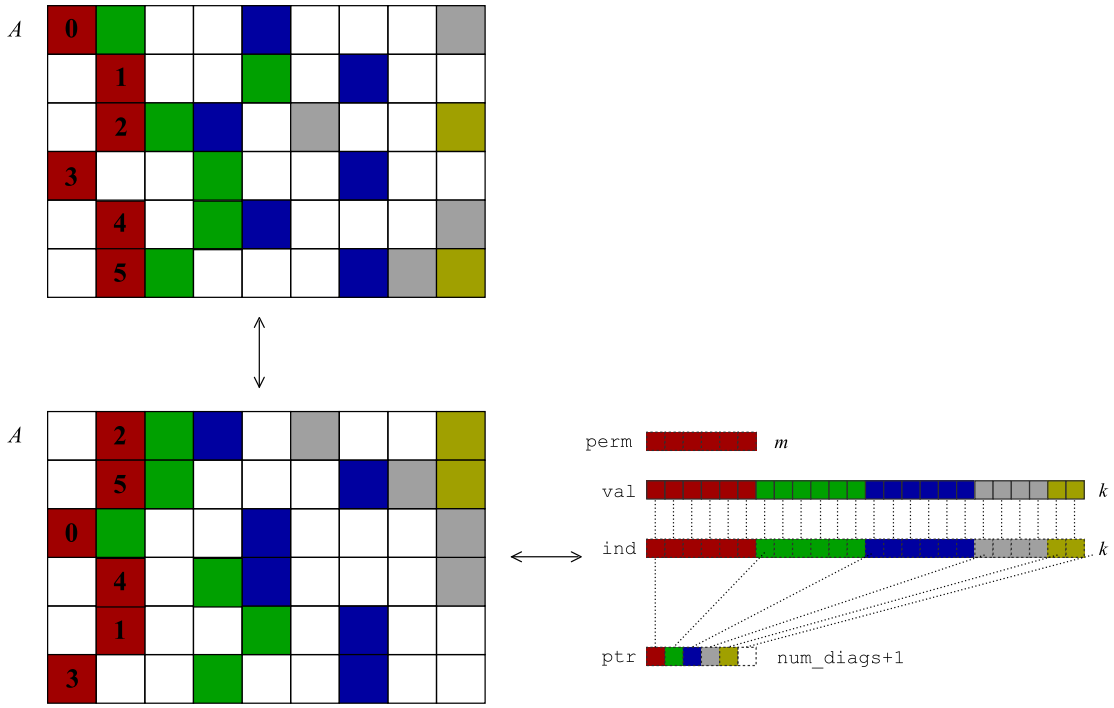

Figure 2.12: **Jagged diagonal format**. In the jagged diagonal representation, the rows of A (*top*) are logically permuted in decreasing order of number of non-zeros per row (*bottom*). The permutation information is stored in `perm`. Elements shaded the same color belong to the same "jagged diagonal." Each jagged diagonal is then stored as a sequence of sparse vectors in `val`, `ind` as with CSR and CSC formats.

Skyline format

Skyline (SKY) format is a composite format which stores the strictly lower triangle of A in CSR, the strictly upper triangle in CSC, and the diagonal is stored in an array. This format was particularly convenient in early implementations for Gaussian elimination, *i.e.*, computing the decomposition $A = LU$, where L is a lower triangular matrix and U is upper triangular. We will not be interested in SKY in this dissertation, and we therefore refer the reader to other discussions [107].

Block compressed stripe formats

The class of formats referred to as *blocked compressed sparse stripe* formats are designed to exploit naturally occurring dense block structure typical of matrices arising in finite element

method (FEM) simulations. For an example of a matrix amenable to block storage, recall the FEM matrix of Figure 1.2 which consists entirely of dense 8×8 blocks. Conceptually, the block compressed sparse stripe formats replace each non-zero in the compressed sparse stripe format by an $r \times c$ dense block.[5] The case of $r = c = 1$ is exactly the compressed stripe storage described in Section 2.1.4.

Here, we describe the block compressed sparse row (BCSR) format. The $r \times c$ BCSR format generalizes CSR: A is divided into $\lceil \frac{m}{r} \rceil$ block rows, and each block row is stored as a sequence of dense $r \times c$ blocks. Figure 2.13 (*top*) shows an example of a 6×9 matrix stored in 2×3 BCSR. The values of A are stored in an array `val` of $K_{rc}rc$ elements, where K_{rc} is the number of non-zero blocks. The blocks are stored consecutively by row, and each block may be stored in any of the dense formats for general matrices (*e.g.*, row- or column-major) described in Section 2.1.2. The starting column index of each block is stored in an array `ind`, and the offset within `ind` of the first index in each block row is stored in `ptr`. Each block is treated as a full dense block, which may require filling in explicit zero elements. We discuss the relationships among fill, the overall size of the data structure, and performance when we examine the *register blocking optimization* based on BCSR format in Chapter 3.

Blockings are not unique, as can be seen by comparing the last block row between Figure 2.13 (*top*) and (*bottom*). Different libraries and systems have chosen different conventions for selecting blocks. The original SPARSITY system chose to always align blocks so that the first column j of each block was always chosen so that $j \bmod c = 0$ [164]. By contrast, the SPARSKIT library uses a greedy approach which scans each block row column-by-column, starting a new $r \times c$ block upon encountering the first column containing a non-zero.

The pseudo-code implementing SpMV using BCSR format is as follows, where we have assumed that r divides m and c divides n:

[5] All implementations of which we are aware treat only square blocks, *i.e.*, $r = c$. We present the straightforward generalization, particularly in light of the experimental results of Section 1.3.

```
     type val : real[K_rc · r · c]
     type ind : int[K_rc]
     type ptr : int[m/r + 1]
1    foreach block row I do
2        i_0 ← I · r /* starting row */
3        Let ŷ ← y_{i_0:(i_0+r−1)} /* Can store in registers */
4        for L = ptr[I] to ptr[I + 1] − 1 do
5            j_0 ← ind[L] · c /* starting column */
6            Let x̂ ← x_{j_0:(j_0+c−1)} /* Can store in registers */
7            Let Â ← a_{i_0:(i_0+r−1),j_0:(j_0+c−1)}
                 /* Â = block of A stored in val[(L · r · c) : ((L + 1) · r · c − 1)] */
8            Perform r×c block multiply, ŷ ← ŷ + Â · x̂
9        Store ŷ
```

where $a : b$ denotes a closed range of integers from a to b inclusive. Since r and c are fixed, the block multiply in line 8 can be fully unrolled, and the elements of x and y can be reused by keeping them in registers (lines 3 and 6). We discuss implementations of SpMV using BCSR in more detail in Chapter 3.

Variable block row format

The variable block row (VBR) format generalizes the BCSR format by allowing block rows and columns to have variable sizes. This format is more complex than the preceeding formats. Moreover, SpMV in this format is difficult to implement efficiently because, unlike BCSR format, the block size changes in the inner-most loop, requiring a branch operation if we wish to unroll block multiplies and keep elements of x and y in registers as in BCSR. Indeed, we know of no implementations of SpMV in this format which are as fast as any of the formats described in this chapter on our evaluation platforms.[6] However, VBR serves as a useful intermediate format in a technique for exploiting blocks of multiple sizes, as discussed in Chapter 5.

We illustrate VBR format in Figure 2.14, where we show a $m{\times}n{=}6{\times}8$ matrix A containing $k = 19$ non-zeros. Consider a partitioning of this matrix into $M = 3$ block rows

[6] The two libraries implementing this format are SPARSKIT and the NIST Sparse BLAS [267, 258].

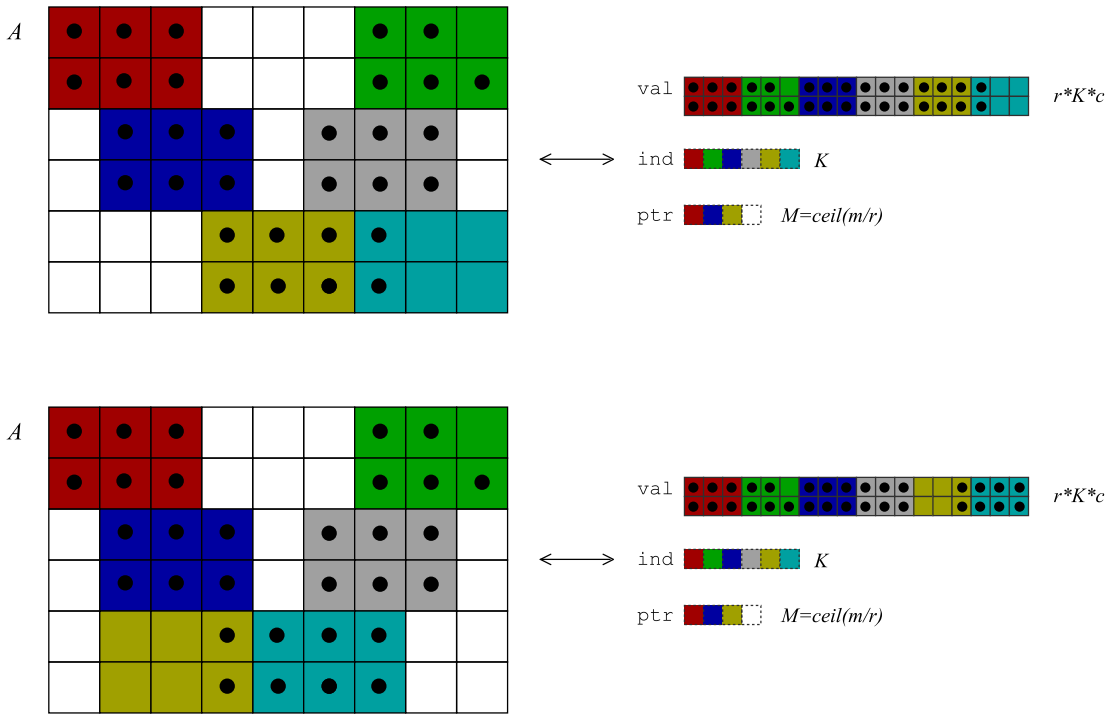

Figure 2.13: **Block compressed sparse row (BCSR) format**. (*Top*) In a 2×3 BCSR format, A is divided into $\lceil \frac{m}{r} \rceil = 3$ block rows, and each row is stored as a sequence of 2×3 blocks in an array `val`. There are $K = 6$ blocks total in this example. The elements of a given block have been shaded the same color, and solid black dots indicate structural non-zeros. To fill all blocks, explicit zeros have been filled in (*e.g.*, the $(0, 9)$ element). Each block may be stored in any of the dense formats (*e.g.*, row-major, column-major; see Section 2.1.2). The column index of the $(0, 0)$ element of each block is stored in the array `ind`. The element `ptr`$[I]$ is the offset in `ind` of the first block of block row I. (*Bottom*) Blockings are not unique. Here, we show a different blocking of the same matrix A (*top*).

and $N = 4$ block columns as shown, yielding $K = 6$ blocks, each shaded with a different color. The VBR data structure is composed of the following 6 arrays:

- `brow` (length $M + 1$): starting row positions in A of each block row. The I^{th} block row starts at row `brow`$[I]$ of A, ends at `brow`$[I + 1] - 1$, and `brow`$[M] = m$.
- `bcol` (length $N + 1$): starting column positions in A of each block column. The J^{th} block column starts at column `bcol`$[J]$ of A, ends at `bcol`$[J + 1] - 1$, and `brow`$[N] = n$.
- `val` (length k): non-zero values, stored block-by-block. Blocks are laid out by row.

- `val_ptr` (length $K+1$): starting offsets of each block within `val`. The b^{th} block starts at position `val_ptr`$[b]$ in the array `val`. The last element `val_ptr`$[K] = k$.
- `ind` (length K): block column indices. The b^{th} block begins at column `bcol`[`ind`$[b]$].
- `ptr` (length $M+1$): starting offsets of each block row within `ind`. The I^{th} block row starts at position `ptr`$[I]$ in `ind`.

The pseudo-code for SpMV using VBR is as follows:

```
    type brow : int[M + 1]
    type bcol : int[N + 1]
    type val : real[k]
    type val_ptr : int[K + 1]
    type ind : int[K]
    type ptr : int[M + 1]
1   foreach block row I do
2       i_0 ← brow[I] /* starting row index */
3       r ← brow[I + 1] − brow[I] /* row block size */
4       Let ŷ ← y_{i_0:(i_0+r−1)}
5       for b = ptr[I] to ptr[I + 1] − 1 do /* blocks within I^th block row */
6           J ← ind[b] /* block column index */
7           j_0 ← bcol[J] /* starting column index */
8           c ← bcol[J + 1] − bcol[J] /* column block size */
9           Let x̂ ← x_{j_0:(j_0+c−1)}
10          Let Â ← a_{i_0:(i_0+r−1),j_0:(j_0+c−1)}
                /* Â = block of A stored in val[val_ptr[b] : (val_ptr[b + 1] − 1)] */
11          Perform r×c block multiply, ŷ ← ŷ + Â · x̂
12      Store ŷ
```

Unlike the BCSR code, r and c are not fixed throughout the computation, making it difficult to unroll line 11 in the same way that we can unroll the block computation in the BCSR code. In particular, we would need to introduce branches to handle different fixed block sizes. The implementation in SPARSKIT uses 2-nested loops to perform the block multiply [267]. We would not expect VBR to perform very well due to the overheads incurred by these loops.

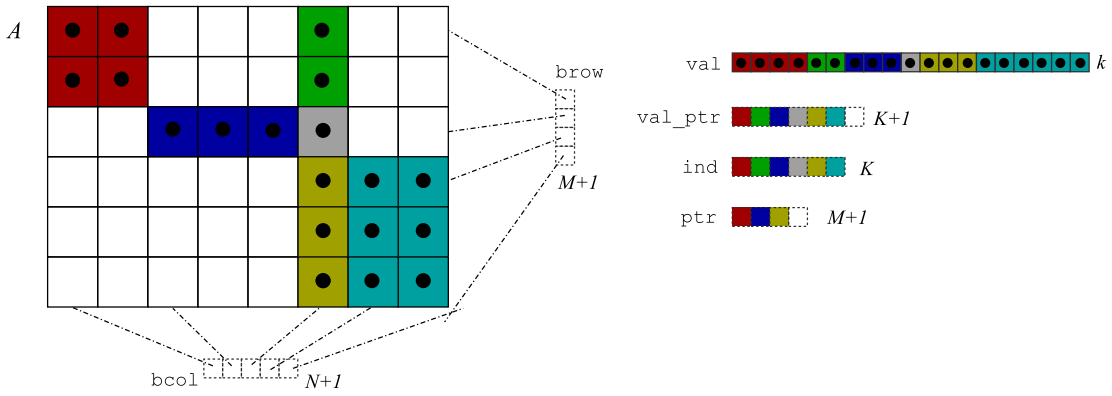

Figure 2.14: **Variable block row (VBR) format**. We show an example of a sparse matrix A with $k = 19$ non-zeros. A is logically partitioned into $M = 3$ block rows, $N = 4$ block columns, yielding $K = 6$ non-zero blocks. The starting positions of each block row and block column are stored in `brow` and `bcol`, respectively. Non-zero values are stored in `val`, and the starting positions of each block of values are stored in `val_ptr`. Block column indices are stored in `ind`, and the beginning of the indices belonging to a given block row are stored in `ptr`.

2.2 Experimental Comparison of the Basic Formats

This section compares implementations of SpMV using a subset of the formats described in Section 2.1. We sometimes refer to these implementations collectively as the "baseline implementations." We make our comparisons across a variety of matrices and machine architectures. The high-level conclusions of these experiments are as follows:

- CSR and MSR formats tend to have the best performance on a wide class of matrices and on a variety of superscalar architectures, among the basic formats considered (and, in particular, omitting the BCSR format.) Thus, either of these formats would appear to be a reasonable default choice if the user knows nothing about the matrix structure. In subsequent chapters, "reference" performance always refers to CSR performance.

- Comparing across architectures, we show that for the most part, none of the basic formats yield significantly more than 10% of peak machine speed. This observation, coupled with the results of Section 1.3 arguing for search-based methods, motivate our aggresive exploitation of matrix structure.

We review the experimental setup (Section 2.2.1) before presenting and discussing the experimental results (Section 2.2.2).

2.2.1 Experimental setup

Our experimental method, as with all the experiments of this dissertation, follows the discussion in Appendix B. When referring to a "platform," we refer to both a machine and a compiler. Thus, measurements reflect both characteristics of the machine architecture plus the quality of the compiler's code generation. In these experiments, we make an effort to use the best compiler flags, pragmas, and keywords that enable vectorization where possible, and in the C implementations, to eliminate false dependencies due to aliasing.

The baseline implementations are taken from the SPARSKIT library. We consider both the Fortran implementations (as written in the original SPARSKIT library) along with C implementations. (Our C implementations are manual translations of the Fortran-based SPARSKIT code.) As discussed in Appendix B, all matrix values are stored in IEEE double-precision (64-bit) floating point, and all indices are stored as 32-bit integers. We compare the CSR, CSC, DIAG, ELL, MSR, and JAD formats. We omit the SKY format since it is functionally equivalent to CSR and CSC. We omit comparison to the COO and VBR format because these implementations were considerably slower (by roughly up to an order of magnitude) than the worst of the formats considered. (In the case of VBR, refer to the discussion about unrolling in Section 2.1.4.)

None of the matrices in this suite consist of only full diagonals. Therefore, our implementation of the DIAG format actually splits the matrix into the sum $A = A_1 + A_2$, where A_1 is stored in diagonal format, A_2 is stored in CSR format, and the non-zero structures of A_1 and A_2 are disjoint. Our criteria for storing a given diagonal of A in A_1 are that (1) the length diagonal must be no less than 85% of the dimension of A, and (2) that diagonal itself must be 90% full. The first condition keeps diagonal storage manageable. For instance, an $n \times n$ permutation matrix would require n^2 storage in the worst case in DIAG format were no such condition imposed. The second condition ensures that replacing a "mostly full" diagonal still leads to a reduction in the total storage. In particular, a diagonal of length n which is 90% full requires that we store at least (8 bytes + 4 bytes) $\times .9n = 10.8n$ bytes in a CSR-like format (1 64-bit real + 1 32-bit int per non-zero), but only approximately $8n$ bytes in diagonal format. Thus, the CSR-like format requires

$10.8/8 = 1.35\times$ more storage.

Recall that JAD includes a logical row permutation, and is otherwise roughly equivalent to CSC and CSR formats (Section 6). The JAD implementation we consider includes a permutation of y on input and an inverse permutation of y on output. The cost of these permutations is reflected in the performance data we report; in practice, if SpMV is performed many times, these permutations could be moved to occur before the first multiplication and after the last, thereby amortizing the permutation cost.

The implementation of the BCSR format is complicated by issues of how to choose the block size and handle explicit fill. Section 1.3 alludes to these difficulties but demonstrates that it is possible to achieve significantly more than 10% of peak machine speed by choosing an appropriate implementation. We consider such an implementation, which includes the choice of a possibly non-square block size, to be among our proposed optimizations, especially since non-square block sizes have been only addressed in any depth by SPARSITY[165, 164]. We therefore defer detailed analyses of BCSR performance to Chapter 3.

2.2.2 Results on the Sparsity matrix benchmark suite

We present the observed performance over the matrices and platforms shown in Appendix B in Figures 2.15–2.18. All figures show both absolute performance (in Mflop/s) and the equivalent performance normalized to machine peak on the y-axis. Matrices which are small relative to the largest cache on each platform have been omitted (see Appendix B).

For ease of comparison across platforms, the fraction of peak always ranges from 0 to 12.5%. To aid comparisons among matrices, we divide the matrices into five classes:

1. Matrix 1: A dense matrix stored in sparse format.

2. Matrices 2–9: Matrices from FEM applications. The non-zero structure of these matrices tends to be dominated by a single square block size, and all blocks are uniformly aligned.

3. Matrices 10–17: These matrices also arise in FEM applications, and possess block structure. However, the block structure consists of a larger mix of block sizes than matrices 2–9, or have irregular alignment of blocks.

4. Matrices 18–39: These matrices come from a variety of applications (*e.g.*, chemical process simulation, finance) and do not have much regular block structure.

5. Matrices 40–44: These matrices arise in linear programming applications.

See Chapter 5 and Appendix F for more information on how these matrix classes differ.

Our discussion is organized by comparisons between platforms, comparisons between formats within a given platform, and comparisons between classes of matrices.

Comparing across platforms

The best performance of any baseline format is typically about 10% or less of machine peak. To see how much better than 10% we might expect to do on any platform, we summarize in Table 2.1 the data of Figures 2.15–2.18, and show how those data compare to dense matrix-vector multiply performance. Specifically, we summarize absolute performance and fraction of machine peak across platforms and over all formats in three cases:

1. SpMV performance for a dense matrix stored in sparse format (*i.e.*, Matrix 1);

2. The best SpMV performance over Matrices 2–44;

3. The best known performance of available dense BLAS matrix-vector multiply implementations (see Appendix B). We compare against the performance achieved with the double-precision routine, DGEMV.

DGEMV performance (item 3 above) serves as an approximate guide to the best SpMV performance we might expect, since there is no index storage overhead associated with DGEMV. Comparing (1) and (3) roughly indicates the performance overhead from storing and manipulating extra integer indices: across all platforms, DGEMV is between 1.48× faster (IBM Power4) and 3.88× faster (Sun Ultra 3). Thus, if we could optimally exploit the non-zero matrix structure and eliminate all computation with indices, we might expect this range of speedups. Furthermore, we might expect to be able to run at nearly 20% or more of peak machine speed.

Finally, observe that (1) and (2) are often nearly equal, indicating that it may be possible to reproduce the best possible performance on Matrix 1 on actual sparse matrices. Indeed, on the Power4 platform, the best performance on a sparse matrix was slightly faster (about 4%) than on the dense matrix in sparse format.

Platform	Matrix 1 (dense) Mflop/s	Matrix 1 (dense) Fraction of Peak	Matrices 2–44 Mflop/s	Matrices 2–44 Fraction of Peak	DGEMV Mflop/s	DGEMV Fraction of Peak	DGEMV Speedup over SpMV
Ultra 2i	34	5.1%	34	5.2%	58	8.7%	1.68×
Ultra 3	83	4.6%	60	3.4%	322	17.9%	3.88×
Pentium 3	42	8.4%	40	8.1%	96	19.2%	2.29×
Pentium III-M	77	9.6%	75	9.4%	147	18.4%	1.92×
Power3	153	10.2%	152	10.1%	260	17.3%	1.69×
Power4	607	11.7%	500	9.6%	900	17.3%	1.48×
Itanium 1	196	6.1%	172	5.4%	310	9.7%	1.58×
Itanium 2	296	8.2%	277	7.7%	1330	36.9%	4.49×

Table 2.1: **Summary across platforms of baseline SpMV performance**. We show absolute SpMV performance (Mflop/s) and fraction of peak for three implementations: (1) the best performance over all baseline formats for Matrix 1, a dense matrix stored in sparse format, (2) the best performance over all baseline formats and sparse Matrices 2–44, and (3) best known performance of the dense BLAS matrix-vector multiply routine, DGEMV. In the last column, we show the speedup of DGEMV over the best of items (1) and (2). The speedup of DGEMV roughly indicates that we might expect a maximum range of speedups for SpMV between 1.48–4.49×.

Comparing performance among formats

The fastest formats overall tend to be the CSR and MSR formats on all platforms except the Itanium 1 platform. We discuss the Itanium 1 in more detail at the end of this section.

Recall that the difference between CSR and MSR formats is that in MSR, we extract the main diagonal and store it without indices. There are generally no differences in performance of more than a few percent between these formats, indicating this separation of the main diagonal only is not particularly beneficial. Indeed, the performance of the DIAG implementations, which in general would separate the main diagonal and other nearly full diagonals, was never faster than CSR except for the dense matrix in sparse format on two platforms (Ultra 3 and Power4). This observation indicates that while there may be some performance benefit to a DIAG implementation (or, a DIAG +CSR hybrid implementation this case), there may not be sufficient diagonal structure in practice to exploit it.

Although the CSC and CSR formats use the same amount of storage, the performance of CSC can be much worse than CSR on the surveyed machines. The main difference in the access patterns of these formats is that the inner-loop of CSR computes a dot product, whereas the inner loop of CSC computes a vector scale operation (or "AXPY" operation,

in Basic Linear Algebra Subroutines (BLAS) lingo [203]). If the matrix is structurally symmetric and there are p non-zeros per row/column, then each dot-product will perform $2p$ loads, to the row of A and vector x, and 1 store to y; by contrast, the AXPY requires $2p$ loads of a column of A and elements of y, interleaved with p stores to y. Thus, the performance difference could reflect artifacts of the architecture which cause even cached stores to incur some performance hit.

The performance of the JAD format is generally worse than that of CSC. Recall that JAD implementation is similar to the CSC implementation, and that our implementation of JAD includes the cost of two permutations of the destination vector y. Thus, the difference in performance between JAD and CSC may reflect these permutation costs, which could be amortized over many SpMV operations. Nevertheless, we would not expect JAD to be faster than CSC on superscalar architectures.

The ELL implementation generally yields the worst performance of all formats. Recall that ELL is best suited to matrices with an average number of non-zeros per row nearly equal to the maximum number of non-zeros per row. This condition is really only true for the Matrix 1 (dense) and Matrices 2 and 11 (in both, 93% of rows are within 1 non-zero of the maximum number of non-zeros per row). The performance difference between ELL performance on Matrices 1, 2, and 11, and performance all other matrices reflects this fact. Nevertheless, ELL performance is still worse than CSR/MSR even on the dense matrix, so at least on superscalar architectures, there is no reason to prefer a pure ELL implementation over CSR.

While ELL and JAD formats were usually the worst formats, there are a number of notable exceptions on the Itanium 1 platform. On Matrix 1 (dense), ELL and JAD were 1.38× faster than CSR, and on Matrix 11, ELL was 1.47× faster than CSR. Otherwise, ELL was only marginally faster than CSR. We do not at present know why ELL performance was only competitive with CSR in a few cases on the Itanium 1 only, and to a lesser extent in a few cases on Itanium 2. These ELL results suggest that on platforms like the Itanium 1, there could be an appreciable benefit to a hybrid ELL/CSR format, in which we use ELL format to store a subset of the non-zeros satisfying the ELL uniform non-zeros per row assumption, and the remainder of the non-zeros in CSR (or another appropriate) format.

Comparing performance across matrices

There are clear performance differences between the various classes of matrices. Performance within both classes of FEM matrices, Matrices 2–17, tends to be higher than Matrices 18–44. Barring a few exceptions, the main point is that these particular classes of matrices really are structurally distinct in some sense, and we might expect that improving the absolute performance of Matrices 18–44 will be more challenging than on Matrices 2–17, which tend to have performance that more closely matches that of the Matrix 1.

2.3 Note on Recursive Storage Formats

In moving from dense formats to the various sparse matrix formats, we see that the random access property is relaxed to varying degrees. Indeed, even in the case of general dense formats, relaxing the random access property has enabled new formats that lead to high performance implementations of dense linear algebra algorithms. Recently, Andersen, Gustavson, *et al.*, have advocated a *recursive* data layout, which, roughly speaking, stores a matrix in a quad-tree format [12, 13]. The intent is to match the data layout to the natural access pattern of recursive formulations of dense linear algebra algorithms. Several such *cache-oblivious* algorithms have been shown to move the asymptotically minimum number of words between main memory and the caches and CPU, without explicit knowledge of cache configuration (sizes and line sizes) [302, 124]. Furthermore, language-level support for recursive layouts have followed, including work on converting array indices from traditional row/column-major layouts to recursive layouts by Wise, *et al.* [329], and work on determining recursive versions of imperfectly-nested loop code by Yi, *et al.* [334]. To date, the biggest performance pay-offs have been demonstrated for matrix multiply and LU factorization which are essentially computation-bound (*i.e.*, $O(n^3)$ flops compared to $O(n^2)$ storage). Nevertheless, it is possible that recursive formats may have an impact on sparse kernels as well, though there is at present little published work on this topic aside from some work on sparse Cholesky and LU factorization [170, 104].

2.4 Summary

Our experimental evaluation confirms the claim of Chapter 1 that 10% of peak or less is typical of the basic formats on modern machines based on cache-based superscalar micro-

processors (Section 2.2). We further conclude that CSR and MSR formats are reasonable default data structures when nothing else is known about the input matrix structure. Indeed, CSR storage is the base format in the PETSc scientific computing library [27], as well as SPARSKIT [267]. However, our analysis is restricted to platforms based on cache-based superscalar architectures, while several of the surveyed formats were designed with vector architectures in mind.

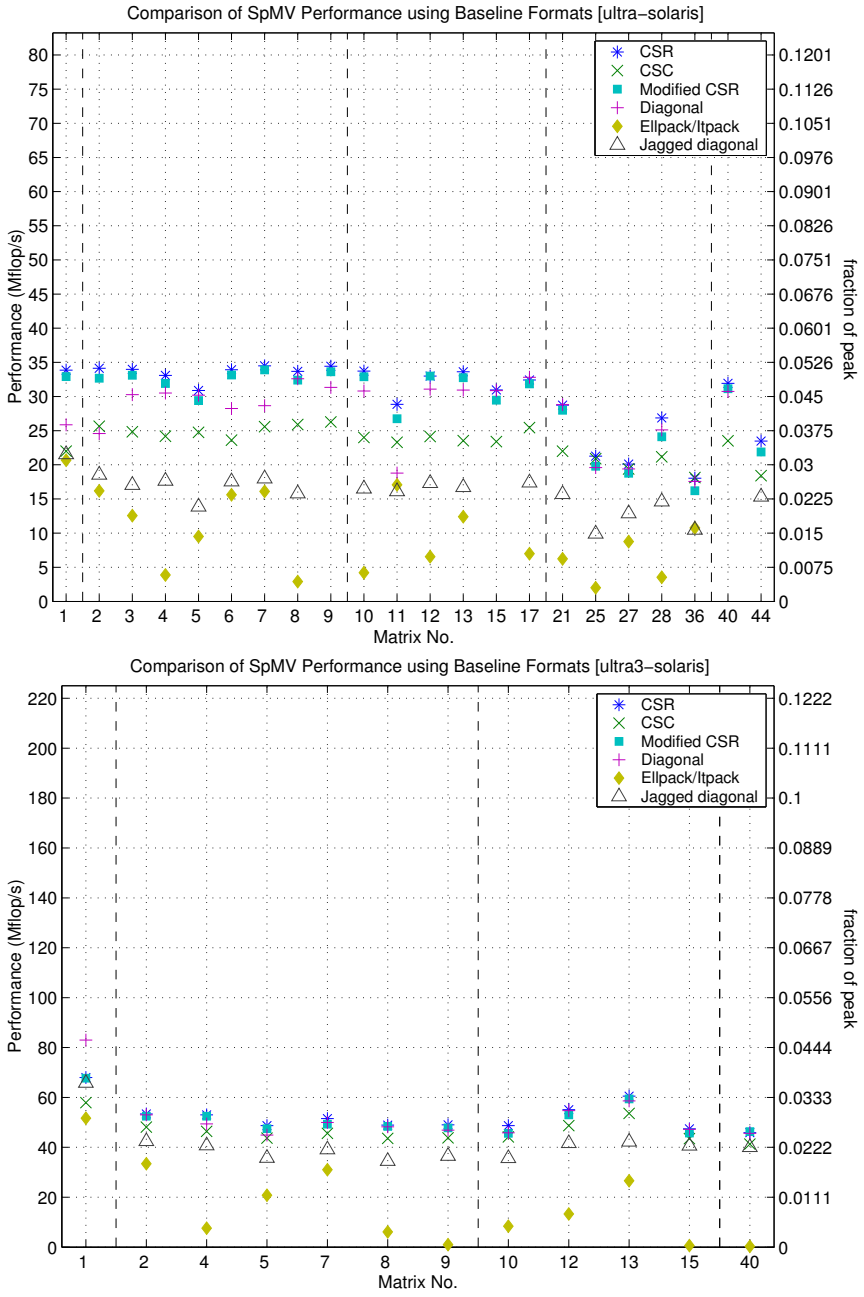

Figure 2.15: **SpMV performance using baseline formats on Matrix Benchmark Suite #1: Sun Ultra 2i (*top*) and Ultra 3 (*bottom*) platforms.** This data is also tabulated in Appendix C.

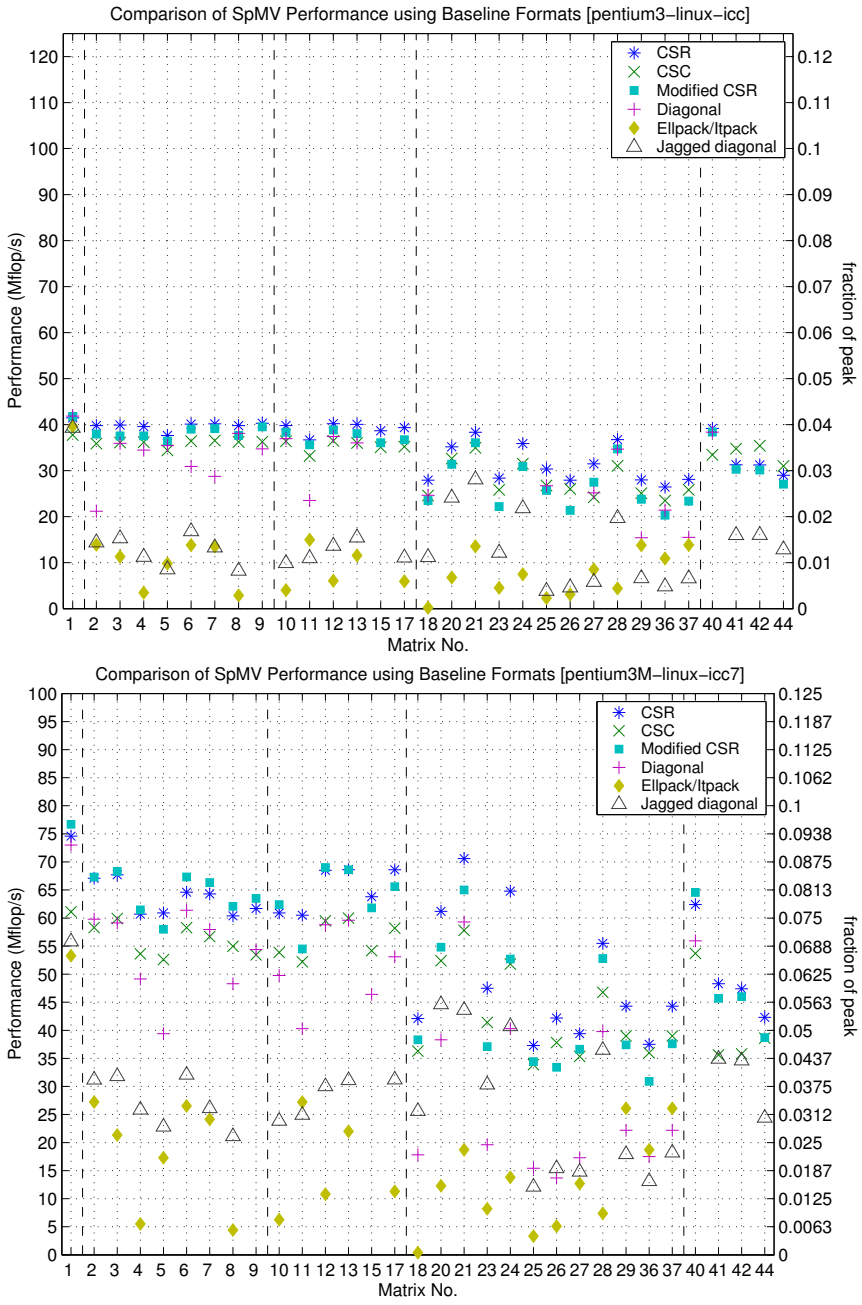

Figure 2.16: **SpMV performance using baseline formats on Matrix Benchmark Suite #1: Intel Pentium III (*top*) and Pentium III-M (*bottom*) platforms**. This data is also tabulated in Appendix C.

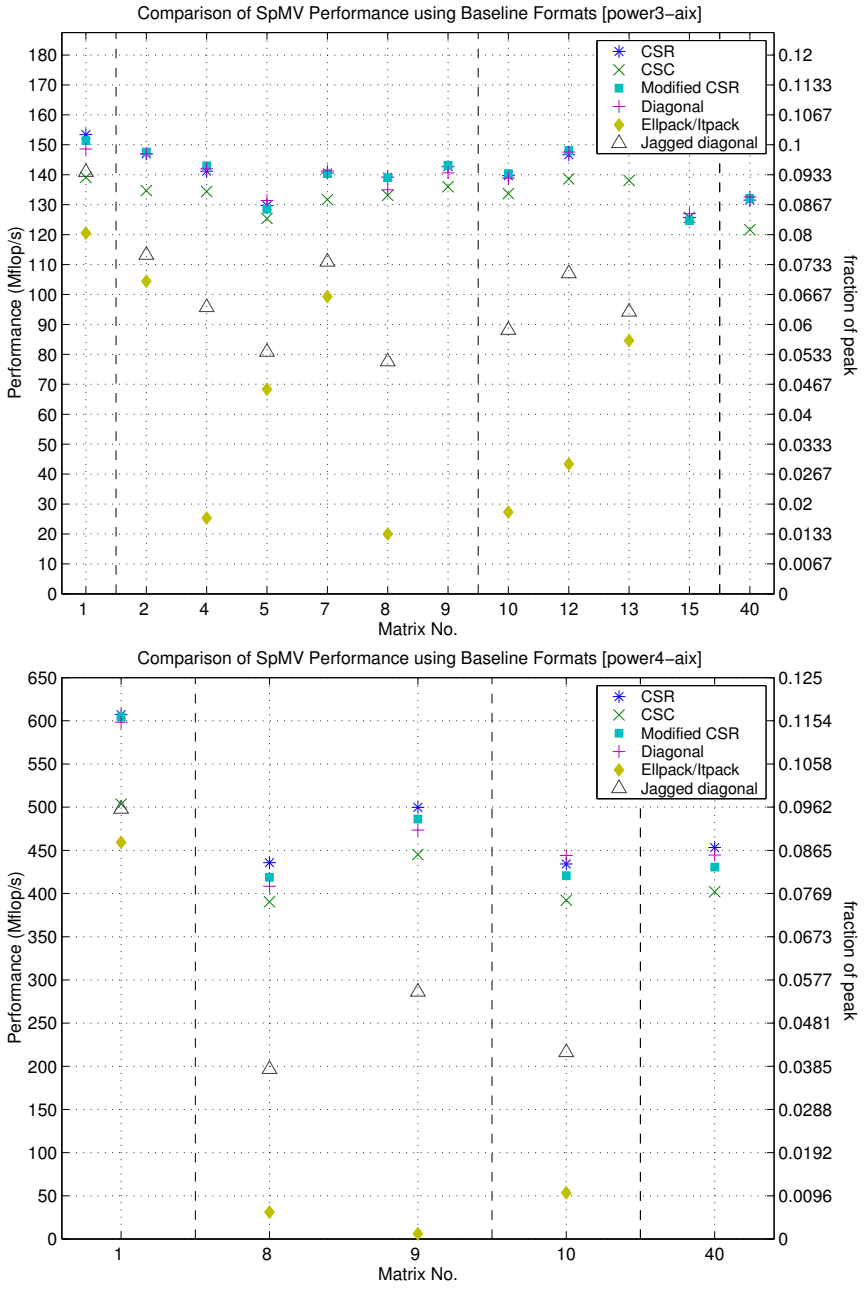

Figure 2.17: **SpMV performance using baseline formats on Matrix Benchmark Suite #1: IBM Power3 (*top*) and Power4 (*bottom*) platforms.** This data is also tabulated in Appendix C.

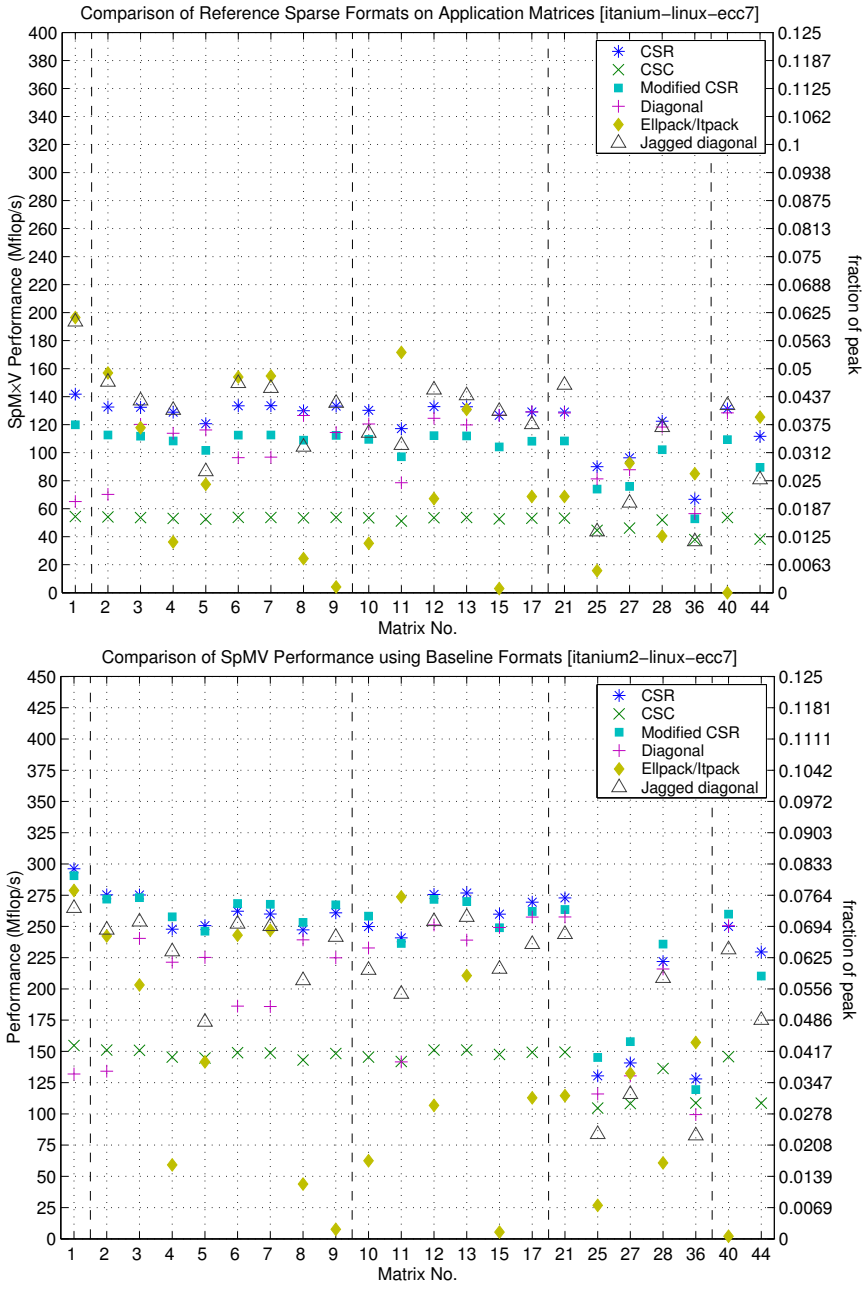

Figure 2.18: **SpMV performance using baseline formats on Matrix Benchmark Suite #1: Intel Itanium 1 (*top*) and Itanium 2 (*bottom*) platforms.** This data is also tabulated in Appendix C.

Chapter 3

Improved Register-Level Tuning of Sparse Matrix-Vector Multiply

Contents

The challenge of choosing the right tuning parameters, as illustrated in Section 1.3, motivates our use of automated tuning techniques based on empirical search. This chapter addresses the problem of how to select register-level blocking (or tiling) sizes for sparse matrix-vector multiply (SpMV) by improving a tuning heuristic previously proposed for the original SPARSITY system [167, 164]. We refer to the original heuristic as the SPARSITY Version 1 heuristic. Our "Version 2" heuristic improves the robustness of block size selection on recent machine architectures which exhibit the highly irregular performance

behavior observed in Section 1.3. This new heuristic replaces the original heuristic in the current SPARSITY system. Here, we justify the engineering design choices behind the implementation of this new heuristic, examining the steps required and quantifying their costs.

We review SPARSITY's implementation of register blocking in Section 3.1. Briefly, the matrix is stored in block compressed sparse row (BCSR) format, as described in Section 2.1.4, and SpMV is expressed as a sequence of operations on blocks of data that fit within machine registers. Though similar to register-level blocking or tiling for dense matrices [201], blocking in the sparse case depends on the non-zero structure of the sparse matrix. As Section 1.3 argues, the block size depends critically on the hardware platform as well. Section 3.1 revisits this argument for a very regular "sparse matrix," namely, a dense matrix stored in sparse BCSR format. Even in the absence of irregular memory access patterns, surprising performance behavior persists.

Trying all or a subset of block sizes is likely to be infeasible if the matrix is known only at run-time, owing to the cost of simply converting the matrix to blocked format. We show empirically that just converting a matrix from unblocked compressed sparse row (CSR) format to a blocked format can cost as much as 40 1×1 SpMVs, depending on the platform and matrix. These costs are acceptable in important application contexts like solving linear systems or computing eigenvalues by iterative methods, where hundreds (or more) of SpMVs can be required for a given matrix [30, 176, 93, 64, 268].

Our goal then is to develop a heuristic for selecting a block size, $r \times c$, that is reasonably accurate, but whose cost is small compared to the cost of converting the matrix. The irregularity of performance as a function of r and c means that developing an accurate but simple analytical performance model will be difficult. Instead, the SPARSITY Version 1 heuristic used the following empirically-based three-step procedure, described in more detail in Section 3.2.

1. Benchmark the register blocked routines for all $r \times c$ block sizes (up to some limit to be defined) on a sample matrix, namely, a dense matrix stored in sparse format. This benchmark is independent of the user's sparse matrix, and therefore needs to be executed only once per machine or machine architecture.

2. When the user's matrix is known at run-time, estimate the amount of natural block structure indirectly by estimating a quantity called the *fill ratio*. The fill ratio, as defined in Section 3.1.1, depends on the matrix and on $r \times c$, and the fill ratio is

estimated for all $r \times c$.

3. Evaluate a performance model that combines the benchmarking data with the fill ratio estimates to select r and c.

The costs of steps 2 and 3 are particularly important to minimize because they occur at run-time. Our Version 2 heuristic follows the above procedure, but improves steps 2 and 3 as described in Section 3.2. These improvements lead to increased prediction accuracy on recent machines like the Itanium 1 and Itanium 2, where Version 1 heuristic chooses an implementation that can achieve as little as 60% of the best possible performance over all r and c. The Version 2 heuristic in contrast frequently selects the best block size, and yielding performance that is nearly always 90% or more of the best. In addition, we show that the run-time costs of the Version 2 heuristic implementation of steps 2 and 3 is between 1–11 unblocked SpMVs, depending on the platform and matrix, on 8 current platforms and 44 test matrices. This cost is the penalty for evaluating the heuristic if it turns out no blocking is required, and we discuss ways in which future work could try to reduce these costs. However, if blocking is beneficial, we show the overall block size selection procedure *including* conversion is never more than 42 1×1 SpMVs on our test machines and matrices, meaning the cost of evaluating the heuristic is modest compared to the cost of conversion.

As noted by Im, *et al.*, SPARSITY's hybrid off-line/run-time tuning methodology for register block size selection is an instance of a more general framework [165]. Indeed, we adapt this framework in subsequent chapters to other kernels and optimizations (*e.g.*, sparse triangular solve (SpTS) in Chapter 6 and sparse $A^TA \cdot x$ (SpA^TA) in Chapter 7), thereby demonstrating the effectiveness and applicability of the approach.

Some of the material in this chapter also appears in recent papers [316, 165]. This chapter discusses the costs of tuning in more detail.

3.1 Register Blocked Sparse Matrix-Vector Multiply

We begin with a review of register blocked sparse matrix-vector multiply (SpMV), as implemented by the SPARSITY system [167, 164]. The SPARSITY implementation of register blocking uses block compressed sparse row (BCSR) format as the base storage format (Section 3.1.1). We present simple analyses and experimental results to help build the reader's intuition for what performance behavior we might reasonably expect in contrast to what

we observe in practice. This section revisits in greater detail the introductory argument of Section 1.3 that motivated the use of empirical search-based methods for tuning.

3.1.1 Register blocking overview

Consider the SpMV operation $y \leftarrow y+Ax$, where A is an $m \times n$ sparse matrix stored in BCSR format. BCSR is generally presented assuming square block sizes, but here we consider the generalization of early descriptions of BCSR from square block sizes to arbitrary rectangular $r \times c$ blocks, as proposed in SPARSITY [164] and described in Section 6. BCSR is designed to exploit naturally occuring dense blocks by reorganizing the matrix data structure into a sequence of small (enough to fit in register) dense blocks.

BCSR logically divides the matrix into $\frac{m}{r} \times \frac{n}{c}$ submatrices, where each submatrix is of size $r \times c$. Assume for simplicity that r divides m and that c divides n. Only those blocks which contain at least one non-zero are stored, and blocks within the same block row are stored consecutively. We assume the SPARSITY convention in which each block is stored in row-major order. The SpMV computation proceeds block-by-block: for each block, we can reuse the corresponding c elements of the source vector x and r elements of the destination vector y by keeping them in registers, assuming a sufficient number is available. The case of $r = c = 1$ is precisely the case of compressed sparse row (CSR) format.

The C implementation of SpMV using 2×3 BCSR appears in Figure 3.1 (*bottom*). The matrix is represented by three arrays: `Aval, Aind,` and `Aptr`. Here, `Aval` is an array storing all the values, `Aind` stores the column indices for each block, and `Aptr` stores the starting position within `Aind` of each block row. For reference, we show a conventional dense matrix-vector multiply code in Figure 3.1 (*top*) where `A` is assumed to be stored in row-major order, and we use a C-style pointer idiom to traverse the elements of `A` (`A++` in line D3 and `A[0]` in line D4). The line numbers for both the dense and sparse codes are labeled to show the correspondence between the two implementations. For example, where the dense code loops over each column `j` within each row `i` (*top*, line D3), the sparse code loops over 2×3 blocks within each block row beginning at row `i`, column `j` (lines S3a–b). The multiplication by each 2×3 block in the sparse code is fully unrolled, and the $c = 3$ corresponding elements of the source vector elements (`x0,x1,x2`) are each reused $r = 2$ times for each block. The destination vector elements (`y0,y1`) can be kept in registers while processing all of the blocks within a block row, *i.e.*, throughout the execution of the

loop in line S3a of Figure 3.1 (*bottom*).

BCSR potentially stores fewer column indices than CSR—one index per block instead of one per non-zero. The effect is to reduce memory traffic by reducing storage overhead. However, imposing a uniform $r \times c$ block size for an arbitrary matrix may require filling in explicit zero values to fill each stored block, resulting in extra zero storage and computation, as discussed in Section 2.1.4. We define the *fill ratio* to be the number of stored values (*i.e.*, number of non-zeros originally plus explicit zeros) divided by the number of non-zeros in the original matrix. By this definition, the fill ratio is always at least one. Whether conversion to a register blocked format is profitable depends highly on the fill and, in turn, the non-zero pattern of the matrix. Indeed, it is even possible to reduce the overall size of the data structure when the fill ratio is greater than 1.

For instance, consider the 50×50 submatrix shown in Figure 3.2 (*left*), where "true" or "ideal" non-zeros are shown by blue dots. (The matrix shown is Matrix 13-`ex11`; see Appendix B.) Block substructure exists, but does not appear to be uniform. Nevertheless, suppose we wish to store this matrix in 3×3 BCSR format. We show the result in Figure 3.2 (*right*), where we impose a uniformly aligned logical grid of 3×3 cells and show explicit zeros by red x's. The fill ratio for the entire matrix turns out to be 1.5, meaning that we perform 1.5× more flops than necessary due to filled in zeros. Nevertheless, on a Pentium III platform, the running time with the 3×3 blocked matrix was two-thirds as much as the unblocked case, and the total storage (in bytes) is only 1.07× more than the unblocked case. As we later discuss (Section 3.2), the improvement in execution time in this case is due to the fact that the code implementing the 3×3 is much faster than the unblocked code, thereby compensating for the overhead due to fill. This example shows that filling in zeros can lead to significantly faster implementations, without necessarily increasing storage by much if at all.

We can express the precise relationship among the size of the data structure, fill ratio, and block size, as follows. Let k be the number of non-zeros in A, and let K_{rc} be the number of $r \times c$ non-zero blocks required to store the matrix in $r \times c$ BCSR format. For 1×1 blocks, $K_{1,1} = k$. The matrix requires storage of $K_{rc} \cdot rc$ double precision values, K_{rc} integers for the column indices, and $\lceil \frac{m}{r} \rceil + 1$ integers for the row pointers. Denote the fill ratio by $f_{rc} = \frac{K_{rc} \cdot rc}{k}$, which is always at least 1 as discussed previously. Since operations on sparse matrices involve both floating point and integer data, we will assume there are γ integers per floating point word. For example, if we use 64-bit double-precision floating

```
D0  void dense_mvm_reference( int m, int n, const double* A, // row-major
       const double* x, double* y )
    {
        int i;
D1      for( i = 0; i < m; i++ ) { // loop over rows
D2          register double y0 = y[i];
            int j;
D3          for( j = 0; j < n; j++, A++ )  // loop over columns in row i
D4              y0 += A[0]*x[j];
D5          y[i] = y0;
        }
    }
```

```
S0  void sparse_mvm_bcsr_2x3( int M, int n,
        const double* Aval, const int* Aind, const int* Aptr,
        const double* x, double* y )
    {
        int I;
S1      for( I = 0; I < M; I++, y += 2 ) { // loop over block rows
S2          register double y0 = y[0], y1 = y[1];
            int jj;

            // loop over non-zero blocks
S3a         for( jj = Aptr[I]; jj < Aptr[I+1]; jj++, Aval += 6 ) {
S3b             int j = Aind[jj];
S4a             register double x0 = x[j], x1 = x[j+1], x2 = x[j+2];

S4b             y0 += Aval[0]*x0; y1 += Aval[3]*x0;
S4c             y0 += Aval[1]*x1; y1 += Aval[4]*x1;
S4d             y0 += Aval[2]*x2; y1 += Aval[5]*x2;
            }
S5          y[0] = y0;  y[1] = y1;
        }
    }
```

Figure 3.1: **Example C implementations of matrix-vector multiply for dense and sparse BCSR matrices**. Here, `M` is the number of block rows (number of true rows is m =2*`M`) and `n` is the number of matrix columns. (*Top*) An example of a C implementation of matrix-vector multiply, where A is stored in row-major storage, with the leading dimension equal to `n`. (*Bottom*) A C implementation of SpMV assuming 2×3 BCSR format. Here, multiplication by each block is fully unrolled (lines S4b–S4d).

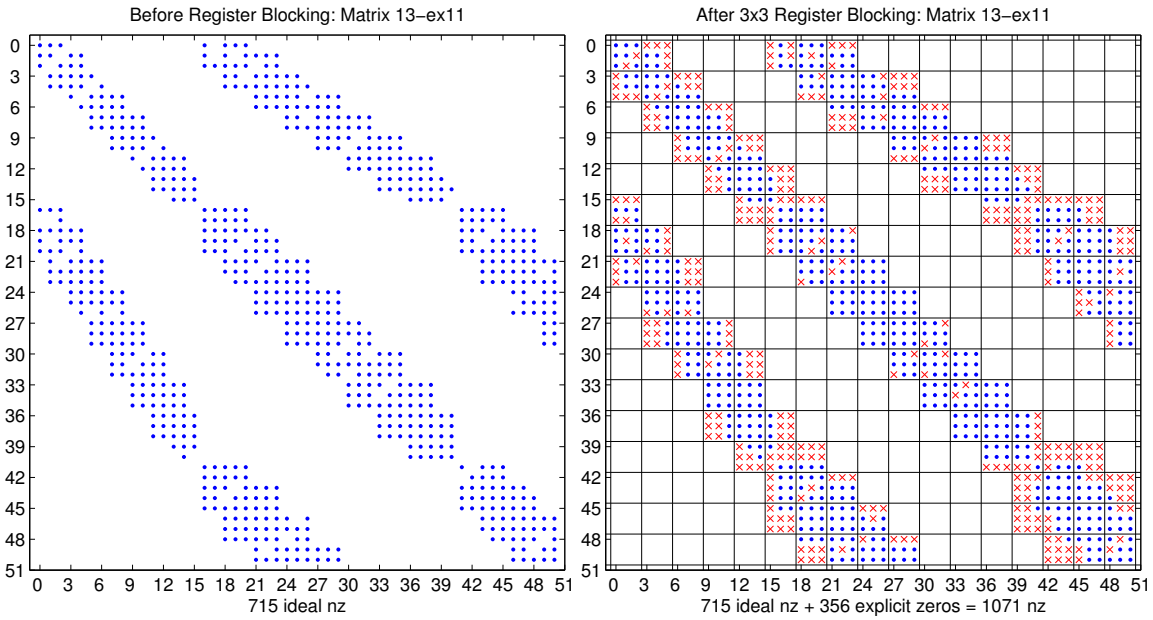

Figure 3.2: **Example of a non-obvious blocking**. (*Left*) A 50×50 submatrix of Matrix 13-`ex11` from the matrix test set (Appendix B). Non-zeros are shown by blue dots. (*Right*) The same matrix when stored in 3×3 register blocked format. We impose a uniformly aligned logical grid of 3×3 cells, and fill in explicit zeros (shown by red x's) to ensure that all blocks are full. The fill ratio here (for the entire matrix) turns out to be 1.5, but the SpMV implementation is nevertheless 1.5 times faster than the unblocked case on the Pentium III platform. Total storage (in bytes) increases only by about 7%.

point values and 32-bit integers on a particular machine, then $\gamma = 2$. The total size $V_{rc}(A)$ of the matrix data structure in floating point words is:

$$\begin{aligned} V_{rc}(A) &= \underbrace{K_{rc} \cdot rc}_{\text{values}} + \underbrace{\frac{1}{\gamma} K_{rc}}_{\text{col. indicies}} + \underbrace{\frac{1}{\gamma}\left(\left\lceil \frac{m}{r} \right\rceil + 1\right)}_{\text{row ptrs.}} \\ &= k f_{rc}\left(1 + \frac{1}{\gamma rc}\right) + \frac{1}{\gamma}\left(\left\lceil \frac{m}{r} \right\rceil + 1\right) \end{aligned} \tag{3.1}$$

If there were little or no fill (*e.g.*, for a dense matrix stored in sparse format), then increasing the block size from 1×1 to $r \times c$ would reduce the overhead for storing the column indices by a factor of rc. To gain an intuitive understanding of Equation (3.1), consider the case when $k \gg m$, so that we can ignore the row pointers, and $\gamma = 2$. (The ratio $\frac{k}{m}$ is typically $O(10)$ to $O(100)$, as shown in Appendix B.) Then, the compression ratio, or ratio of unblocked

storage to blocked storage, can be approximated as follows:

$$\frac{V_{1,1}(A)}{V_{rc}(A)} \approx \frac{\frac{3}{2}k}{kf_{rc}\left(1+\frac{1}{2rc}\right)} = \frac{3}{2}\cdot\frac{1}{f_{rc}\left(1+\frac{1}{2rc}\right)} \tag{3.2}$$

Thus, the maximum compression ratio is $\frac{3}{2}$ if we can choose a sufficiently large block size without any fill. A corollary is that in order to maintain the same amount of storage as the unblocked case, we can tolerate fill ratios of at most $\frac{3}{2}$. This observation explains why storage increased only by a modest amount in the example of Figure 3.2.

In SPARSITY, register blocking is implemented by (1) a special code generator to output the $r\times c$ code shown in Figure 3.1, and (2) a heuristic for selecting r and c, given the matrix. We defer a discussion of and subsequent improvement to the SPARSITY Version 1 heuristic described by Im [164, 167] to Section 3.2.

3.1.2 Surprising performance behavior in practice

By analogy to tiling in the dense case, the most difficult aspect of applying register blocking is knowing on which matrices to apply it and how to select the block size. This fact is illustrated for a particular sparse matrix in the example of Section 1.3 (Figure 1.3). There, we see experimentally the surprising performance behavior as the block size varies, motivating our use of automated empirical search.

What may be even more surprising is that the irregular behavior occurs even if we repeat the same experiment on a very "regular" sparse problem: the case of a *dense* matrix stored in BCSR format, as we show below. The implication is that irregular memory access alone does not explain irregularities in performance. Below, we argue that the performance behavior we might reasonably expect can differ considerably from what we observe in practice.

In the case of a dense matrix in BCSR format, there is no fill, assuming the dimensions of the matrix are either sufficiently large or a multiple of the block size. We could reasonably expect performance to increase smoothly with increasing r, c for the following reasons:

- *The storage overhead decreases with increasing rc.* We only need to store 1 integer index per block of rc non-zero values.
- *The instruction overhead per flop decreases with increasing rc.* Because we have unrolled the $r\times c$ block multiply, the innermost loop contains a constant number of

integer operations per $2rc$ flops. Specifically, the loop in Figure 3.1, line S3a executes 1 branch (at the end of each iteration), 1 loop bound comparison (line S3a), 2 iteration variable updates (line S3a), and 1 integer load (line S3b) for every $2rc$ flops (lines S4b–d).

- *There should be no significant instruction cache thrashing issues, provided we limit the register block size.* When does the fully unrolled block multiply exceed the instruction cache capacity? The innermost loop contains $3rc$ instructions for the multiplies, adds, and loads, if we ignore the $O(1)$ number of integer operations as r and c become large. The size of the innermost loop is $24rc$ bytes if we generously assume 8 bytes per instruction. The smallest L_1 instruction cache (I-cache) of our test platforms is 8 KB, so the largest dimension for a square block size in which the unrolled code will still fit in the I-cache is $\sqrt{8192/24} \approx 18$. In our experiments, we will only consider block sizes up to 12×12.

- *The memory access pattern is regular for a dense matrix stored in BCSR format.* The code remains the same as that shown in Figure 3.1, but consecutive values of `j` in line S3b will have a regular fixed-stride pattern (assuming sorted column indices). In other words, from the processor's perspective, the source vector loads (line S4a) are executed as a sequence of stride 1 loads, in contrast to the case of a general sparse matrix in which the value of `j` in S3b could change arbitrarily across consecutive iterations. On machines with hardware prefetching capabilities, this regular access pattern should be detectable.

- *There should not be a significant degree of stalling due to branch mispredictions.* The cost of mispredictions can be high due to pipeline flushing. For instance, the Ultra 3 has a 14-stage pipeline, so a branch mispredict could in the worst case cause a 14-cycle stall. However, we claim that branch mispredicts cannot fully explain the observed performance irregularities. Suppose we choose the dimension of the dense matrix to be $n \sim O(1000)$. Then, the trip count of the innermost loop will be long relative to typical pipeline depths. Therefore, we can expect that common branch prediction schemes should predict that the branch be taken by default, with a mispredict rate of approximately $\frac{1}{n}$.

The main capacity limit should be the number of registers. To execute the code of Figure 3.1

(*bottom*), we need $r+c+1$ floating point registers to hold one matrix element, r destination vector elements, and c source vector elements. Thus, we would expect performance to increase smoothly with increasing rc, but only up to the limit that $r+c+1 \leq R$ where R is the number of visible machine registers.

The performance observed in practice does not match the preceeding expectations. Figures 3.3–3.6 show the performance (Mflop/s) of SpMV using $r\times c$ BCSR storage for a dense matrix as r and c vary from 1×1 up to 12×12. We show data for 8 platforms, organized in pairs by vendor/processor family. Within each plot, every $r\times c$ implementation is shaded by its performance and also labeled by its speedup relative to the 1×1 implementation. Table 3.1 shows relevant machine characteristics and summary statistics of these plots. The data largely confirm the main conclusions of the sparse matrix example shown in Section 1.3 (Figure 1.3):

- *Knowledge of the "natural" block size of a matrix coupled with knowledge of the number of floating point registers is insufficient to predict the best block size.* Performance increases smoothly with increasing rc on only 2 of the 8 platforms—the Ultra 3 and Pentium III-M—and, to a lesser extent, on the Ultra 2i. The drop-off in performance on the Ultra 2i (upper-right corner of Figure 3.3 (*top*)) occurs approximately when $r+c+1$ exceeds $R=16$, and therefore might be explained by register pressure. However, on the Pentium III-M which has only 8 registers, performance continues to increase as $r+c+1$ increases well beyond $R=8$ (Figure 3.4 (*top*)). On the Itanium 1 and Itanium 2, the best performance occurs when $c \leq 2$, while the machine has a considerable number of registers ($R=128$; see Figure 3.6).

- *Performance can be a very irregular function of $r\times c$, and varies between platforms.* Furthermore, the value of $r\times c$ which attains the best absolute performance varies from platform to platform (Table 3.1).

  Even within a processor family, there can be considerable variation between processor generations. For instance, compare the Pentium III to the more recently released Mobile Pentium III (Pentium III-M) platform (Figure 3.4). The two platforms differ qualitatively in that performance as a function of r and c is more smooth and flat on the Pentium III-M than on the Pentium III. Although absolute performance of SpMV is higher on the more recent Pentium III-M, performance as a fraction of peak is somewhat lower than on the older platform: in the best case, we can achieve

21.4% of peak on the Pentium III compared to 15.2% on the Pentium III-M. Indeed, the improvement in peak moving from the Pentium III to the Pentium III-M is not matched by an equivalent improvement in performance. The peak performance of the Pentium III-M is 1.6× faster than the Pentium III (Table 3.1), but the ratio of the maximum performance is only $122/107 \approx 1.14$ times faster, and the ratio of the median performance data is $120/88 \approx 1.36$ times faster.

Also consider differences between the Power3 and Power4 platforms (Figure 3.5): the Power3 performance is nearly symmetric with respect to r and c—compare the upper-left corner, where $r > c$, to the lower-right corner, where $r < c$. In contrast, performance is higher on the Power4 when $r > c$ compared to the case of $r < c$.

Performance on the Itanium 1 and Itanium 2 platforms (Figure 3.6) is characteristically similar in the sense that (a) performance is best when c is 2 or less and at particular values of r, (b) there is swath of values of r and c in which performance is worse than or comparable to the 1×1 performance, and (c) performance increases again toward the upper-right corner of the plot ($rc \gtrsim 16$). However, choosing the right block size is much more critical on the Itanium 2, where the maximum speedup is 4.07× (at 4×2), compared to 1.55× (at 4×1) on the Itanium 1.

- *Significant performance improvements are possible, even compared to tuned dense matrix-vector multiply (DGEMV).* (A list of DGEMV implementations to which we compare are described in Appendix B.) As shown in Table 3.1, the best SpMV performance is typically close to or in excess of tuned DGEMV preformance, the notable exception being the Ultra 3. On the other platforms, this observation indicates that a reasonable but coarse bound on SpMV performance is DGEMV performance, provided the sparse matrix possesses exploitable dense block structure. The two cases in which SpMV is faster than DGEMV (Ultra 2i and Pentium III) indicates that these DGEMV implementations can most likely be better tuned.

  On the Ultra 3, DGEMV runs at 17% of machine peak, compared to the best SpMV performance running at 5% peak. Nevertheless, blocked SpMV performance is about 1.8× faster than the 1×1 performance, indicating there is some value to tuning.

These results for a dense matrix in sparse format reaffirm the conclusions of Section 1.3. The main difference in the example we have just considered is that we have eliminated the

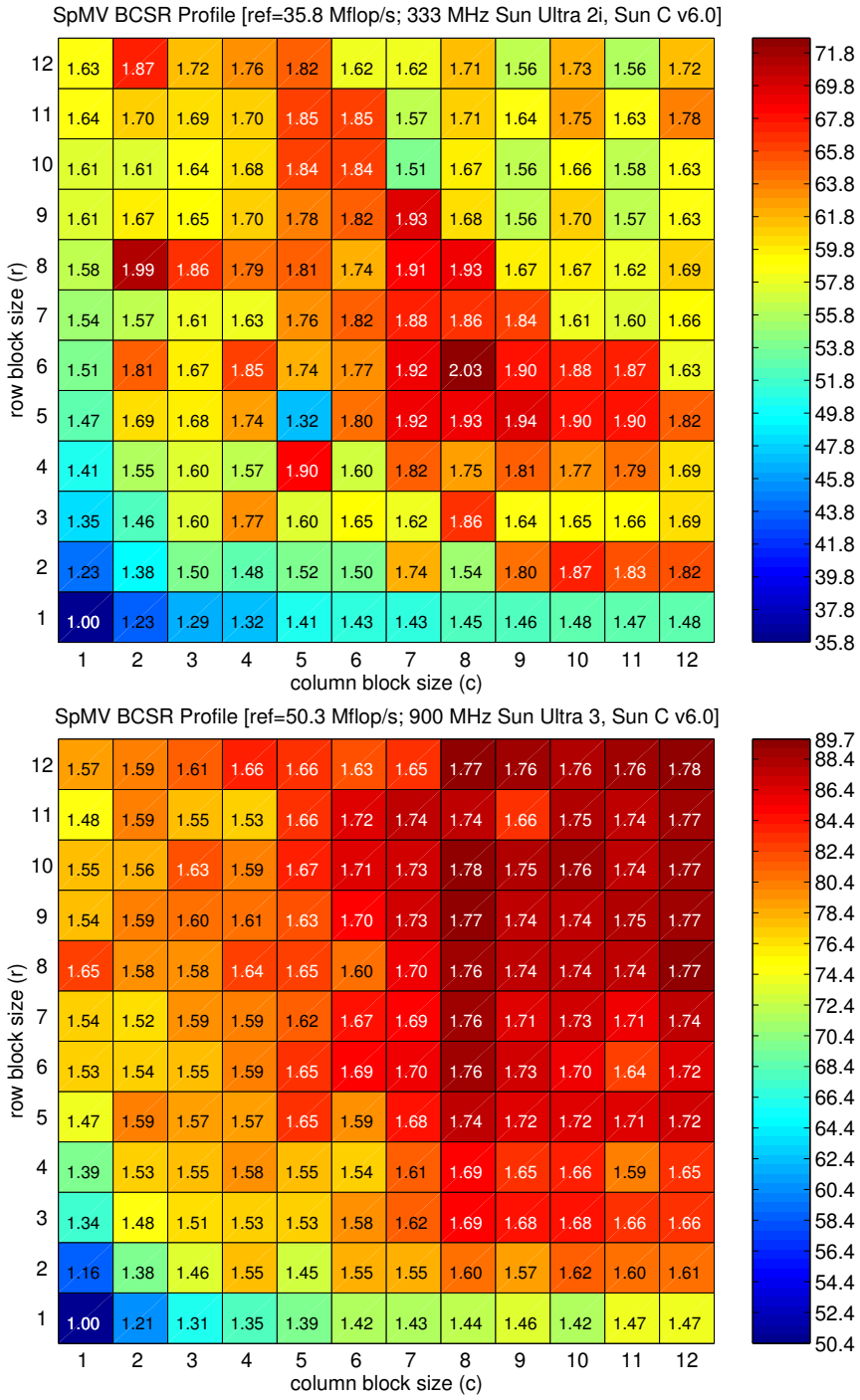

Figure 3.3: **SpMV BCSR Performance Profiles: Sun Platforms**. The performance (Mflop/s) of $r \times c$ register blocked implementations on a dense $n \times n$ matrix stored in BCSR format, on block sizes up to 12×12. Results shown for the Sun Ultra 2i (*top*) and Ultra 3 (*bottom*). On each platform, each square is an $r \times c$ implementation shaded by its performance, in Mflop/s. Each implementation is labeled by its speedup relative to the 1×1 implementation.

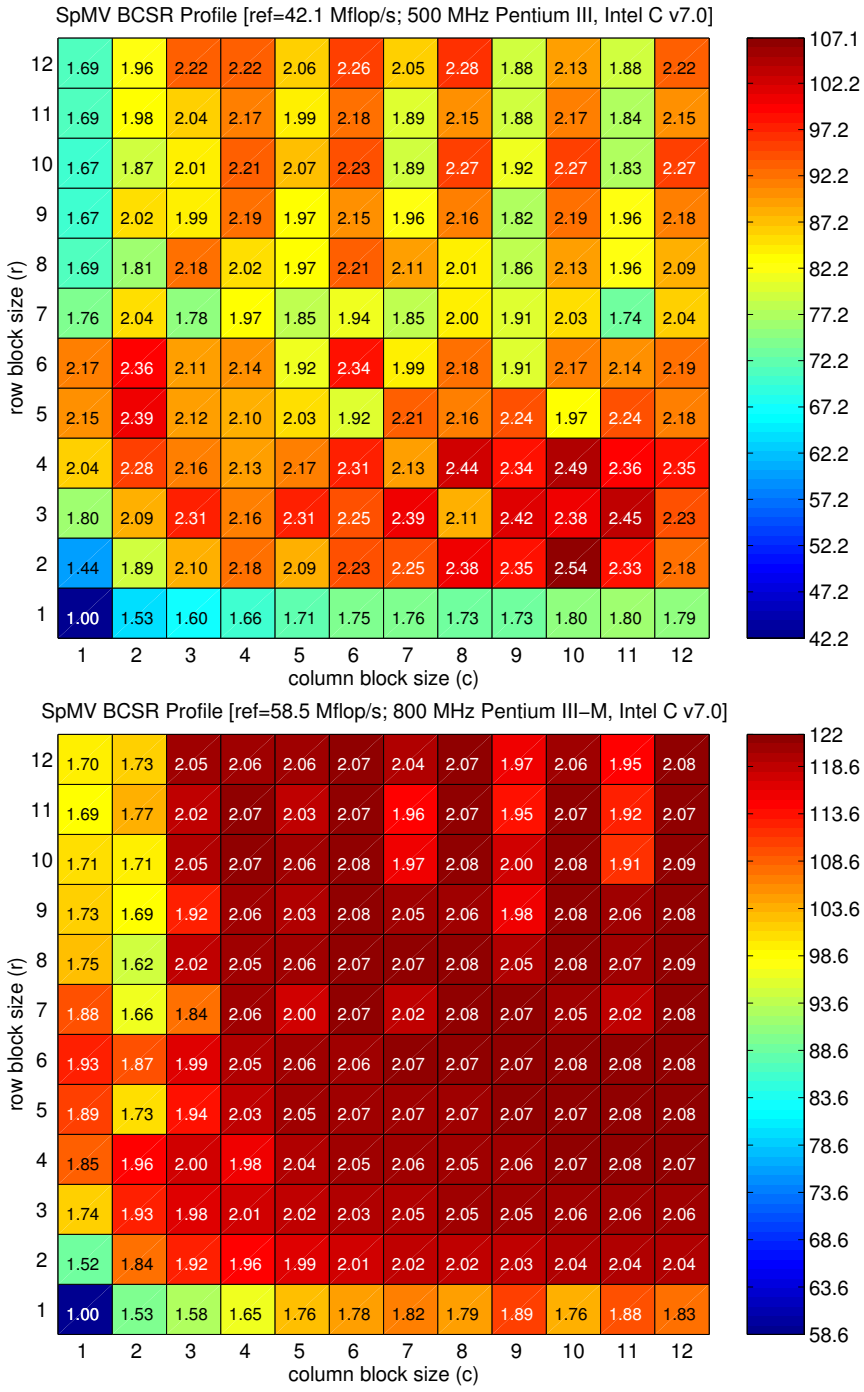

Figure 3.4: **SpMV BCSR Performance Profiles: Intel (x86) Platforms**. The performance (Mflop/s) of $r \times c$ register blocked implementations on a dense $n \times n$ matrix stored in BCSR format, on block sizes up to 12×12. Results shown for the Intel (x86) Pentium III (*top*) and Pentium III-M (*bottom*). On each platform, each square is an $r \times c$ implementation shaded by its performance, in Mflop/s. Each implementation is labeled by its speedup relative to the 1×1 implementation.

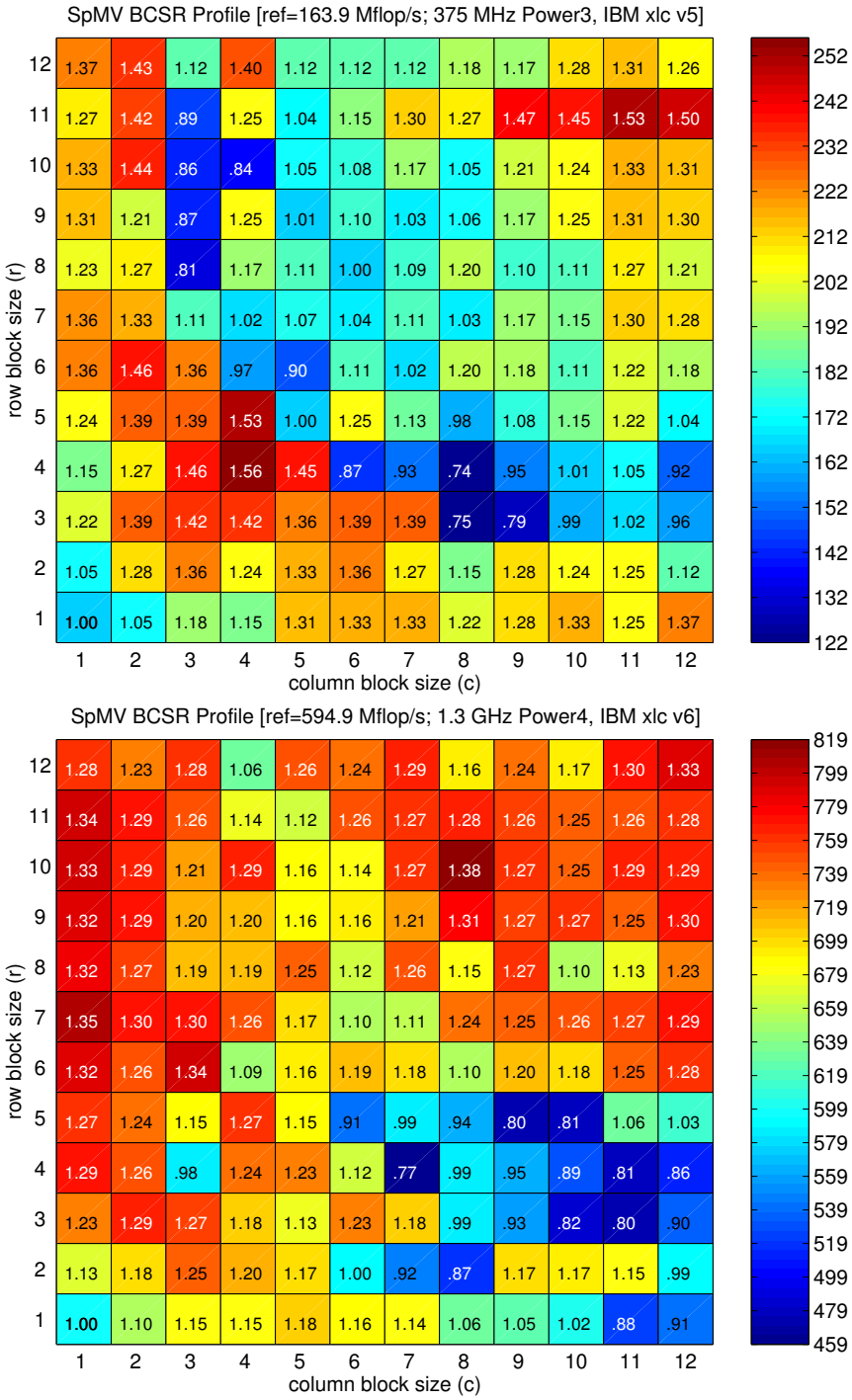

Figure 3.5: **SpMV BCSR Performance Profiles: IBM Platforms**. The performance (Mflop/s) of $r \times c$ register blocked implementations on a dense $n \times n$ matrix stored in BCSR format, on block sizes up to 12×12. Results shown for the IBM Power3 (*top*) and Power4 (*bottom*). On each platform, each square is an $r \times c$ implementation shaded by its performance, in Mflop/s. Each implementation is labeled by its speedup relative to the 1×1 implementation.

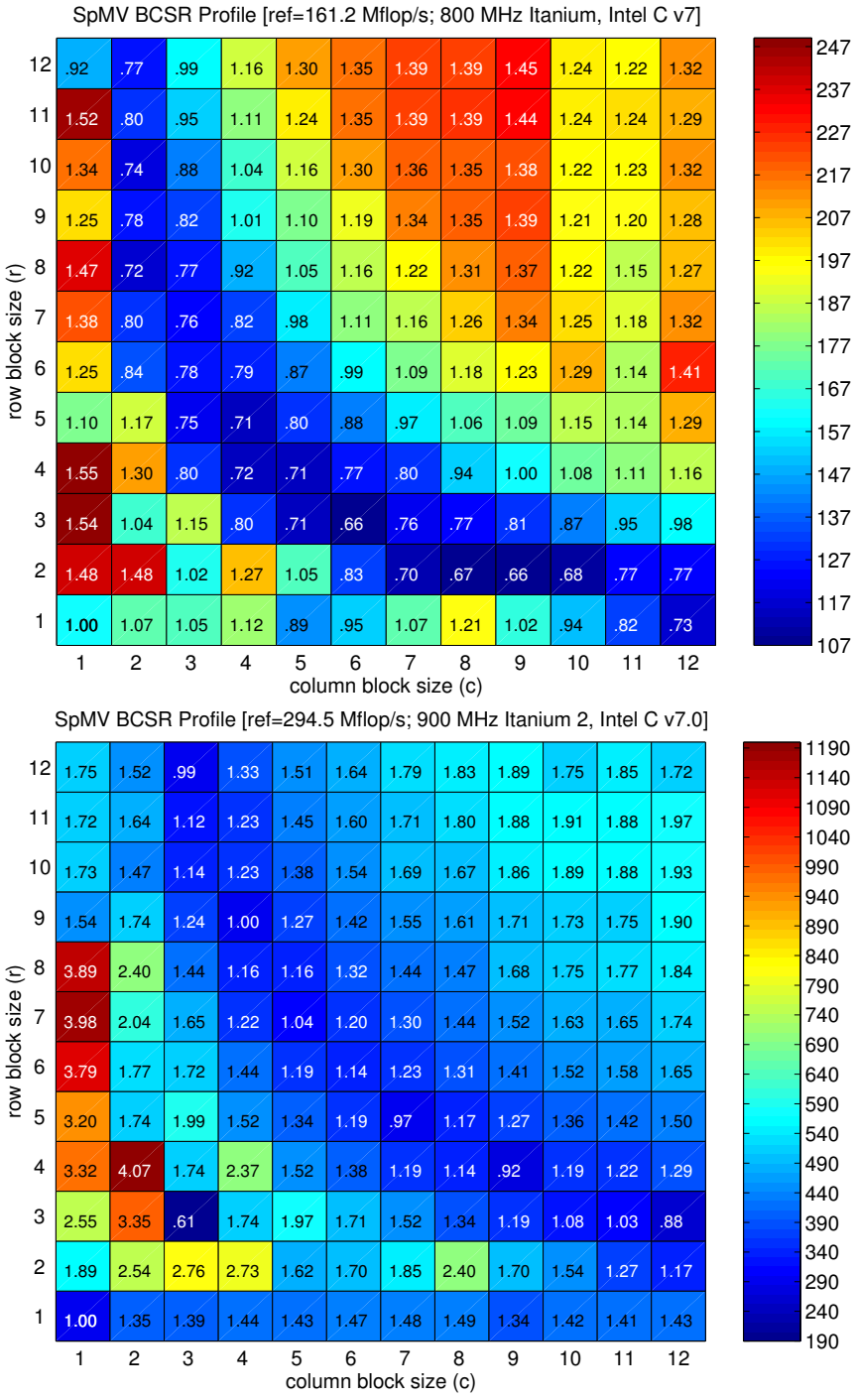

Figure 3.6: **SpMV BCSR Performance Profiles: Intel (IA-64) Platforms**. The performance (Mflop/s) of $r \times c$ register blocked implementations on a dense $n \times n$ matrix stored in BCSR format, on block sizes up to 12×12. Results shown for the Intel (IA-64) Itanium (*top*) and Itanium 2 (*bottom*). On each platform, each square is an $r \times c$ implementation shaded by its performance, in Mflop/s. Each implementation is labeled by its speedup relative to the 1×1 implementation.

Platform				*Dense Matrix Register Profile*			
Processor Compiler (no. of regs)	Peak Mflop/s	DGEMM Mflop/s	DGEMV Mflop/s	1×1 Mflop/s	Best Mflop/s	$r\times c$	Median Mflop/s
Ultra 2i Sun cc v6 (16)	667	425 [.637]	59 [.088]	36 [.053]	73 [.109]	6×8	60 [.089]
Ultra 3 Sun cc v6 (32)	1800	1600 [.888]	311 [.172]	50 [.027]	90 [.050]	12×12	82 [.045]
Pentium III Intel C v7 (8)	500	330 [.660]	58 [.116]	42 [.084]	107 [.214]	2×10	88 [.176]
Pentium III-M Intel C v7 (8)	800	640 [.800]	147 [.183]	59 [.073]	122 [.152]	8×12	120 [.15]
Power3 IBM xlc v7 (32)	1500	1300 [.866]	260 [.173]	164 [.109]	256 [.170]	4×4	198 [.132]
Power4 IBM xlc v7 (32)	5200	3500 [.673]	915 [.175]	595 [.114]	819 [.157]	10×8	712 [.136]
Itanium 1 Intel C v7 (128)	3200	2100 [.656]	315 [.098]	161 [.050]	250 [.078]	4×1	178 [.055]
Itanium 2 Intel C v7 (128)	3600	3500 [.972]	1330 [.369]	295 [.081]	1200 [.333]	4×2	451 [.125]

Table 3.1: **Summary of SpMV register profiles (dense matrix)**. We summarize a few machine characteristics (no. of double-precision floating point registers, peak machine speed, and performance of tuned double-precision dense matrix-matrix and matrix-vector multiply routines, DGEMM and DGEMV) and data from Figures 3.3–3.6. Performance data are shown in Mflop/s, with fraction of peak shown in square brackets.

run-time source of irregular memory access patterns. Thus, the irregularity of performance behavior as a function of r and c is not due only to irregular memory access. To make a direct comparison between sparse profiles of Figure 1.3 and the dense profiles of Figures 3.3–3.6, consider the set of $r\times c$ values such that $r, c \in \{1, 2, 4, 8\}$. The absolute performance data is lower in sparse profiles than in the dense. However, the performance relative to the 1×1 performance in each case is qualitatively similar, though not exactly the same.

Instead, the complexity of performance we have observed most likely reflects both

the overall complexity of the underlying hardware and the difficulty of optimal instruction scheduling. Even for earlier, simpler pipelined RISC architectures, it is well-known that the off-line (compiler) problem of optimally scheduling a basic block is NP-complete [155]. Thus, any sequence of instructions emitted by the compiler is most likely an approximation to the best possible schedule (which will in turn depend on load latencies that vary with where in the memory hierarchy data resides). However, what is encouraging about these data is that, compared to DGEMV, reasonably good performance for SpMV appears to nevertheless be possible, *provided* the right tuning parameters (a good block size) can be selected.

3.2 An Improved Heuristic for Block Size Selection

Both the SPARSITY Version 1 heuristic and Version 2 heuristic are based on the idea that the data of Figures 3.3–3.6 captures the irregularities of performance as r and c vary, and that the fill ratio quantifies how many extra flops will be performed due to explicit zeros. The Version 2 heuristic is as follows:

1. Once per machine, compute the *register (blocking) profile*, or the set of observed SpMV performance values (in Mflop/s) for a dense matrix stored in sparse format, at all block sizes from 1×1 to 12×12. Denote the register profile by $\{P_{rc}\,(\text{dense})\,|1 \leq r, c \leq 12\}$.

2. When the matrix A is known at run-time, compute an estimate $\hat{f}_{rc}\,(A, \sigma)$ of the true fill ratio $f_{rc}(A)$ for all $1 \leq r, c, \leq 12$. Here, σ is a user-selected parameter ranging from 0 to 1 which controls the accuracy of the estimate, as we describe in Section 3.2.1. (Our fill estimation procedure ensures that $\hat{f}_{rc}\,(A, 1) = f_{rc}(A)$.)

3. Choose r, c that maximizes the following *estimate* of register blocking performance $\hat{P}_{rc}\,(A, \sigma)$,

$$\hat{P}_{rc}\,(A, \sigma) = \frac{P_{rc}\,(\text{dense})}{\hat{f}_{rc}\,(A, \sigma)} \tag{3.3}$$

Although we refer to Equation (3.3) as a performance estimate, our interest is not to predict performance precisely, but rather to use this quantity to compute a relative ranking of block sizes.

The SPARSITY Version 1 heuristic implemented a similar procedure which chose r and c independently. In particular, the block size $r_{\mathrm{h}} \times c_{\mathrm{h}}$ was chosen by maximizing the following two ratios separately:

$$r_{\mathrm{h}} = \mathrm{argmax}_{1 \le r \le 12} \frac{P_{rr}\,(\mathrm{dense})}{\hat{f}_{r,1}\,(A, \sigma)} \tag{3.4}$$

$$c_{\mathrm{h}} = \mathrm{argmax}_{1 \le c \le 12} \frac{P_{cc}\,(\mathrm{dense})}{\hat{f}_{1,c}\,(A, \sigma)} \tag{3.5}$$

The Version 1 heuristic is potentially cheaper to execute than the Version 2 heuristic heuristic because we only need to estimate the fill for $r + c$ values, rather than for all $r \cdot c$ values. However, only the diagonal entries ($P_{i,i}\,(\mathrm{dense})$) of the profile contribute to the estimate. Performance along the diagonals of the profile do not characterize performance well in the off-diagonals on platforms like the Itanium 1 and Itanium 2 (Figure 3.6). Furthermore, we show that the cost of estimating the fill for all r and c, which grows linearly with σ, can be kept small relative to the cost converting the matrix (Section 3.2.2 and Section 3.3).

3.2.1 A fill ratio estimation algorithm

We present a simple algorithm for computing the fill ratio estimate $\hat{f}_{rc}\,(A, \sigma)$ of a matrix A stored in CSR format. This algorithm samples the non-zero structure, and is user-controlled by a tunable parameter σ specifies what fraction (between 0 and 1) of the matrix is sampled to compute the estimate, and thereby controls the cost and accuracy of the estimate. In particular, the cost of computing the estimate is $O(\sigma k r_{\max} c_{\max})$, where k is the number of non-zeros. This cost scales linearly with respect to σ. (Recall from the discussion of Section 3.1 that we fix $r_{\max} = c_{\max} = 12$.) Regarding accuracy, when $\sigma = 1$, the fill ratio is computed exactly. Following a presentation of the the algorithm, we discuss the accuracy and cost trade-offs as σ varies.

Basic algorithm

The pseudocode for the fill estimation algorithm is shown in Figure 3.7. The inputs to procedure **EstimateFill** are an $m \times n$ matrix A with k non-zeros stored in CSR format, the fraction σ, a *particular* row block size r, and the *maximum* column block size $c_{\max}$. The procedure returns the fill ratio estimates $\hat{f}_{rc}\,(A, \sigma)$ at a particular value of r and for all $1 \le c \le c_{\max}$. (We use the MATLAB-style colon notation to specify a range of indices,

EstimateFill(A, r, $c_{\max}$, σ):

1 Initialize array `Num_blocks`$[1 : c_{\max}] \leftarrow 0$ `/*` no. of blocks at each $1 \leq c \leq c_{\max}$ `*/`
2 Initialize `nnz_visited` $\leftarrow 0$ `/*` no. of non-zeros visited `*/`
3 **repeat** $\max(1, \sigma \lceil \frac{m}{r} \rceil)$ **times**
4 Choose a block of r consecutive rows $i_0 : i_0 + r - 1$ in A with $i_0 \mod r = 0$
5 Initialize array `Last_block_index`$[1 : c_{\max}] \leftarrow -1$
6 **foreach** non-zero $A(i, j) \in A\,(i_0 : i_0 + r - 1, :)$ in column-major order **do**
7 `nnz_visited` $\leftarrow$ `nnz_visited` $+ 1$
8 **foreach** $c \in 1 : c_{\max}$ **do**
9 **if** $\left\lfloor \frac{j}{c} \right\rfloor \neq$ `Last_block_index`$[c]$ **then**
`/*` $A(i, j)$ is the first non-zero in a new block `*/`
10 `Last_block_index`$[c] \leftarrow \left\lfloor \frac{j}{c} \right\rfloor$
11 `Num_blocks`$[c] \leftarrow$ `Num_blocks`$[c] + 1$
12 **return** $\hat{f}_{rc}\,(A, \sigma) \leftarrow$ `Num_blocks`$[c] \cdot r \cdot c /$ `nnz_visited`

Figure 3.7: **Pseudocode for a fill ratio estimation algorithm**. The inputs to the algorithm are the $m \times n$ matrix A, what fraction σ of the matrix to sample, a particular row block size r, and a range of column block sizes from 1 to $c_{\max}$. The output is the fill ratio estimate $\hat{f}_{rc}\,(A, \sigma)$ for all $1 \leq c \leq c_{\max}$.

e.g., "$1 : n$" is short-hand for $1, 2, \ldots, n$.) To compute the fill ratios for all $r_{\max} \cdot c_{\max}$ block sizes, we call the procedure for each r between 1 and $r_{\max}$.

EstimateFill loops over $\sigma \lceil \frac{m}{r} \rceil$ block rows of A (line 3), maintaining an array `Num_blocks`$[1 : c_{\max}]$ (line 1). Specifically, `Num_blocks`$[c]$ counts the total number of $r \times c$ blocks needed to store the block rows scanned in BCSR format. We discuss block row selection (line 4) in more detail below. **EstimateFill** enumerates the non-zeros of each block row (line 6) in "column-major" order: matrix entries are logically ordered so that entry $(i, j) < (i + 1, j)$ and $(m, j) < (0, j + 1)$, assuming zero-based indices for A, and entries are enumerated in increasing order. This ordering ensures that all non-zeros within the block row in column j are visited before those in column $j + 1$. To implement this enumeration efficiently, we require that the column indices of A within each row of CSR format be sorted on input. For each non-zero $A(i, j)$ and block column size c (line 8), if $A(i, j)$ does not belong to the same block column as the previous non-zero visited (line 9),

then we have encountered the first non-zero in a new $r \times c$ block (lines 10–11). We compute and return the fill ratios for all c based on the block counts `Num_blocks`$[c]$ and the total non-zeros visited (line 12).

There are a variety of ways to choose the block rows (line 4). For instance, block rows may be chosen uniformly at random, but must be done so *without replacement* to ensure that the estimates converge to truth when $\sigma = 1$. The Version 1 heuristic examined every $\lceil \frac{1}{\sigma} \rceil$-th block row for each r. We adopt the Version 1 heuristic convention in the Version 2 heuristic to ensure repeatability of the experiments.

The following is an easy way to enumerate the non-zeros of a given block row of a CSR matrix in column-major order, assuming sorted column indices. We maintain an integer array `Cur_index`$[1:r]$ which keeps a pointer to the current column index in each row, starting with the first. At each iteration, we perform a linear search of `Cur_index`$[1:r]$ to select the non-zero $A(i,j)$ with the smallest column index, and update the corresponding entry `Cur_index`$[i]$. The asymptotic cost of selecting a non-zero is $O(r)$.

Since the Version 1 heuristic only needs fill estimates at $r \times 1$ and $1 \times c$ block sizes, a simpler algorithm was used in the original SPARSITY implementation. However, the two algorithms return identical results on these block sizes, given the same convention for selecting block rows.

Asymptotic costs

The asymptotic cost of executing the procedure **EstimateFill** shown in Figure 3.7 for all $1 \leq r \leq r_{\max}$ is $O(\sigma k r_{\max} c_{\max})$, where k is the number of non-zeros in A. To simplify the analysis, assume that the number of rows $m \leq k$, and that every $r \leq r_{\max}$ divides m, and that $r_{\max} \sim O(c_{\max})$.

First, consider a single execution of **EstimateFill** for a fixed value of r, and for simplicity further assume that r divides m. The total cost is dominated by the time to execute lines 9–11 in the innermost loop, which each have $O(1)$ cost. The outermost loop in Figure 3.7 (line 3) executes $\sigma \frac{m}{r}$ times, assuming $\sigma \frac{m}{r} \geq 1$. The loop in line 6 executes approximately $r \frac{k}{m}$ times on average, assuming $\frac{k}{m}$ non-zeros per row. Thus, lines 9–11 will execute $c_{\max} \cdot r \frac{k}{m} \cdot \sigma \frac{m}{r} = \sigma k c_{\max}$ times. To execute **EstimateFill** for all $1 \leq r \leq r_{\max}$ will therefore incur a cost of $O(\sigma k \cdot r_{\max} c_{\max})$. This cost is linear with respect to k, and therefore has the same asymptotic costs as SpMV itself.

Since we assume an $O(r)$ linear search procedure to enumerate the non-zeros in column major order in line 6 of Figure 3.7, there is an additional overall cost of $O(\sigma k \cdot r_{\max}^2)$, bringing the total asymptotic costs to $O\left(\sigma k \cdot r_{\max}(r_{\max} + c_{\max})\right)$. Since we consider $r_{\max}$ and $c_{\max}$ to be of the same order, we can regard the overall cost as being $O(\sigma k r_{\max} c_{\max})$.

3.2.2 Tuning the fill estimator: cost and accuracy trade-offs

Although we know the cost of fill estimation varies linearly with σ, implementating the heuristic requires more precise knowledge of the true cost ("including constants") and how prediction accuracy varies with σ. In this section, we empirically evaluate the relationship among σ, the actual execution time of algorithm **EstimateFill**, and how closely the performance at the block size $r_{\mathrm{h}} \times c_{\mathrm{h}}$ selected by the heuristic approaches the best performance at the block size $r_{\mathrm{opt}} \times c_{\mathrm{opt}}$ determined by exhaustive search, on a few of the matrices in the test set. We present data which suggests that in practice, choosing $\sigma = .01$ keeps the actual cost of fill estimation to a few unblocked SpMVs, while yielding reasonably good accuracy.

We executed C implementations of the Version 2 heuristic (Section 3.2) on 3 matrices and 4 platforms (Ultra 2i, Pentium III-M, Power4, and Itanium 2), while varying σ. Platform characteristics (cache sizes, compiler flags) appear in Appendix B. The three matrices were taken also from Appendix B, and were selected because (1) their non-zero patterns exhibit differing structural characteristics, and (2) they were all large enough to exceed the size of the largest cache on the 4 platforms:

- Matrix 9-`3dtube` (pressure tube model): Matrix 9 arises in a finite element method (FEM) simulation, and consists mostly of a single uniformly aligned block size (96% of non-zeros are contained within dense 3×3 blocks). (See Appendix F for more detail on the non-zero distributions, alignments, and how these data are determined.)

- Matrix 10-`ct20stif` (engine block model): Matrix 10 also comes from an FEM application, but contains a mix of block sizes, aligned irregularly. (The 2 block sizes containing the largest fractions of total non-zeros are 6×6, which contains 39% of non-zeros, and 3×3, which contain 15% of non-zeros. Refer to Appendix F.)

- Matrix 40-`gupta1` (linear programming problem): Matrix 40 does not have any obvious block structure.

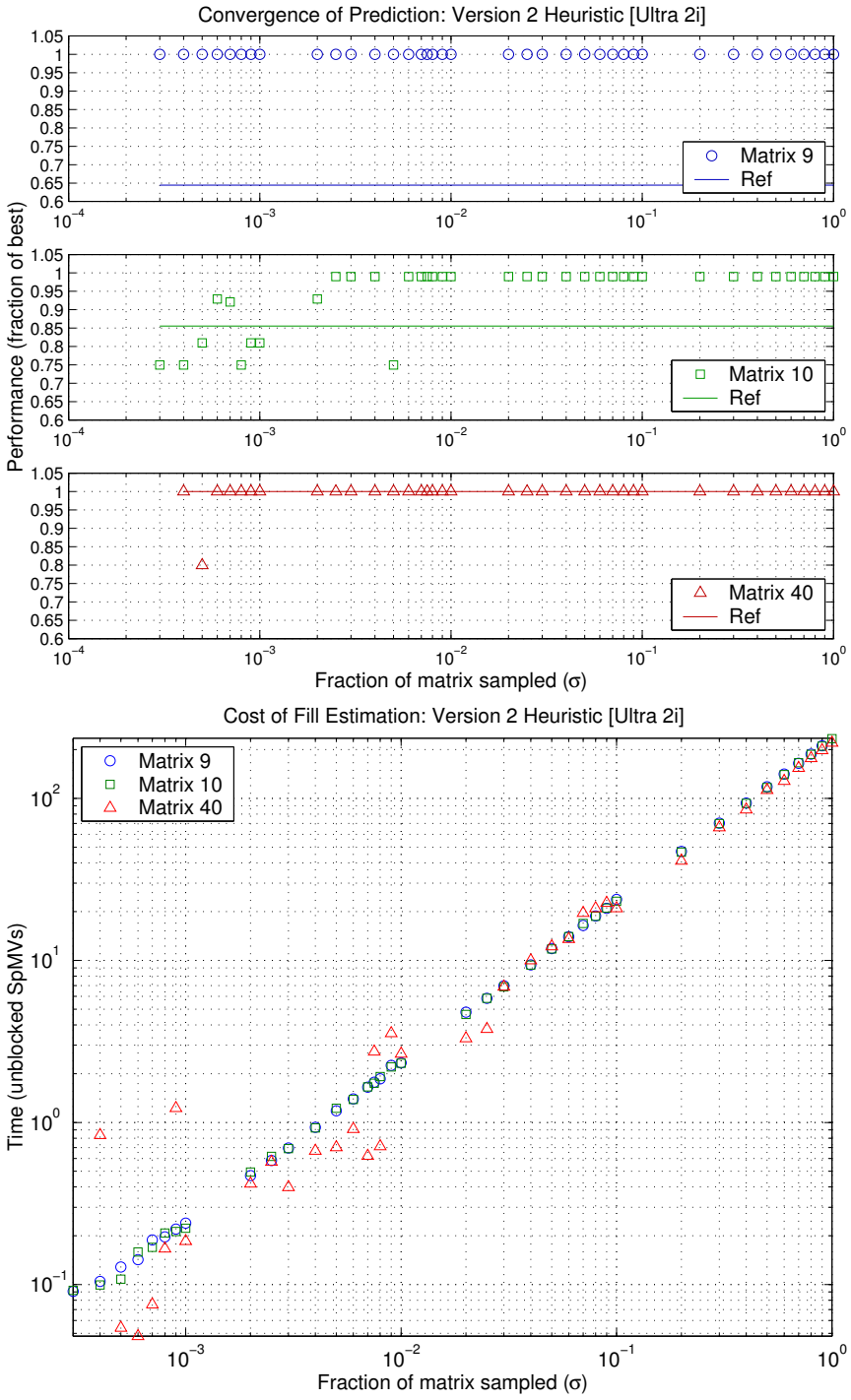

Figure 3.8: **Accuracy and cost trade-off example: Matrices 9, 10, and 40 on Ultra 2i**. (*Top*) Performance of the implementation chosen by the heuristic as σ varies. We show data for three test matrices, where performance (y-axis) is shown as a fraction of the best performance over all $1 \leq r, c \leq 12$. (*Bottom*) Cost of executing the heuristic as σ varies. Time (y-axis) is shown as multiples of the time to execute a single unblocked (1×1) SpMV on the given matrix. These data are tabulated in Appendix D.

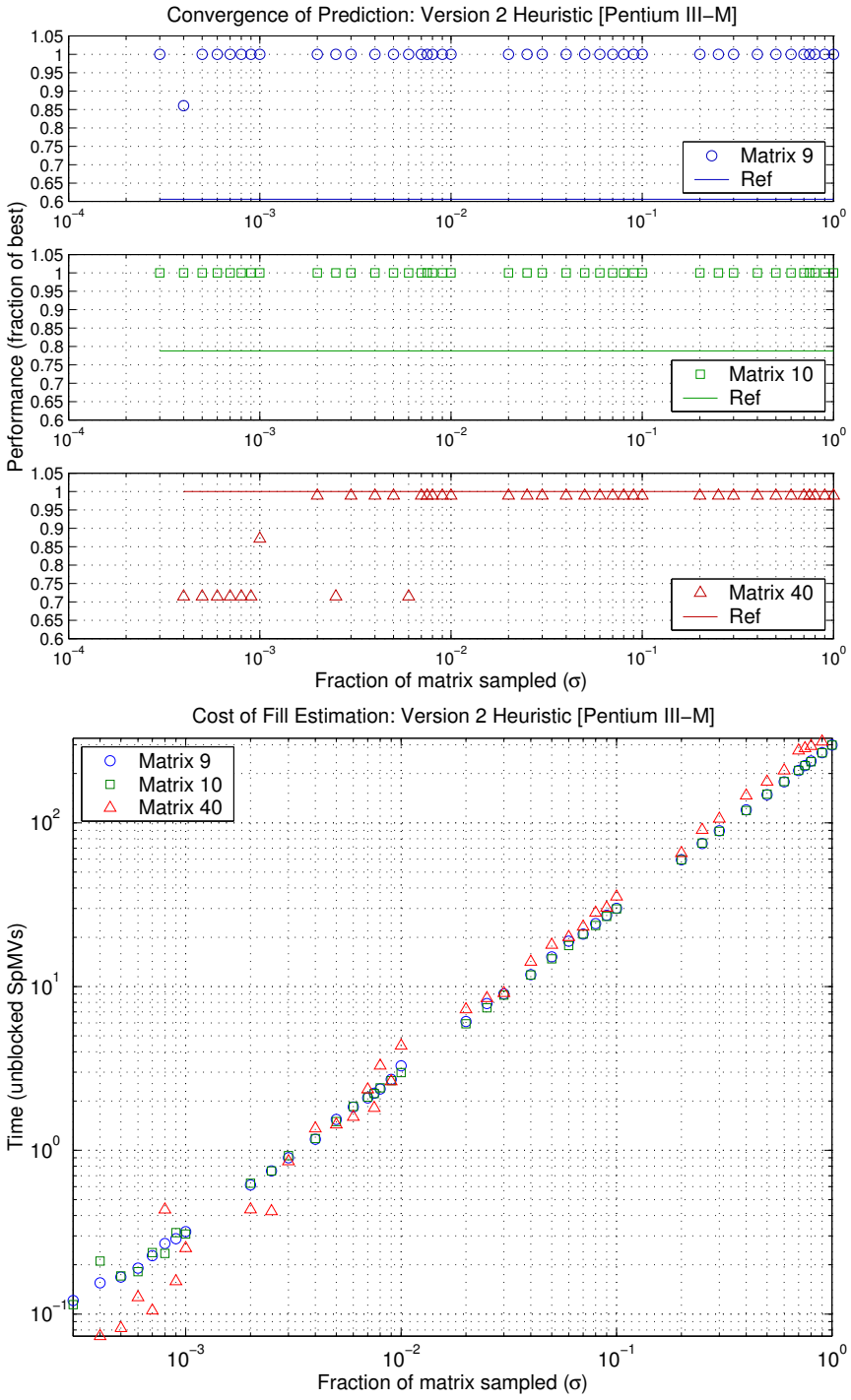

Figure 3.9: **Accuracy and cost trade-off example: Matrices 9, 10, and 40 on Pentium III-M**. (*Top*) Performance of the implementation chosen by the heuristic as σ varies. We show data for three test matrices, where performance (y-axis) is shown as a fraction of the best performance over all $1 \leq r, c \leq 12$. (*Bottom*) Cost of executing the heuristic as σ varies. Time (y-axis) is shown as multiples of the time to execute a single unblocked (1×1) SpMV on the given matrix. These data are tabulated in Appendix D.

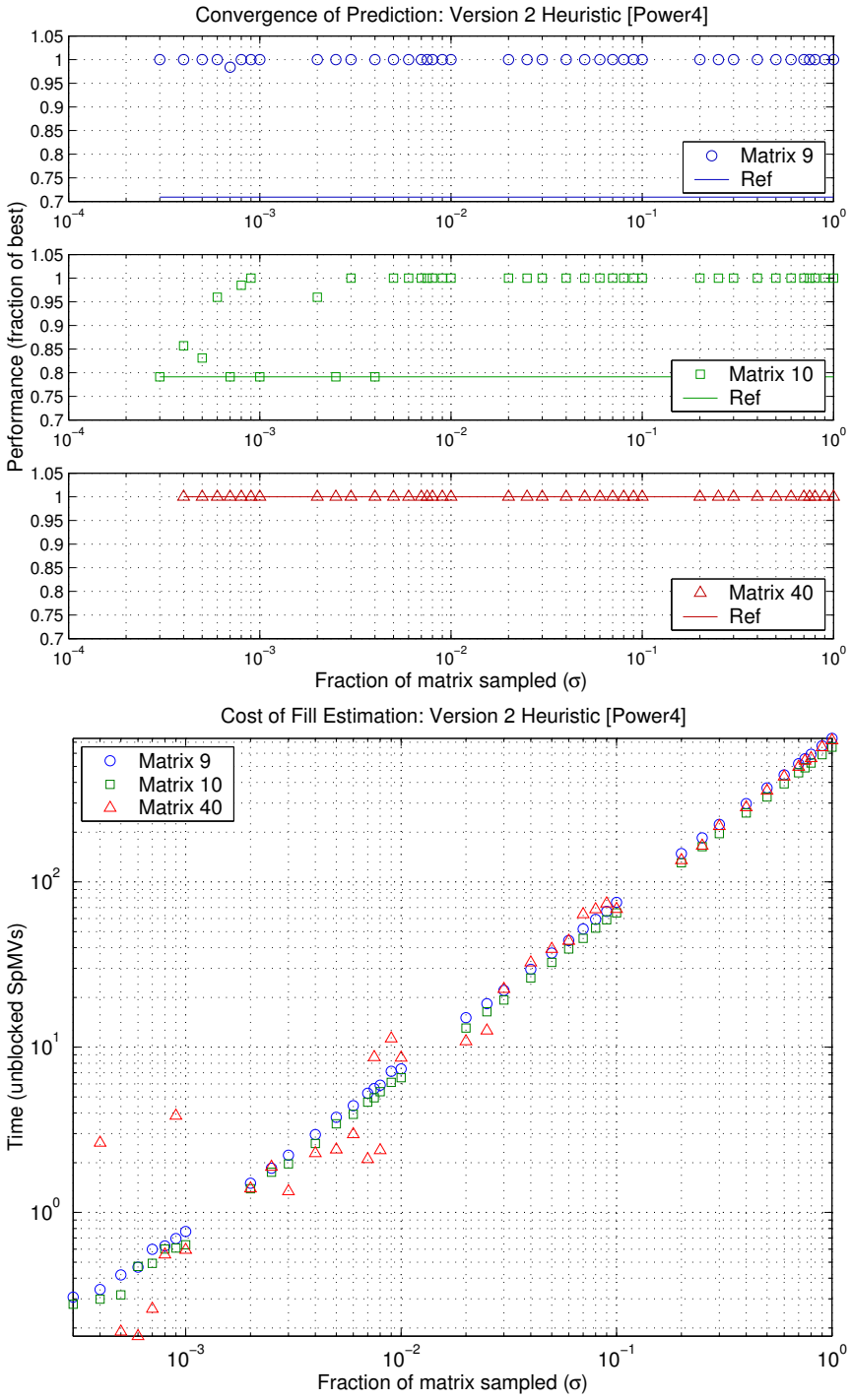

Figure 3.10: **Accuracy and cost trade-off example: Matrices 9, 10, and 40 on Power4.** (*Top*) Performance of the implementation chosen by the heuristic as σ varies. We show data for three test matrices, where performance (y-axis) is shown as a fraction of the best performance over all $1 \leq r, c \leq 12$. (*Bottom*) Cost of executing the heuristic as σ varies. Time (y-axis) is shown as multiples of the time to execute a single unblocked (1×1) SpMV on the given matrix. These data are tabulated in Appendix D.

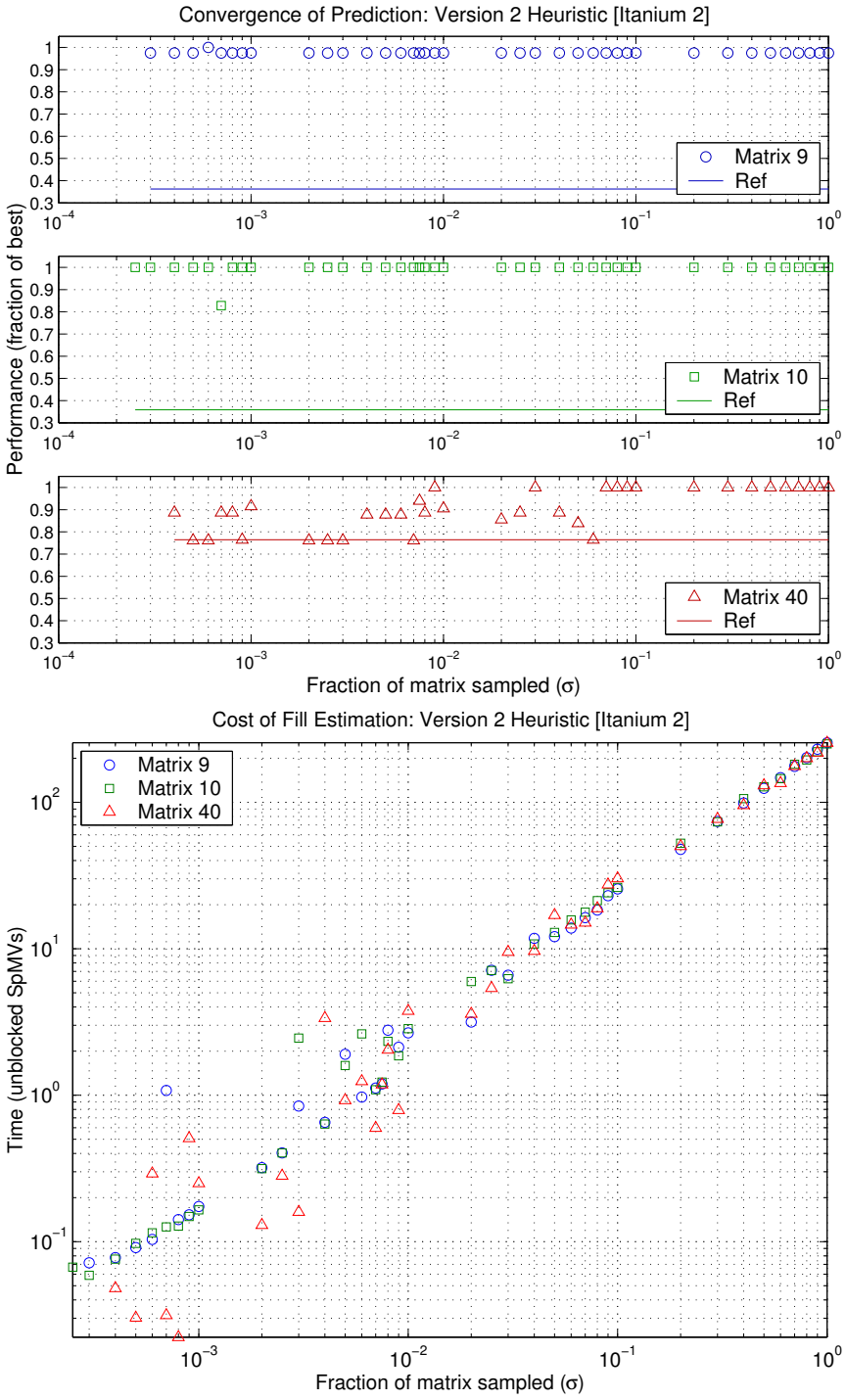

Figure 3.11: **Accuracy and cost trade-off example: Matrices 9, 10, and 40 on Itanium 2.** (*Top*) Performance of the implementation chosen by the heuristic as σ varies. We show data for three test matrices, where performance (y-axis) is shown as a fraction of the best performance over all $1 \leq r, c \leq 12$. (*Bottom*) Cost of executing the heuristic as σ varies. Time (y-axis) is shown as multiples of the time to execute a single unblocked (1×1) SpMV on the given matrix. These data are tabulated in Appendix D.

For each matrix, machine, and value of σ, we ran **EstimateFill** to predict a block size $r_{\mathrm{h}} \times c_{\mathrm{h}}$. We also ran exhaustive searches over all block sizes, and denote the best block size by $r_{\mathrm{opt}} \times c_{\mathrm{opt}}$. We also measured the time to execute **EstimateFill**. For each of the 4 platforms, Figures 3.8–3.11 present the following data:

- The performance of the $r_{\mathrm{h}} \times c_{\mathrm{h}}$ implementation as a fraction of the $r_{\mathrm{opt}} \times c_{\mathrm{opt}}$ implementation, for each of the 3 matrices (top 3 plots of Figures 3.8–3.11) and values of σ. We show the performance of the unblocked code by a solid horizontal line.

- The time to execute **EstimateFill**, in multiples of the time to execute the unblocked SpMV routine for each matrix (bottom plot of Figures 3.8–3.11).

(These data are also tabulated in Appendix D.) Using these figures, we can choose a value of σ and determine how close the corresponding prediction was to the best possible (top 3 plots of each figure), and the corresponding cost (bottom plot).

We make the following conclusions based on Figures 3.8–3.11:

1. *When the exact fill ratio is known ($\sigma = 1$), performance at the predicted block size is optimal or near-optimal on all platforms.* Performance at $r_{\mathrm{h}} \times c_{\mathrm{h}}$ is always within 5% of the best for these three matrices. This observation confirms that Equation (3.3) is a reasonable quantity to try to estimate.

   However, perfect knowledge of the fill ratio does not *guarantee* that the optimal block size is selected. For instance, the optimal performance and block size for Matrix 9 on Itanium 2 is 720 Mflop/s at 6×1 (see Table D.4). The heuristic selects 3×2, which runs at a near-optimal 702 Mflop/s. We emphasize that Equation (3.3) is a heuristic performance estimate.

2. *Mispredictions that lead to performance worse than the reference are possible, depending on σ.* Two notable instances are (1) Matrix 10 on Ultra 2i in a number of cases when $\sigma \leq .005$, and (2) Matrix 40 on Itanium 2 when $\sigma \leq .06$. Since Equation (3.3) does not predict performance perfectly even when the fill ratio is known exactly, we should always check at run-time to make sure that performance at the selected block size is not much worse than 1×1 SpMV. At a minimum, this will cost at least one additional SpMV.[1]

[1] Many modern machines support the PAPI hardware counter library, which provides access to CPU cycle

3. *At $\sigma = .01$, the cost of executing the heuristic is between 1 and 10 SpMVs.* This can be seen by observing the bottom plot of Figures 3.8–3.11. We conclude that this value of σ is likely to have a reasonable cost on most platforms.

4. *The predictions have stabilized by $\sigma = .01$ in all but one instance.* The predictions tend to be the same after this value of σ. The exception is Matrix 40 on the Itanium 2, where the predictions do not become stable until $\sigma \geq .07$. Examining the bottom plots of Figures 3.8–3.11, we see that the cost at this value of σ is about 11 SpMVs, but can range from 20–40 SpMVs on the other three platforms.

There are many ways to address the problem of how to choose σ in a platform and matrix-specific way. For instance, we could monitor the stability of the predictions as more of the matrix is sampled, while simultaneously monitoring the elapsed time so as not to exceed a user-specified maximum. (Confidence interval estimation is an example of a statistical technique which could be used to monitor and make systematic decisions regarding prediction stability [260].) However, in the remainder of this chapter, we settle on the use of $\sigma = .01$ on all platforms, where the observations above justify this choice as a reasonable trade-off between cost and prediction accuracy.

3.3 Evaluation of the Heuristic: Accuracy and Costs

This section evaluates the overall accuracy and total run-time cost of tuning using the Version 2 heuristic. We implemented the Version 2 heuristic according to the guidelines and results of Section 3.2, and evaluated the heuristic against exhaustive search on the 8 platforms and 44 test matrices listed in Appendix B. This data leads us to the following empirical conclusions:

1. *The Version 2 heuristic nearly always chooses an implementation within 10% of the best implementation found by exhaustive search in practice* (Section 3.3.1). The sole exception is Matrix 27 on Itanium 1, for which the heuristic selects an implementation which is 86% as fast as the best by exhaustive search.

   In addition, we find that even exact knowledge of the fill ratio ($\sigma = 1$) does not lead to significantly better predictions, confirming that our choice of $\sigma = .01$ is reasonable.

counters (as well as cache miss statistics), thus providing one portable way to use an accurate timer [60]. In addition, the most recent revision of the FFTW package (FFTW 3) also contains a standard interface just for reading the cycle counter, and is available on many additional platforms [123].

2. *The total cost of tuning, including execution of the heuristic and conversion to blocked format, is at most 43 unblocked SpMV operations in practice* (Section 3.3.2). This total cost depends on the machine, and can even be as low as 5–6 SpMVs (Ultra 3, Pentium III, and Pentium III-M).

Our implementation of the heuristic includes a run-time check in which the unblocked SpMV routine is also executed once to ensure that blocking is profitable. This additional execution is reflected in reported costs for the heuristic.

For each platform, we omit matrices which fit in the largest cache level, following the methodology outlined in Appendix B. The matrices are organized as follows:

- Matrix 1 (D): Dense matrix stored in sparse format, shown for reference.
- Matrices 2–9 (FEM 1): Matrices from FEM simulations. The majority of non-zeros in these matrices are located in blocks of the same size, and these blocks are uniformly aligned on a grid, as shown by solid black lines in Figure 3.2 (*right*).
- Matrices 10–17 (FEM 2): Matrices from FEM simulations where a mixture of block sizes occurs, or the blocks are not uniformly aligned, or both.
- Matrices 18–39 (Other): Matrices from non-FEM applications which tend not to have much if any regular block structure.
- Matrices 40–44 (LP): Matrices from linear programming applications which also tend not to have regular block structure.

The structural properties of the matrices are discussed in more detail in Chapter 5 and Appendix F.

This discussion focuses on the accuracy and cost of the heuristic. We revisit this performance data when comparing absolute performance to our upper bounds performance model in Chapter 4.

3.3.1 Accuracy of the Sparsity Version 2 heuristic

Figures 3.12–3.15 summarizes how accurately the Version 2 heuristic predicts the optimal block size. For each platform and matrix, we show

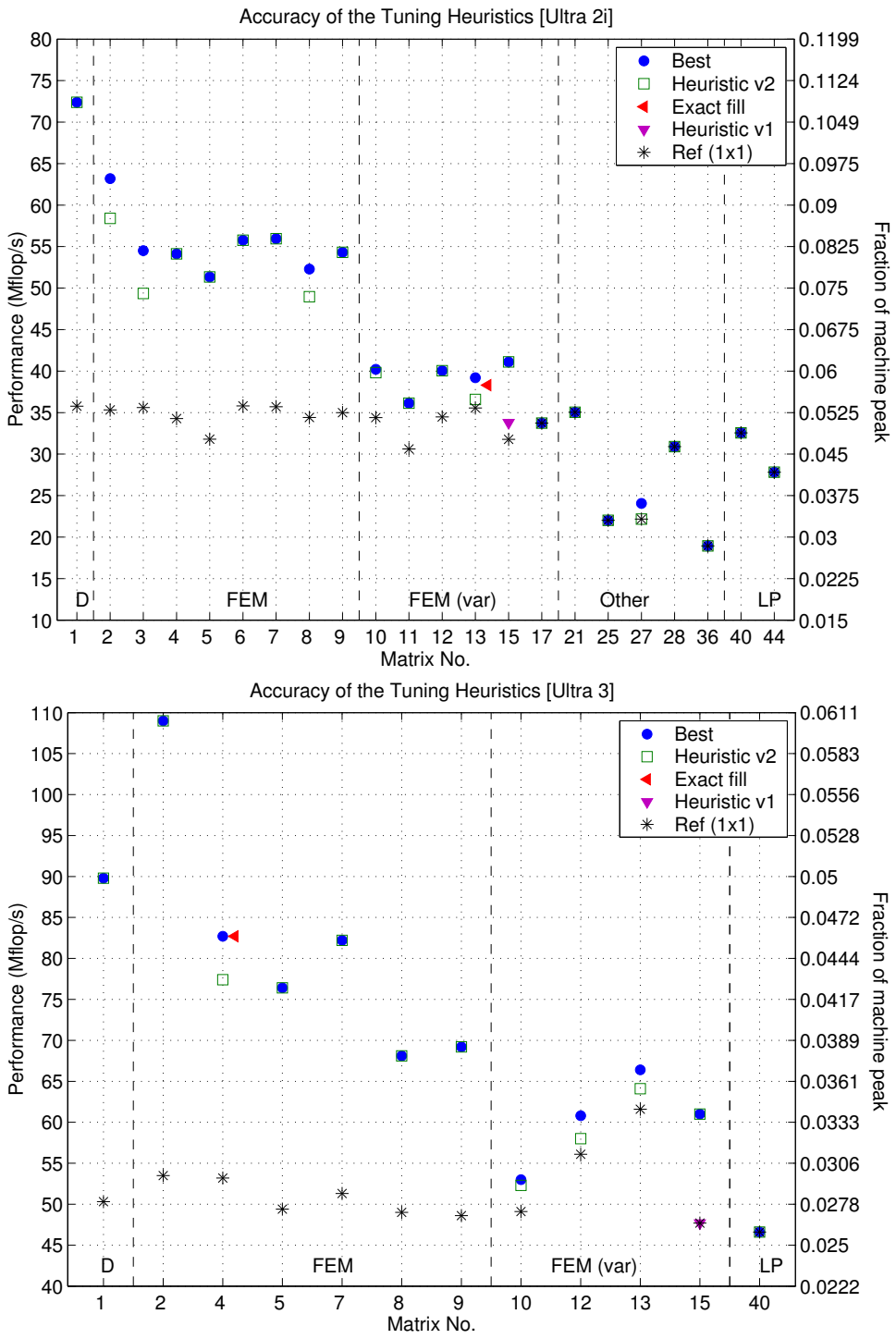

Figure 3.12: **Accuracy of the Version 2 heuristic for block size selection: Ultra 2i and Ultra 3**. (*Top*) Ultra 2i (*Bottom*) Ultra 3

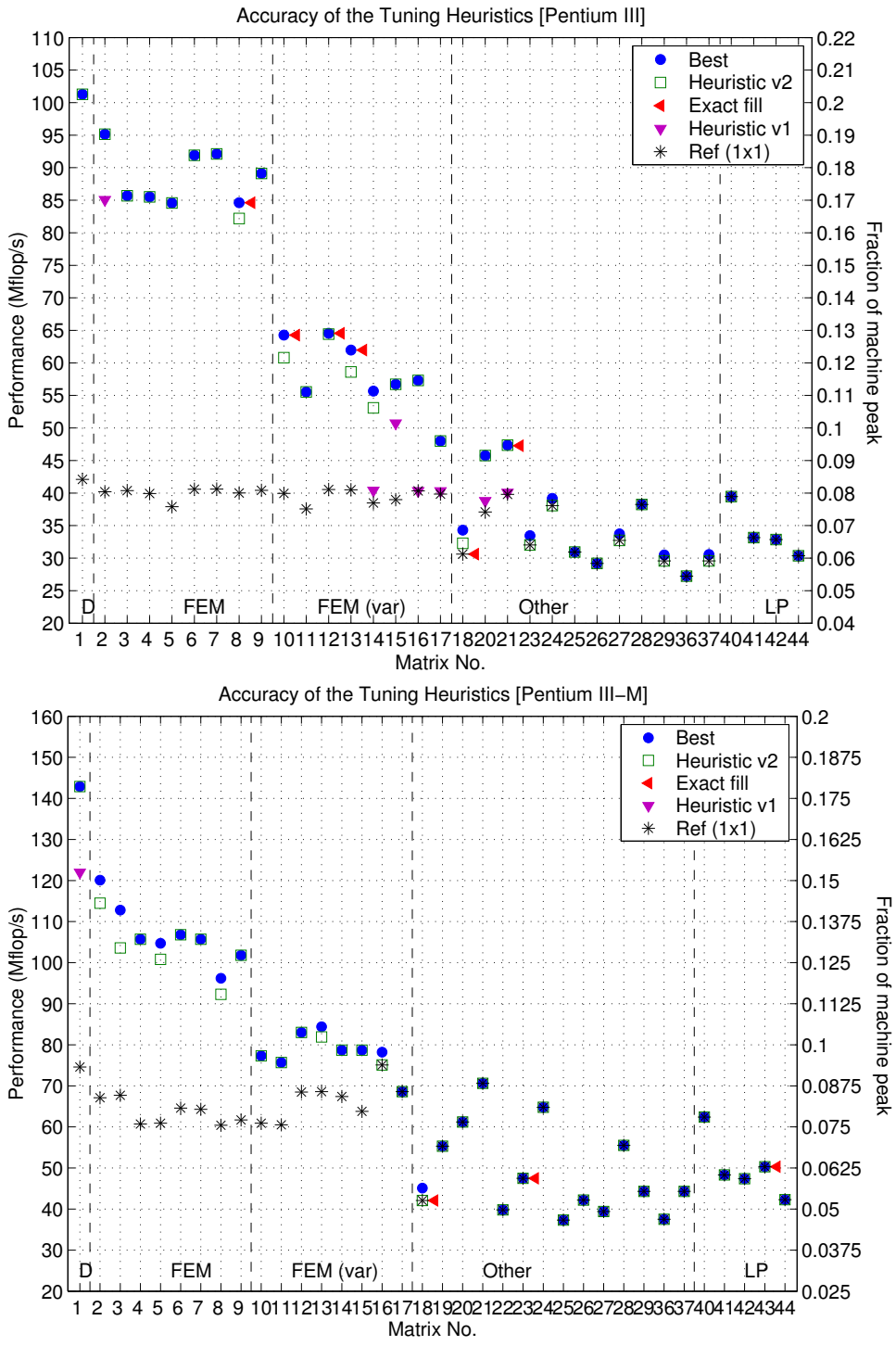

Figure 3.13: **Accuracy of the Version 2 heuristic for block size selection: Pentium III and Pentium III-M**. (*Top*) Pentium III (*Bottom*) Pentium III-M

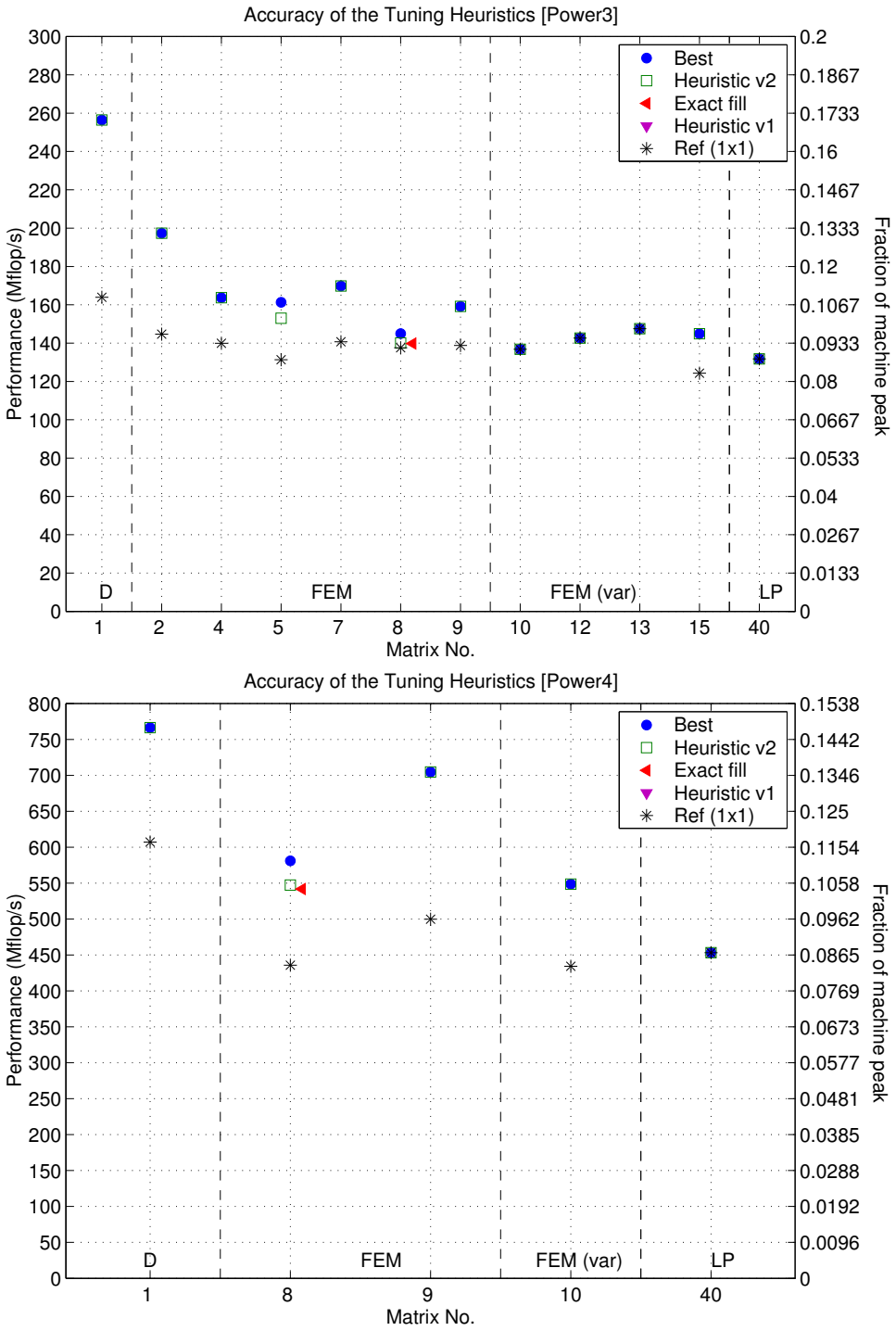

Figure 3.14: **Accuracy of the Version 2 heuristic for block size selection: Power3 and Power4**. (*Top*) Power3 (*Bottom*) Power4

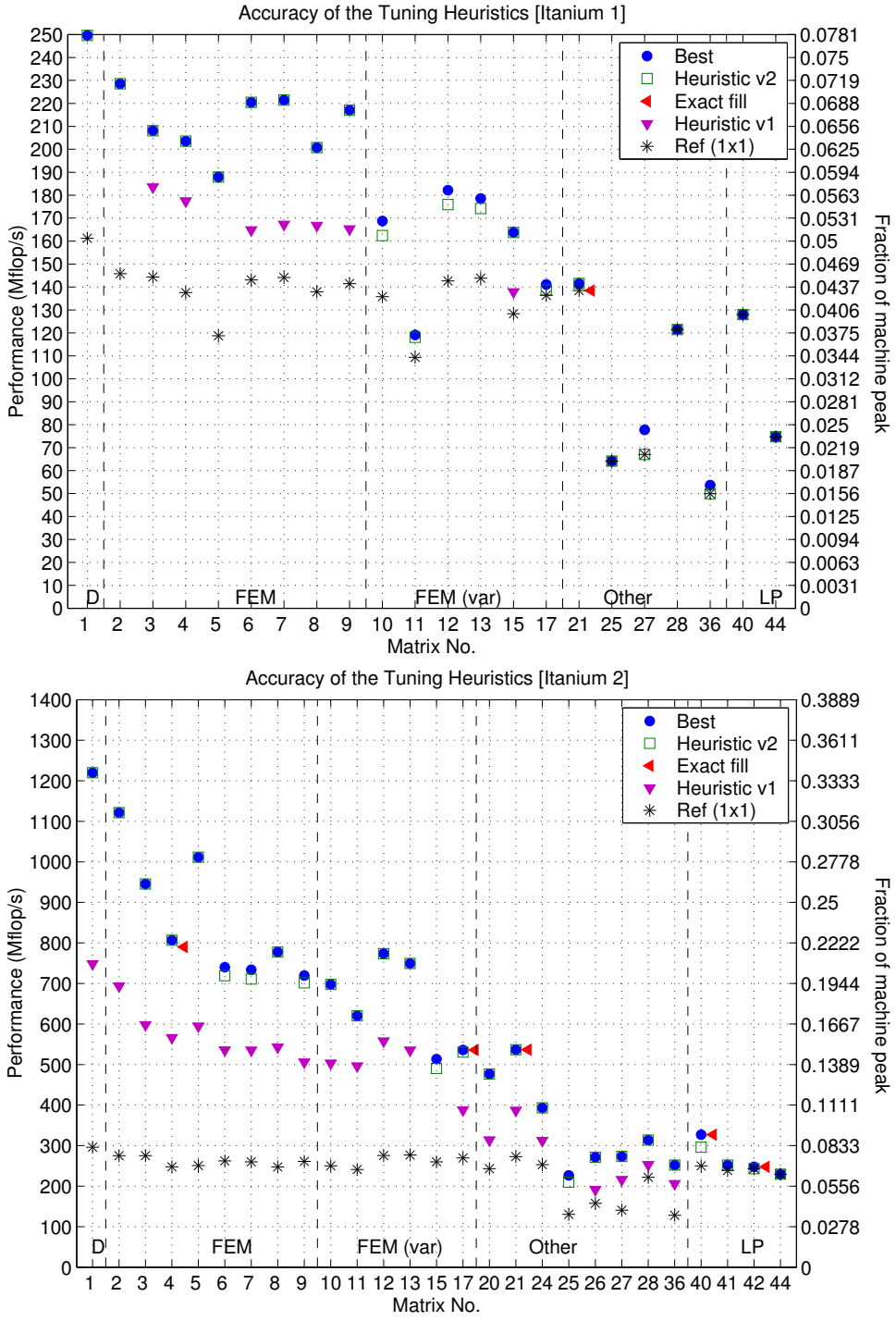

Figure 3.15: **Accuracy of the Version 2 heuristic for block size selection: Itanium 1 and Itanium 2.** (*Top*) Itanium 1 (*Bottom*) Itanium 2

- **Best** (solid blue dots): Performance of the best implementation selected by an exhaustive search. We denote the optimal block size by $r_{\text{opt}} \times c_{\text{opt}}$.
- **Version 2 heuristic** (hollow green squares): Performance of the implementation chosen by the Version 2 heuristic, where $\sigma = .01$. We denote the block size selected by $r_{\text{h}} \times c_{\text{h}}$.
- **Exact fill** (solid red left-pointing triangle): Performance of the implementation chosen by the Version 2 heuristic, where $\sigma = 1$. We denote the corresponding block size by $r_{\text{e}} \times c_{\text{e}}$. To reduce clutter, we only show this data point if $r_{\text{e}} \times c_{\text{e}}$ differs from $r_{\text{h}} \times c_{\text{h}}$. The presence of this data point indicates how much better we could do if we had exact knowledge of the fill ratio.
- **Version 1 heuristic** (solid purple downward-pointing triangles): Performance of the implementation chosen by Version 1 heuristic. To give the original heuristic maximum advantage, we choose exact knowledge of fill, *i.e.*, $\sigma = 1$. To reduce clutter, we only show this data point if the performance of the implementation chosen by the heuristic is either greater than or less than the $r_{\text{h}} \times c_{\text{h}}$ implementation by more than 10%.
- **Reference** (black asterisks): Performance of the unblocked (1×1) implementation.

(This data, including actual block sizes, is also tabulated in Appendix D.)

The performance at $r_{\text{h}} \times c_{\text{h}}$ is nearly always 90% or more of the performance at $r_{\text{opt}} \times c_{\text{opt}}$. In addition, the Version 2 heuristic noticeably improves prediction accuracy on the Itanium 1 and Itanium 2 platforms, where performance of the Version 1 heuristic is as low as 60% of the best (*e.g.*, Matrix 2 and Matrix 5 on Itanium 2, Figure 3.15). There is an improvement on Matrices 14–17 and Matrices 20–21 on Pentium III, Figure 3.13 (*top*), as well. Since the Version 1 heuristic only considers the diagonal component of the register profile, it must necessarily miss the strong irregularity of performance as a function of r and c, particularly on the Itanium 1 and Itanium 2 platforms.

We confirm that $\sigma = .01$ is a reasonable choice in practice. When $r_{\text{e}} \times c_{\text{e}}$ differs from $r_{\text{h}} \times c_{\text{h}}$, the difference in actual performance is never more than a few percent.

The Version 2 heuristic prediction is not within 90% in one instance: Matrix 27 on Itanium 1. In this case, heuristic selects the reference implementation, whereas the best implementation is actually the 3×1 implementation (Table D.11). The reference

Version 2 heuristic prediction						
Rank	$r \times c$	Register Profile P_{rc} (dense)	Fill $\hat{f}_{rc}(A, \sigma = 1)$	Performance Estimate $\hat{P}_{rc}(A, \sigma = 1)$	True Mflop/s	True Rank
1	1×1	161	1.00	161	67	4
2	2×1	238	1.53	156	72	2
3	3×1	248	1.94	128	78	1
4	1×2	173	1.53	113	45	15
5	4×1	250	2.39	105	72	3

Table 3.2: **Top 5 predictions compared to actual performance: Matrix 27 on Itanium 1**. We show the top 5 performance estimates after evaluating Equation (3.3) for Matrix 27 on Itanium 1. The true Mflop/s and rank at each block size are shown in the last two columns.

performance is 86% of the best performance. Although this performance is reasonable, let us consider the factors that cause the heuristic to select a suboptimal block size.

Inspection of Table D.11 reveals that even if the fill ratio were known exactly, Equation (3.3) still predicts that the 1×1 implementation will be the fastest. Table 3.2 shows the top 5 performance estimates after evaluation of Equation (3.3), compared to the actual observed Mflop/s and true ranking (last two columns). Columns 2–5 show the components of Equation (3.3). We see that 4 of the actual top 5 implementations—3×1, 2×1, 4×1, and 1×1—are indeed within the top 5 predictions, though the relative ranking does not otherwise precisely reflect truth.

There are at least two possible ways to handle cases such as this one. One approach is to perform a limited search among the top few predictions, if the cost of conversion is small relative to the expected number of uses. An alternative is to replace the dense profile values, P_{rc} (dense), with performance on some other canonical matrix, such as a random blocked matrix. Since the dense profile eliminates the kinds of irregular memory references which are relatively more pronounced in Matrices 18–44 than in Matrices 2–17, we might reasonably suspect mispredictions to occur. Indeed, there is a substantial gap in the magnitude of the performance estimate, $\hat{P}_{rc}(A, 1)$ compared to the actual observed performance, indicating that we might try to better match profiles to matrices.

Both possible solutions are avenues for future investigation. Nevertheless, for a wide range of machines and matrices, we conclude that heuristic as presented in Section 3.2 appears sufficient to predict reasonably good block sizes in the vast majority of cases.

3.3.2 The costs of register block size tuning

In addition to being reasonably accurate, we show that the total cost of tuning when $\sigma = .01$ is at most 43 unblocked SpMV operations on the test matrices and platforms. Furthermore, the time to tune tends to be dominated by the cost of converting the matrix from CSR to BCSR. We show these costs on 8 platforms in Figures 3.16–3.19. Each plot breaks down the total cost of tuning into two components for all matrices:

1. **Heuristic** (green bar): The cost (in unblocked SpMVs) of computing the fill estimate for $\sigma = .01$, selecting the block size by maximizing Equation (3.3), plus one additional execution of the unblocked SpMV if the predicted block size is not 1×1.

2. **Conversion** (yellow bar): The cost (in unblocked SpMVs) of converting the matrix from 1×1 to $r_h \times c_h$ format, plus the cost of executing the $r_h \times c_h$ routine once. If the heuristic determines that 1×1 is the best block size, then the conversion time is shown as zero. Each bar is also labeled above by the fraction of total time accounted for by this conversion cost.

We include the cost of executing both the unblocked and final blocked routine once in order to approximate the best case cost of a run-time check that ensures the selected block size is faster.

The cost of conversion is typically the most expensive of the two major steps. In only two instances is the cost of conversion less than 50% of the total cost: Matrix 1-`dense` on the Ultra 2i and Pentium III. When blocking is profitable, the cost of conversion constitutes as much as 96% of the total tuning cost (Matrix 28 on Itanium 2, Figure 3.19).

The cost of each step also varies from platform to platform. Figure 3.20 summarizes the data in Figures 3.16–3.19 across all platforms. Specifically, we show the minimum, median, and maximum costs of (1) evaluating the heuristic plus one unblocked SpMV (green solid circles), (2) converting the matrix plus one blocked SpMV (yellow solid squares), and (3) the total costs of both steps (red solid diamonds). For the conversion cost, we only compute the summary statistics in the cases in which a block size other than 1×1 was predicted by the heuristic. The following is a high-level summary of the main features of this data:

- The median total cost is just over 31 SpMVs in the worst case, on Itanium 1. The maximum total cost is 43 SpMVs, and also occurs on Itanium 1 though this cost is

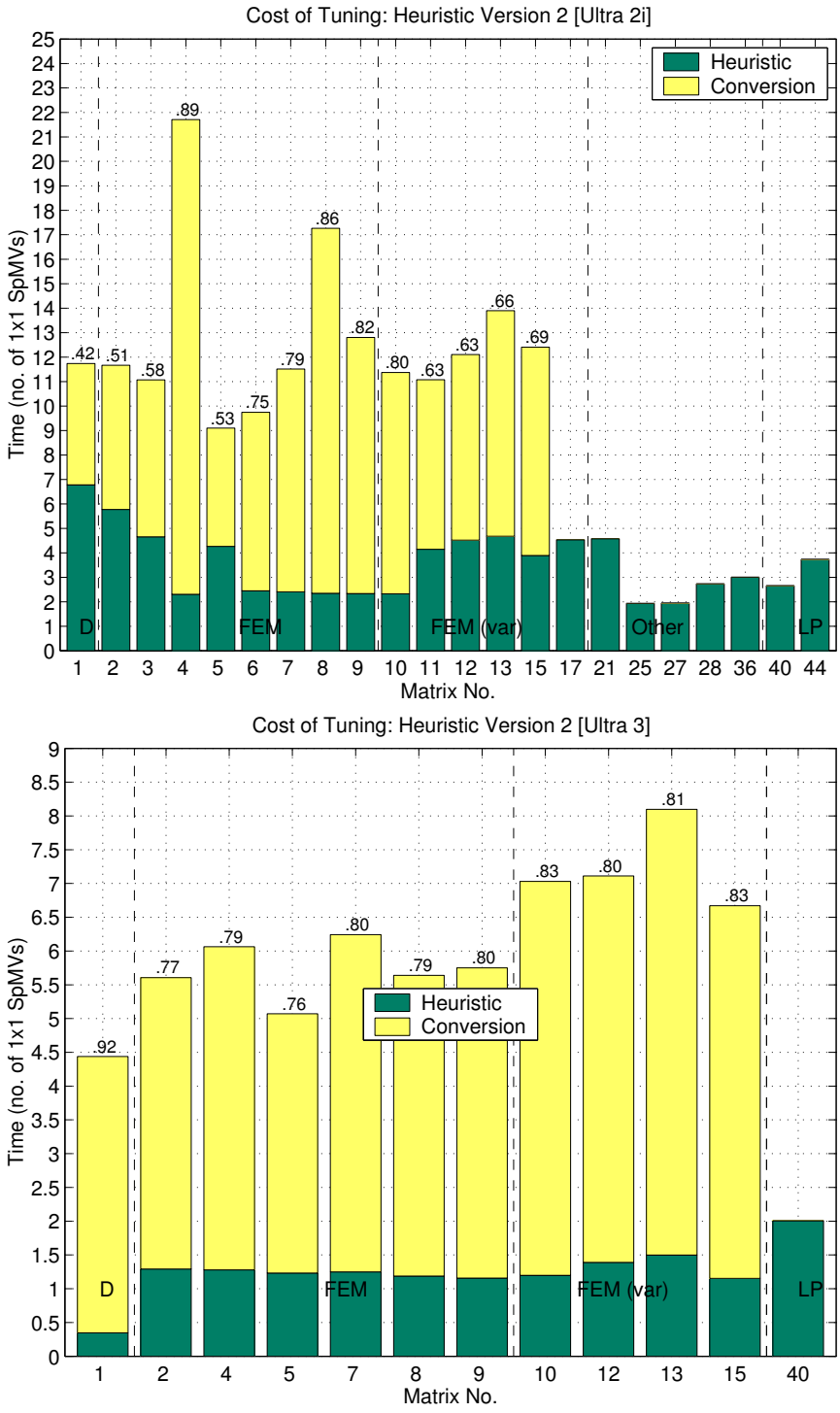

Figure 3.16: **Cost of register block size tuning: Ultra 2i and Ultra 3**. (*Top*) Ultra 2i (*Bottom*) Ultra 3

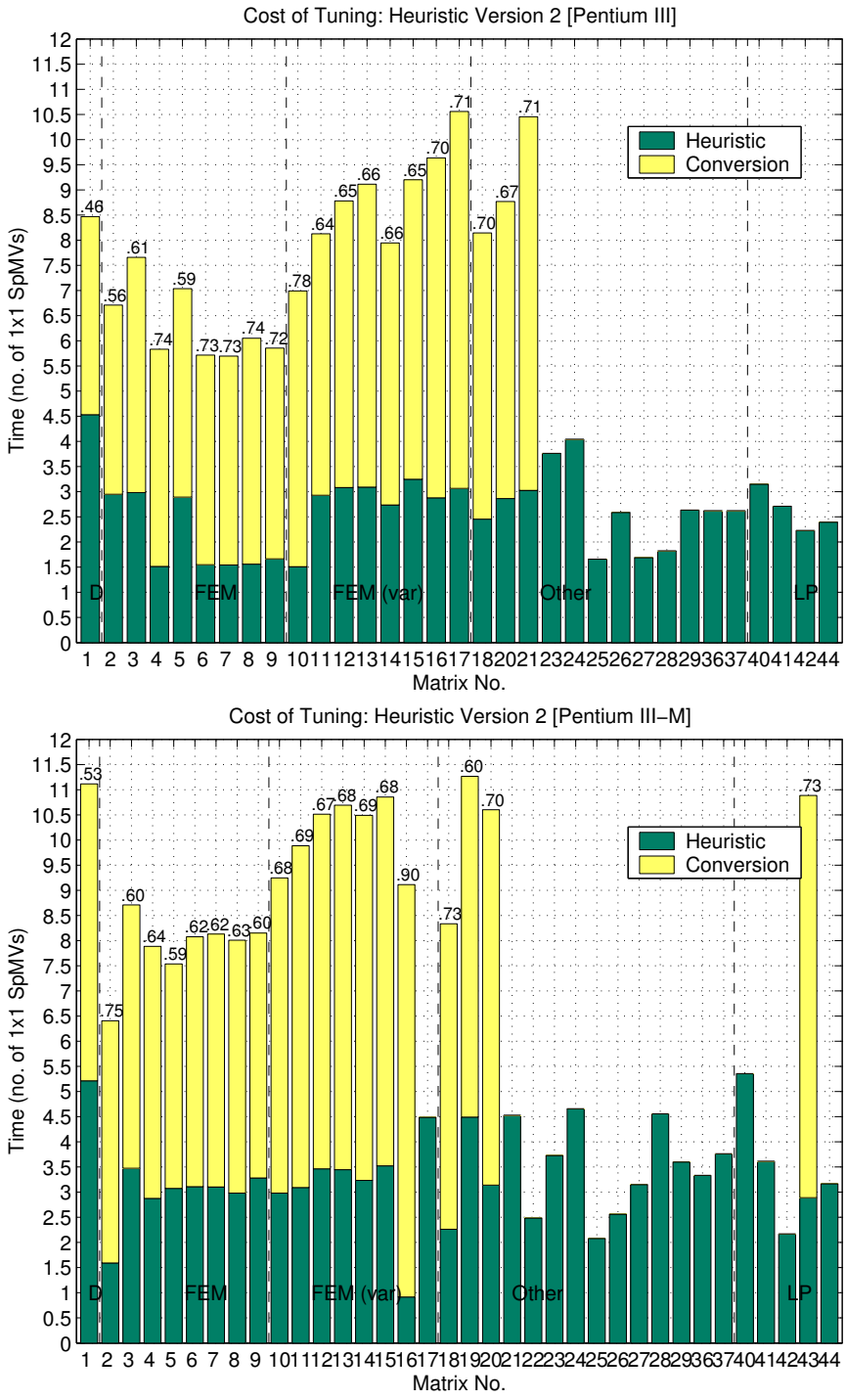

Figure 3.17: **Cost of register block size tuning: Pentium III and Pentium III-M**. (*Top*) Pentium III (*Bottom*) Pentium III-M

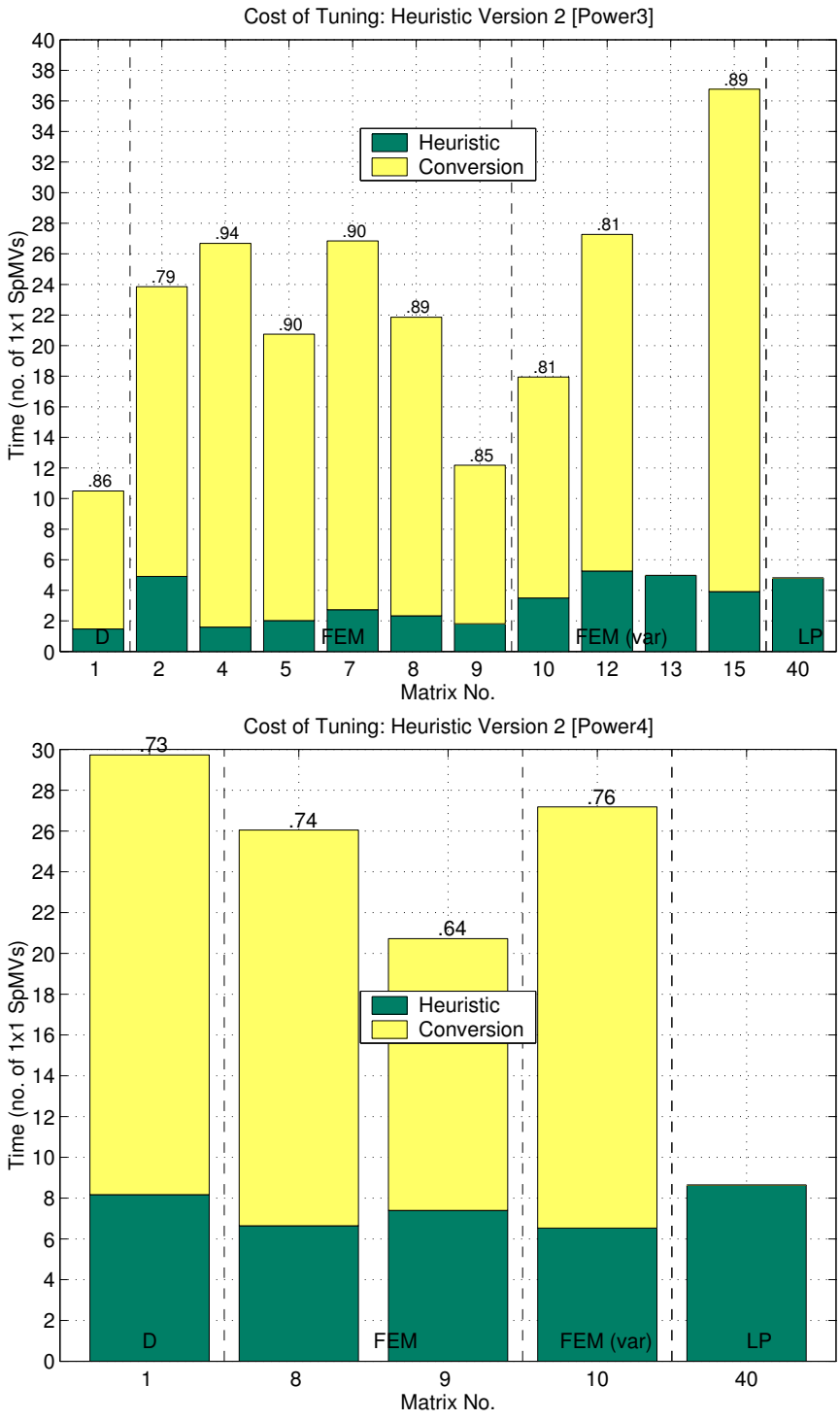

Figure 3.18: **Cost of register block size tuning: Power3 and Power4.** (*Top*) Power3 (*Bottom*) Power4

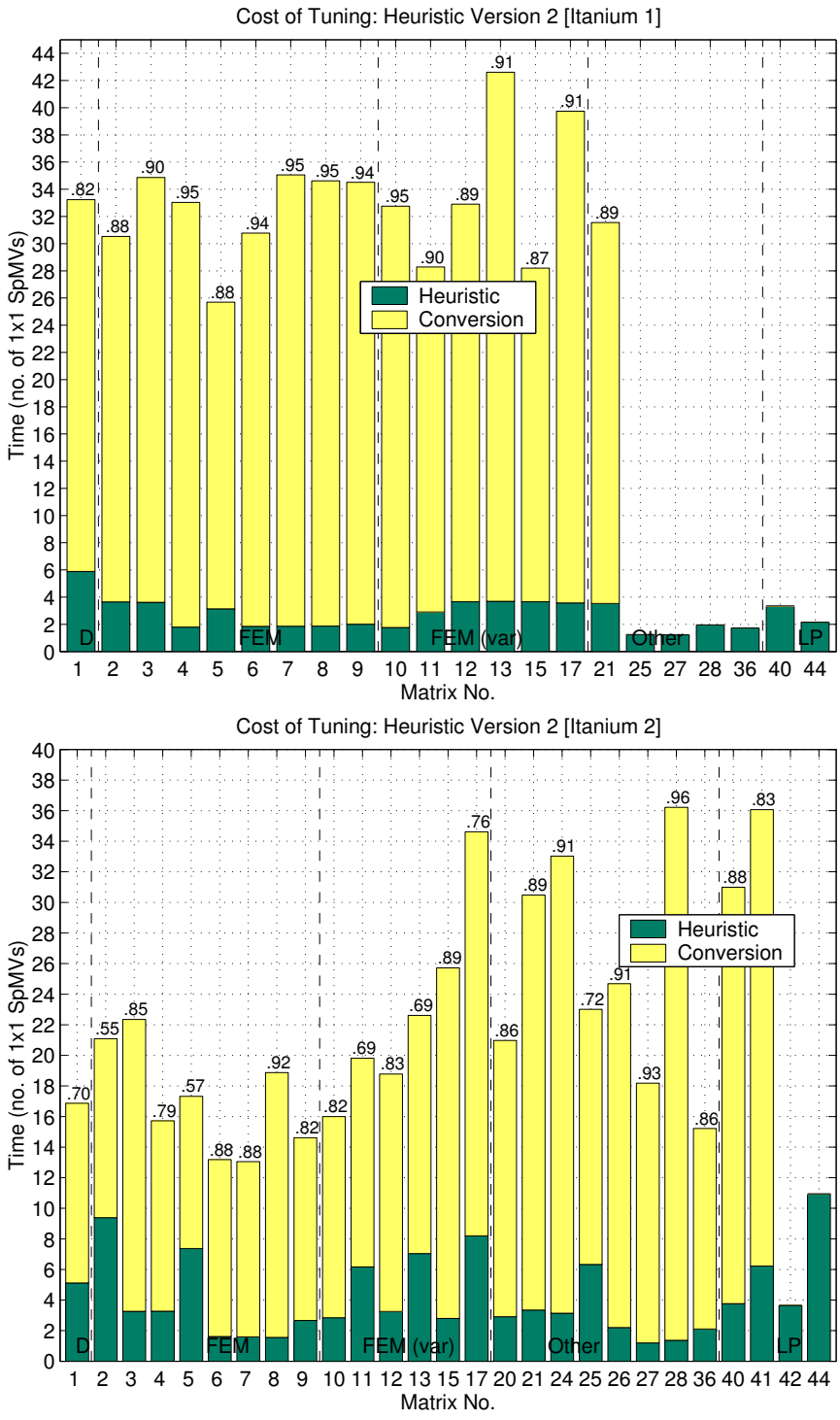

Figure 3.19: **Cost of register block size tuning: Itanium 1 and Itanium 2**. (*Top*) Itanium 1 (*Bottom*) Itanium 2

reduced on Itanium 2. This indicates that tuning will be most beneficial in applications where many SpMVs are performed for the same matrix. However, this cost depends on the platform. The total cost is relatively the most expensive on the Power and Itanium architectures.

- The median cost of evaluating the heuristic is always less than 7.5 unblocked SpMVs, while the median cost of conversion is between 5 and 20 SpMVs. The median cost of conversion is at least twice the median cost of the heuristic—thus, data structure conversion is a good target for future work on reducing the overall tuning cost.

We do not fully understand by how much we can reduce the data structure conversion time. We use the conversion routine implemented in SPARSITY, and did not attempt to optimize or profile this code to identify bottlenecks. Roughly speaking, this routine makes two passes through the matrix, once to determine the final output size, and once to copy the values. The cost of allocating the new data structure is included in the time we measure. Additional profiling and a closer inspection of this routine is needed.

There can also be some benefit to reducing the cost of evaluating the heuristic as well, particularly when it turns out blocking is not needed. For instance, the heuristic costs 11 unblocked SpMVs for Matrix 44 on Itanium 2. As discussed in Section 3.2.2, there are a number of ways in which this cost can be reduced. One of the simplest ways is to reduce $r_{\max}$ and $c_{\max}$, since the cost of fill estimation is proportional to $r_{\max} \cdot c_{\max}$. The largest block size selected in the matrix test suite is 8×8, which would reduce the cost of estimation by more than half (from 12×12=144 to 64). Another is to abandon the fixed sampling fraction σ in favor of an adaptive technique that detects when the performance estimate has stabilized. These ideas are opportunities for future work.

3.4 Summary

This chapter demonstrates the steps of tuning and quantifies their cost, in the particular case of register block size selection. Our contribution is an improvement to the hybrid off-line/run-time heuristic originally proposed by SPARSITY system [164, 167], particularly on recent platforms based on the Itanium 1 and Itanium 2 processors. Our "Version 2 heuristic" replaces the heuristic proposed in the original SPARSITY system [164, 167]. The main idea is to estimate the fill efficiently for a larger set of possible block sizes than was

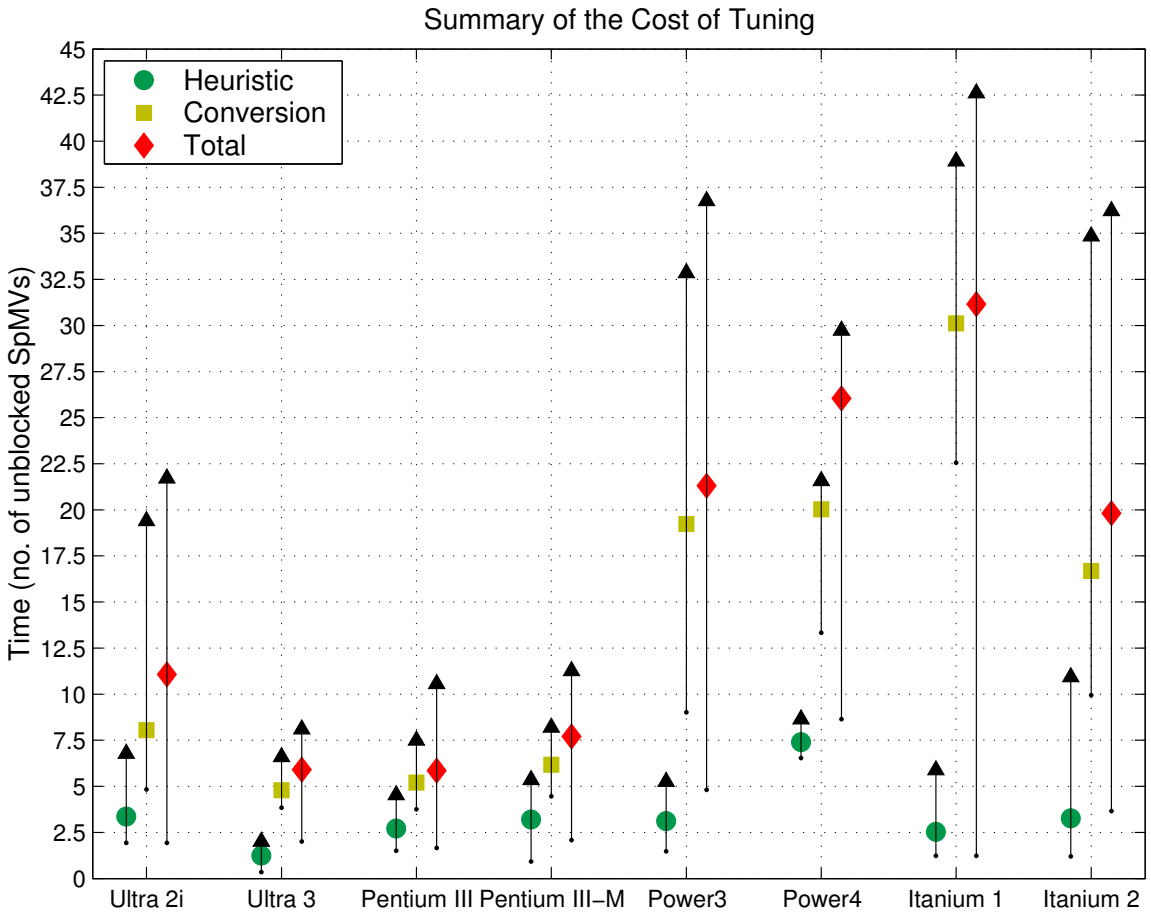

Figure 3.20: **Summary of the costs of tuning across platforms**. For each platform, we show the minimum, median, and maximum costs for (1) evaluating the heuristic, plus executing one unblocked SpMV, (2) converting the matrix to blocked format plus one execution of the blocked SpMV, provided a block size other than 1×1 is selected and (3) the total cost of both steps. Cost is measured in units of unblocked SpMVs.

previously done in the Version 1 heuristic, thereby enabling use of more of the information contained in the irregular performance profiles shown in Section 3.1.

The register profiles shown in Figures 3.3–3.6 raise questions about whether performance would be so irregular if, for each $r \times c$ block size, we could select the best instruction schedule. After all, we currently rely on the compiler to schedule the unrolled loop body shown in Figure 3.1 (*bottom*). This question remains open. One way to resolve it is to apply an automated PHiPAC/ATLAS-style search over possible instruction schedules [46, 325]. However, as Chapter 4 suggests for SpMV, any absolute improvements in performance are

likely to be limited. The main benefit to trying to find better schedules would be to simplify block size selection in accordance with a user's expectations, thereby greatly reducing or even eliminating the cost of having to execute a heuristic.

Still, our heuristic is engineered to keep the cost small relative to the cost of just converting a user's matrix from CSR to BCSR format, which is a lower bound on the cost of any tuning process where blocking is indeed more efficient than not blocking. For each platform and matrix category (FEM 2–9, FEM 10–17, and Other/LP 18–44), we observe that the median costs of heuristic evaluation range between 1–7.5 SpMVs, and between 5–31 SpMVs for conversion. The maximum total costs (heuristic evaluation plus conversion) range from 8 SpMVs (on Ultra 3) to 43 SpMVs (on Itanium 1).

When blocking is not profitable, the cost of fill estimation can seem high. Though we selected a fixed sampling fraction $\sigma = .01$ for the purposes of evaluating the heuristic, the ideal value of σ varies directly by matrix, and indirectly by machine (through the register profile). Adaptive schemes, as discussed in Section 3.2.2 and again in Section 3.3, are an obvious opportunity for refinement.

Though we consider only register block size selection in this chapter, subsequent chapters revisit the off-line/run-time approach to tuning kernels like sparse triangular solve, and evaluation of $A^TA \cdot x$. We find that this tuning methodology is effective in these other contexts as well.

Chapter 4

Performance Bounds for Sparse Matrix-Vector Multiply

Contents

This chapter presents machine-specific upper and lower bounds on the performance (Mflop/s) of sparse matrix-vector multiply (SpMV). We derive these bounds in the specific case when *register blocking* optimizations, as implemented in the Sparsity system [167, 164], have been applied. As discussed in our review of register blocking (Section 3.1), register-level reuse is made possible by appropriate exploitation of the matrix non-zero structure. Our bounds in turn depend on the problem through this structure. They also depend on the

hardware through the effective costs of cache hits and misses, and on the cache line sizes (Section 4.1).

The primary goal in developing the upper bound in particular is to estimate the best possible performance, given a machine, matrix, and data structure but *independent* of any particular instruction mix or ordering. Upper bounds act as a target against which we can evaluate the quality of the generated and tuned code. A gap between the observed performance of SpMV and the upper bound indicates the potential pay-off from additional low-level tuning (*e.g.*, from better instruction selection and scheduling). An additional goal is to better understand how a variety of factors, like matrix non-zero structure and memory hierarchies, influence the performance achieved in practice.

We evaluate register blocked SpMV on the 8 hardware platforms, and 44 test matrices that span a variety of applications. Our central findings, presented in detail in Section 4.2, can be summarized as follows:

1. SPARSITY's generated and compiled code can frequently achieve 75% or more of the performance upper bound, particularly for the class of matrices with uniform natural block structure that arise in finite element method (FEM) applications. This observation limits the improvement we would expect from additional low-level tuning, and furthermore motivates the ideas for achieving higher performance which we pursue in subsequent chapters, including (1) consideration of higher-level matrix structures (*e.g.*, multiple register block sizes, exploiting symmetry, matrix reordering), and (2) optimizing kernels with more opportunity for data reuse (*e.g.*, sparse matrix-multiple vector multiply, multiplication of A^TA by a vector).

2. For matrices from FEM applications, typical speedups range between $1.4 - 4.1\times$ on nearly all platforms. This result confirms the importance of register blocking on modern cache-based superscalar architectures.

3. The fraction of machine peak achieved by register blocking correlates with a measure of *machine balance* that is based on our model's cache parameters. Balance measures the number of flops that can be executed in the average time to perform a load from main memory. A relationship between balance and achieved performance hints at a possible way to evaluate how efficiently we might expect SpMV to run on a given architecture.

4. A simple consequence of our model is that strictly increasing line sizes should be used in multi-level memory hierarchies for applications dominated by unit stride streaming memory accesses (*e.g.*, Basic Linear Algebra Subroutines (BLAS) 1 and BLAS 2 routines). For SpMV in particular, we show how to compute approximate speedups when this architectural parameter varies. On a real system with equal L_1 and L_2 line sizes, we show it might be possible to speed up absolute SpMV performance by up to a factor of 1.6× by doubling the L_2 line size.

The methodology for deriving bounds presented in this chapter also serves as a framework for the development of similar bounds in later chapters for other kernels and data structures. Comparisons against the bounds allow us to identify new opportunities to apply previously developed automatic low-level tuning technology (in systems such as ATLAS [324] or PHiPAC [46]) to sparse kernels.[1]

This chapter greatly expands on work which appeared in a recent paper [316].

4.1 A Performance Bounds Model for Register Blocking

Observed performance depends strongly on the particular low-level instruction selection and scheduling decisions. In this section, we derive instruction mix- and schedule-independent bounds on the best possible performance of sparse matrix-vector multiply (SpMV) assuming block compressed sparse row (BCSR) format storage. Our primary goal is to quantify how closely the generated code approaches the best possible performance. Our bounds depend on both the matrix non-zero structure (through the fill ratio at a given value of $r \times c$) and the machine (through the latency of access at each level of the memory hierarchy).

The key high-level assumptions underlying our bounds are summarized in Section 4.1.1. Briefly, our bounds model consists of two components, each of which makes modeling assumptions. The first component is a model of execution time for kernels with streaming memory access behavior. This model considers only the costs of load and store operations. We argue below that such a model is appropriate for operations like SpMV, which largely enumerates non-zero values of the matrix while computing relatively few flops per memory reference. The second component is an analysis of the number of cache misses for a given matrix, matrix data structure, and kernel. We optimistically ignore conflict

[1] In particular, see Chapter 7 when we discuss the performance of the kernel, $y \leftarrow y + A^T Ax$.

misses to obtain performance upper bounds. We derive these two components in detail in Section 4.1.2 and Section 4.1.3, respectively. Changes to the bounds for other kernels and data structures are essentially isolated to the latter component that models hits and misses.

Although we are primarily interested in upper bounds on performance, for completeness we also briefly discuss lower bounds. Contrasting the two types of bounds helps to emphasize the assumptions of the model.

4.1.1 Modeling framework and summary of key assumptions

We derive upper and lower bounds on the performance P, measured as a rate of execution in units of Mflop/s, for a sparse kernel given a matrix and machine. Specifically, we define P as

$$P = \frac{F \times 10^{-6}}{T} \tag{4.1}$$

where F is the minimum number of flops required to execute a given kernel for a given sparse matrix, and T is the execution time in seconds of a given implementation. If A has k non-zeros,[2] then for SpMV, $F = 2k$. The execution time T will depend on the particular data structure and implementation. Note that F depends only on the matrix and not on the implementation, so comparing P between two different implementations is equivalent to comparing (inverse) execution time. Thus, we can fairly compare the value of P for two register blocked implementations with different block sizes and fill ratios as if we were comparing (inverse) execution time. We never include operations with filled-in zeros in F.

To obtain an upper bound on P, we need a lower bound on T since F is fixed. We make two guiding assumptions:

1. In our model of T, we *only consider the cost of memory references*, ignoring the cost of all other operations (*e.g.*, flops, ALU operations, branches). We present the details of this model in Section 4.1.2.

2. Our model of T is in turn proportional to the weighted sum of cache misses at each level of the memory hierarchy. Thus, we can further bound T from below by computing lower bounds on cache misses. To obtain such bounds, we *only count compulsory misses*, *i.e.*, we ignore conflict misses. We present cache miss bounds in Section 4.1.3.

[2] Throughout this dissertation, we distinguish only between "zero" and "non-zero" values. That is, if the user provides our system with some representation of A which contains zero values, we treat these structurally and logically as "non-zero" values, including them in k. However, if any of our methods add explicit structural zero entries, then these are *not* counted in k; this point is discussed further in this section.

In subsequent chapters, we apply these same assumptions to derive bounds for other kernels and data structures.

4.1.2 A latency-based model of execution time

Our model of execution time T assumes that the cost of the operation is dominated by the cost of basic memory operations (loads and stores). The primary motivation for such a model is the observation that SpMV is essentially a streaming application. Recall from the discussion of Chapter 2 that, abstractly, SpMV can be described as the operation

$$\forall a_{i,j} \neq 0 : y_i \leftarrow y_i + a_{i,j} \cdot x_j, \tag{4.2}$$

and, furthermore, that the key to an efficient implementation of SpMV is efficient enumeration of the non-zero $a_{i,j}$ values. There is no reuse of $a_{i,j}$, assuming all the matrix values are distinct. Furthermore, there are only 2 flops executed per non-zero element.[3] In the best case, x and y are cached and the time to perform SpMV is at least the time to read all the matrix elements, *i.e.*, the time to stream through the matrix A. For most applications, A is large and SpMV is thus limited by the time to stream the matrix from main memory.

In our approach to modeling execution time we assume the following:

1. We ignore the cost of non-memory operations, including flops, branches, and integer (ALU) operations. As discussed in Section 3.1, there are a constant number of integer operations per $2rc$ flops, so the decision to neglect these operations is likely to be valid as rc increases.

2. To each load or store operation, we assign a cost (access latency) based on which level of the memory hierarchy holds the data operand.

3. We ignore the cost of TLB misses. Since we are modeling kernels with streaming memory access patterns, we expect predominantly unit stride access. Page faults will only occur once per l_{TLB} words, where l_{TLB} is the TLB page size. Typically, $l_{\text{TLB}} \sim O(1000)$ double-precision words (see the tabulated machine parameters in Appendix B), compared to the cost of a TLB miss which is typically $O(10)$ cycles. Thus, the amortized cost per word is $O(\frac{1}{100})$ cycles.

[3] In subsequent chapters, we consider optimizations and kernels that can reuse the elements of A, including (1) the case when A is symmetric, *i.e.*, $a_{i,j} = a_{j,i}$, (2) multiplication by multiple x vectors, and (3) sparse kernels with explicit reuse of A such as $y \leftarrow y + A^T Ax$.

Consider a machine with κ levels of cache, where the cost of executing a memory operation (either a load or a store) on data residing in the L_i cache is α_i seconds. Let α_{mem} be the cost in seconds of executing a memory operation on data residing in main memory. Suppose we know for a given kernel, matrix, storage format, and machine, that the number of hits (accesses) to the L_i cache is H_i, and the number of "memory hits" (memory accesses) is H_{mem}. We then posit the following model for T:

$$T = \sum_{i=1}^{\kappa} \alpha_i H_i + \alpha_{\text{mem}} H_{\text{mem}} \tag{4.3}$$

We can equivalently express T in terms of loads, stores, and cache misses. Let Loads be the number of load operations, Stores be the number of store operations, and M_i be the number of L_i misses. Since $H_1 = \text{Loads} + \text{Stores} - M_1$, $H_i = M_{i-1} - M_i$ for $2 \leq i < \kappa$, and $H_{\text{mem}} = M_\kappa$,

$$T = \alpha_1(\text{Loads} + \text{Stores}) + \sum_{i=1}^{\kappa-1}(\alpha_{i+1} - \alpha_i)M_i + (\alpha_{\text{mem}} - \alpha_\kappa)M_\kappa \tag{4.4}$$

For a sensible memory hierarchy, the latencies will satisfy the condition $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_\kappa \leq \alpha_{\text{mem}}$. Thus, if we can count Loads and Stores exactly, then Equation (4.4) shows we can further bound T from below by computing lower bounds on M_i.

We have assigned the same cost to load and store operations. This is a reasonable approximation since we do relatively few stores compared to loads, as we later show.

4.1.3 Lower (and upper) bounds on cache misses

We further bound the expression for execution time given by Equation (4.4) from below by computing lower bounds on cache misses M_i at each L_i level of the memory hierarchy. This section gives expressions for Loads, Stores, and the lower (and upper) bounds on M_i for a given matrix and the sparse kernel SpMV, where the matrix is stored in BCSR format. The following two assumptions underlie the lower bounds:

1. We ignore conflict misses. For SpMV, we further ignore capacity misses. (We do consider capacity issues for some of the other kernels examined in this dissertation.) Thus, we only consider compulsory misses for SpMV. This assumption is equivalent to assuming infinite cache capacity and fully associatve caches. We do, however, account for cache line sizes.

2. We assume maximum advantage from spatial locality in accesses to the source vector. For blocked matrices, *e.g.*, matrices arising in finite element method (FEM) problems, this assumption is reasonable since we will load blocks of the source and destination vector (see Figure 3.1, lines S2 and S4a). For randomly structured matrices, this assumption will not generally be true—for each source vector access, we will load an entire cache line, of which we will only use 1 element. However, this assumption is subsumed by the first assumption. That is, under the general condition that the matrix has at least one non-zero per column, we will touch every element of the source vector at least once.

Suppose that the L_i cache has capacity C_i and line size l_i, both in units of floating point words. By "word," this dissertation assumes double-precision floating point values of which the size of a floating point word is 8 bytes (64-bits). An 8 KB L_1 cache with 32 byte lines has $C_1 = 1024$ and $l_i = 4$. (Though we ignore cache capacity for SpMV, we do consider capacity issues for some of the other kernels appearing in subsequent chapters.) To describe the matrix data structure, we assume the notation of Section 3.1.1: A is an $m \times n$ sparse matrix with k non-zeros, K_{rc} is the number of $r \times c$ blocks required to store the matrix in $r \times c$ BCSR format, and the fill ratio is $f_{rc} = \frac{K_{rc} \cdot rc}{k}$.

We count the number of loads and stores as follows. Every matrix entry is loaded exactly once. Thus, lines S3b–S4d of Figure 3.1, which load the rc elements of a block, will each execute K_{rc} times. As suggested by Figure 3.1, line S2, we assume that all r entries of the destination vector can be kept in registers for the duration of a block row multiply. Thus, we only need to load each element of the destination vector once, and store each element once. Similarly, we assume that the c source vector elements can be kept in registers during the multiplication of each block (Figure 3.1, line S4a), thus requiring a total of $K_{rc}c = \frac{kf_{rc}}{r}$ loads of the source vector. In terms of the number of non-zeros and the fill ratio, the total number of loads of floating point and integer data is

$$\begin{aligned} \text{Loads}(r,c) &= \underbrace{kf_{rc} + \frac{kf_{rc}}{rc} + \left\lceil \frac{m}{r} \right\rceil + 1}_{\text{matrix}} + \underbrace{\frac{kf_{rc}}{r}}_{\text{source vec}} + \underbrace{m}_{\text{dest vec}} \\ &= kf_{rc}\left(1 + \frac{1}{rc} + \frac{1}{r}\right) + m + \left\lceil \frac{m}{r} \right\rceil + 1 \end{aligned} \tag{4.5}$$

and the total number of stores is Stores $= m$, which is independent of r and c. The source vector load term depends only on r, introducing a slight asymmetry in the number of loads

as a function of block size. If the time to execute all loads were equal, then we might expect performance to grow more quickly with increasing r than with increasing c.

Next, consider the number of misses at the L_i cache. One compulsory L_i read miss per cache line is incurred for every matrix element (value and index) and destination vector element. The source vector miss count is more complicated to predict. If the source vector size is less than the size of the L_i cache, then in the best case we would incur only 1 compulsory miss per cache line for each of the n source vector elements. Thus, a lower bound $M_{\text{lower}}^{(i)}$ on L_i misses is

$$M_{\text{lower}}^{(i)}(r,c) = \frac{1}{l_i}\left[kf_{rc}\left(1+\frac{1}{\gamma rc}\right)+\frac{1}{\gamma}\left(\left\lceil\frac{m}{r}\right\rceil+1\right)+m\right]+\frac{n}{l_i}. \tag{4.6}$$

The factor of $\frac{1}{l_i}$ accounts for the L_i line size by counting only one miss per line.

In contrast to this lower bound, consider the following crude upper bound on cache misses. In the worst case, we will miss on every access to a source vector element due to capacity and conflict misses; thus, an upper bound on misses is

$$M_{\text{upper}}^{(i)}(r,c) = \frac{1}{l_i}\left[kf_{rc}\left(1+\frac{1}{\gamma rc}\right)+\frac{1}{\gamma}\left(\left\lceil\frac{m}{r}\right\rceil+1\right)+m\right]+\frac{kf_{rc}}{r}. \tag{4.7}$$

Only the last terms differ between Equation (4.7) and Equation (4.6). Any refinements to these bounds would essentially alter this term, by considering, for example, the degree of spatial locality inherent in a particular matrix non-zero pattern.

In the case of the Itanium 1 and Itanium 2 platforms, the L_1 cache is used only for integer data [168]. Thus, we would drop all terms associated with floating point data in $M_{\text{lower}}^{(1)}(r,c)$ to reflect this architectural property.

4.2 Experimental Evaluation of the Bounds Model

This section compares the performance of register blocking in practice to the predictions of the upper (and lower) bounds model described in Section 4.1. We measure the running times and, where available, memory traffic (cache misses) using the PAPI hardware counter library [60] on the eight platforms and suite of 44 test matrices described in Appendix B. The test matrices are organized into the following 5 categories:

- Matrix 1 (D): A dense matrix in sparse format (as in the register profiles of Figures 3.3–3.6), shown for reference.

- Matrices 2–9 (FEM): Matrices from FEM applications, characterized by a predominantly uniform block structure (a single block size aligned uniformly).
- Matrices 10–17 (FEM var.): Matrices from FEM applications, characterized by more complex block structure, namely, by multiple block sizes, or a single block size with blocks not aligned on a fixed grid.
- Matrices 18–39 (Other): Matrices from various non-FEM applications, including chemical process simulations, oil reservoir modeling, and economic modeling applications, among others.
- Matrices 40–44 (LP): Matrices from linear programming problems.

In evaluating a given platform, we consider only the subset of these matrices whose size exceeds the capacity L_κ cache. (See Appendix B for a detailed description of the experimental methodology.) The dense matrix is shown mostly for reference. Of the remaining 4 categories, we find that three distinct groups emerge when examining absolute performance: FEM Matrices 2–9, FEM Matrices 10–17, and Matrices 18–44.

Before looking at performance, we first demonstrate the two components of our model, beginning with the latency-based model of execution time (Section 4.2.1), followed by an experimental validation of our load and cache miss counts (Section 4.2.2).

We then put these two components together and compare the performance predicted by the bounds to actual performance in Section 4.2.3, which contains the four main results of this chapter:

1. We find that the register blocked implementations can frequently achieve 75% or more of the performance upper bound, placing a fundamental limit on improvements from additional low-level tuning. (The Ultra 3 is the main exception, achieving less than 35% of the bound.)
2. For matrices from FEM applications, we find typical speedups in the range of $1.4-4.1\times$ on most of the platforms. (The Power3 is the notable exception, achieving speedups of $1.3\times$ or less in the best case.)
3. We show that the fraction of machine peak achieved for SpMV is correlated to a machine-specific measure of balance derived from our latency model, providing a

coarse way to characterize how well a given platform can perform SpMV relative to machine peak.

4. We demonstrate the importance of strictly increasing cache line sizes for multi-level memory hierarchies, a simple and direct consequence of our performance model. For instance, on the Pentium III, where the L_1 and L_2 line sizes are equal, doubling the L_2 line size could yield a speedup of 1.6× in the best case.

4.2.1 Determining the model latencies

We use microbenchmarks to measure the latencies α_i and α_{mem} that appear in our execution time model, Equation (4.4). We summarize the values of these latencies, along with other relevant cache parameters, in Table 4.1. A range of values indicates "best" and "worst" case latencies, as measured by microbenchmarks. The best case latency corresponds to the minimum latency seen by an application under the condition of unit stride streaming memory access. The worst case represents the cost of non-unit stride or, in the case of the Itanium 2 and Power4, the cost of random memory access. (We use the worst case latencies in computing performance *lower* bounds, but will not otherwise be interested in these values.) This section explains what microbenchmarks we run and how we determine the latency values. We first discuss why the latency model and measurements are important.

The latencies not only allow us to evaluate our performance bounds model, but also serve as a machine-specific indicator of sustainable bandwidth, as suggested by the rightmost column of Table 4.1, labeled "Sustainable fraction of peak memory bandwidth." To see what this column represents, consider the average time per load in our model when streaming through a single array with unit stride access. Suppose we execute l_κ such unit stride loads, where all the data initially resides in main memory. We incur $M_\kappa = 1$ cache miss at the L_κ cache, and $M_i = l_\kappa / l_i$ misses at each of the L_i caches, for $i < \kappa$. Upon substitution into Equation (4.4), the time to execute these loads in our model is:

$$T = \alpha_1 l_\kappa + \sum_{i=1}^{\kappa-1} (\alpha_{i+1} - \alpha_i) \frac{l_\kappa}{l_i} + (\alpha_{\text{mem}} - \alpha_\kappa)$$

and the average time per load can be written as follows:

$$\frac{T}{l_\kappa} = \alpha_1 \left(1 - \frac{1}{l_1}\right) + \sum_{i=2}^{\kappa} \alpha_i \left(\frac{1}{l_{i-1}} - \frac{1}{l_i}\right) + \frac{\alpha_{\text{mem}}}{l_\kappa} \tag{4.8}$$

Platform **Processor** Clock rate Peak mem. bw	*Cache Parameters* Capacity Line size Min–Max latency L_1	L_2	L_3	Mem	Sustainable Fraction of of Peak Bandwidth [STREAM]
Ultra 2i 333 MHz 664 MB/s	16 KB 16 B 2 cy	2 MB 64 B 6–7 cy	—	256 MB — 38–66 cy	0.50 [0.32]
Ultra 3 900 MHz 5 GB/s	64 KB 32 B 1–2 cy	8 MB 64 B 5–11 cy	—	4 GB — 28–200 cy	0.31 [0.10]
Pentium III 500 MHz 680 MB/s	16 KB 32 B 2 cy	512 KB 32 B 18 cy	—	128 MB — 25–60 cy	0.76 [0.51]
Pentium III-M 800 MHz 915 MB/s	16 KB 32 B 1–2 cy	256 KB 32 B 5–18 cy	—	256 MB — 40–60 cy	0.65 [0.62]
Power3 375 MHz 2 GB/s	64 KB 128 B 0.5–2 cy	8 MB 128 B 9 cy	—	1 GB — 35–139 cy	0.71 [0.47]
Power4 1 GHz 11 GB/s	32 KB 128 B 0.7–1.4 cy	1.5 MB 128 B 4.4–91 cy	16 MB 512 B 21.5–1243 cy	4 GB — 60–10000 cy	0.36 [0.21]
Itanium 1 800 MHz 2 GB/s	16 KB 32 B 0.5–2 cy	96 KB 64 B 0.75–9 cy	2 MB 64 B 21–24 cy	1 GB — 36–85 cy	0.61 [0.53]
Itanium 2 900 MHz 6 GB/s	32 KB 64 B 0.34–1 cy	256 KB 128 B 0.5–4 cy	1.5 MB 128 B 3–20 cy	2 GB — 11–60 cy	0.97 [0.63]

Table 4.1: **Machine-specific parameters for performance model evaluation**. We show the machine-specific parameters used to evaluate the performance model presented in Section 4.1. Beneath each platform's processor name, we show the clock rate and theoretical peak main memory bandwidth, as reported by the platform vendor. For cache and memory latencies, a range indicates estimated best and worst cases, used for the performance upper and lower bounds, respectively. Latencies are determined using the Saavedra-Barrera benchmark [269], except on the Power4 and Itanium 2 platforms where we use the PMaC-MAPS benchmark [282]. For all machines, we take the number of integers per double to be $\gamma = 2$. The final column shows sustainable bandwidth β_s according to our model, as a fraction of peak bandwidth. Beneath this fraction, we show the results of the STREAM Triad benchmark [217], as a fraction of peak bandwidth, in square brackets.

From this average time, we can compute a *sustainable memory bandwidth* for streaming unit stride data access: at 8 B/word, the sustainable memory bandwidth is $\beta_s = \frac{l_K}{T} \times 8 \cdot 10^{-6}$ MB/s. We adopt the term "sustainable memory bandwidth" as used by McCalpin to refer to the achievable, as opposed to peak, bandwidth [217]. The last column of Table 4.1 shows this sustainable bandwidth as a fraction of the vendor's reported theoretical peak memory bandwidth (shown in column 1). We use the minimum latency where a range is specified in Table 4.1 to evaluate β_s. Beneath this normalized value of β_s, we show the bandwidth reported by the STREAM Triad benchmark for reference, in square brackets and also as a fraction of peak memory bandwidth. Comparing β_s to the STREAM Triad bandwidth, we see that β_s is a bound on the STREAM bandwidth.

Examining the last column of Table 4.1 shows that the sustainable bandwidth β_s is often a noticeably reduced fraction of the vendor's reported main memory bandwidth. This fraction varies between as little as 31% of peak bandwidth (Ultra 3), or as much 97% (Itanium 2), with a median value of 63%. It has been noted elsewhere that peak memory bandwidth is often an overly optimistic indicator of true data throughput from memory [217]. In the particular case of SpMV, this fact frequently leads to bounds on performance for operations like SpMV that greatly exceed what is realized in practice [140]. We claim that our latency model, with the latencies we have used, allows us to compute more realistic bounds on performance.

Below, we illustrate how we determine the access latencies using two examples. The first example uses the Saavedra-Barrera microbenchmark, which we use on all but the Power4 and Itanium 2 platforms. For these two machines, we use the Memory Access Pattern Signature (MAPS) microbenchmark, which has been hand-tuned for a variety of recent hardware platforms [282]. What the examples show is that what we measure and call an access "latency" is really a measure of inverse throughput for streaming workloads. These latencies abstract away all the mechanisms of a memory system design that hide true latency (*e.g.*, pipelining and buffering to support multiple outstanding misses [328]).

Example 1: Saavedra-Barrera microbenchmark

The Saavedra-Barrera microbenchmark measures the average time per load when repeatedly streaming through an array of length N (a power of 2) at stride s [269]. This benchmark varies both N and s, and the output can be used to determine the cache capacities, line

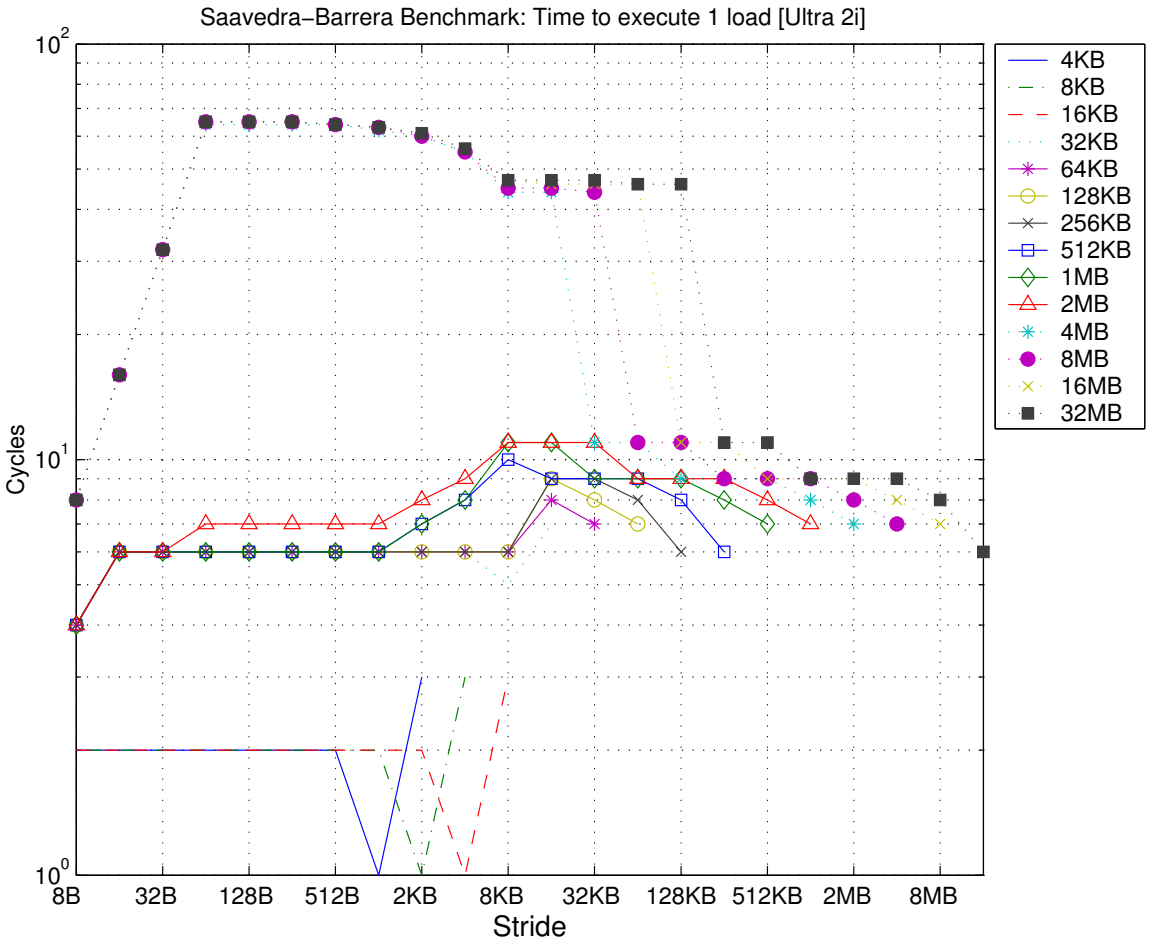

Figure 4.1: **Sample output from the Saavedra-Barrera microbenchmark on the Ultra 2i**. The Saavedra-Barrera microbenchmark measures the average time to execute a load when streaming through an array of length N at stride s [269]. The machine shown, based on the Sun 333 MHz Ultra 2i processor, has a 16 KB direct-mapped L1 cache, a two-way associative 2 MB L2 cache, a TLB of size 64 with 8 KB pages. Load time is shown in cycles along the y-axis. Each line corresponds to a given value of N, and the stride s is shown along the x-axis. Parameters such as cache capacities, line sizes, associativities, page sizes, and the number of TLB entries can generally be deduced from the output of this benchmark.

sizes, associativities, and effective latencies (α_i) at all levels of the memory hierarchy. The latencies are particularly important because they allow us to evaluate the execution time model, Equation (4.4), completely when we know the number of cache misses (*e.g.*, from hardware counters) exactly.

Figure 4.1 shows an example of the output of the Saavedra-Barrera microbench-

mark. The average load time, in cycles, is shown on the y-axis. Each line represents a fixed value of N, and stride s appears on the x-axis. The word size is 8 B, so "unit" stride corresponds to $s = 8$ B. Observe that for all array sizes up to $N = 16$ KB, the time to access the data is 2 cycles. (The blips at the last two data points for each of these lines are due to issues with timing resolution.) This confirms that the L_1 cache size is 16 KB, and further indicates that the effective access latency when data resides in the L_1 cache is $\alpha_1 = 2$ cycles, independent of the stride.

We compute α_2 for this example as follows. First, note that a second plateau at 6–7 cycles occurs for sizes up to $N = 2$ MB, the size of the L_2 cache. This plateau starts at $s = 16$ B, confirming that the L_1 line size is 16 B. The minimum access latency would thus appear to be 6 cycles, but we need to check this since there may be cache and memory hardware mechanisms (such as pipelining and buffering to support multiple outstanding misses) that allow faster commit rates at unit strides. For arrays between 32 KB and 2 MB in size, the average unit stride load time is 4 cycles. Since these arrays fit within the L2 cache, which has 64 B lines ($l_2 = 8$ words), for every l_2 loads we will incur no L_2 misses ($M_2 = 0$), and $M_1 = 1$ misses in L_1. After substituting these values into our execution time model, Equation (4.4), we find that the average load time for in-L_2 data is

$$\begin{aligned} \frac{T_{\text{in-}L_2}}{l_2} = 4 \text{ cy} &= \alpha_1 \left(1 - \frac{1}{l_1}\right) + \alpha_2 \frac{1}{l_1} \\ &= (2 \text{ cy}) \left(1 - \frac{1}{2 \text{ words}}\right) + \alpha_2 \left(\frac{1}{2 \text{ words}}\right) \end{aligned}$$

Solving this equation yields $\alpha_2 = 6$ cycles, which happens to confirm the minimum value of the L_2 plateau. Both of these empirically determined values α_1 and α_2 match what is described in the Ultra 2i processor manual [291].

Finally, the memory latency α_{mem} can be determined similarly. We note a third plateau at 66 cycles, beginning at $s = 64$ B (the L_2 line size). Again, we need to check the unit stride case, which has an average latency of 8 cycles by solving Equation (4.8) for α_{mem}:

$$\frac{T_{\text{in-mem}}}{l_2} = 8 \text{ cy} = (2 \text{ cy}) \left(1 - \frac{1}{2}\right) + (6 \text{ cy}) \left(\frac{1}{2} - \frac{1}{8}\right) + \frac{\alpha_{\text{mem}}}{8}$$

which yields a minimum effective memory load time of $\alpha_{\text{mem}} = 38$ cycles. This value is smaller than the 66 cycle plateau, indicating the presence of hardware mechanisms that allow for more efficient transfer of data in the unit stride case.

Our main use of the Saavedra-Barrera benchmark is to compute access latencies in the manner shown above. However, Figure 4.1 is clearly rich with information; we refer the interested reader to the original work by Saavedra-Barrera for more details on decoding the output of the benchmark [269].

Example 2: MAPS microbenchmark

On the Power4 and Itanium 2 platforms, the Saavedra-Barrera benchmark could not be run reliably due to artifacts from timing and loop overhead. Instead, we used the MAPS microbenchmark [282], which is sufficient to determine memory latencies though it does not provide the same level of information about cache parameters as the Saavedra-Barrera benchmark.

MAPS measures, for arrays of various lengths, the average time per load for (1) a unit stride access pattern, and (2) a random access pattern. We show an example of the output of the MAPS microbenchmark in Figure 4.2 for the Power4 platform.

For arrays that fit within the 32 KB L_1 cache, the average load time in the unit stride case is flat at 0.7 cycles; in the random data case, the minimum load time is approximately 1.4 cycles. Thus, we show α_1 =0.7–1.4 cycles in Table 4.1. (A fractional cycle time is possible in this case because the Power4 can commit up to 2 loads per cycle [34].) The remaining minimum latencies in the unit stride case can be computed by using the methodology of Example 1 above, where we use the in-cache plateaus for the average execution time. For α_2, the average load time for data residing within the 1.5 MB L_2 cache leads to

$$\frac{T_{\text{in-}L_2}}{l_2} = .93 \text{ cy} \quad = \quad (0.7) \cdot \left(1 - \frac{1}{16 \text{ words}}\right) + \alpha_2 \left(\frac{1}{16}\right)$$

or $\alpha_2 \approx 4.4$ in the unit stride case. To compute the maximum value of α_2, we use $\alpha_1 = 1.4$ cycles, and take the average time to be $T_{\text{in-}L_2}/l_2 = 7$ cycles, the value on the random access curve at .75 MB, or half the L_2 size. This leads to an upper value for α_2 of 91 cycles. For the L_3 cache, we determine the best case value for α_3 to be 21.5 cycles, from:

$$\frac{T_{\text{in-}L_3}}{l_3} = 2 \text{ cy} \quad = \quad (.7)\left(1 - \frac{1}{16}\right) + (4.4)\left(\frac{1}{16} - \frac{1}{16}\right) + \alpha_3 \frac{1}{16}$$

The middle term in the preceeding equation is zero, since the line sizes l_1 and l_2 are the same. The maximum value, based on the average load time of 79 cycles at half the L_3 cache size (8 MB), is 1243 cycles. Finally, we calculate the minimum memory latency to be 60

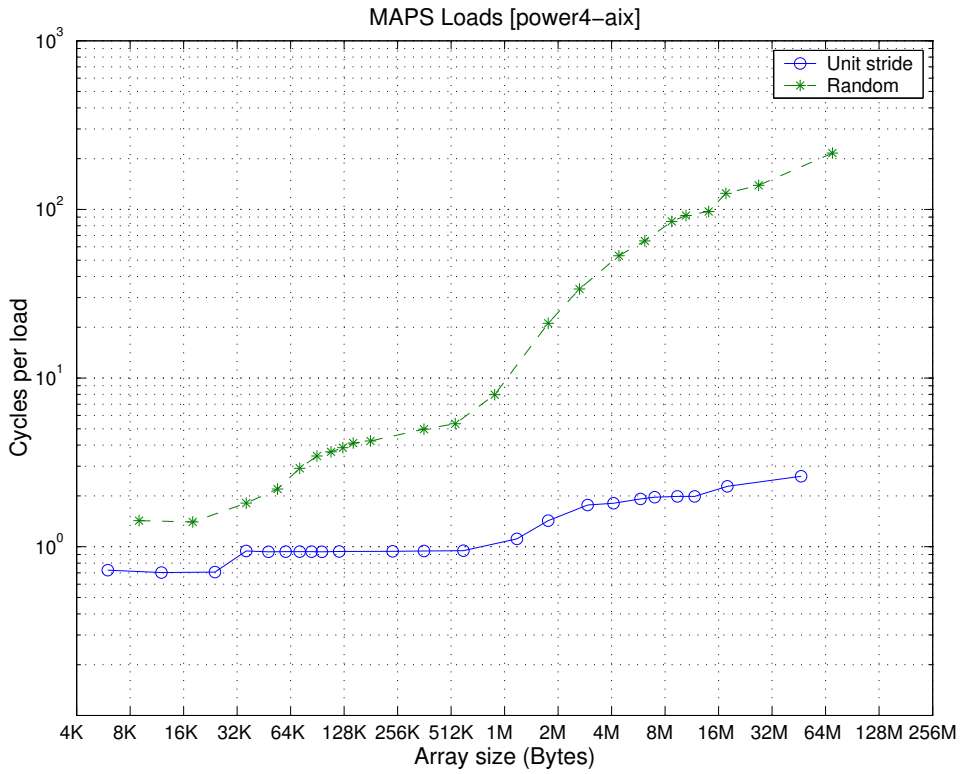

Figure 4.2: **Sample output from the MAPS microbenchmark on the Power4**. The MAPS microbenchmark measures the average time per load in cycles (y-axis) for arrays of various lengths N (x-axis). MAPS is carefully hand-tuned for each architecture, and tests two basic access patterns: a unit stride pattern and a random pattern.

cycles, using the maximum average time in the unit stride case of 2.6 cycles, as shown in Figure 4.2:

$$\frac{T_{\text{in-mem}}}{l_3} = 2.6 \text{ cy} \quad = \quad (.7)\left(1 - \frac{1}{16}\right) + (4.4)\,(0) + (21.5)\left(\frac{1}{16} - \frac{1}{64}\right) + \alpha_{\text{mem}} \cdot \frac{1}{64}$$

Using the maximum average load time of 216 cycles on the random curve, we compute a very pessimistic upper bound on α_{mem} to be approximately 10,000 cycles. Although these latencies seem extremely large, we note that the L_3 line size is, at 512 B, is a factor of 4 longer than the largest line size on any of the other platforms. Thus, truly random access is likely to waste a considerable amount of bandwidth. In addition, the Saavedra-Barrera benchmark measures strided accesses, which may still be detected by hardware prefetching mechanisms while random accesses are not.

4.2.2 Cache miss model validation

This section evaluates the accuracy of the second component of our performance bounds model which counts memory traffic, including cache misses. Specifically, we compare our analytic load count, Equation (4.5), and our cache miss lower and upper bounds, Equations (4.6)–(4.7), to experimentally observed counts of these quantities for register blocked SpMV on the 44 test matrices. We show data for the subset of the 8 platforms listed in Table 4.1 on which the PAPI hardware counter library was available: Ultra 2i, Pentium III, Power3, Itanium 1, and Itanium 2.

The data show that we model loads and misses reasonably well, particularly on the class of FEM matrices. Thus, we assert that the counting aspect of the performance model is a good approximation to reality. We summarize the minimum, median, and maximum ratio of actual counts to those predicted by the model for both loads and cache misses in Table 4.2. This section explores the count data in more depth. In addition to evaluating the accuracy of our models, we make the following remarks:

1. We find that matrices from the different broad classes of applications appear to be fairly distinct from one another when examining the load counts. Put another way, load counts (when normalized by the number of non-zeros in the unblocked matrix) are a useful indirect indicator of matrix block structure and density.

2. Our lower bound model of cache misses is particularly accurate at the largest cache levels, and less accurate for small cache sizes, as indicated in Table 4.2. In principle, the cache miss bounds could be made more accurate at the smaller cache sizes by accounting for capacity and/or conflict misses, which we currently ignore. We argue that the relative magnitudes of the cache misses at all levels are such that less accurate modeling in the smaller caches is often acceptable, particularly since we are interested in reasonable time bounds and not exact predictions.

Validating load counts

Figures 4.3–4.6 compare the number of loads predicted by our model to the actual number of load instructions executed. (This data appears in tabulated form in Tables E.1–E.5.) We focus on loads because the number of stores are nearly always m, and they vary neither with block size (barring spilling) nor the number of non-zeros. Matrices are shown along

the x-axis. For each matrix, we ran register blocked SpMV for all $r \times c$ up to 12×12, selected the block size $r_{\text{opt}} \times c_{\text{opt}}$ with the smallest observed running time, and report the following:

- **Model load count**: the number of loads predicted by Equation (4.5) at the value $r_{\text{opt}} \times c_{\text{opt}}$, shown by a dashed blue line.
- **Actual load count**: the number of measured loads at the best block size, $r_{\text{opt}} \times c_{\text{opt}}$. By default, we show these counts using green solid circles, but consider two additional distinctions for subsequent analysis. If the block size is "small," which we define as the condition $r_{\text{opt}} \cdot c_{\text{opt}} \leq 2$, then we show the actual load count using a black hollow square. If the average number of non-zeros per row is less than or equal to 10 (*i.e.*, $\frac{k}{m} \leq 10$), then we show the load count by a red plus symbol. (These two conditions can be true simultaneously.)

Our primary goal in this section is to verify that the model's load counts approximate reality reasonably well.

In addition, the load data reveals how the block structure and density vary among the different matrix application classes. First, note that Figures 4.3–4.6 present the load count data normalized by the number of non-zeros k in the unblocked matrix. From Equation (4.5), we expect this quantity to be

$$\frac{\text{Loads}(r,c)}{k} = f_{rc}\left(1 + \frac{1}{rc} + \frac{1}{r}\right) + \frac{m}{k} + \frac{1}{k}\left(\left\lceil \frac{m}{r} \right\rceil + 1\right) \tag{4.9}$$

The limiting cases of Equation (4.9) reflect different kinds of matrix structure. A matrix with abundant uniform block structure will tend to have a "large" block size ($r, c \gg 1$), f_{rc} near 1, and many non-zeros per row ($k \gg m$, *i.e.*, relatively dense structure). This case suggests a simple lower limit of 1 for Equation (4.9). Intuitively, loads of the matrix values dominate the overall load count for this kind of structure. In the absence of block structure but with k still much greater than m, Equation (4.9) is approximately 3. Roughly speaking, the load count is dominated by the number of matrix value, index, and source vector element loads. If the matrix is very sparse and has little block structure, then k/m will be shrink toward 1, meaning Equation (4.9) could be as high as 5 as the relative number of destination vector and row pointer loads increases. Equation (4.9) is an interesting quantity because it distinguishes matrix block and density structure.

The normalized load counts observed on the Ultra 2i (Figure 4.3) and Power3 (Figure 4.4) show both that our model can be very accurate, and that the structure of

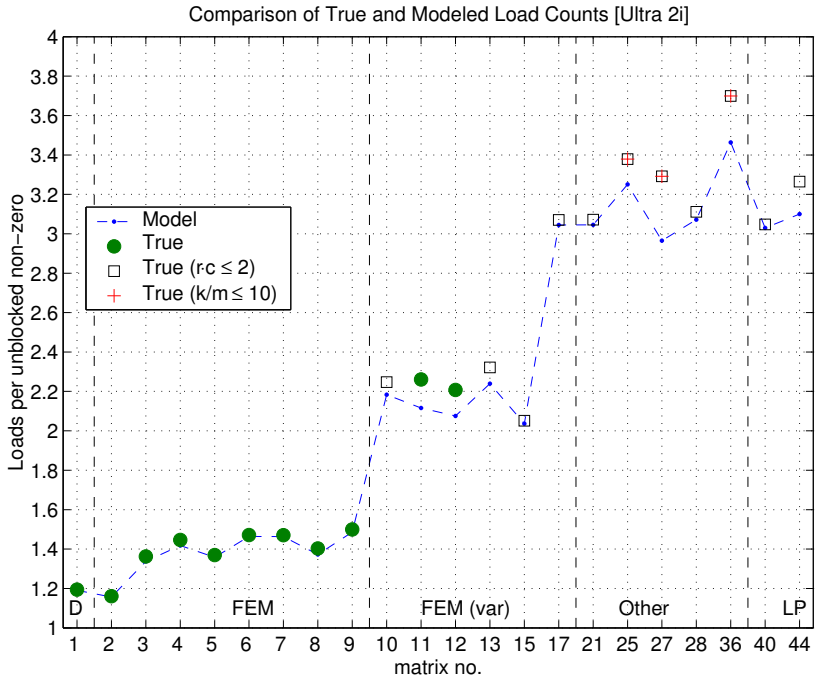

Figure 4.3: **Comparison of analytic and measured load counts: Ultra 2i**.

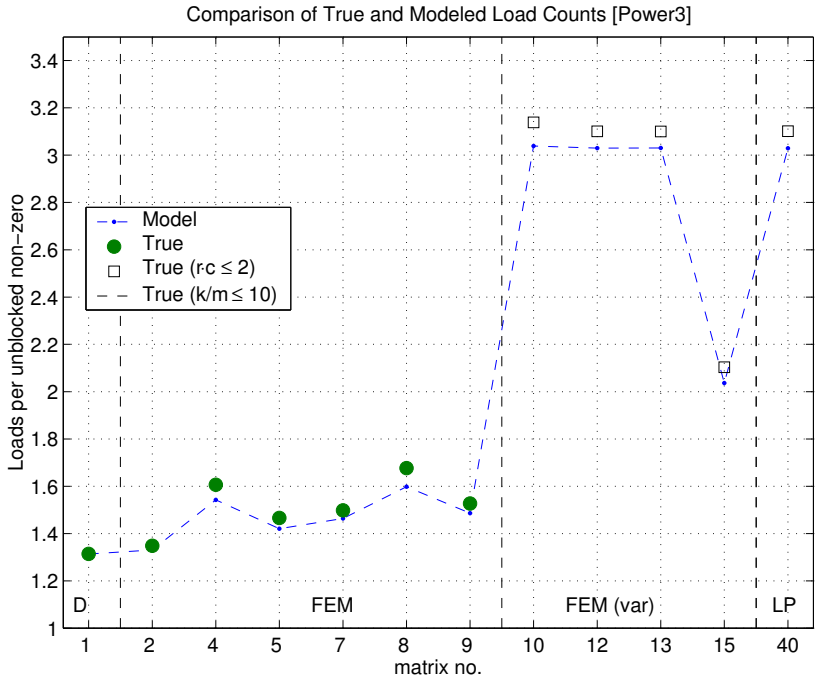

Figure 4.4: **Comparison of analytic and measured load counts: Power3**.

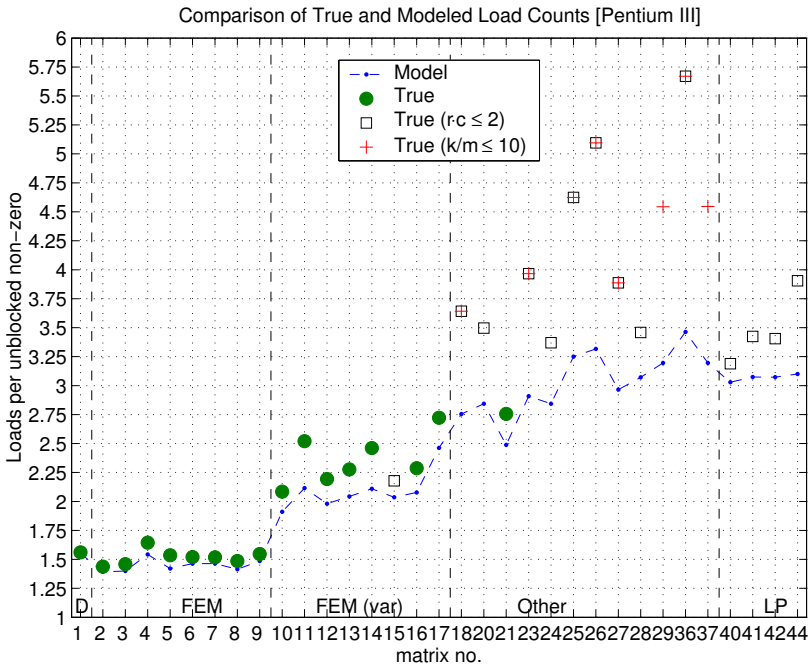

Figure 4.5: **Comparison of analytic and measured load counts: Pentium III**.

the different application matrices is quite distinct. Regarding model accuracy, the ratio of actual to model load counts are no more than 1.12 on either platform, with a median value of approximately 1.02 on the Ultra 2i and 1.03 on the Power3, as summarized in Table 4.2.[4] Regarding structure, we see that the FEM Matrices 2–9 have normalized load counts of 1.7 or less, and therefore most closely resemble the dense matrix on the basis of loads. In other words, these counts simply reflect their more uniform block structure compared to the other matrices. In contrast, FEM Matrices 10–17 have normalized load counts of approximately 2 (Ultra 2i) or 3 (Power3), and tend to have small $r_{\text{opt}} \times c_{\text{opt}}$. The remaining matrices have normalized load counts of 3 or more on the Ultra 2i, with Matrices 25, 27, and 36 being relatively sparse (fewer than 10 non-zeros per row).

The remaining platforms roughly confirm these observations, with some exceptions. We consider each platform in turn.

The median ratio of actual to model load counts is approximately 1.11 on the Pentium III (Figure 4.5), and is considerably higher—up to 1.63—on Matrices 18–37. Indeed, Matrices 26 and 36 even exceed the approximate upper limit of 5, despite the fact that

[4] These summary statistics can be derived from the detailed tabulated data shown in Appendix E.

$r_{\text{opt}} \times c_{\text{opt}}$ is 1×1 in both cases. The extra loads are due to spill of local integer variables, as we were able to confirm by inspection of the assembly code. On the Pentium III, there are 8 integer registers of which only 4 are typically free for general purpose use. The code of Figure 3.1, even in the 1×1 case, nominally uses about 7 integer variables (`I,y,jj,Aval,j` and the two loop bounds `M` and `Aptr[I+1]`), which clearly exceeds 4 registers. The spilling is associated with the integer iteration variables associated with the outermost loop of Figure 3.1, which explains why it is particularly evident in the case when the innermost loop count is low, *i.e.*, when k/m is small. We conclude that our performance upper bound will be optimistic on the Pentium III for Matrices 18–37 due to these extra operations.

The median ratio of actual to model load counts on the Itanium 1 and Itanium 2 platforms (Figure 4.6) are also reasonably good, at 1.11 and 1.12, respectively. A few anomalously larger ratios occur with Matrices 27 and 36 on Itanium 1, and Matrices 25, 26, and 36 on Itanium 2. The median ratio of actual to model loads is between 1.2–1.25 in these 5 instances. These matrices share a low density (fewer than 10 non-zeros per row), but we do not know precisely why the load counts would be relatively higher than on other matrices on this platform. Nevertheless, as with the Pentium III, we can simply conclude that that our performance bounds will be optimistic in these instances.

Validating cache miss bounds

Figures 4.7–4.11 compare the number of misses predicted by our model to the actual number of actual misses reported by PAPI. (This data appears in tabulated form in Tables E.1–E.5.) Matrices are shown along the x-axis. For each matrix, we ran register blocked SpMV for all $r \times c$ up to 12×12, selected the block size $r_{\text{opt}} \times c_{\text{opt}}$ with the smallest observed running time, and report the following for each cache:

- **Cache miss lower bound**: the number of misses predicted by Equation (4.6) at the value $r_{\text{opt}} \times c_{\text{opt}}$, shown by small dots and a dashed black line.
- **Cache miss upper bound**: the number of misses predicted by Equation (4.7) at the value $r_{\text{opt}} \times c_{\text{opt}}$, shown by asterisks and a dashed blue line.
- **Actual miss count**: the number of measured misses at the best block size, $r_{\text{opt}} \times c_{\text{opt}}$, shown by solid green circles.

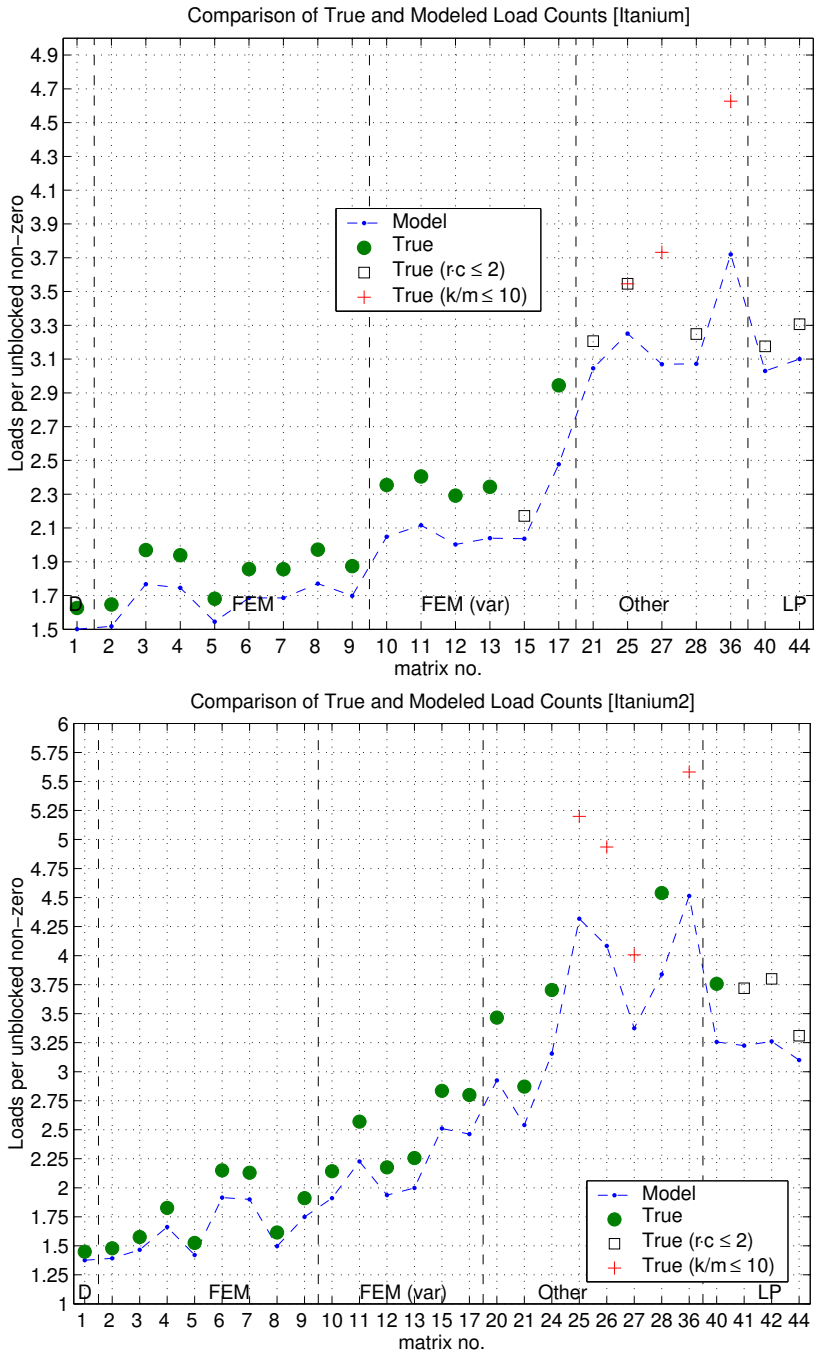

Figure 4.6: **Comparison of analytic and measured load counts: Itanium 1 (top) and Itanium 2 (bottom).**

	Ratio of Actual Counts to Lower Bound								
	Loads			$L_{\kappa-1}$			L_{κ}		
Platform	Min	Median	Max	Min	Median	Max	Min	Median	Max
Ultra 2i	1.00	1.02	1.11	1.07	1.14	1.80	1.00	1.02	1.07
Pentium III	1.00	1.11	1.64	1.02	1.09	1.75	1.00	1.01	1.36
Power3	1.00	1.03	1.05	1.04	1.45	1.84	1.03	1.04	1.08
Itanium 1	1.05	1.10	1.24	1.00	1.02	1.41	1.00	1.01	1.11
Itanium 2	1.05	1.13	1.24	1.01	1.02	1.42	1.00	1.01	1.05

Table 4.2: **Summary of load and cache miss count accuracy**. We show minimum, median, and maximum of ratio of actual load instructions executed to those predicted by Equation (4.5) (columns 2–4). A ratio of 1 would indicate that the model predicts the actual counts exactly. In addition, we show minimum, median, and maximum ratio of actual cache misses to those predicted by the lower bound, Equation (4.6). We show data for the L_1 (columns 5–7) and L_2 (columns 8–10) cache on all platforms except the Itanium 1 and Itanium 2, where we show L_2 (columns 5–7) and L_3 (columns 8–10) data. (The L_1 cache on the Itanium platforms do not cache floating point data [168].)

We are particularly interested in how closely the actual miss counts approach the cache miss lower bound. Refer to Table 4.2 for summary statistics, which we discuss below. For the largest caches, the ratio of actual misses to the lower bound is usually not much more than 1, while the at the smallest caches, the ratios are relatively larger.

In the large caches, the median ratios are all less than 1.04. Even the maximum ratio is less than 1.11, except on the Pentium III which, as we discussed for load counts above, suffers from spilling due to the small number of general purpose integer registers.

In the small caches, the median ratios are as high as 1.45 (Power3), the maxima all exceed 1.4 (Table 4.2), and, roughly speaking, the ratios tends to increase with increasing matrix number (Figures 4.7–4.11). These data suggest that the lower bounds could be refined by accounting for capacity and conflict misses. However, to assess the effect of under-predicting cache misses in the smaller caches, we need to examine the relative contribution of misses at each level to the overall execution time T, *i.e.*, the $(\alpha_{i+1} - \alpha_i)M_i$ terms in Equation (4.4).

For example, consider Matrix 40 on the Power3. From Figure 4.9, the actual cache misses are $M_1 = .18k$ and $M_2 = .10k$. Thus, $(\alpha_2 - \alpha_1)M_1 = (9 - .5 \text{ cy})(0.18k)$, or $1.53k$ cycles, while $(\alpha_{\text{mem}} - \alpha_2)M_2 = 26(.10k) = 2.60k$ cycles. Thus, the relative contribution to execution time from M_2 is larger than from M_1 by a factor of $2.60/1.53 \approx 1.70\times$ in this particular case. By underestimating M_1, we will certainly underestimate T, but the

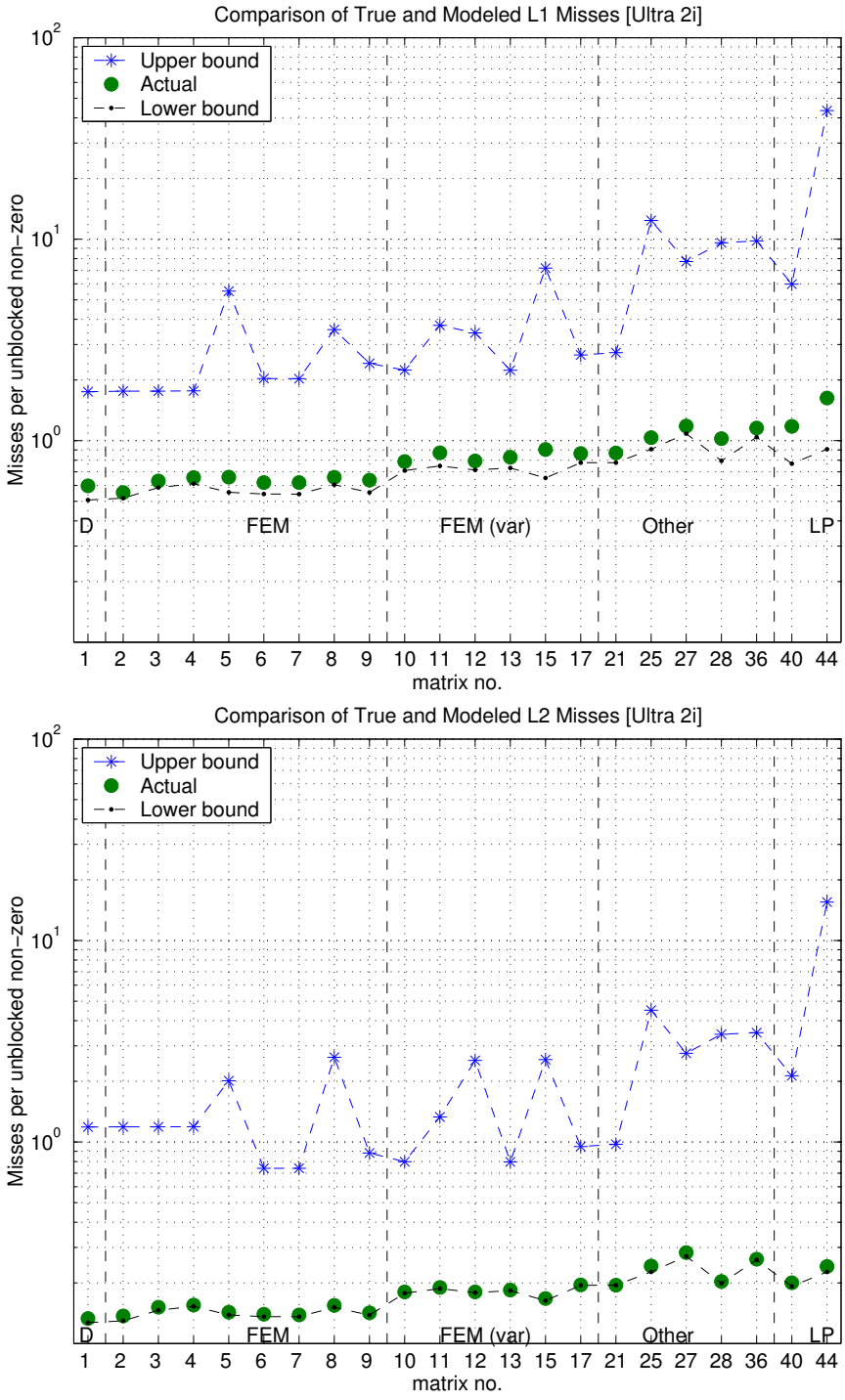

Figure 4.7: **Comparison of analytic cache miss bounds to measured misses: Ultra 2i**. Actual counts of L_1 misses (*top*) and L_2 misses (*bottom*), as measured by PAPI (solid green circles), compared to the analytic lower bound, Equation (4.6) (solid black line), and upper bound, Equation (4.7) (blue asterisks). The counts have been normalized by the number of non-zeros in the unblocked matrix. Matrices (x-axis) that fit within the L_κ cache have been omitted.

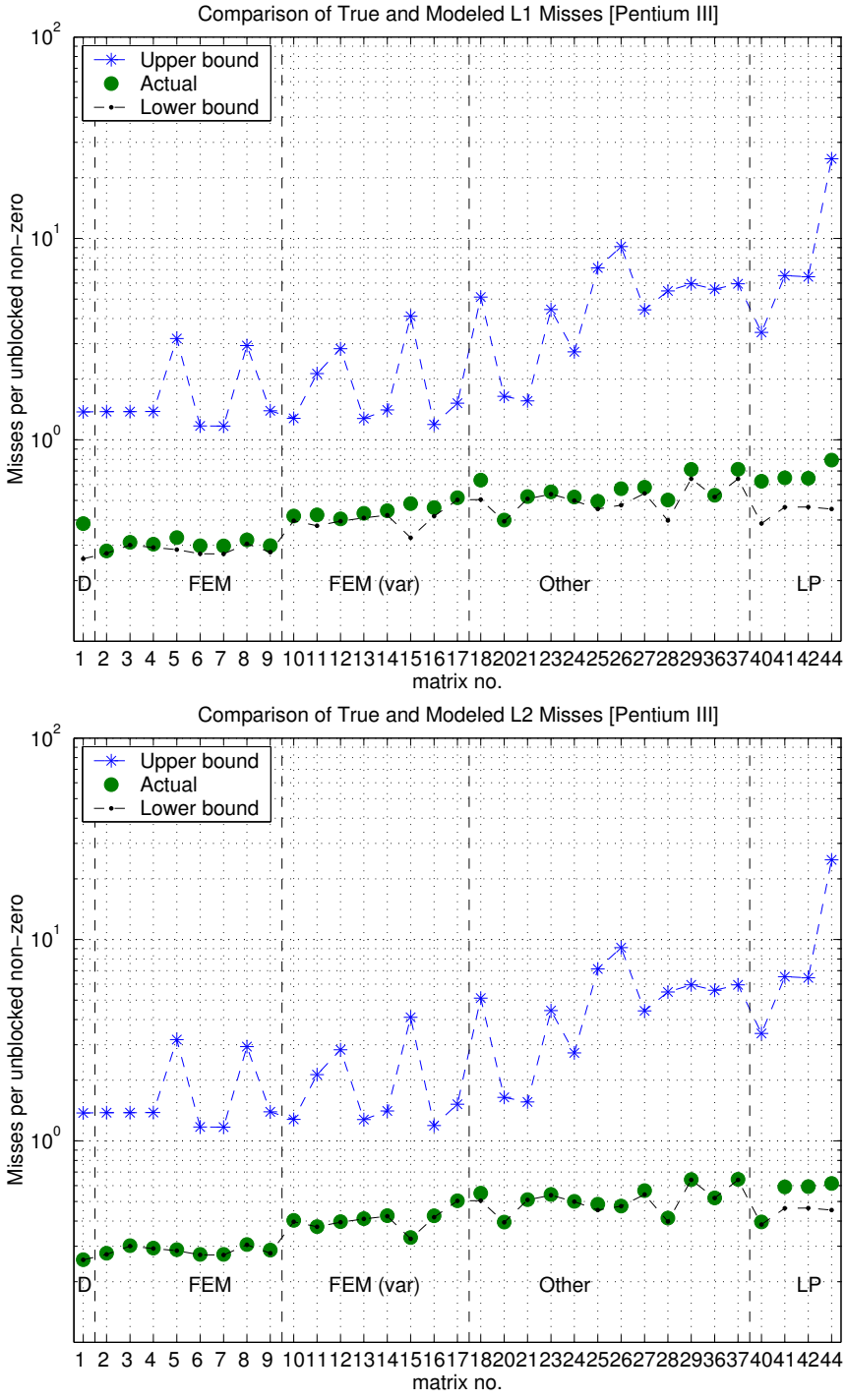

Figure 4.8: **Comparison of analytic cache miss bounds to measured misses: Pentium III**. Actual counts of L_1 misses (*top*) and L_2 misses (*bottom*), as measured by PAPI (solid green circles), compared to the analytic lower bound, Equation (4.6) (solid black line), and upper bound, Equation (4.7) (blue asterisks). The counts have been normalized by the number of non-zeros in the unblocked matrix. Matrices (x-axis) that fit within the L_κ cache have been omitted.

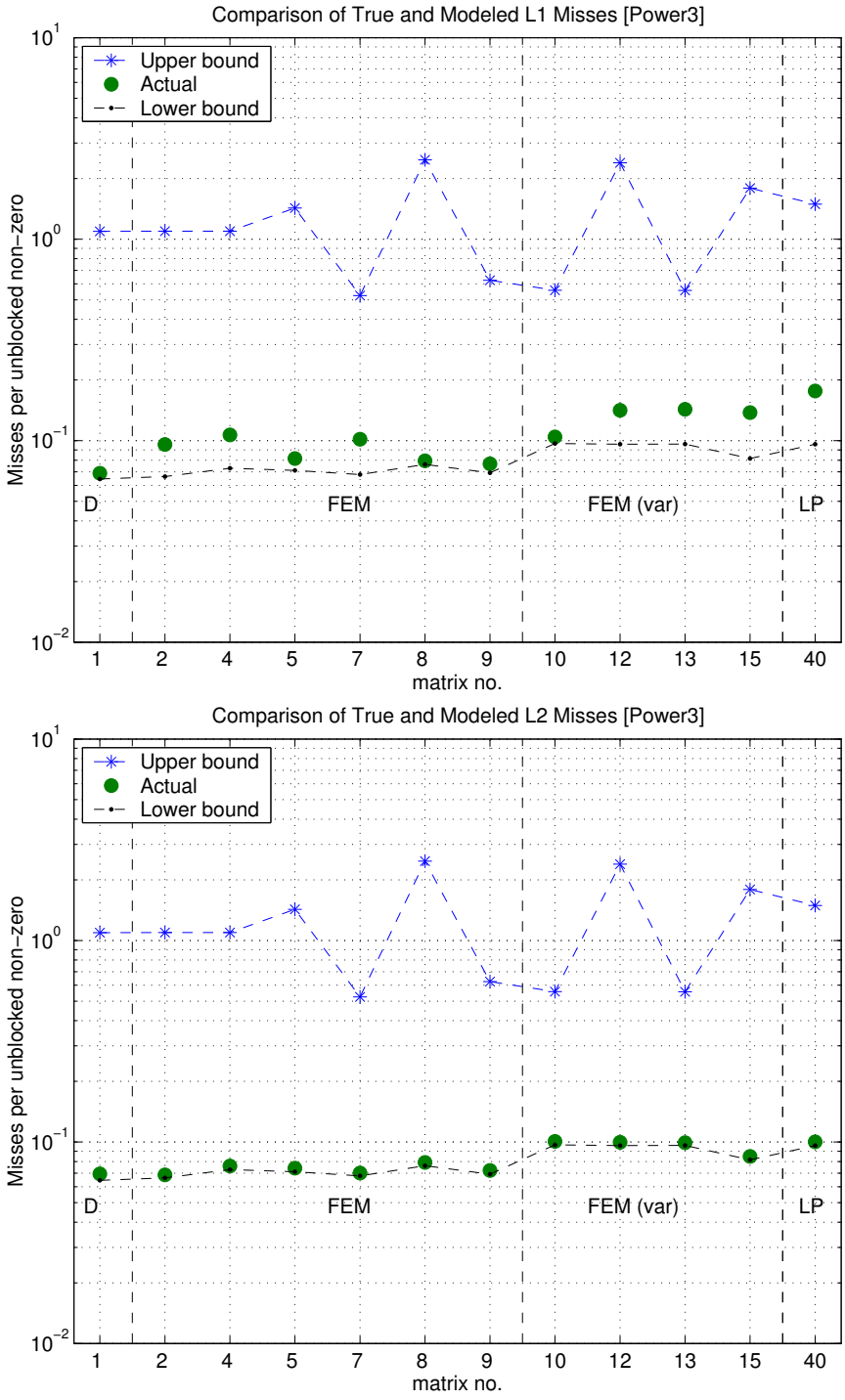

Figure 4.9: **Comparison of analytic cache miss bounds to measured misses: Power3**. Actual counts of L_1 misses (*top*) and L_2 misses (*bottom*), as measured by PAPI (solid green circles), compared to the analytic lower bound, Equation (4.6) (solid black line), and upper bound, Equation (4.7) (blue asterisks). The counts have been normalized by the number of non-zeros in the unblocked matrix. Matrices (x-axis) that fit within the L_κ cache have been omitted.

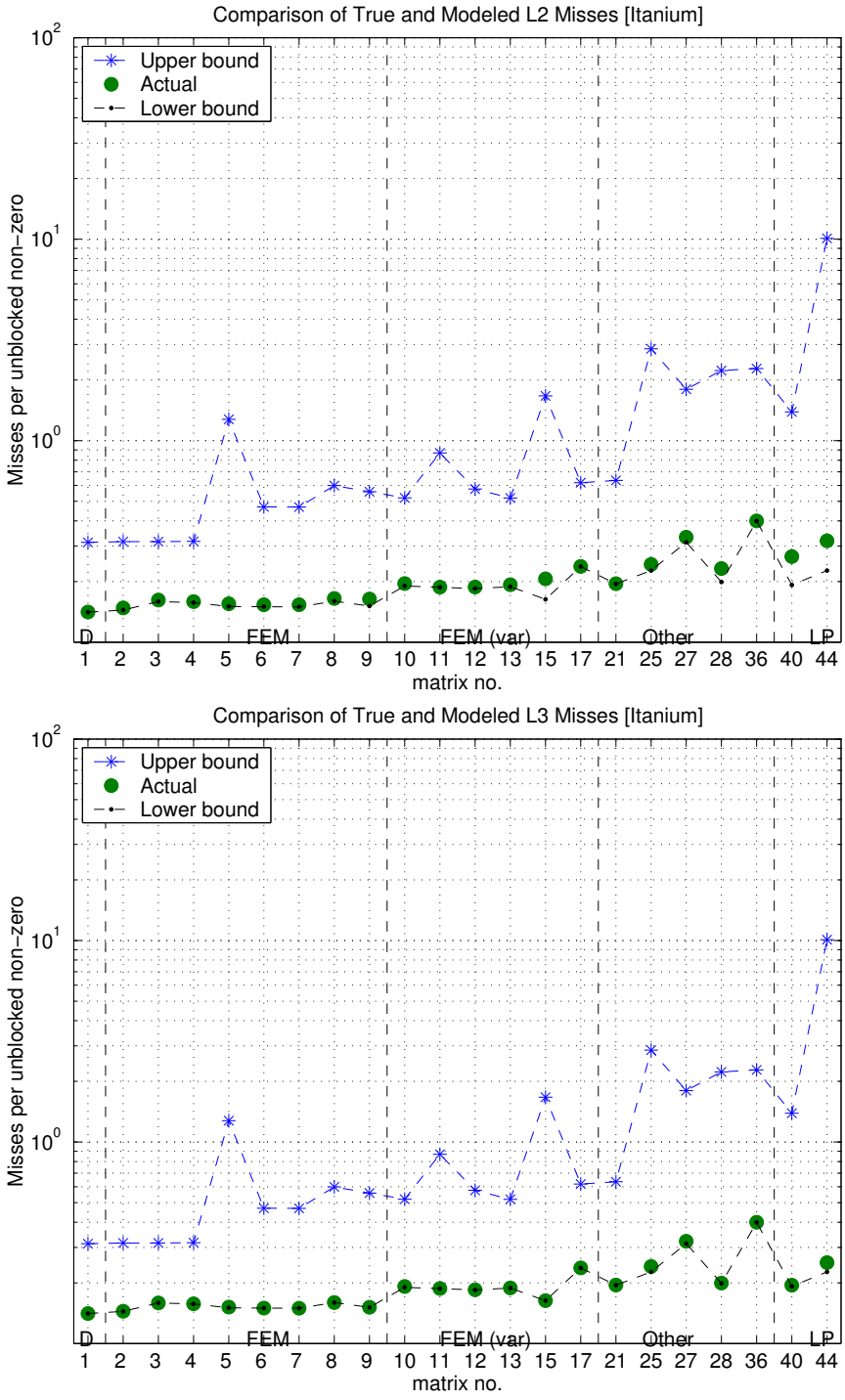

Figure 4.10: **Comparison of analytic cache miss bounds to measured misses: Itanium 1**. Actual counts of L_2 misses (*top*) and L_3 misses (*bottom*), as measured by PAPI (solid green circles), compared to the analytic lower bound, Equation (4.6) (solid black line), and upper bound, Equation (4.7) (blue asterisks). The counts have been normalized by the number of non-zeros in the unblocked matrix. Matrices (x-axis) that fit within the L_κ cache have been omitted.

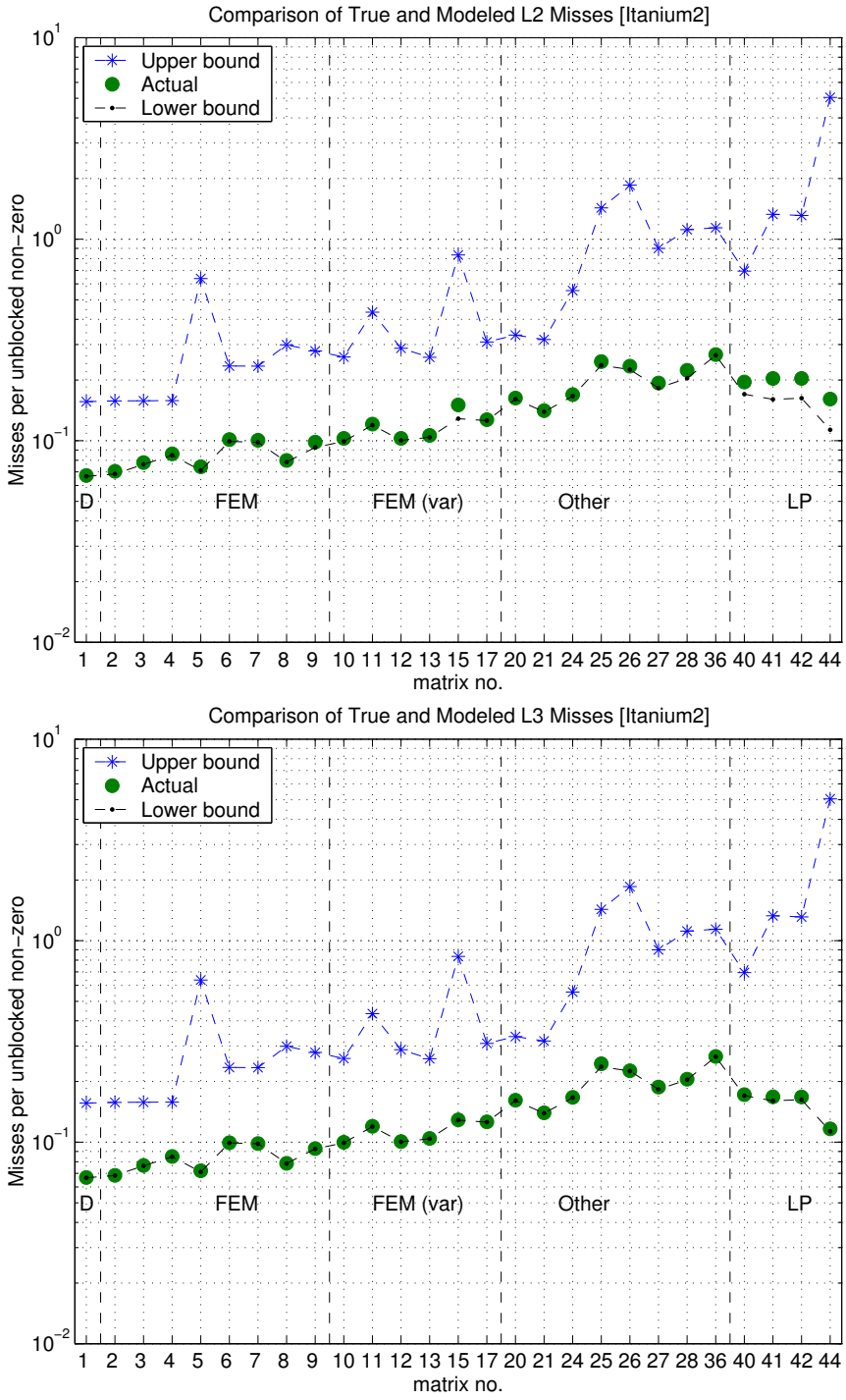

Figure 4.11: **Comparison of analytic cache miss bounds to measured misses: Itanium 2**. Actual counts of L_2 misses (*top*) and L_3 misses (*bottom*), as measured by PAPI (solid green circles), compared to the analytic lower bound, Equation (4.6) (solid black line), and upper bound, Equation (4.7) (blue asterisks). The counts have been normalized by the number of non-zeros in the unblocked matrix. Matrices (x-axis) that fit within the L_κ cache have been omitted.

ultimate impact on T depends on the latencies and absolute cache miss values. We consider the breakdown of the terms contributing to T in more detail in Section 4.2.3, where we try to understand some of the architectural implications of the performance bounds model.

4.2.3 Key results: observed performance vs. the bounds model

This section evaluates the performance of the best register blocked SpMV implementations generated by SPARSITY against the best performance predicted by the upper bounds. We organize our discussion around the following key findings:

1. *For FEM matrices, we can frequently achieve 75% or more of the performance upper bound.* This result is summarized graphically in Figure 4.16. In short, provided we can select the optimal block size (addressed in Chapter 3), additional performance gains from low-level tuning will thus be limited.

2. *For FEM Matrices 2–9, typical speedups range between* $1.4-4.1\times$ *on 7 of the 8 evaluation platforms.* Speedups across platforms are summarized in Figure 4.17. Speedups are smallest on the Power3, where even for Matrices 2–9, maximum speedup is less than $1.3\times$.

   On the remaining matrices, speedups are modest owing to their non-zero structure. Nevertheless, maximum speedups of up to $2.8\times$ are possible on FEM Matrices 10–17, and up to $2\times$ on Matrices 18–44.

3. *The fraction of machine peak achieved by register blocked SpMV correlates well with a machine-specific measure of balance related to our latency model.* Callahan, *et al.*, define *machine balance* to be

$$\text{Balance} = \frac{\text{Peak performance (flops / time)}}{\text{Main memory bandwidth (words / time)}} \tag{4.10}$$

   which measures the amount of work (flops) that can be performed per word read from memory [65]. We define *sustainable balance* to be Equation (4.10) with peak machine speed for the numerator, and β_s, defined in Section 4.2.1, for the denominator. We show a simple relationship between sustainable balance the achieved SpMV performance on all classes of matrices. Thus, this measure of balance is an intuitively simple way to infer a given machine's ability to run SpMV well. A graphical summary of the relationship between observed SpMV performance and machine balance appears in Figure 4.18.

4. *For a multi-level memory hierarchy, our performance model favors cache designs with strictly increasing cache line sizes.* Owing to the predominantly streaming behavior of SpMV, the model implies that on a machine with a multi-level memory hierarchy in which the L_i and L_{i+1} cache have the same line size, the larger cache is effectively "transparent." This fact becomes apparent when we look at how the model charges execution time to each level of the memory hierarchy, depicted graphically in Figure 4.19. Indeed, this conclusion applies more broadly to all applications dominated by stride-1 streaming memory behavior (*e.g.*, Basic Linear Algebra Subroutines (BLAS) Level 1 and Level 2 calculations).

The main experimental evidence for these claims appears in Figures 4.12–4.15. Each figure shows performance data for one of the 8 platforms and the subset of 44 benchmark matrices that exceed the size of the largest cache. We specifically compare the performance of the following implementations and bounds model predictions:

- **Analytic upper bound**: The highest value of the analytic upper bound on performance (Mflop/s) over all block sizes, shown by a blue solid line. We compute the upper bound as discussed in Section 4.1 using the minimum latencies shown in Table 4.1. We denote the block size of the implementation shown by $r_{\text{up}} \times c_{\text{up}}$.

- **PAPI upper bound**: An upper bound on performance for the $r_{\text{up}} \times c_{\text{up}}$ implementation, represented by pink solid triangles. To obtain the PAPI upper bound, we substitute measured cache misses into Equation (4.4) and use the minimum latencies shown in Table 4.1. This calculation is equivalent to assuming precise knowledge of true memory operations and cache misses, and therefore represents a more realistic upper bound than the analytic bound. On the Ultra 3, Pentium III-M, and Power4, we omit the PAPI upper bound since PAPI was not available for these machines at the time of this writing.

- **Actual best**: The best measured performance over all block sizes for the SPARSITY-generated implementations. Let the block size of the implementation shown in the figure be $r_{\text{opt}} \times c_{\text{opt}}$. We show the performance of the best implementation using three different markers: a solid green circle by default, with two additional cases. First, if the block size is small (where we define "small" to mean $r_{\text{opt}} \cdot c_{\text{opt}} \leq 2$), we use a black hollow square. Second, if fill led to the total size $V_{r_{\text{opt}} c_{\text{opt}}}(A)$ of the blocked data

structure exceeding the size of the 1×1 data structure by more than 25%, then we show the performance using a red solid marker. (Matrix data structure size is given by Equation (3.1) in Section 3.1.1.) These two conditions can occur simultaneously, in which case both markers are shown.

- **Reference**: The unblocked (1×1) implementation is represented by asterisks.
- **Analytic lower bound**: We show the value of the performance lower bound for the block size $r_{\text{up}} \times c_{\text{up}}$using a solid black line. This bound was obtained by evaluating Equation (4.4) with the maximum latencies shown in Table 4.1 and the upper bound on cache misses given by Equation (4.7).

In general, $r_{\text{opt}} \times c_{\text{opt}}$ and $r_{\text{up}} \times c_{\text{up}}$ will not necessarily be equal, and the upper bound at $r_{\text{opt}} \times c_{\text{opt}}$ will be closer to the true $r_{\text{opt}} \times c_{\text{opt}}$ performance. We show the bound at $r_{\text{up}} \times c_{\text{up}}$ since we are most interested in how fast SpMV runs independent of scheduling issues. To see when $r_{\text{opt}} \times c_{\text{opt}}$ and $r_{\text{up}} \times c_{\text{up}}$ agree, refer to the detailed tables in Appendix E.

To help the reader interpret the data of Figures 4.12–4.15, we discuss two platforms as examples: the Ultra 2i and Itanium 2. Following these examples, subsequent sections address each of our 4 key conclusions, summarizing the data on all platforms.

On the Ultra 2i, Figure 4.12 (*top*), the reference performance is nearly flat at 35 Mflop/s, or 5.25% of machine peak, on FEM Matrices 2–17. Reference performance on the remaining matrices is much more variable, but also never exceeds 35 Mflop/s. The analytic upper bound indicates that considerable speedups should be possible, and that nearly 10% of machine peak may be possible. This bound tends to decrease with increasing matrix number, and exhibits approximately three plateaus at Matrices 2–9, Matrices 10–17, and Matrices 18–44. The differences in these plateaus reflects the differences in non-zero structure among these groups, as we observed for the normalized load counts in Section 4.2.2. The PAPI upper bound closely tracks the analytic upper bound. Since the PAPI bound "models" misses exactly, the fact that it is typically within 90% of the analytic bound on all but Matrix 44 indicates that our modeling of misses is reasonable on this platform.

How does the actual best performance compare, both to the upper bounds and to the reference? For the dense Matrix 1, SpMV performance is very close to the upper bounds, indicating that in the absence of irregular memory references, the upper bound is nearly attainable. For FEM Matrices 2–9, the actual best performance is 1.4–1.65× faster

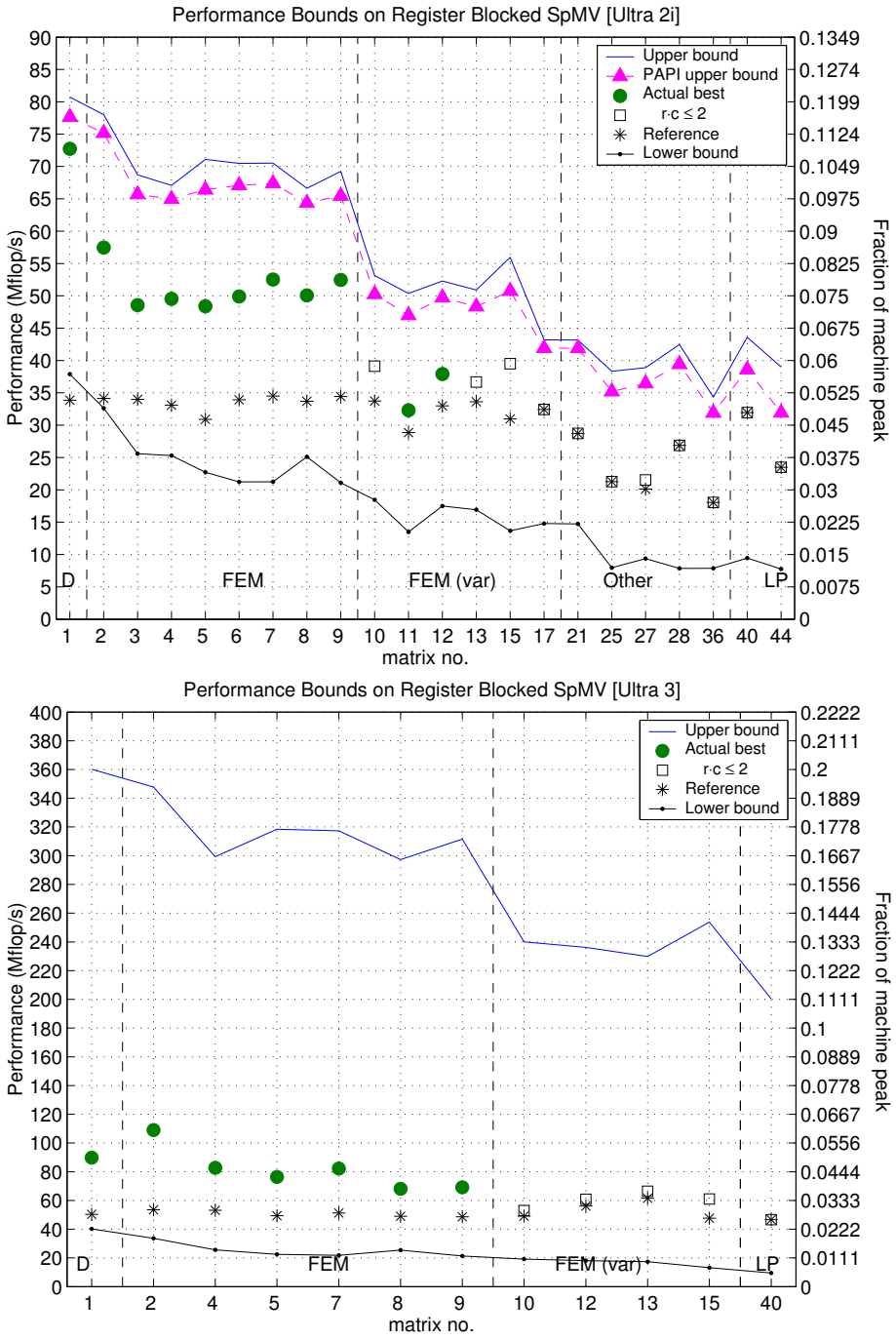

Figure 4.12: **Comparison of observed performance to the bounds: Ultra 2i and Ultra 3**. (*Top*) Performance data on Ultra 2i. DGEMV performance: 59 Mflop/s. Peak: 667 Mflop/s (*Bottom* Performance data on Ultra 3. DGEMV: 311 Mflop/s. Peak: 1.8 Gflop/s. Note: PAPI was not available on this platform.

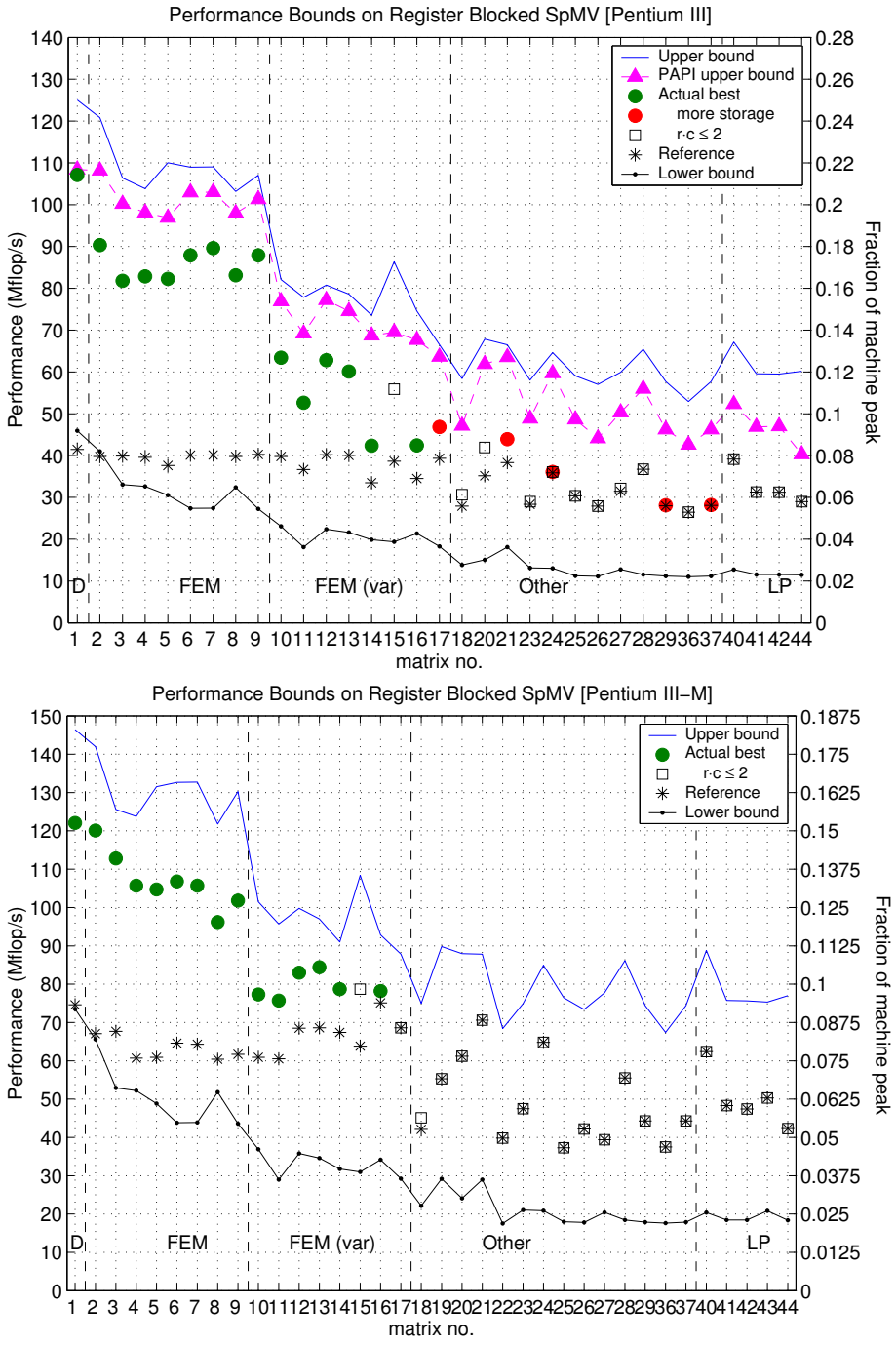

Figure 4.13: **Comparison of observed performance to the bounds: Pentium III and Pentium III-M**. (*Top*) Performance data on Pentium III. DGEMV performance: 58 Mflop/s. Peak: 500 Mflop/s. (*Bottom* Performance data on Pentium III-M. DGEMV: 150 Mflop/s. Peak: 800 Mflop/s. Note: PAPI was not available on this platform.

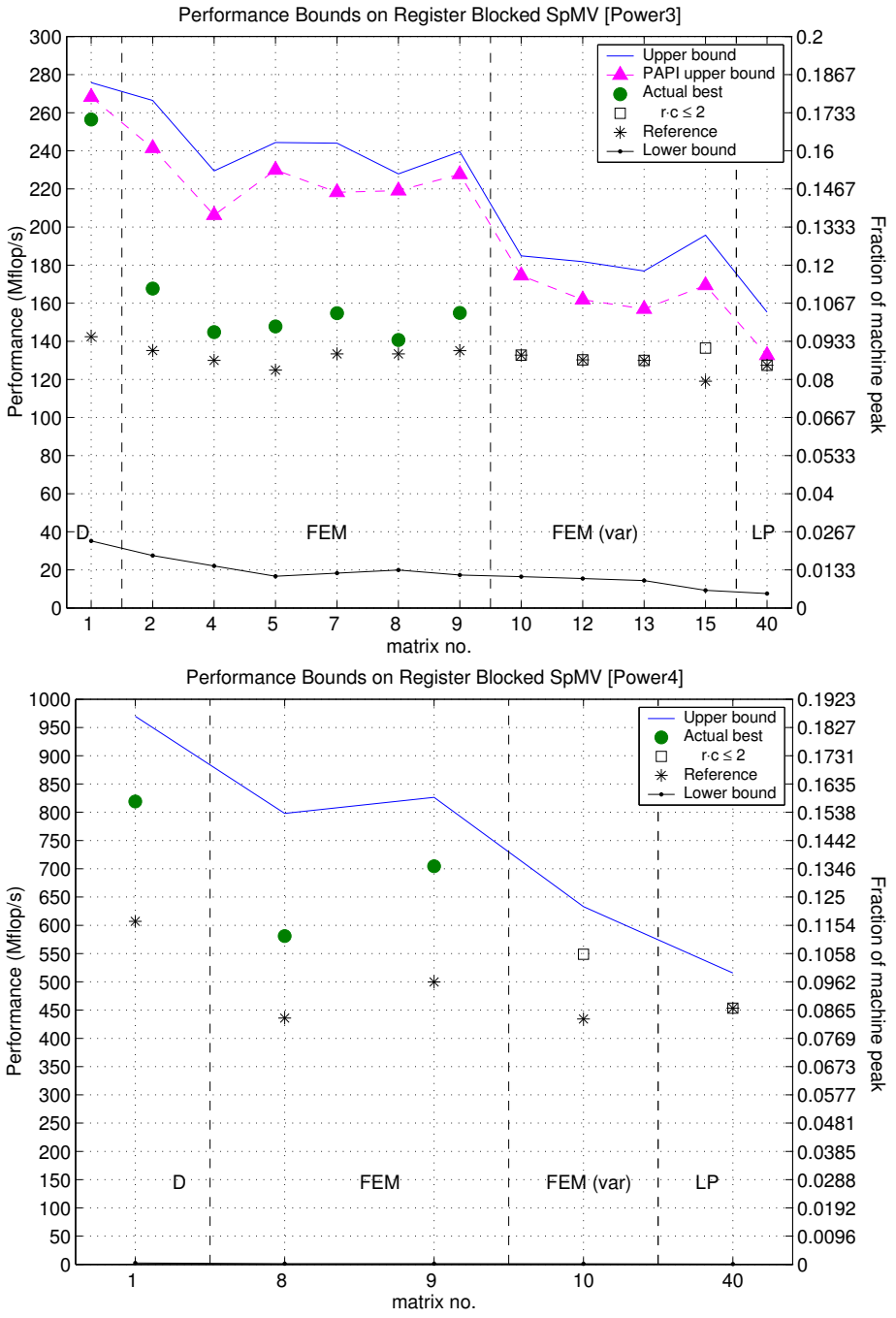

Figure 4.14: **Comparison of observed performance to the bounds: Power3 and Power4**. (*Top*) Performance data on Power3. DGEMV performance: 260 Mflop/s. Peak: 1500 Mflop/s. (*Bottom* Performance data on Power4. DGEMV: 915 Mflop/s. Peak: 5.2 Gflop/s. Note: PAPI was not available onthis platform.

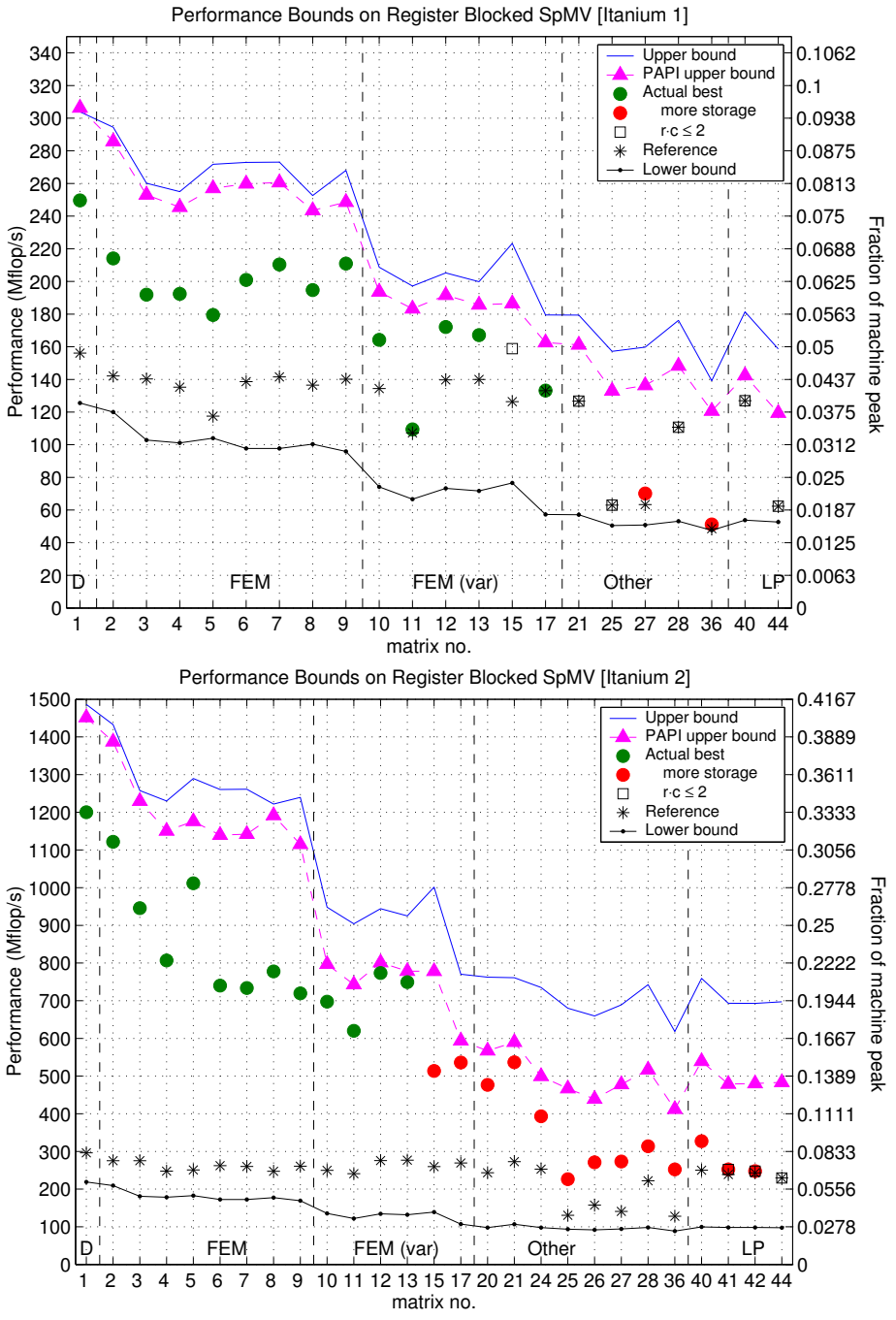

Figure 4.15: **Comparison of observed performance to the bounds: Itanium 1 and Itanium 2**. (*Top*) Performance data on Itanium 1. DGEMV performance: 315 Mflop/s. Peak: 3.2 Gflop/s. (*Bottom* Performance data on Itanium 2. DGEMV: 1.33 Gflop/s. Peak: 3.6 Gflop/s.

than the reference, with a median speedup of 1.5×. Furthermore, the best performance is within 75–84% of the PAPI upper bound. Thus, additional low-level tuning will could yield an additional improvement in execution time by an additional factor of at most 1.2–1.3—an appreciable but limited benefit.

For Matrices 10–17, speedups are more modest, ranging from none (Matrix 17) to at most 1.3× (Matrix 15), with a median speedup of 1.16×; the fraction of the upper bound achieved is also somewhat smaller, ranging from 70%–81%, with a median of just 79%. For the remaining matrices, speedups are no larger than 1.1×, and the fraction of the upper bound achieved ranges widely from less than 60% up to 84%, with a median of 70%. Since the analytic and PAPI bounds are close, errors in miss modeling probably do not explain why the bound might be optimistic. However, most of the implementations shown use a small block size ($r_{\text{opt}} \cdot c_{\text{opt}} \leq 2$). As discussed in Section 4.1.2, we expect the overhead from integer operations and branch mispredictions to become less visible as $r \cdot c$ *increases*. Thus, the upper bound on Matrices 10–44 may be optimistic as a result of our assumption to neglect the cost of everything except memory references, though this cannot be proven from the data alone.

The trend of decreasing performance with increasing matrix number persists on the Itanium 2, Figure 4.15 (*bottom*). However, there are differences compared to the Ultra 2i. For instance, the fraction of the PAPI upper bound achieved is highest for Matrices 10–17, with a median value of 90%. In contrast, Matrices 2–9 have a median fraction of only 67%. Nevertheless, the fraction of machine peak achieved is much higher on Itanium 2 than on the Ultra 2i—Itanium 2 achieves up to 31% of machine peak on Matrix 2, compared to about 8.5% for the same matrix on the Ultra 2i. By this measure, Itanium 2 would appear to be a better machine for SpMV. This observation leads us to ask whether and how we can compare different architectures with respect to their SpMV performance.

On Itanium 2, the relative gap between the analytic and PAPI upper bounds becomes much larger as matrix number increases, particularly, Matrices 20–44, than on the Ultra 2i. As noted in Section 4.2.2, the analytic bounds undercount the actual number of loads executed, and we therefore expect this discrepancy. We do not presently know the precise reason why actual load counts are high, but assuming measured load counts reflect truth, we should take the PAPI upper bound as a more realistic bound.

A third way in which Itanium 2 differs from the Ultra 2i is the fact that an increase in matrix storage due to fill can nevertheless lead to considerable speedups. On Matrices

15–40, storage increases of more than 25% show surprising speedups of up to 2×. This demonstrates the potentially non-trivial ways in which performance, tuning parameters, and matrix structure interact on modern machines.

This brief sample of observations raises a number of general questions, which we now address by looking at the data across all platforms:

1. How does achieved performance compare to the upper bound, across all platforms and the three classes of matrices corresponding to the plateaus on the Ultra 2i?

2. What is the range of speedups across all platforms? Thus, if there is any doubt about the upper bounds or what fraction of the bounds we can achieve, can we nevertheless assert that good improvements are possible?

3. How do the platforms compare in terms of the fraction of machine peak achievable? Can we characterize machines that yield high fractions of peak?

4. Can the upper bounds model tell us anything about how architectures or memory hierarchies should be designed for SpMV?

1. Proximity to the performance upper bound

Our first key result is that we can frequently achieve at least 75% of the performance upper bound, particularly for the class of FEM Matrices 2–17.

Consider the following three groups of matrices: FEM Matrices 2–9, FEM Matrices 10–17, and Matrices 18–44 from various non-FEM applications, including linear programming. For each platform and group, we compute the minimum, median, and maximum fraction of upper bound achieved by best implementations. The "upper bound" is taken to be the PAPI upper bound on all platforms for which we had PAPI data, and the analytic upper bound otherwise. We show these summary statistics in Figure 4.16.

For FEM Matrices 2–9, the median fraction is at least 78% on 5 of the 8 platforms, and the maximum fraction is 84% or more on 6 of the 8 platforms. For FEM Matrices 10–17, the median fraction is 78% or more on 7 of the 8 platforms, though there is only 1 matrix in this group on the Power4. The median fractions for Matrices 18–44 tends to be much lower—below 70% on all 5 platforms for which there are at least two matrices in this class that exceeded the largest cache. Furthermore, the spread between minimum and maximum values tends to be much larger than on the other platforms. However, the

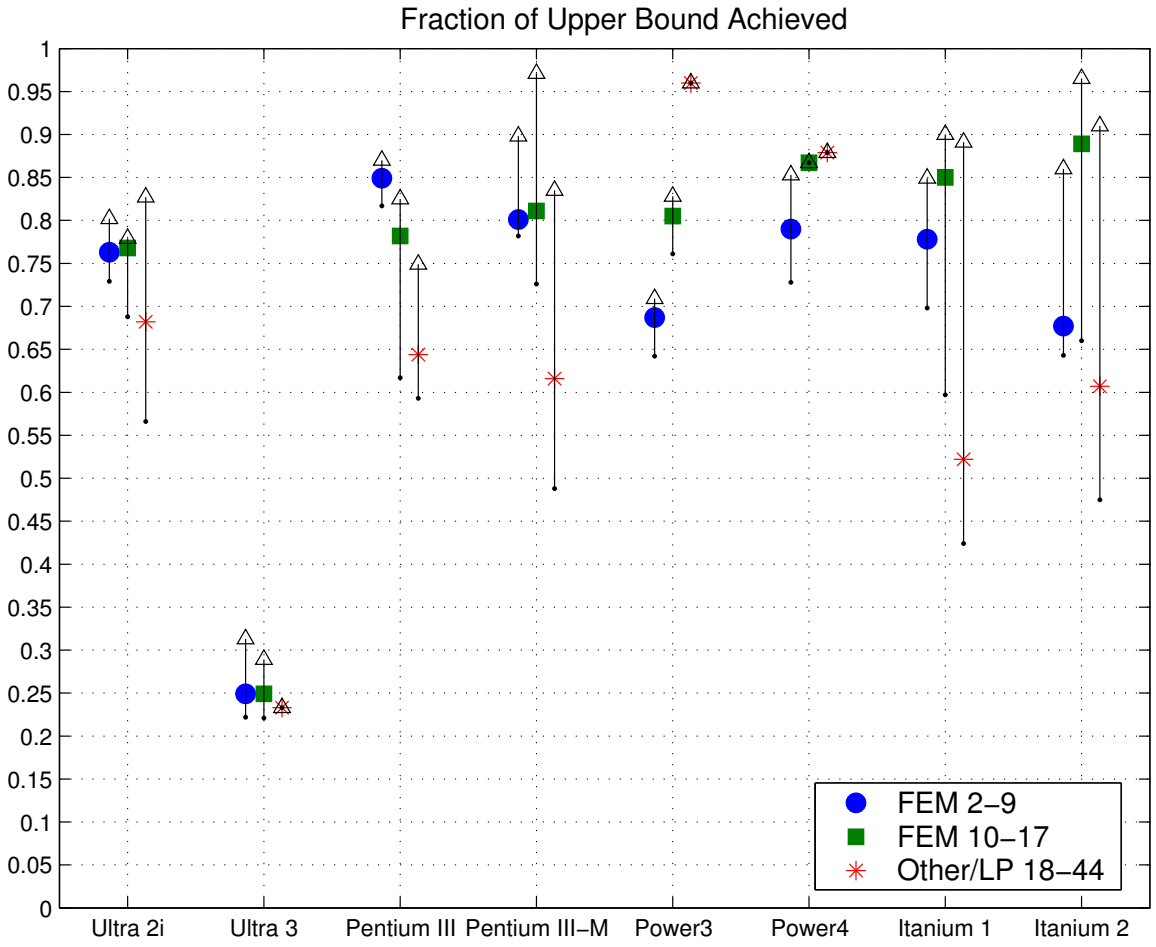

Figure 4.16: **Fraction of performance upper bound achieved**. We summarize the fraction of the performance upper bound achieved across all platforms and separated by application. Median fractions for FEM Matrices 2–9 are shown by blue solid circles, for FEM Matrices 10–17 by green solid squares, and for all remaining matrices by red asterisks. Arrows indicate a range from the minimum fraction (a dot) to the maximum (a triangle). We use the PAPI upper bound on all platforms except the Ultra 3, Pentium III-M, and Power4, where we instead use the analytic upper bound model of Section 4.1.

maximum fraction can exceed 75% on those same platforms, and is above 75% on all but the Ultra 3 and Pentium III platforms. In short, we can frequently achieve 75% of the upper bound on FEM matrices, where we expect blocking to pay off.

The Ultra 3 achieves an anomalously low fraction of the upper bound. However, DGEMV performance, at 311 Mflop/s (Table 3.1), is 86% of the 360 Mflop/s upper bound on the dense matrix shown in Figure 4.12 (*bottom*). Thus, the upper bound is likely to be

reasonable. With improved scheduling and low-level tuning, we should expect to achieve a much greater fraction of the upper bound.

2. Speedups across platforms

We summarize minimum, median, and maximum values of speedups for each platform and matrix group in Figure 4.17. The best median speedups of at least 1.4× are achieved on FEM Matrices 2–9 on all platforms but the Power3. Maximum speedups exceed 1.7× on 5 of the 8 platforms, and can reach as high as 4.1× (Itanium 2). Indeed, even the Ultra 3, on which the fraction of the upper bound and fraction of machine peak are very low, still achieves speedups of at least 1.4× and up to 2×, with a median speedup of over 1.5×.

On FEM Matrices 10–17, median performance is modest, exceeding 1.4× on only 2 of the 8 platforms. Though disappointing, this behavior is not surprising given that the block structure of these matrices is quite different from FEM Matrices 2–9. We address the question of what kind of block structure is present, and what techniques (such as variable blocking and splitting) can be used to exploit that structure in Chapter 5.

Matrices 18–44 show the smallest speedups—maximum speedups are at most 1.2× on all but 1 platform. Recall that the best block sizes for these matrices are typically small (Section 4.2.2), raising the question of whether the small block size implementations can be better tuned. Since the bounds also tend to be optimistic on these matrices, it is currently unclear how much better we could do at small block sizes. More refined models of misses and a better understanding of the instruction overhead are needed to resolve this question.

3. Correlations between SpMV performance and machine balance

We show that the fraction of machine peak achieved by SpMV is correlates well with a measure of *machine balance* based on our latency parameters. Machine balance is a machine-dependent but application-independent parameter which characterizes the rate of computation in the CPU relative to the rate at which memory can feed the CPU [65]. Balance is traditionally defined to be the peak flop execution rate divided by the main memory bandwidth. We assume the unit of balance to be flops per double in this discussion.

Recalling the general definition of machine balance given by Equation (4.10), we define *sustainable balance* with respect the the sustainable memory bandwidth β_s (Section 4.2.1) according to our model. Section 4.2.1 argues that this bandwidth is a more

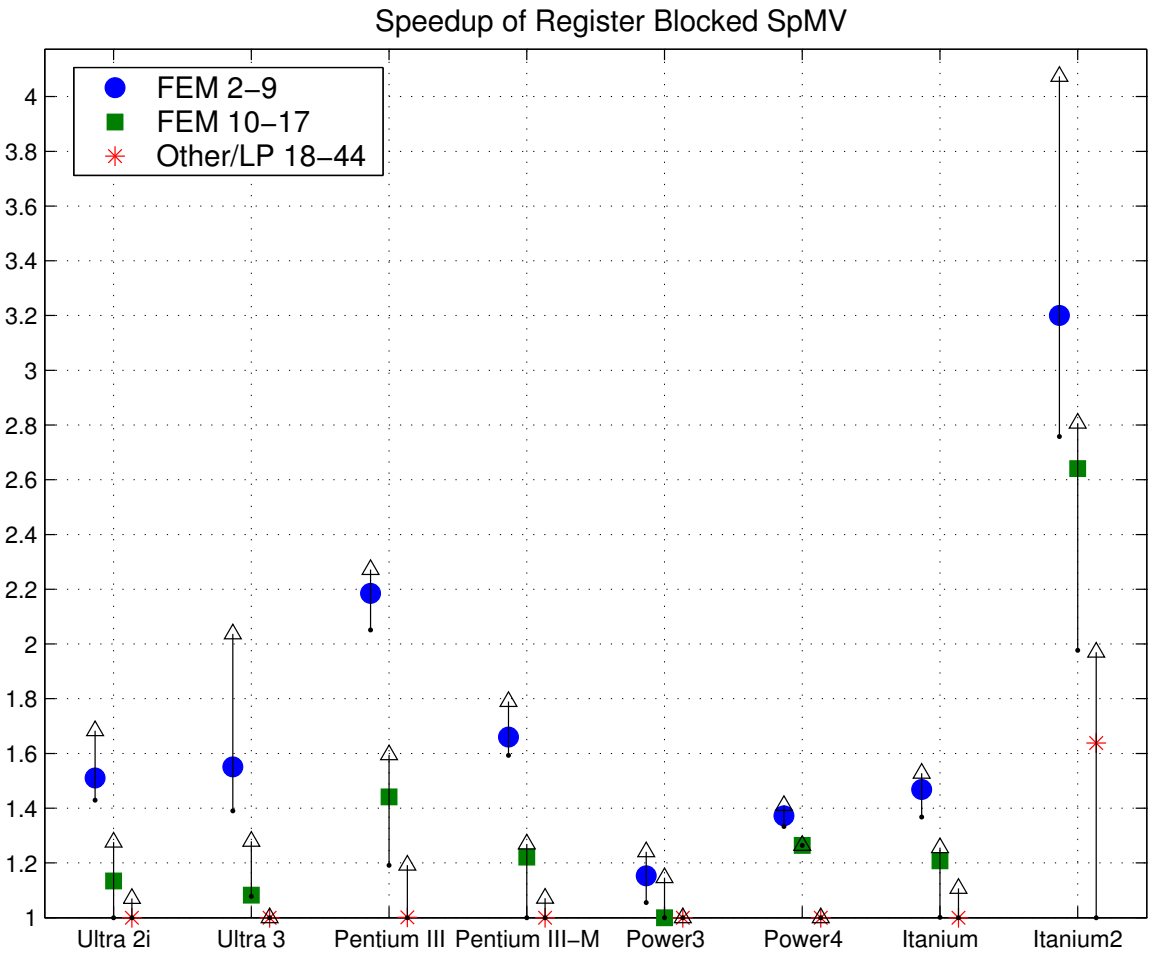

Figure 4.17: **Summary of speedup across platforms**. We summarize the speedup of the best implementation over all block sizes relative to the unblocked (1×1) implementation. For each platform, we separate data by application. Median fractions for FEM Matrices 2–9 are shown by blue solid circles, for FEM Matrices 10–17 by green solid squares, and for all remaining matrices by red asterisks. Arrows indicate a range from the minimum fraction to the maximum.

realistic measure of memory bandwidth than the manufacturer's reported peak value. Let μ denote a platform (microprocessor) with peak performance $\rho(\mu)$ (in Mflop/s) and sustainable bandwidth $\beta_s(\mu)$ (in millions of doubles per second). The sustainable balance of μ is defined to be $B(\mu) = \rho(\mu)/\beta_s(\mu)$.

The kernel DGEMV will only be compute-bound if we can read data from memory at a rate of at least 1 double for every two flops, considering only the time to read the matrix from main memory (*i.e.*, ignoring source and destination vector loads). Thus, the ideal

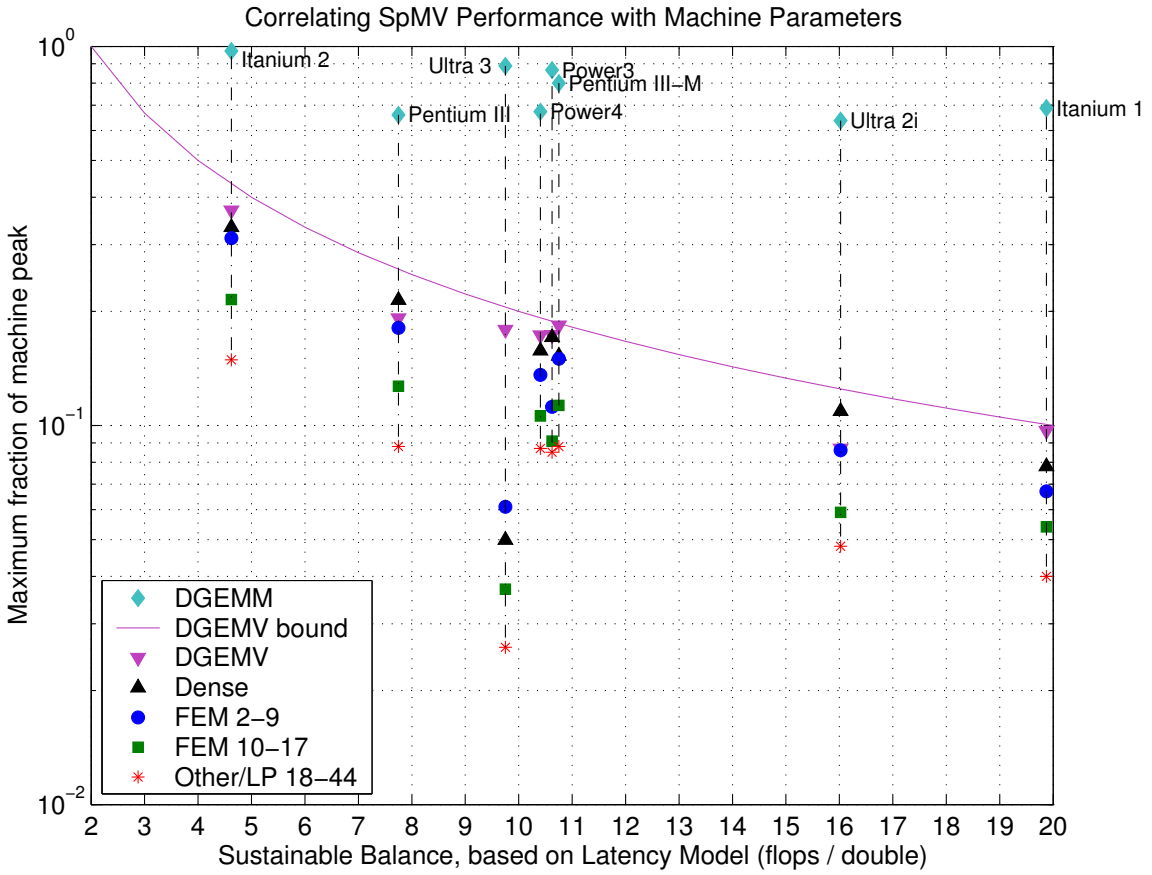

Figure 4.18: **Correlating register blocked SpMV performance with a measure of machine balance**. For each platform, we show the maximum fraction of machine peak achieved, in each of four matrix groups, as a function of a measure of sustainable balance $B(\mu)$ based on our latency model. The four matrix groups are: the dense register profile (maximum fraction over all $r \times c$ is shown), FEM Matrices 2–9, FEM Matrices 10–17, and Matrices 18–44. Data for a given platform are connected by a vertical line. Platform names appear next to the DGEMM data point (blue diamonds). The DGEMV bound is the best possible fraction of peak when performing 2 flops per double (*i.e.*, 2 divided by the sustainable balance).

balance for DGEMV is $B(\mu) \leq 2$. DGEMV could attain machine peak on a hypothetical machine with such a balance. For SpMV, since the index structure requires more data movement per matrix element (but varies by matrix), the ideal value of balance is strictly less than 2 flops per double.

We show the correlation between $B(\mu)$ and achieved SpMV performance in Figure 4.18. For each platform and matrix group, we plot the maximum fraction of machine

peak achieved for register blocked SpMV, versus $B(\mu)$ for each machine μ. For reference, we show the best fraction of peak achieved for a dense matrix in sparse format by a black solid upward-pointing triangle, and the performance of DGEMM and DGEMV taken from Table 3.1, shown by cyan diamonds and purple downward-pointing triangles, respectively. A solid vertical line connects the data points for a single machine. The name of the platform appears to the immediate right of the DGEMM data point. Finally, we show a solid purple line corresponding to a bound on DGEMV performance. This bound is simply $2/B(\mu)$, since DGEMV performs at most 2 flops per double. Although none of the eight machines has a sustainable balance of 2 or less, Figure 4.18 shows a general trend: the fraction of achieved peak increases as balance decreases, independent of the type of matrix.

The Ultra 3 is an outlier. However, DGEMV runs at 311 Mflop/s, or approximately 17% of machine peak. Thus, if it were possible to tune SpMV more carefully, we would expect to confirm the trend shown in Figure 4.18.

The relationship between $B(\mu)$ and achieved performance confirms the memory-bound character of matrix-vector multiply operations (dense or sparse), and furthermore hints at a method for characterizing a machine's suitability for performing these operations efficiently via the model parameters α_i and α_{mem}. However, since we primarily determine these parameters empirically, we cannot at present provide much insight into what specific aspects of memory system design influence the performance of these operations. Nevertheless, moving toward such an understanding is a clear opportunity for future work.

4. Implications for memory hierarchies: strictly increasing cache line sizes

The performance model presented in Section 4.1 favors strictly increasing cache line sizes for multi-level memory hierarchies. We illustrate this point, and present a simple example which shows how much we might speed up SpMV by varying the relative line sizes.

Our model of execution time, Equation (4.3), assigns a cost of $\alpha_i H_i$ to all hits H_i to the L_i cache. Since $\alpha_i \leq \alpha_{i+1}$, we would prefer to hit in L_i instead of L_{i+1}. Equal line sizes, $l_i = l_{i+1}$, implies $M_{\text{lower}}^{(i)} = M_{\text{lower}}^{(i+1)}$ according to Equation (4.6). Thus, assuming the true number of cache misses M_i in the L_i cache is exactly $M_{\text{lower}}^{(i)}$, $H_{i+1} = M_i - M_{i+1} = 0$. Thus, any miss in the L_i cache is not serviced by the L_{i+1} cache, and is instead forwarded to the next level at a higher cost. The L_{i+1} cache effectively becomes unused.

Figure 4.19 (*top*) shows $\alpha_i H_i$ at all cache levels and $\alpha_{\text{mem}} H_{\text{mem}}$ for Matrix 40

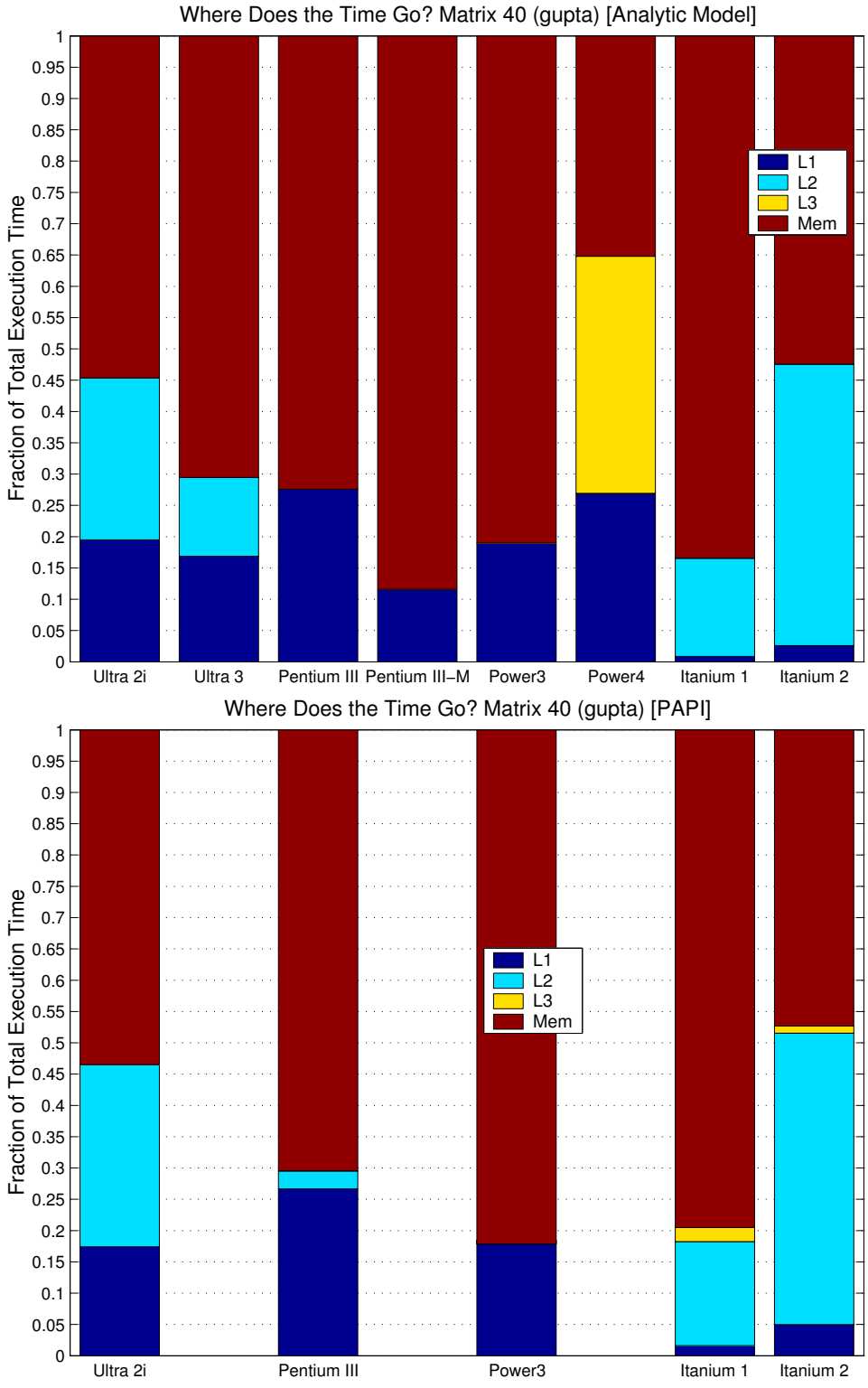

Figure 4.19: **Breakdown of how the model assigns cost to each level of the memory hierarchy**. We show the relative contribution to total execution time of each term $\alpha_i H_i$ in Equation (4.3). (*Top*) Execution time breakdown based on substituting our analytic load count and cache miss lower bound. (*Bottom*) Execution time breakdown based on substituting PAPI load and cache miss data into the execution time model.

on all 8 platforms, where we substitute our analytic model of loads and lower bounds on cache misses into Equation (4.4). Specifically, each bar represents the total execution time according to the model on a given platform, and is segmented into regions shaded by the relative cost $\alpha_i H_i/T$, where T is the total execution time.

The fraction of total execution time assessed to memory accesses is at least 60% on all platforms except the Power4 and Itanium 2, and as high as 88% on the Pentium III-M. This observation confirms the general intuition that memory accesses dominate overall execution time for SpMV. However, caches play an important role, especially on the Power4 and Itanium 2 where they account for 47–65% of total execution time.

However, some of the caches have "disappeared." On the Pentium III, Pentium III-M, Power3, and Power4 platforms, $l_1 = l_2$, there is no contribution to the total execution time from the L_2 cache. On both Itanium platforms, $l_2 = l_3$, and accesses to the L_3 cache accounts for none of the total time. To confirm that we are properly modeling cache misses, we show $\alpha_i H_i/T$ when we substitute true cache misses as measured by PAPI into the execution time model, and show the results in Figure 4.19 (*bottom*). Even with exact cache misses, the larger caches account for very little of the total execution time in our model in the case of equal line sizes.

To see the impact of strictly increasing line sizes, consider the following simple example which shows how much we can potentially speed up SpMV by increasing the line size on a hypothetical machine with two levels of cache, and $\gamma = 2$ integers per double. Let the L_2 line size be a multiple of the L_1 line size, $l_2 = \sigma l_1$, where σ is an integer power of 2. Assume we execute SpMV on a general $n \times n$ sparse matrix with k non-zeros, but no natural block structure, so that the 1×1 implementation is fastest over all block sizes. Further suppose that $k \gg n$, so that (1) we can approximate the load count of Equation (4.5) by $\text{Loads}(1,1) \approx 3k$, where we have ignored the terms for row pointers and destination vector loads, and (2) we can ignore stores. We approximate the misses by first assuming $M_i = M_{\text{lower}}^{(i)}$, and then approximating the lower bound Equation (4.6) as follows:

$$\begin{aligned} M_1 = M_{\text{lower}}^{(1)}(1,1) &\approx \frac{1}{l_1} \cdot \frac{3}{2}k, \\ M_2 = M_{\text{lower}}^{(2)}(1,1) &\approx \frac{1}{l_2} \cdot \frac{3}{2}k \approx \frac{1}{\sigma}M_1 \end{aligned}$$

Substituting these loads and misses into Equation (4.4), an approximate lower bound on

execution time T_σ to perform SpMV is then

$$\begin{aligned} T_\sigma &= \alpha_1 \text{Loads}(1,1) + (\alpha_2 - \alpha_1) M_1 + (\alpha_{\text{mem}} - \alpha_2) \frac{M_1}{\sigma} \\ &= \alpha_1 (3k) + \left[\alpha_2 - \alpha_1 + \frac{\alpha_{\text{mem}} - \alpha_2}{\sigma} \right] \cdot \frac{3k}{2l_1} \end{aligned} \tag{4.11}$$

When the line sizes are equal, $\sigma = 1$ and Equation (4.11) includes a term corresponding proportional to a full memory latency α_{mem}. As σ increases, this term goes down by $\frac{1}{\sigma}$, while the contribution from the L_2 increases toward α_2. As expected, increasing σ from 1 shifts the cost from memory to the L_2 cache.

How much faster can SpMV go as σ increases? Suppose we are somehow able to keep all the cache and memory latencies fixed while increasing the line size. This assumption is difficult to realize in practice since the latencies will depend on the relative line sizes, but is useful for bounding potential improvements. Figure 4.20 shows the speedup T_1/T_σ for each of the following three platforms: Pentium III, Pentium III-M, and Power3. Speedups are greatest on the Pentium III-M platform, which has the largest gap between α_2 and α_{mem}: increasing the L_2 line size to the next largest power of 2 ($\sigma = 2$) yields a potential 1.6× speedup over the case of equal line sizes.

The speedups shown are likely to be the maximum possible speedups, since increasing the L_2 line size will tend to increase α_2 as well. Let us instead suppose that when we double the L_2 line size we also double the L_2 latency, but keep all other latencies fixed. On the Pentium III-M, this yields a 1.47× speedup instead of a 1.6×. Although somewhat reduced compared to the more optimistic case of keeping all line sizes fixed, this speedup indicates the potential utility of maintaining strictly increasing line sizes in multi-level memory hierarchies.

4.3 Related Work

For dense matrix algorithms, a variety of sophisticated static models for selecting transformations and tuning parameters have been developed, each with the goal of providing a compiler with sufficiently precise models for selecting memory hierarchy transformations and parameters such as tile sizes [70, 130, 219, 66, 330]. However, it is difficult to apply these analyses directly to sparse matrix kernels due to the presence of indirect and irregular memory access patterns, and the strong dependence between performance and matrix structure that may only be known at run-time.

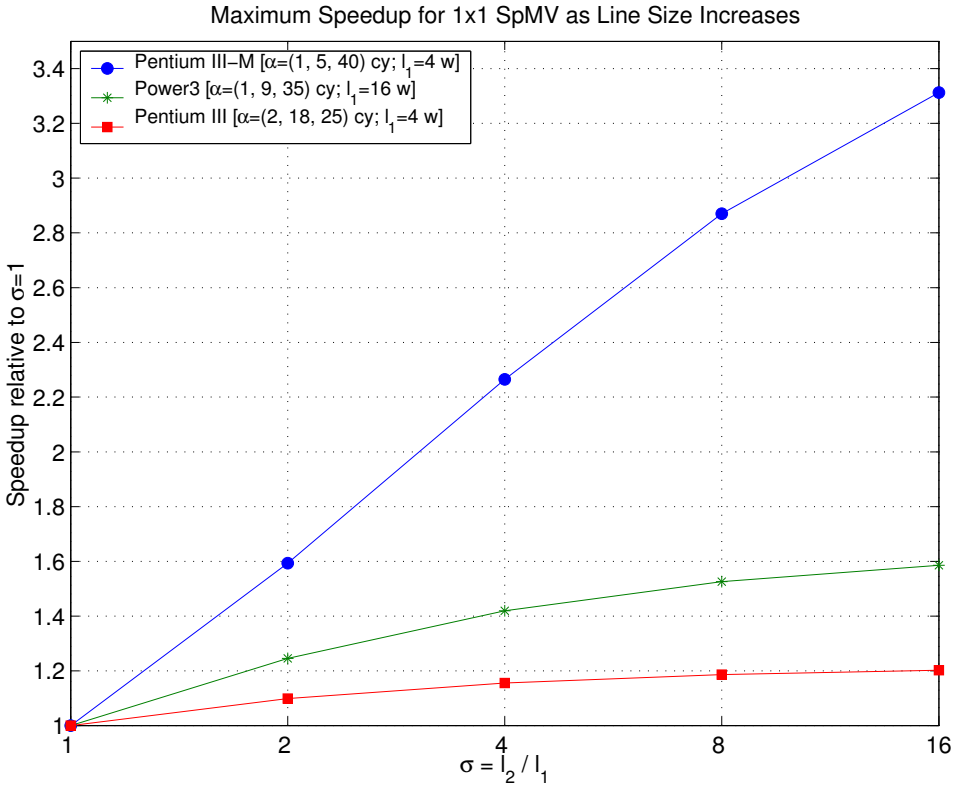

Figure 4.20: **Approximate potential speedup from increasing cache line size while keeping cache and memory latencies fixed**. We show the approximate speedup T_1/T_σ on systems with two cache levels, where the L_1 and L_2 line sizes are related by $l_2 = \sigma l_1$, and the cache and memory latencies are fixed. T_σ is given by Equation (4.11). Each curve shows the speedup using the latency parameters and L_1 cache line size given by one of the following three platforms: Pentium III, Pentium III-M, and Power3.

Nevertheless, there have been a few notable modeling efforts for SpMV. The earliest work of which we are aware is the study by Temam and Jalby [294]. They developed a sophisticated model of cache misses for SpMV, under the main assumptions of (1) square banded matrices with a uniformly random distribution of non-zeros within the band, (2) a compressed sparse row (CSR) format data structure, and (3) a machine with a single cache level. They show how cache misses vary with matrix structure (dimension, density, and bandwidth) and cache parameters (line size, associativity, and capacity). Unlike our simple model, their model includes (approximations of) conflict misses, particularly self-interference misses. Their two main conclusions are as follows. First, they find that cache line size has the greatest affect on misses, while associativity has the least. This finding

supports our decision to primarily model misses based on line size. Second, they show that self-interference misses (the largest contributor to conflict misses in their model) can be minimized by reducing the matrix bandwidth and maximizing cache capacity. Since we ignore conflict misses altogether, cache capacity was not an important factor in our model. Interestingly, this observation about self-interference misses implies (as the authors suggest) that an effective way to reduce cache misses is to block the matrix by bands; in contrast, cache blocking of sparse matrices has been implemented using rectangles in the SPARSITY system [164]. As far as we know, band blocking has yet to be tested in practice.

The model we have presented differs from the Temam and Jalby model in several ways. First, we assume multi-level memory hierarchies. Second, we are most interested not only in cache miss counts, but also in the effect of misses on *execution time*. We model time explicitly via effective cache and memory access latencies. Third, we neglect all conflict misses. The cache miss data presented in Section 4.2.2 suggests that this assumption is reasonable in practice, particularly as the cache capacity grows. Fourth, we model a blocked data structure. We are able to capture aspects of the non-zero structure through explicit modeling of the fill ratio under blocking, a departure from the uniformly random distribution assumption. Despite these differences, we view our work as largely complementary to the Temam and Jalby work. In particular, our modular model allows us to substitute different models of cache misses, which could include a version of the Temam and Jalby model, possibly adapted to different structural non-zero distributions.

Two additional studies have updated the Temam and Jalby models for the sparse matrix-multiple vector multiply (SpMM) kernel, $Y \leftarrow Y + AX$, where X, Y are dense matrices and A is a sparse matrix. SpMM has more opportunities for reuse and blocking, and these two studies consider cache-level blocking with respect to X and Y (*i.e.*, they do not explicitly reorganize A, assumed to be stored in CSR format, into cache-sized blocks as in SPARSITY's cache blocking [164]). Fraguela, *et al.*, consider analytic modeling of the SpMM kernel, and refine the conflict miss modeling of Temam and Jalby to include more accurate counts of cross-interference misses [117]. The basic assumptions about random non-zero structure and caches are the same. They show how their model can be used to predict block sizes that minimize overall cache misses under different loop orderings. Navarro, *et al.*, consider simulations of the SpMM kernel, [231]. Like DGEMM, the existence of densely stored multiple vectors in SpMM greatly increases the possibility and influence of TLB misses. They conclude that X should be reorganized into row-major storage when possible,

and study the trade-offs between cache and TLB misses under different loop orderings on a simulated DEC Alpha platform. Their work shows that in extending our models to the multiple-vector case, TLB misses will be an important component.

A related class of codes are stencil calculations, for which Leopold has developed tight lower bounds on capacity misses [207]. Investigating the full implications of their model for locality-enhancing transformations or architectures specialized for stencil operations are opportunities for future work.

Gropp, *et al.*, consider performance upper bounds modeling of a particular computational fluid dynamics code, which includes SpMV on a particular matrix as a large component [139]. They consider two types of bounds on their application. The first bound is based on the time to move just the matrix data at the rate reported by the STREAM Triad benchmark [217]. Our model bounds the STREAM benchmark as well, as shown in Table 4.1. Since the STREAM bandwidth is often less than 75% of our upper bound, and since our SpMV code achieves 75% or more of the upper bound in many cases on the same platforms, our upper bound is more likely to be a true "bound." The second bound Gropp, *et al.*, present is based on instruction issue limits, *i.e.*, by counting the number and type of all instructions produced in the compiler-generated assembly code, and bounding the time to execute them by ignoring dependencies and assuming maximum utilization of CPU functional units. They apply this bound to a flux calculation and not SpMV, and find that their bound is even more optimistic than their memory-based bound. Nevertheless, our data shows that their analysis, possibly refined to consider dependencies, could be useful in refining our SpMV bounds when the block size is small.

Heber, *et al.*, develop, study, and tune a fracture mechanics code [153] on Itanium 1. However, we are interested in tuning for matrices that come from a variety of domains and on several machine architectures. Nevertheless, their methodology for examining instruction issue limits and the output of the Intel compiler, combined with recent work on Itanium-specific tuning [81, 297, 173, 37], could shed light on how to improve instruction scheduling more generally on the Itanium processor family for SpMV and other sparse kernels.

Although we have used the Saavedra-Barrera [269] and MAPS benchmarks [282] to determine access latencies, a number of other microbenchmarks have been developed to determine cache parameters [112, 298], though these benchmarks do not appear to provide qualitatively different information from what we were able to obtain or needed for these models. Nevertheless, an interesting question is whether new microbenchmarks could be

implemented that assess how and to what extent other aspects of memory system design (*e.g.*, pipelining and buffering to support multiple outstanding misses [328]) contribute to performance.

4.4 Summary

The main contribution of this chapter is a performance upper bounds model specialized to register blocked SpMV. This model is based on (1) characterizing the machine by the visible latency to access data at each level of the memory hierarchy, and by the cache line sizes, and (2) lower bounds on cache misses that account for matrix structure.

Intuitively, the time to perform SpMV is dominated by the time to read the matrix. Indeed, our count of loads and lower bound on cache misses are very similar to the expression of the matrix volume $V_{rc}(A)$ given by Equation (3.1), owing to the dominant cost of reading A. In this sense, the size of the sparse matrix data structure is a fundamental algorithmic limit to SpMV. Thus, we can view the problem of data structure selection as a data compression problem, possibly opening new lines of attack for future work.

The proximity of SPARSITY-generated code, when compiled with vendor compilers, to the performance upper bound on matrices from FEM applications indicates that additional low-level tuning will yield limited gains, at least for matrices which have natural uniform dense block structure. Viewed another way, these bounds are good predictors of performance achievble in practice on a variety of architectures. We show that a simple relationship exists between the characterization of the machine using our model to achieved fraction of machine peak. However, our use of measured effective latency parameters α_i and α_{mem} obscures which particular aspects of memory system and processor design keep these parameters small. There is a clear opportunity to try to characterize more specifically what aspects of machine design yield good SpMV performance.

One aspect of machine design which is prevalent in practice (on 5 of the 8 evaluation platforms) but a performance penalty in our model is the use of equal line sizes between two levels of the memory hierarchy. A simple consequence of the our performance bounds model is the importance of strictly increasing line sizes for register blocked SpMV. Gradual refinements of the model to incorporate additional architectural features may yield additional insights, in the spirit of similar attempts by Temam and Jalby [294].

Other possible refinements to our model include (1) better modeling of conflict

misses and the spatial locality inherent in a given matrix structure, and (2) explicit modeling of instruction issue limitations, in the spirit of Gropp, *et al.* [139]. Both refinements could lead to tighter upper bounds for matrices like Matrices 18–44 which lack easily exploitable block structure, as well as insights into how to improve low-level scheduling and tuning in both software (the compiler) and hardware (through additional or new CPU resources).

Chapter 5

Advanced Data Structures for Sparse Matrix-Vector Multiply

Contents

We propose two techniques in this chapter to extend register blocking's performance improvements for sparse matrix-vector multiply (SpMV) and potential storage savings to more complex matrix non-zero patterns:

1. To handle matrices composed of multiple, irregularly aligned rectangular blocks, we present in Section 5.1 a technique in which we split the matrix A into a sum $A =$

$A_1 + A_2 + \ldots + A_s$, where each term is stored in unaligned block compressed sparse row (UBCSR) format. Matrices from finite element method (FEM) models of complex structures lead to this kind of structure, and the strict alignment imposed by register blocking (as implemented in SPARSITY and reviewed in Chapter 3) typically leads to extra work from explicitly filled-in zeros. Combining splitting with the UBCSR data structure attempts to reduce this extra work. The main matrix- and machine-specific tuning parameters in the split UBCSR implementation are the number of splittings s and the block size for each term A_i. We show speedups that can be as high as 2.1× over not blocking at all, and as high as 1.8× over the standard implementation of register blocking described in Chapter 3. Even when performance does not improve, storage can be significantly reduced.

2. For matrices with diagonal substructure, including complex compositions of non-zero diagonal runs, we propose row segmented diagonal (RSDIAG) format in Section 5.2. The main matrix- and machine-specific tuning parameter is an unrolling depth. We show that implementations based on this format can lead to speedups of 2× or more for SpMV, compared to a compressed sparse row (CSR) format implementation.

These results complement the body of existing techniques developed in the context of the SPARSITY system, including combinations of symmetry [204], cache blocking [235, 165, 164], multiplication by multiple vectors [204, 165, 164], and reordering to create block structure [228]. Section 5.3 summarizes the kinds of performance improvements for SpMV that one might expect from all of these techniques, including those explored in this chapter.

The experimental data presented in this chapter was collected on the following subset of the 8 platforms listed in Appendix B: Ultra 2i, Pentium III-M, Power4, and Itanium 2. For each technique, we present results for a subset of the 44 SPARSITY benchmark suite, as well as a number of supplemental matrices described in each section. (All matrices, including sources when available, are listed in Appendix B.)

5.1 Splitting variable-block matrices

Chapters 3–4 note differences in the structure of finite element method (FEM) Matrices 10–17 compared to FEM Matrices 2–9, making typical speedups from uniformly aligned register blocking on the former class of matrices lower than those on the latter. Here, we distinguish

the structure of these two classes by a characterization of the matrix block structure based on variable block row (VBR) format, as discussed in Chapter 2. In VBR, the matrix block structure is defined by logically partitioning rows into block rows and columns into block columns. When Matrices 10–17 start in VBR, we find that they differ from Matrices 2–9 primarily in two ways:

- **Unaligned blocks**: The register blocking optimization as proposed and implemented in SPARSITY (and reviewed in Chapter 3) assumes that each $r \times c$ block is uniformly aligned: if the upper-leftmost element of the block is at position (i, j), then register blocking assumes $i \bmod r = j \bmod c = 0$. When Matrices 12 and 13 are stored in VBR format, we find that most non-zeros are contained in blocks of the same size, but $i \bmod r$ and $j \bmod c$ are distributed uniformly over all possible values up to $r - 1$ and $c - 1$, respectively.

- **Mixtures of "natural" block sizes**: Matrices 10, 15, and 17 possess a mix of block sizes, at least when viewed in VBR format.

(We treat Matrix 11, which contains a mix of blocks and diagonals, in Section 5.2; Matrices 14 and 16 are eliminated on our evaluation platforms due to their small size.)

Unaligned block rows can be handled by simply augmenting the usual block compressed sparse row (BCSR) format with an additional array of row indices `Arowind` such that `Arowind`$[I]$ contains the starting index of block row I. We refer to this data structure as unaligned block compressed sparse row (UBCSR) format. An example of the 2×3 UBCSR routine appears in Figure 5.1, where we use the same line numbering scheme shown for the 2×3 BCSR example of Figure 3.1. The two implementations differ by only one line—line S2 of Figure 3.1 has been replaced by lines S2a and S2b in Figure 5.1.

To handle multiple blocks sizes, we can compute the distribution of work (*i.e.*, non-zero elements) over block sizes from the VBR data structure, and then split the matrix A into a sum of matrices $A = A_1 + \ldots + A_s$, where each term A_l holds all non-zeros of a particular block size and is stored in UBCSR format. This section considers structurally disjoint splittings (*i.e.*, A_i and A_j have no non-zero positions in common when $i \neq j$) with up to 4-way splittings (*i.e.*, $2 \leq s \leq 4$).

Section 5.1.1 below provides a motivating example for UBCSR format, and discusses the block size distribution characteristics of the augmented matrix test set used in

```
S0  void sparse_mvm_ubcsr_2x3( int M, int n,
        const double* Aval,
        const int* Arowind, const int* Aind, const int* Aptr,
        const double* x, double* y )
    {
        int I;
S1      for( I = 0; I < M; I++, y += 2 ) { // loop over block rows
S2a         int i = Arowind[I];  // block row starting index
S2b         register double y0 = y[i+0], y1 = y[i+1];
            int jj;

            // loop over non-zero blocks
S3a         for( jj = Aptr[I]; jj < Aptr[I+1]; jj++, Aval += 6 ) {
S3b             int j = Aind[jj];
S4a             register double x0 = x[j], x1 = x[j+1], x2 = x[j+2];

S4b             y0 += Aval[0]*x0; y1 += Aval[3]*x0;
S4c             y0 += Aval[1]*x1; y1 += Aval[4]*x1;
S4d             y0 += Aval[2]*x2; y1 += Aval[5]*x2;
            }
S5          y[0] = y0;  y[1] = y1;
        }
    }
```

Figure 5.1: **Example C implementations of matrix-vector multiply for dense and sparse UBCSR matrices**. Here, `M` is the number of block rows stored and `n` is the number of matrix columns. Multiplication by each block is fully unrolled (lines S4b–S4d). Only lines S2a–S2b differ from the BCSR code of Figure 3.1.

this section. (In addition to Matrices 10–17, we use 5 more test matrices with irregular block structure and/or irregular alignments that also arise in FEM applications.) We discuss a variation on conversion to VBR format that allows for some fill in Section 5.1.2. As in the case of register blocking, fill can allow for some additional compression of the overall data structure. We further specify precisely how we select and convert a given matrix to a split format in Section 5.1.3. This discussion is necessary since there may be many possible ways to split an arbitrary matrix. We present experimental results in Section 5.1.4 which show that performance approaching that of Matrices 2–9 is possible, and that an important by-product of the split formulation is a significant reduction in matrix storage.

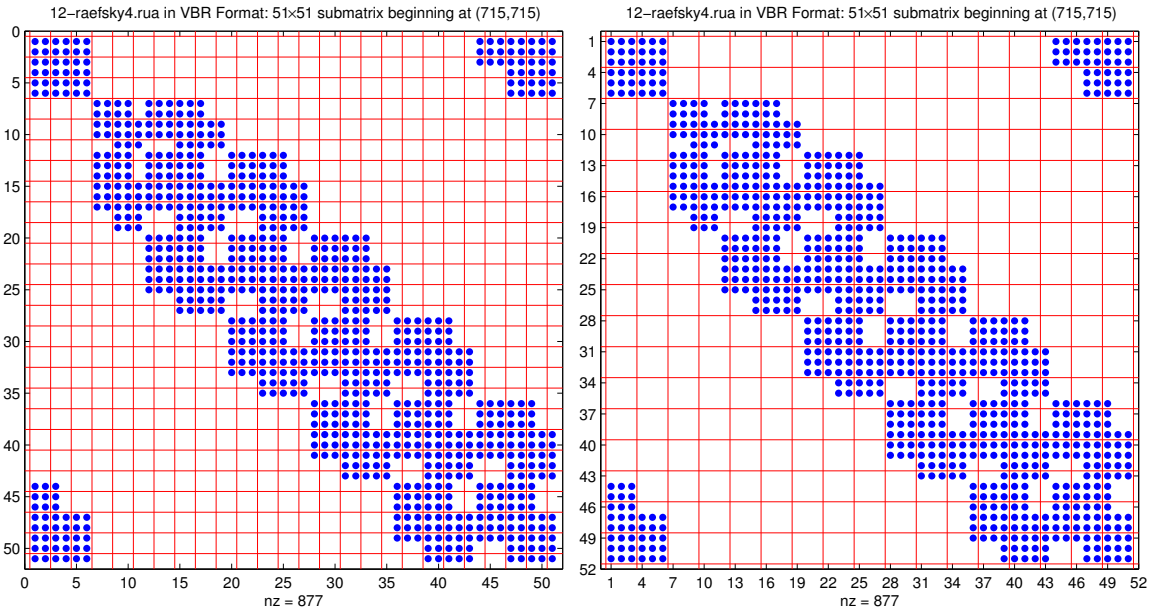

Figure 5.2: **Uniform block sizes can inadequately capture "natural" block structure: Matrix 12-`raefsky4`.** We show the 51×51 submatrix beginning at element (715, 715) of Matrix 12-`raefsky4` when uniformly aligned 2×2 (*left*) and 3×3 (*right*) logical grids have been imposed, as would occur with register blocking. These grids do not precisely capture the true non-zero structure, leading to fill ratios of 1.23 for 2×2 blocking, and 1.46 for 3×3 blocking.

5.1.1 Test matrices and a motivating example

Figure 5.2 shows the 51×51 submatrix beginning at the (715, 715) entry of Matrix 12. In the left plot, we superimpose the logical grid of 3×3 cells that would be imposed under register blocking, and in the right plot we superimpose the grid of 2×2 cells, where the corresponding fill ratios are 1.46 and 1.24, respectively. These blocks sizes are optimal on some platforms (see Chapter 3). Although there is abundant block structure, uniform blocking does not perfectly capture it.

Table 5.1 compares the best observed performance for Matrix 12 (column 3) to both a reference implementation using compressed sparse row (CSR) format storage (column 6) and the best observed performance on Matrices 2–9 (column 2) on 7 of the evaluation platforms used in Chapter 3 (column 1). We also show the best block size and fill ratio for Matrix 12 (columns 4 and 5). We observe speedups over the reference in all cases. However, if we compute the fraction of the best performance on Matrices 2–9 (by dividing column 3 by column 2) and then take the median over all platforms, we find the median fraction to be only 69%. Since there are evidently abundant blocks, this motivates us to ask whether

Platform	Matrices 2–9 Maximum Mflop/s	Matrix 12-raefsky4 Best Mflop/s	$r \times c$	Fill	1×1 Mflop/s
Ultra 2i	57	38	2×2	1.24	33
Ultra 3	109	61	2×1	1.13	56
Pentium III	90	63	3×3	1.46	40
Pentium III-M	120	83	2×2	1.24	68
Power3	168	130	1×1	1.00	130
Itanium 1	214	172	3×1	1.24	140
Itanium 2	1122	774	4×2	1.48	276

Table 5.1: **Best performance and block sizes under register blocking: Matrix 12-raefsky4.** Summary of the best performance under register blocking (column 3), the best register block size $r_{\text{opt}} \times c_{\text{opt}}$ (column 4), and the fill ratio at $r_{\text{opt}} \times c_{\text{opt}}$ (column 5) for Matrix 12-raefsky4. This example shows the typical gap between performance achieved on Matrices 10–17 and the best performance on Matrices 2–9 (column 1). This data is taken from Chapter 4.

we can achieve higher fractions by better exploiting the actual block structure.

VBR serves as a useful an intermediate format for understanding the block structure. (See Chapter 2 for a detailed description of VBR.) Support for VBR is included in a number of sparse matrix libraries, including SPARSKIT [267], and the NIST Sparse BLAS [258]. The main drawback to using VBR is that it is difficult to implement efficiently in practice. The innermost loops of the typical VBR implementation carry out multiplication by an $r \times c$ block. However, this block multiply cannot be unrolled in the same way as BCSR because the column block size c may change from block to block within a block row. (See Chapter 2.1.4 for a more detailed discussion of this issue.)

Nevertheless, we can quickly characterize the block structure of a matrix in VBR format by scanning the data structure to determine the distribution of non-zeros over block sizes. We show the same 51×51 submatrix of Matrix 12 as it would be blocked in VBR format in Figure 5.3 (*top*). We used a routine from the SPARSKIT library to convert the matrix from CSR to VBR format. This routine partitions the rows by looping over rows in order, starting at the first row, and placing rows with identical non-zero structure in the same block. The same procedure is used to partition the columns. The distribution of non-zeros can be obtained in one pass over the resulting VBR data structure. For Matrix 12, the maximum block size in VBR format turns out to be 3×3. In Figure 5.3 (*bottom-left*), we show the fraction of non-zeros contained in all blocks of a given size $r \times c$, where

$1 \leq r, c \leq 3$. Each square represents a value of $r \times c$ shaded by the fraction of non-zeros for which it accounts, and labeled by that fraction. A label of '0' indicates that the fraction is zero when rounded to two digits, but there is at least 1 block at the given size. For Matrix 12, 96% of the non-zeros occur in 3×3 blocks.

Although the matrix is dominated by 3×3 blocks, these blocks are not uniformly aligned on row boundaries as assumed by BCSR (and register blocking). In Figure 5.3 (*bottom-right*), we show the distributions of $i \bmod r$ and $j \bmod c$, where (i, j) is the starting position in A of each 3×3 block. The first row index of a given block row can start on any alignment, with 26% of block rows having $i \bmod r = 1$, and the remainder split equally between $i \bmod r = 0$ and 2. This observation motivates the use of UBCSR.

When evaluting the UBCSR data structure and splitting for variable block sizes, we augment test matrices Matrices 10–17 with 5 additional matrices from FEM applications. We summarize the variable block test set in Table 5.2. This table includes a short list of dominant block sizes after conversion to VBR format, along with the fraction of non-zeros for which those block sizes account. The reader may assume that the dominant block size is also irregularly aligned except in the case of Matrix 15. For more information on the distribution of non-zeros and block size alignments, refer to Appendix F.

5.1.2 Altering the non-zero distribution of blocks using fill

The SPARSKIT CSR-to-VBR conversion routine only groups rows (or columns) when the non-zero patterns between rows (columns) matches exactly. However, this convention can be too strict on some matrices in which it would be profitable to fill in zeros, just as with register blocking. Below, we discuss a simple variation on the SPARSKIT routine that allows us to create a partitioning based on a measure of similarity between rows (columns).

First, consider the example of Matrix 13. According to Table 5.2, this matrix has relatively few block sizes larger than the trivial unit block size (1×1). However, the 52×52 submatrix of Matrix 13, depicted in Figure 5.4 shows that a few isolated zero elements break up potentially larger blocks.

Although there are many ways to account for cases like this one, we introduce a simple change to the conversion routine based on the following measure of similarity between rows (columns). Let u and v be two sparse column vectors whose non-zero elements are equal to 1. Let k_u and k_v be the number of non-zeros in u and v, respectively. Let $S(u, v)$

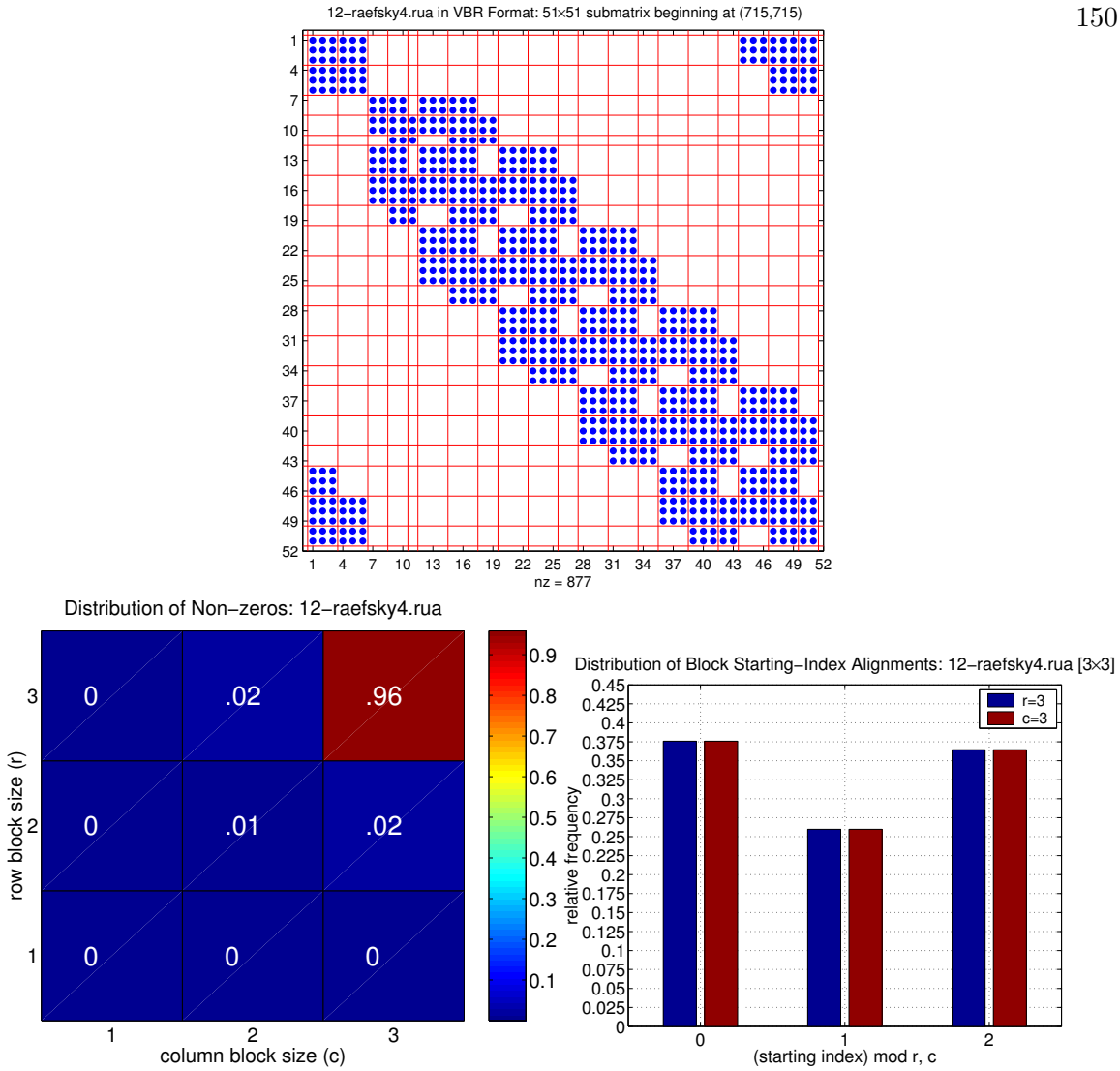

Figure 5.3: **Logical grid (block partitioning) after greedy conversion to variable block row (VBR) format: Matrix 12-`raefsky4`.** (*Top*) We show the logical block partitioning after conversion to VBR format using a greedy algorithm. (*Bottom-left*) Approximately 96% of the non-zero blocks are 3×3. (*Bottom-right*) Let (i, j) be the starting row and column index of each 3×3 block. We see that 37.5% of these blocks have $i \bmod 3 = 0$, 26% have $i \bmod 3 = 1$, and the remaining 36.5% have $i \bmod 3 = 2$. The starting column indices follow the same distribution, since the matrix is structurally (though not numerically) symmetric.

be the following measure of similarity between u and v:

$$S(u, v) = \frac{u^T \cdot v}{\max(k_u, k_v)} \tag{5.1}$$

This function is symmetric with respect to u and v, has a minimum value of 0 when u and v have no non-zeros in common, and a maximum value of 1 when u and v are identical.

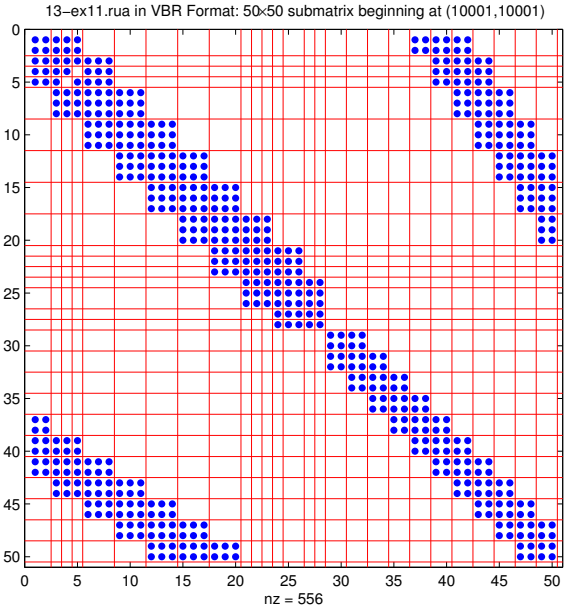

Figure 5.4: **Logical grid (block partitioning) after greedy conversion to VBR format: Matrix 13-ex11.** We show a 50×50 submatrix beginning at position (10001, 10001) in Matrix 13-ex11. The existence of explicit zero entries like those shown in the upper-left corner in positions (4,5) and (5,4) "break-up" the following contiguous blocks: one beginning at (39,3) and ending at (44,5), and another extending from (3,39) to (5,44).

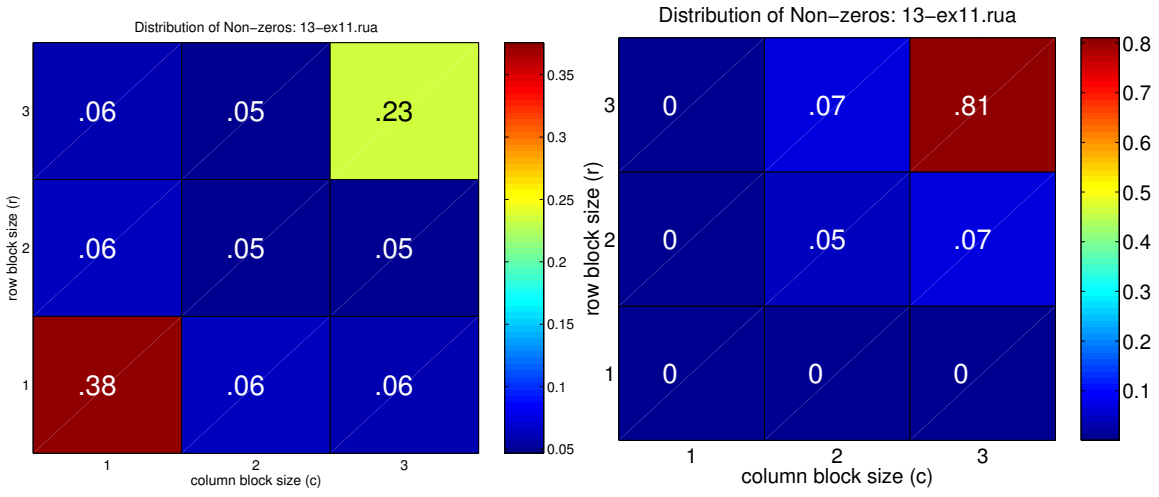

Figure 5.5: **Distribution of non-zeros over block sizes in variable block row format, without and with thresholding: Matrix 13-ex11.** (*Left*) Distribution of non-zeros without thresholding, *i.e.*, $\theta = 1$. Under the block partitioning shown in Figure 5.4, 23% of non-zeros are contained in 3×3 blocks, and 38% in 1×1 blocks. (*Right*) Distribution with thresholding at $\theta = 0.7$. The fraction of non-zeros in 3×3 blocks increases to 81%. The fill ratio is 1.01.

#	Matrix	Dimension	No. of Non-zeros	Dominant block sizes (% of non-zeros)
10	`ct20stif` Engine block	52329	2698463	6×6 (39%) 3×3 (15%)
12	`raefsky4` Buckling problem	19779	1328611	3×3 (96%)
13	`ex11` 3D flow	16614	1096948	1×1 (38%) 3×3 (23%)
15	`vavasis3` 2D partial diff. equ.	41092	1683902	2×1 (81%) 2×2 (19%)
17	`rim` Fluid mechanics problem	22560	1014951	1×1 (75%) 3×1 (12%)
A	`bmw7st_1` Car body analysis	141347	7339667	6×6 (82%)
B	`cop20k_m` Accelerator cavity design	121192	4826864	2×1 (26%) 1×2 (26%) 1×1 (26%) 2×2 (22%)
C	`pwtk` Pressurized wind tunnel	217918	11634424	6×6 (94%)
D	`rma10` Charleston Harbor model	46835	2374001	2×2 (17%) 3×2 (15%) 2×3 (15%) 4×2 (9%) 2×4 (9%)
E	`s3dkq4m2` Cylindrical shell	90449	4820891	6×6 (99%)

Table 5.2: **Variable block test matrices**. Matrices with variable block structure and/or non-uniform alignment: Matrices 10–17 and supplemental matrices Matrices A–E, all arising in FEM problems. Dominant block sizes $r\times c$ are shown in the last column, along with the percentage of non-zeros contained within $r\times c$ blocks shown in parentheses. See also Appendix F.

Based on this similarity measure, we use the following algorithm to compute a block row partitioning of an $m\times n$ sparse matrix A. We assume that A is a pattern matrix, *i.e.*, all non-zero entries are equal to 1. This partitioning is expressed below as a set of lists of the rows of A, where rows in each list are taken to be in the same block row. On input, the caller provides a threshold, θ, specifying the minimum value of $S(u, v)$ at which two rows may be considered as belonging to the same block row. The procedure examines

rows sequentially, starting at row 0, and maintains a list of all row indices `Cur_block` in the current block row. Each row is compared to the first row of the current block, and if their similarity exceeds θ, the row is added to the current block row. Otherwise, the procedure starts a new block row.

```
    Algorithm PartitionRows( A, θ )
1       Cur_block ← [0] /* Ordered list of row indices in current block */
2       All_blocks ← ∅
3       for i = 1 to m − 1 do /* Loop over rows */
4           Let u ← row Cur_block[0] of A /* First row in current block */
5           Let v ← row i of A
6           if S(u, v) ≥ θ then
7               Append i onto Cur_block
            else
8               All_blocks ← All_blocks ∪ Cur_block
9               Cur_block ← [i]
10      All_blocks ← All_blocks ∪ Cur_block
11      return All_blocks
```

We may partition the columns using a similar procedure. However, all of the matrices in Table 5.2 are structurally (but not numerically) symmetric, so the row partition can be used as a column partition. The SPARSKIT CSR-to-VBR routine can take these row and column partitions as inputs, and returns A in VBR format. The conversion routine fills in explicit zeros to make the blocks conform to the partitions.

When we partition Matrix 13 using Algorithm **PartitionRows** and $\theta = 0.7$, the distribution shifts so that 3×3 blocks contain 81% of all stored values (including filled in zeros), instead of just 23% when $\theta = 1$. The fill ratio (stored values including filled in zeros divided by true number of non-zeros) is 1.01 at $\theta = 0.7$. More opportunities for blocking become available at the cost of a 1% increase in flops.

To limit the number of experiments in the subsequent Section 5.1.4, we consider only two values: $\theta = 1$ ("exact match" partitioning) and $\theta = \theta_{\min}$, chosen as follows. For all $\theta \in \Theta = \{0.5, 0.55, 0.6, \ldots, 1.0\}$, we compute the non-zero distribution over block sizes after conversion to VBR format. Denote the block sizes by $r_1 \times c_1$, $r_2 \times c_2$, $\ldots$, $r_t \times c_t$. Consider a splitting $A = A_1 + A_2 + \ldots + A_t$ at θ, where each term A_i contains only the non-zeros

contained in block sizes that are exactly $r_i \times c_i$, stored in UBCSR format. Let $\theta_{\text{min}} \in \Theta$ be the threshold that *minimizes* the total data structure size needed to store all A_i under this splitting. For each of the matrices in Table 5.2, we show the non-zero distributions over block sizes at $\theta = 1$ and $\theta = \theta_{\text{min}}$ in Appendix F.

5.1.3 Choosing a splitting

The split implementations upon which we base our conclusions are selected by the following search procedure. This procedure is not intended for practical use at run-time; instead, we use it simply to select a reasonable implementation of variable block splitting that we can then compare to the best register blocked implementation.

For each of the thresholds θ_{min} and 1 (see Section 5.1.2), we convert the input matrix A into VBR format and we determine the top 3 block sizes accounting for the largest fraction of matrix non-zeros. We then measure the performance of all possible s-way splittings, as computed by procedure **Split** in Section 5.1.3, based on the factors of these block sizes. This section presents data corresponding to the fastest implementation found. We restrict s to $2 \leq s \leq 4$, and force the last term A_s to be stored in CSR format (*i.e.*, to hold 1×1 blocks). However, we allow A_s to have no elements if a particular splitting leads to no 1×1 blocks. For the matrices in Table 5.2 and the four evaluation platforms, we show the best performance and the corresponding split configuration in Tables G.1–G.4.

For example, suppose the top 3 block sizes are 2×2, 3×3, and 8×1. Then the set of all factors dividing the row block size are $R = \{1, 2, 3, 4, 8\}$, and the column factors are $C = \{1, 2, 3\}$. Denote the set of all block sizes by $B = R \times C - \{(1, 1)\}$. For an s-way splitting, we try all $\binom{|B|}{s-1}$ subsets of B of size $s - 1$, and the A_s term is taken to contain 1×1 blocks.

Below, we describe the greedy procedure we use to convert a matrix A to split UBCSR format, given a request to build an s-way splitting of the form $A = A_1 + \ldots + A_s$ using the block sizes, $r_1 \times c_1$, $r_2 \times c_2$, $\ldots$, $r_s \times c_s$.

We first define a procedure **SplitOnce**(A, θ, r, c) which converts A to VBR format (at threshold θ), greedily extracts $r \times c$ blocks based on the VBR structure, returning a matrix B consisting entirely of $r \times c$ blocks and a second matrix $A - B$ containing all "leftover" elements from A.

Algorithm **SplitOnce**(A, θ, r, c)

1 Let $V \leftarrow A$ converted to VBR format at threshold θ
2 Let $B \leftarrow$ empty matrix
3 **foreach** block row I in V, in increasing order of row index **do**
4 **foreach** block b in I of size at least $r \times c$, in increasing order of column index **do**
5 Convert block b into as many non-overlapping but adjacent $r \times c$ blocks as possible, with the first block aligned at the upper left corner of b
6 Add these blocks to B
7 **return** B in UBCSR format, $A - B$ in CSR format

This procedure does not extract *exact* $r \times c$ blocks, but rather extracts as many non-overlapping $r \times c$ blocks as possible from any block of size at least $r \times c$ (lines 4–5).

The procedure to build a split representation of A repeatedly calls **SplitOnce**:

Algorithm **Split**$(A, \theta, r_1, c_1, \ldots, r_s, c_s)$

1 Let $A_0 \leftarrow A$
2 **for** $i = 1$ **to** $s - 1$ **do**
3 $A_i, A_0 \leftarrow$ **SplitOnce**(A_0, θ, r_i, c_i)
4 $A_s \leftarrow A_0$
5 **return** $A_1, \ldots, A_s$

Because of the way in which blocks are extracted by **SplitOnce**, the order in which the block sizes are specified to **Split** matters. For a given list of block sizes, we call **Split** on all permutations, except that $r_s \times c_s$ is always chosen to be 1×1. This procedures also keeps θ fixed over all calls to **SplitOnce**, though in principle one could use a different threshold at each call.

The search procedure is not intended for practical execution at run-time, owing the cost of conversion. For instance, the time to execute **SplitOnce** once is roughly comparable in cost to the conversion cost observed for BCSR conversion—between 5–31 reference sparse matrix-vector multiply (SpMV)s, as discussed in Chapter 3. Developing heuristics to select a splitting, in the spirit of Chapter 3, is an opportunity for future work.

5.1.4 Experimental results

We show that performance over register blocking often improves when we split the matrix according to the distribution of blocks obtained after conversion to VBR format and use the UBCSR data structure. Even when performance does not improve significantly, we are generally able to reduce the overall storage.

The top plots of Figures 5.6–5.9 compare the performance of the following implementations, for each platform and matrix listed in Table 5.2. (For each platform, matrices that fit within the largest cache are omitted.)

- **Best register blocking implementation on a dense matrix in sparse format** (black hollow square): Best performance shown in the register profile for the corresponding platform (Figures 3.3–3.6).

- **Median, minimum, and maximum register blocking performance on Matrices 2–9** (median by a black hollow circle, maximum by a black solid diamond, and minimum by a black solid downward-pointing triangle): For Matrices 2–9, consider the best performance observed after an exhaustive search (blue solid circles shown in Figures 3.12–3.15). We show the median, minimum, and maximum of these values.

- **Splitting and UBCSR storage** (red solid squares): Performance of SpMV when A is split into $A = A_1 + A_2 + \ldots + A_s$, where $2 \leq s \leq 4$. All terms are stored in UBCSR, except for A_s which is stored in CSR format. The implementation for which we report data is the best over a limited search, as described in Section 5.1.3. We follow the same convention of excluding flops by filled in zeros when reporting Mflop/s.

- **Fastest and slowest component under splitting** (blue triangle and dot): We measure the raw performance of executing SpMV for each component A_i. We show the fastest component by a blue triangle, the slowest by a blue dot, and the two components are connected by a vertical dash-dot line. The purpose of including these points is to see (indirectly) to what extent the fastest and slowest component of each splitting contributes to overall performance.

- **Register blocking** (green dots): Best performance with uniformly aligned register blocking, over all $r \times c$ block sizes.

- **Reference** (black asterisks): Performance in CSR format.

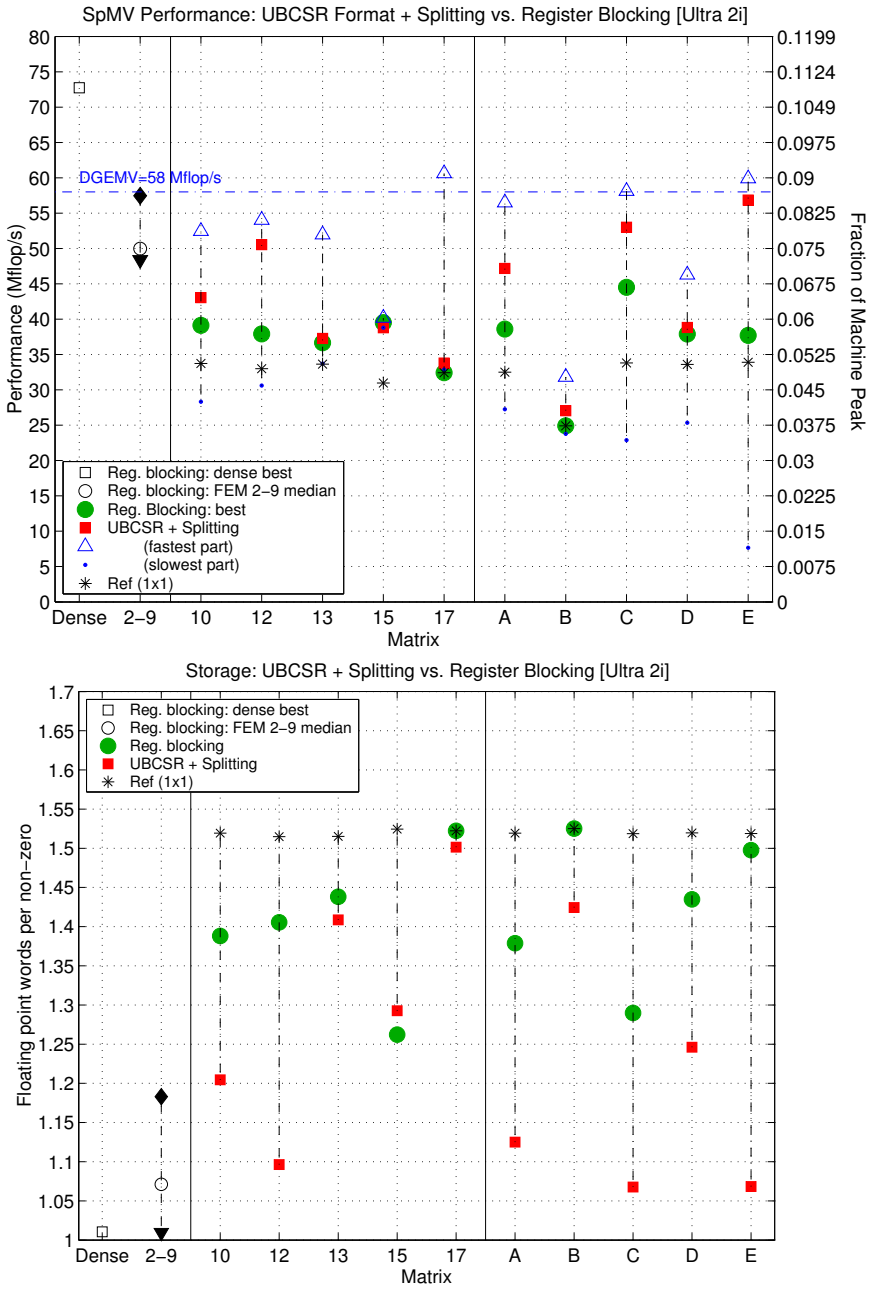

Figure 5.6: **Performance and storage for variable block matrices: Ultra 2i**. (*Top*) Performance, Mflop/s. (*Bottom*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table G.1.

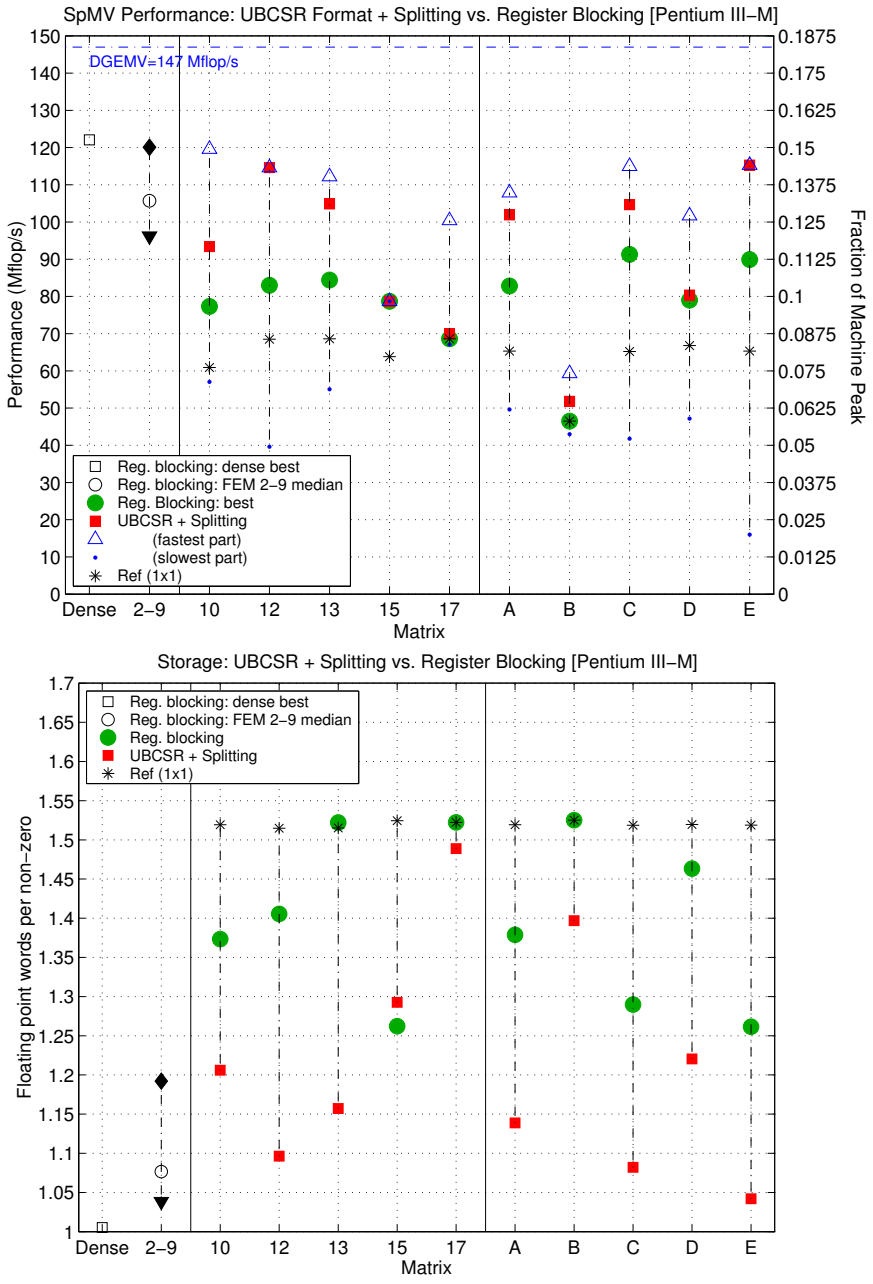

Figure 5.7: **Performance and storage for variable block matrices: Pentium III-M.** (*Top*) Performance, Mflop/s. (*Bottom*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table G.2.

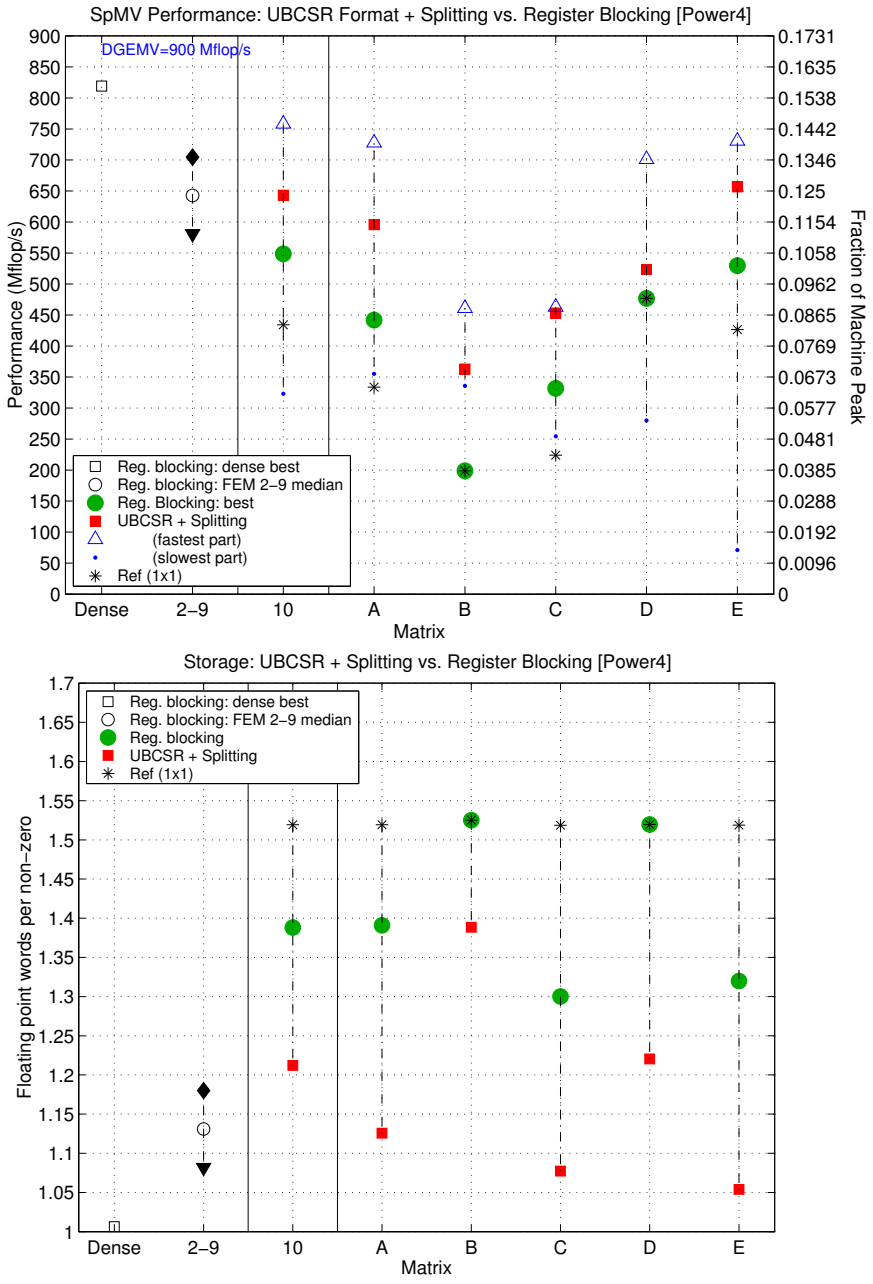

Figure 5.8: **Performance and storage for variable block matrices: Power4.** (*Top*) Performance, Mflop/s. (*Bottom*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table G.3.

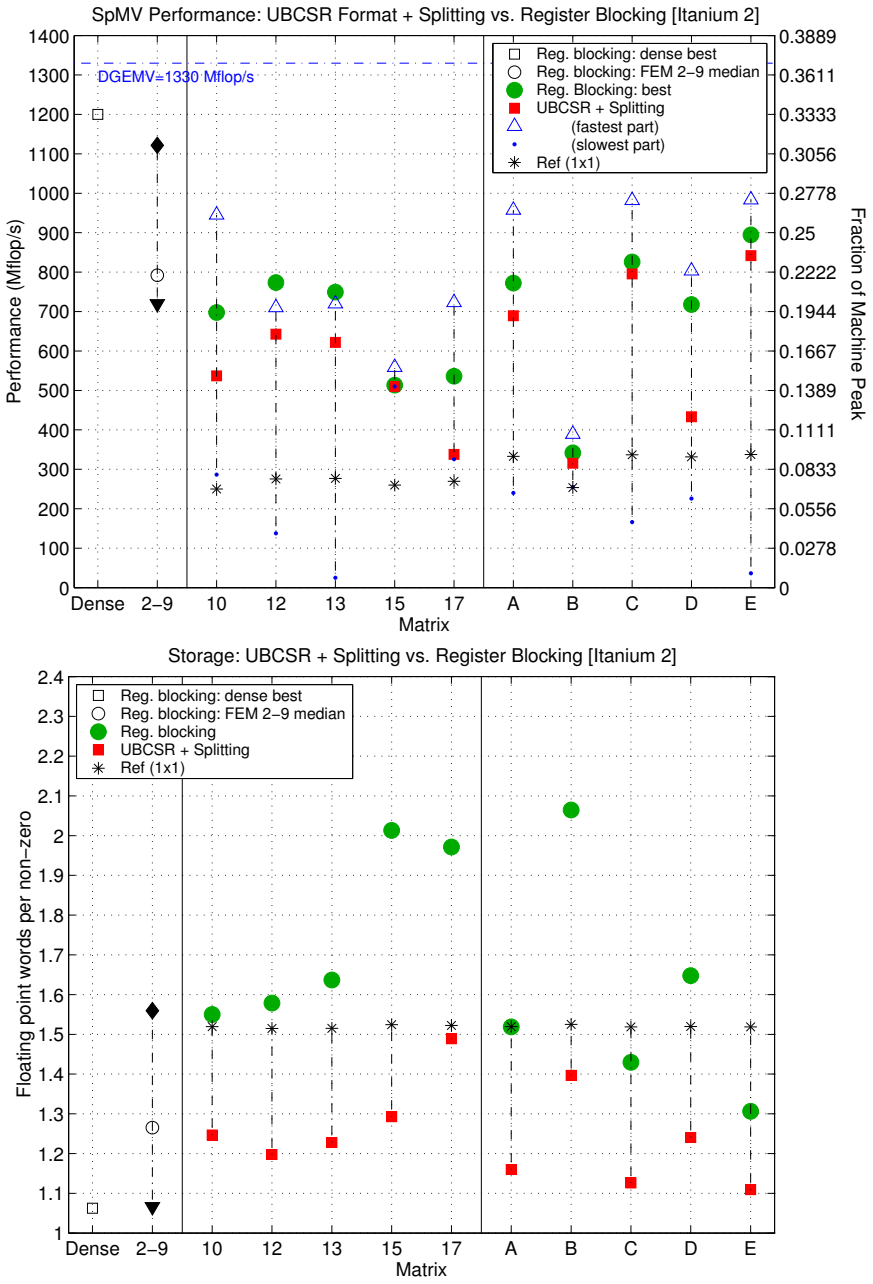

Figure 5.9: **Performance and storage for variable block matrices: Itanium 2**. (*Top*) Performance, Mflop/s. (*Bottom*) Storage, in doubles per ideal non-zero. For additional data on the block sizes used, see Table G.4.

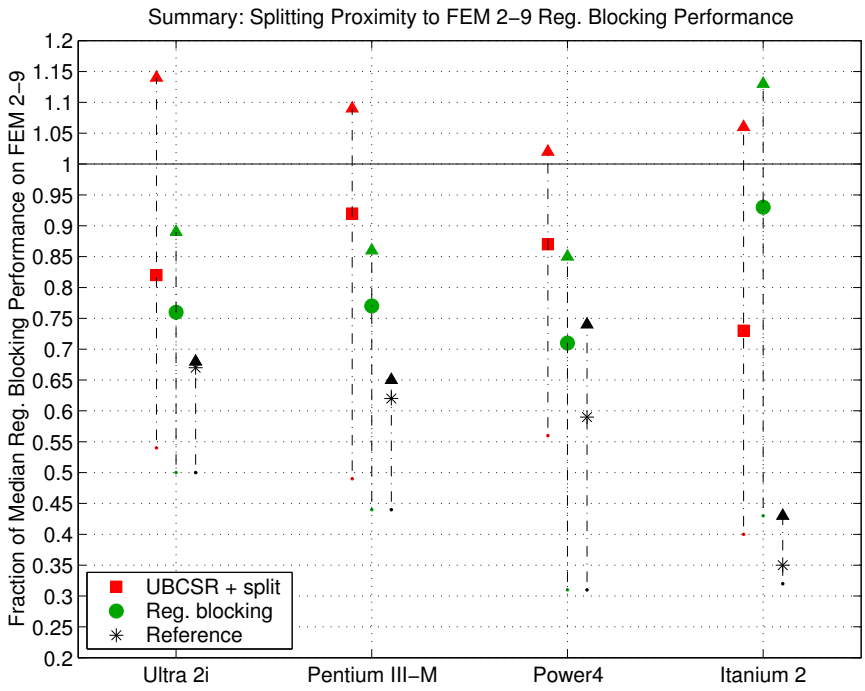

Figure 5.10: **Fraction of median register blocking performance over Matrices 2–9**.

For reference, we also show in the top plots of Figures 5.6–5.9 the performance of tuned dense matrix-vector multiply (DGEMV) on a large (out-of-cache) matrix, shown by a blue horizontal dash-dot line (also labeled by performance in Mflop/s). In the bottom plots of Figures 5.6–5.9, we show the total size (in doubles) of the data structure normalized by the number of true non-zeros (*i.e.*, excluding fill).

We summarize the main observations as follows:

1. *By relaxing the block row alignment using UBCSR storage, it is possible to approach the performance seen on Matrices 2–9.* A single block size and irregular alignment characterize the structure of Matrices 12, 13, A, C, and E (Table 5.2). The best absolute performance under splitting within a given platform is typically seen on these matrices. Furthermore, this performance is roughly comparable to median register blocking performance taken over Matrices 2–9 on the same platform. These two observations suggest that the overhead of the additional row indices in UBCSR is small. We summarize how closely the split implementations approach the performance observed for Matrices 2–9 in Figure 5.10, discussed in more detail below.

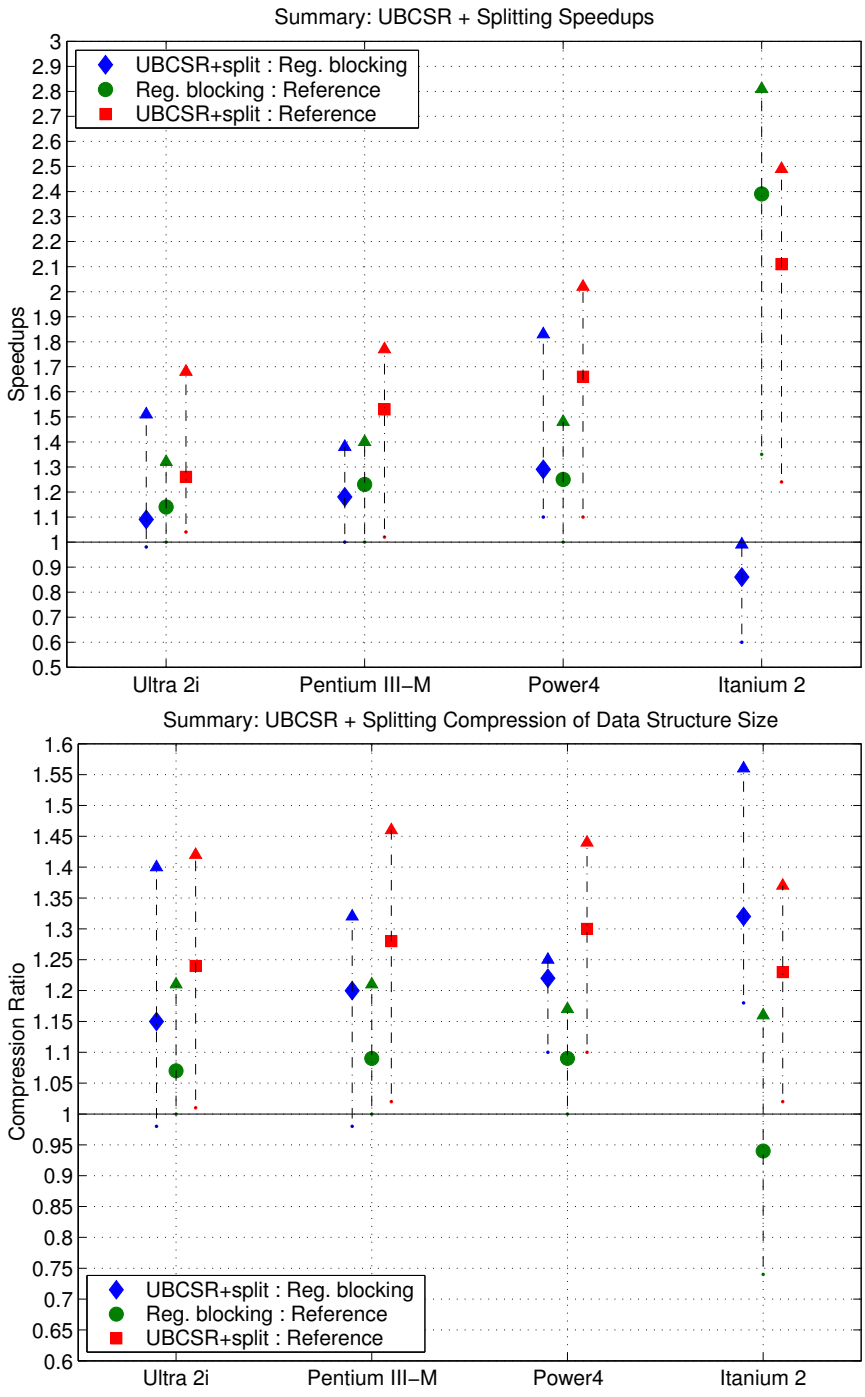

Figure 5.11: **Speedups and compression ratios after splitting + UBCSR storage, compared to register blocking.** (*Top*) We compare the following speedups: UBCSR storage + splitting over register blocking, register blocking over the reference CSR implementation, and UBCSR storage + splitting over the reference. For each platform, we show minimum, median, and maximum speedups for each pair. (*Bottom*) We compare the compression ratios for the same three pairs of implementations.

2. *Median speedups, taken over the matrices in Table 5.2 and measured relative to the reference performance, range from* 1.26× *(Ultra 2i) up to* 2.1× *(Itanium 2).* Furthermore, splitting can be up to 1.8× faster than register blocking alone. We summarize the minimum, median, and maximum speedups in Figure 5.11 (*top*).

3. *Splitting can lead to a significant reduction in total matrix storage.* The compression ratio of splitting over the reference is the size of the reference (CSR) data structure divided by the size of the split+UBCSR data structure. The median compression ratios of splitting over the reference, taken over the matrices in Table 5.2, are between 1.26–1.3×. Compared to register blocking, the compression ratios of splitting can be as high as 1.56×. We summarize the minimum, median, and maximum compression ratios in Figure 5.11 (*bottom*).

These three findings confirm the potential improvements in speed and storage using UBCSR format and splitting. We elaborate on these conclusions below.

1. Proximity to uniform register blocking performance

The performance under splitting and UBCSR storage can approach or even slightly exceed the median register blocking performance on FEM Matrices 2–9. For each platform, we show in Figure 5.10 the minimum, median, and maximum performance on the matrices in Table 5.2. Performance is displayed as a fraction of median register blocking performance taken over Matrices 2–9. We also show statistics for register blocking only and reference implementations. The median fraction achieved by splitting exceeds the median fraction achieved by register blocking on all but the Itanium 2. On the Pentium III-M and Power4, the median fraction of splitting exceeds the maximum of register blocking only, demonstrating the potential utility of splitting and the UBCSR format. The maximum fraction due to splitting slightly exceeds 1 on all platforms.

The data for the individual platforms, Figures 5.6–5.9 (*top*), shows that the best performance is attained on Matrices 12, 13, A, C, and E, which are all dominated by a single unaligned block size (see Table 5.2). However, the fastest component of the splitting is comparable in performance to the median FEM 2–9 performance in at least half the cases on all platforms. Not surprisingly, splitting performance can be limited by the slowest component, which in most cases is the CSR implementation, or in the case of Matrix 15, "small" block sizes like 2×1 and 2×2. On Itanium 2, the fastest component is close to or in excess of

the register blocking performance (Figure 5.9 (*top*)) but overall performance never exceeds register blocking performance. This observation suggests the importance of targeting the CSR (1×1) implementation for low-level tuning, as suggested in the performance bounds analysis of Chapter 4.

2. Median speedups

We compare the following speedups on each platform in Figure 5.11 (*top*):

- Speedup of splitting over register blocking (blue solid diamonds)
- Speedup of register blocking over the reference (green solid circles)
- Speedup of splitting over the reference (red solid squares)

Figure 5.11 (*top*) shows minimum, median, and maximum speedups taken over the matrices in Table 5.2.

Splitting is at least as fast as register blocking on all but the the Itanium 2 platform. Median speedups, taken over the matrices in Table 5.2 and measured relative to the reference performance, range from 1.26× (Ultra 2i) up to 2.1× (Itanium 2). Relative to register blocking, median speedups are relatively modest, ranging from 1.1–1.3×. However, these speedups can be as much as 1.8× faster.

3. Reduced storage requirements

Though the speedups can be relatively modest, splitting can significantly reduce storage requirements. Recall from Section 3.1 that the asymptotic storage for CSR, ignoring row pointers, is 1.5 doubles per non-zero when the number of integers per double is 2. When abundant dense blocks exist, the storage decreases toward a lower limit of 1 double per non-zero. Figures 5.6–5.9 (*bottom*) compare the storage per non-zero between CSR, register blocked, and the splitting implementations. We also show the minimum, median, and maximum storage per non-zero taken over FEM Matrices 2–9 for register blocking. Except for Matrix 15, splitting reduces the storage on all matrices and platforms, and is often comparable to the median storage requirement for Matrices 2–9.

In the case of Matrix 15, the slight increase in storage is due to a small overhead in UBCSR storage. All natural dense blocks are 2×1 or 2×2 and uniformly aligned for this matrix, as shown in Figure F.14.

On Itanium 2, the dramatic speedups over the reference from register blocking come at the price of increased storage—just over 2 doubles per non-zero on Matrices 15, 17, and B. Though the splitting implementations are slower, they dramatically reduce the storage requirement in these cases.

We summarize the overall compression ratios across platforms in Figure 5.11 (*bottom*). We define the compression ratio for format a over format b as the size of the matrix in format b divided by the size in format a (larger ratios mean a requires less storage). We compare the compression ratio for the following pairs of formats in Figure 5.11 (*bottom*):

- Compression ratio of splitting over register blocking (blue solid diamonds)
- Compression ratio of register blocking over the reference (green solid circles)
- Compression ratio of splitting over the reference (red solid squares)

Median compression ratios, taken over the matrices in Table 5.2, for the split/UBCSR representation over BCSR/register-blocking range from 1.15 to 1.3. Relative to the reference, the median compression ratio for splitting ranges from 1.24 to 1.3, but can be as high as 1.45, which is close to the asymptotic limit.

5.2 Exploiting diagonal structure

This section presents performance results for a generalization of a diagonal data structure which we refer to as the row segmented diagonal (RSDIAG) format. RSDIAG is inspired by the types of diagonal substructure that arises in practice (Section 5.2.1). The data structure is organized and parameterized by a tuning parameter—an unrolling depth—that can be selected in a matrix and architecture-specific fashion (Sections 5.2.2–5.2.3). We show that SpMV implementations based on this format can lead to speedups of 2× or more over CSR on a variety of application matrices, and consider examples in which both diagonals and rectangular blocking can be profitably exploited (Section 5.2.4).

5.2.1 Test matrices and motivating examples

Our data structure for diagonals is motivated by the kinds of non-zero patterns that arise in practice, two examples of which we show in Figure 5.12. Figure 5.12 (*left*), a 60×60 submatrix taken from Matrix 11, is an example of mixed block diagonal and diagonal

substructure. The entire matrix consists of interleaved diagonals and 4×4 block diagonals, and the single diagonals account for 25% of all non-zeros. Uniform register blocking is difficult to apply in this case because of the fill required near the single diagonals.

Figure 5.12 (*right*) shows an example of a 80×80 submatrix of a larger matrix that exhibits complex diagonal structure. This structure is not exploited by the usual diagonal (DIAG) format discussed in Chapter 2 because DIAG assumes full or near-full diagonals. This matrix consists of a large number of "diagonal runs" that become progressively longer, with an average run length being roughly 93 elements. Any given row intersects 3–4 such runs on average.

Aside from Matrix 11, the matrices from the original SPARSITY benchmark suite do not exhibit much diagonal or diagonal fragment structure. However, matrices from a number of applications do, so this section applies RSDIAG format to these cases. The diagonal test set, displayed in Table 5.3, includes Matrix 11.

This test set also includes 3 synthetic matrices whose structure mimics the non-zero patterns arising in finite difference discretizations of scalar elliptic partial differential equations on rectangular regions (squares and cubes) with a "natural" ordering of nodes [267, 93]. We refer to these matrices as "stencil matrices." We include 5-point and 9-point stencils on squares (Matrices S1–S2), and a 27-point stencil on a cube (Matrix S3). These matrices consist of 5, 9, and 27 nearly full diagonals, respectively.

The last three columns of Table 5.3 roughly characterize the diagonal structure of the test matrices. For each matrix A, we identify all diagonal runs of length 6 or more, by the procedure described below (Section 5.2.3). We report the fraction of total non-zeros contained in these runs in column 3 and the average run length in column 4. The last column (5) shows the number of non-zeros per row. All but two matrices—Matrices 11 and F—are dominated by diagonal substructure, as suggested by column 3 of Table 5.3. Matrices 11 and F contain block structure that we exploit by splitting, as described in Section 5.2.3.

5.2.2 Row segmented diagonal format

The basis for RSDIAG format is shown in Figure 5.12 (*right*). An input matrix A is divided into *row segments*, or blocks of consecutive rows such that each segment consists of 1 or more diagonal runs equal to the number of rows in the segment. Let s be the number of

#	Matrix Dimension n, No. of non-zeros k	Approx. % of all nzs in diag. runs	Avg. diag. run length	Avg. no. of nzs per row
11	`11-bai` Airfoil $n = 23560$, $k = 484256$	43%	328	20.7
S1	`dsq_S_625` 2D 5-point stencil ($N = 625$) $n = 388129$, $k = 1938153$	100%	1551	5.0
S2	`sten_r2d9` 2D 9-point stencil ($N = 500$) $n = 250000$, $k = 2244004$	100%	747	9.0
S3	`sten_r3d27` 3D 27-point stencil ($N = 42$) $n = 74088$, $k = 1906624$	100%	61	27.0
F	`2anova2` Statistical analysis (ANOVA) $n = 254284$, $k = 1261516$	60%	440	9.0
G	`3optprice` Option pricing (finance) $n = 59319$, $k = 1081899$	96%	71	18.2
H	`marca_tcomm` Markov model: telephone exchange $n = 547824$, $k = 2733595$	>99.5%	477	5.0
I	`mc2depi` Markov model: Ridler-Rowe epidemic $n = 525825$, $k = 2100225$	>99.5%	592	4.0

Table 5.3: **Diagonal test matrices**. A list of matrices with diagonal substructure. The approximate fraction of non-zeros contained in diagonal runs of length 6 or more is listed in column 3. The average diagonal run length is shown in column 4. The average number of non-zeros per row (k/n) is shown in column 5.

such segments. In Figure 5.12 (*right*), each red horizontal line shown separates two row segments. Within the submatrix shown, the smallest segments consist of only 1 row each, the largest segments shown consist of 6 rows each (*e.g.*, rows 51–56), and $s = 40$. The RSDIAG data structure consists of the following:

- the starting row of each segment in an array `seg_starts`, of length $s + 1$.
- the number of diagonals in each segment, stored implicitly in an array `num_diags` of length s

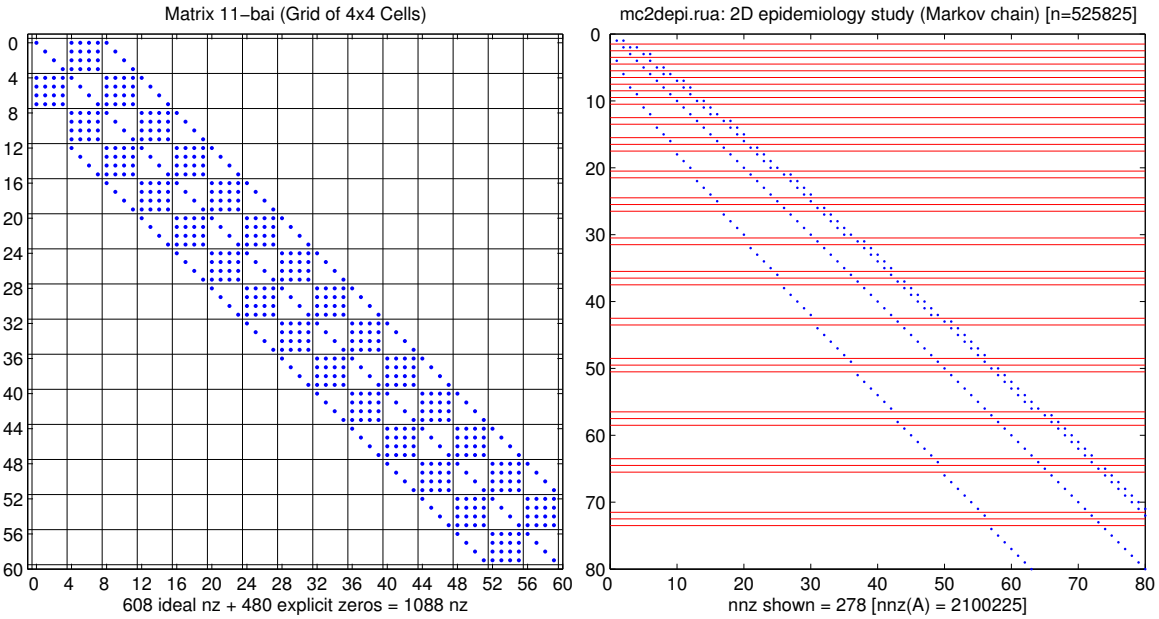

Figure 5.12: **Example of mixed diagonal and block structure: Matrix 11bai**. (*Left*) A 60×60 submatrix of Matrix 11-`bai`, beginning at position $(0, 0)$. This matrix consists of a number of interleaved diagonals and 4×4 block diagonals, with breaks/gaps along the diagonals. Diagonals account for approximately 25% of all non-zeros. The fill ratios under uniform register blocking at 2×2 and 4×4 are 1.23 and 1.70, respectively. (*Right*) Spy plot of a matrix from a Markov chain model used in an epidemiology study [286]. An 80×80 submatrix starting at position $(0, 0)$ in the original matrix. Diagonal fragments continue to lengthen and shrink.

- the starting column index (or source-vector index) of each diagonal in each segment, stored in an array `src_ind`, and
- an array `val` containing all non-zero values, laid out as described below.

The data structure is tuned for a given platform by selecting a tuning parameter u that represents an unrolling depth. For each segment, the non-zero values are laid out by storing u elements from the first diagonal, followed by u elements from the second diagonal, and so on for the remaining diagonals, and then storing the next u elements from the first diagonal, followed by the next u elements from the second diagonal, continuing until all non-zeros have been stored. Since u is fixed for an entire row segment, the code to multiply by a row segment may be unrolled by u. As usual, the best value of u depends on the platform and the available diagonal structure within the matrix.

We show an example of this data structure in Figure 5.13, where the sample matrix A is divided into two row segments (one consisting of 2 diagonals and the other consisting of 3 diagonals). The tuning parameter $u = 2$ in this example. We show the corresponding

$$A = \left(\begin{array}{ccccccc} a_{00} & 0 & 0 & a_{03} & 0 & 0 & 0 \\ 0 & a_{10} & 0 & 0 & a_{14} & 0 & 0 \\ \hline a_{20} & 0 & a_{22} & a_{23} & 0 & 0 & 0 \\ 0 & a_{31} & 0 & a_{33} & a_{34} & 0 & 0 \\ 0 & 0 & a_{42} & 0 & a_{44} & a_{45} & 0 \\ 0 & 0 & 0 & a_{53} & 0 & a_{55} & a_{56} \end{array}\right)$$

$$\texttt{seg\_starts} \leftarrow [0, 2, 6] \qquad \texttt{num\_diags} \leftarrow [2, 3] \qquad \texttt{src\_ind} \leftarrow [0, 3 | 0, 2, 3]$$

$$\texttt{val} \leftarrow [a_{00}, a_{10}, a_{03}, a_{14} | a_{20}, a_{31}, a_{22}, a_{33}, a_{23}, a_{34} | a_{42}, a_{53}, a_{44}, a_{55}, a_{45}, a_{56}]$$

Figure 5.13: **Example of row segmented diagonal storage**. We show a 6×7 matrix A with diagonal substructure. Here, A is partitioned into two row segments: the first segment contains two diagonals, and the second contains three. The values are laid out in an array in blocks of length $u = 2$ taken from each diagonal within a segment.

code—a routine named `sparse_mvm_onerseg_2`—to multiply by *one* given row segment in Figure 5.14, where again $u = 2$. The innermost loop has been unrolled by 2 (lines R4b–R4c). A complete SpMV routine repeatedly calls `sparse_mvm_onerseg_2` (or an equivalent routine at a different unrolling depth), as we show in Appendix H (Figure H.1).

This storage format can be extended to store block diagonals or bands, though we leave this possibility for future work.

5.2.3 Converting to row segmented diagonal format

Just as in Section 5.1, we benchmark split formulations of SpMV that combine diagonal and block structure. Specifically, we split the matrix $A = B + A_{\text{diag}}$, where B is optimized using register blocking and A_{diag} is stored in RSDIAG format. The remainder of this section describes the limited search procedure we use to select, for each matrix and machine, an implementation on which to report results in Section 5.2.4.

First, we extract all diagonal runs of length at least 6, and store them in a matrix A_1 (see note in the following paragraph). All remaining elements are stored in a second matrix A_2. Both A_1 and A_2 are stored in CSR format. If the number of non-zeros in A_2 accounts for less than 5% of the total non-zeros, then we do not split the matrix and instead elect to store all of A in RSDIAG format. (Equivalently, we set $B = 0$.) This case applies

```
R0   void sparse_mvm_onerseg_2( int M, int n_diags,
         const double* val, const int* src_ind,
         const double* x, double* y )
     {
         int I; // iterates block rows
R1a      const double* xpp = x;
R1b      for( br = 0; br < M; br++, y += 2, xpp += 2 )
         {
             int diag;
R2a          const int* indp = src_ind;
R2b          register double y0 = y[0], y1 = y[1];

R3a          for( diag = 0; diag < n_diags;
                  diag++, indp++, val += 2 ) // loop over diagonals
             {
R4a              register const double* xp = &(xpp[indp[0]]);
R4b              y0 += val[0] * xp[0];
R4c              y1 += val[1] * xp[1];
             }
R5a          y[0] = y0;
R5b          y[1] = y1;
         }
     }
```

Figure 5.14: **Example: row segmented diagonal matrix-vector multiply**. This routine multiplies one matrix row segment by a vector, assuming an unrolling depth of $u = 3$. The pointer `y` points to the first corresponding destination vector element. We number the lines to highlight the correspondence between the dense and BCSR SpMV routines shown in Figure 3.1. This routine is called once per row segment. The complete SpMV routine is shown in Appendix H.

to all the matrices in Table 5.3 except Matrices 11 and F.

For Matrices 11 and F, we store A_1 in RSDIAG format, and then evaluate the SPARSITY Version 2 heuristic described in Chapter 3 to optimize A_2 by register blocking.[1] Let B denote the optimized version of A_2. (If the heuristic determines that blocking is not beneficial, A_2 is left unchanged.)

Given u, conversion of either all of A or A_1 to A_{diag} is based on the *maximal* row segments (*i.e.*, the row segments of maximum length). For all unrolling depths u such that $1 \leq u \leq 32$, we convert A_1 to a matrix A_{diag} in RSDIAG format and measure the performance of applying $B + A_{\text{diag}}$.

Some diagonals intersect blocks, and are kept in A_2 for blocking rather than being

[1] In principle, we could also run the UBCSR splitting search procedure described in Section 5.1.3, but this was not necessary for the test matrices with diagonal structure.

placed in A_1 for diagonal storage. For example, see the third sub- and super-diagonals in Figure 5.12 (*left*), which intersect the 4×4 block diagonals. Since only two of the test matrices display prominent block structure, we identify these diagonals manually to exclude them from A_1. In principle, we could instead apply efficient non-zero structure analysis techniques developed by Bik and Wijshoff [44] and Knijnenburg and Wijshoff [192].

Section 5.2.4 reports on the performance of the best implementation found by the above procedure for each matrix and machine over all values of u. These implementations are summarized in Tables H.1–H.4 of Appendix H.

5.2.4 Experimental results

We show that RSDIAG format leads to improvements in performance by up to a factor of 2× or more compared to register blocking on the diagonal matrix test set. Figures 5.15–5.18 compare the absolute performance in Mflop/s and the speedup relative to register blocking for the following three implementations:

- **Best register blocking implementation on a dense matrix in sparse format** (black hollow square): Best performance shown in the register profile for the corresponding platform (Figures 3.3–3.6).
- **Median, minimum, and maximum register blocking performance on Matrices 2–9** (median by a black hollow circle, maximum by a black solid diamond, and minimum by a black solid downward-pointing triangle): For Matrices 2–9, consider the best performance observed after an exhaustive search (blue solid circles shown in Figures 3.12–3.15). We show the median, minimum, and maximum of these values.
- **RSDIAG** (red solid squares; horizontal dash-dot line): The implementation described in Section 5.2.3. For matrices 11 and F, this implementation splits $A = B + A_{\text{diag}}$, where B is optimized by register blocking and A_{diag} contains diagonal fragments stored in RSDIAG format.
- **Register blocking** (green solid circles): The register blocking implementation at the block size selected by the SPARSITY Version 2 heuristic (see Chapter 3).
- **Reference** (black asterisks): The CSR implementation.

Matrices 11, F, G, and H all fit in the L3 cache of the Power4, and we therefore omit these matrices from the Power4 results.

For reference, we also show in Figures 5.15–5.18 (*top*)

- the performance of tuned dense matrix-vector multiply (DGEMV) on a large (out-of-cache) matrix, as a blue horizontal dash-dot line, and
- the performance of tuned dense band matrix-vector multiply (DGBMV), for a large (out-of-cache) matrix with bandwidth 5 centered about the main diagonal, as a purple horizontal dash-dot line.

(For more information on the dense BLAS implementations, see Appendix B.)

We make the following high-level conclusions based on this data:

1. *The performance of the RSDIAG implementations can approach the performance observed on Matrices 2–9 on all platforms except the Itanium 2.* Specifically, the median performance of the RSDIAG implementations on the diagonal test set is 75–85% of the median register blocking performance on Matrices 2–9 on the Ultra 2i, Pentium III-M, and Power4. It may be possible to reduce the gap by unrolling across diagonals within a segment instead of just along the diagonal, as we have described for RSDIAG.

   On Itanium 2, the fraction is smaller (56%), but at least performance is comparable to tuned DGBMV performance.

2. *Median performance in the RSDIAG format is reasonably good compared to dense band matrix-vector multiply performance.* The absolute performance is 80% or more of the tuned DGBMV performance on an out-of-cache workload with a bandwidth 5.

3. *Median speedups, taken over the diagonal test matrices, from RSDIAG relative to register blocking range from* 1.6× *up to nearly* 2.3×. On Matrices 11 and F, which have a mix of diagonal and block structure, the improvement from RSDIAG is relatively more modest—up to 1.4× faster than just register blocking.

4. *Storage using RSDIAG format for the diagonal matrix test set is typically close to the asymptotic minimum value of 1 double per non-zero.* The data structure size depends only on the initial row segmentation, and not on the tuning parameter u since u only affects the organization of the non-zero values (Section 5.2.2). We compare the

Matrix	Storage (doubles per non-zero)				
		Register blocking			
	RSDIAG	Ultra 2i	Pentium III-M	Power4	Itanium 2
11	1.01	1.41	1.41	–	1.83
S1	<1.01	1.60	1.60	1.60	3.25
S2	<1.01	1.56	1.72	1.56	2.30
S3	1.03	1.52	1.67	1.52	2.12
F	1.07	1.60	1.60	–	1.60
G	1.03	1.53	1.82	–	1.82
H	1.01	1.60	1.60	–	2.35
I	<1.01	1.63	1.63	1.63	2.31

Table 5.4: **Comparing storage requirements between row segmented diagonal storage and register blocking**. We compare the storage, in units of doubles per non-zero, between row segmented diagonal storage and register blocking. The specific implementation parameters are listed in Appendix H.

total size of the data structure (in normalized units of doubles per non-zero) between the split/RSDIAG format compared to register blocking on the four platforms and matrices in Table 5.4.

Since many of these matrices have relatively few non-zeros per row, the integer index overhead incurred by register blocking can be high—between 1.5 and 2 doubles per non-zero—due to the relatively high contribution from row pointers (see Chapter 3). On Itanium 2, the storage is especially high due to fill overheads.

These results confirm the potential performance and storage pay-offs from RSDIAG.

Figures 5.15–5.18 (*top*) also show that absolute performance tends to increase as we move from Matrix S1 to Matrix S2 to Matrix S3, while absolute performance tends to decrease in moving from Matrix G to Matrix H to Matrix I. These trends correlate with the number of non-zeros per row displayed in Table 5.3: performance increases as the average number of non-zeros per row increases. Furthermore, the optimal value of u tends either to remain flat or decrease as the number of non-zeros per row increase.

We make the relationships among performance, non-zeros per row, and u explicit in Figures 5.19–5.21, where we present data for the 3 platforms on which data exists for all 6 matrices: Ultra 2i, Pentium III-M, and Itanium 2. Specifically, we show absolute performance on each platform as a function of u for these six matrices. Each series represents a matrix, and in the legend we label and list each series in decreasing order by the average

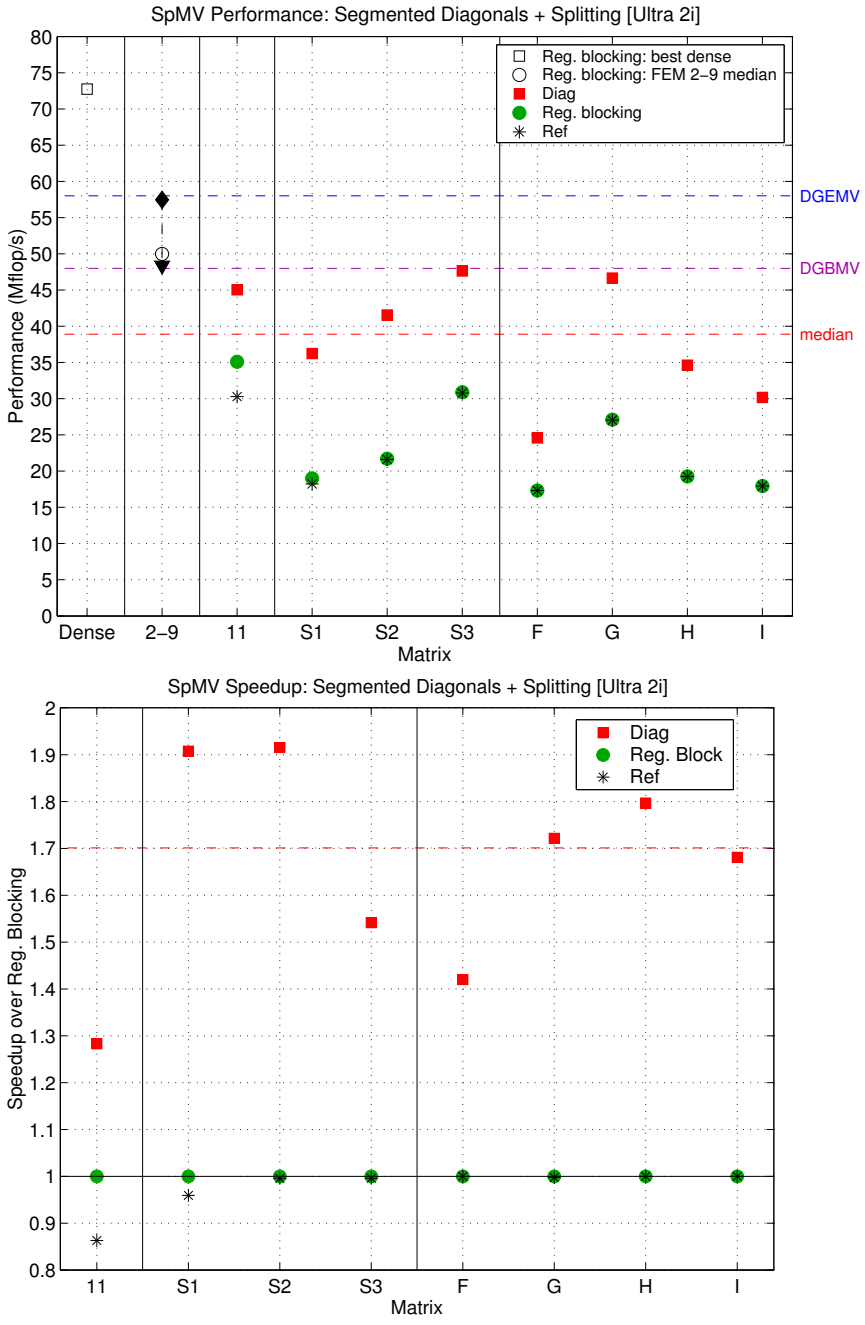

Figure 5.15: **Performance results on diagonal matrices: Ultra 2i.** (*Top*) Performance in Mflop/s of (*Bottom*) Speedup of the RSDIAG implementation over register blocking. This data is also tabulated in Table H.1.

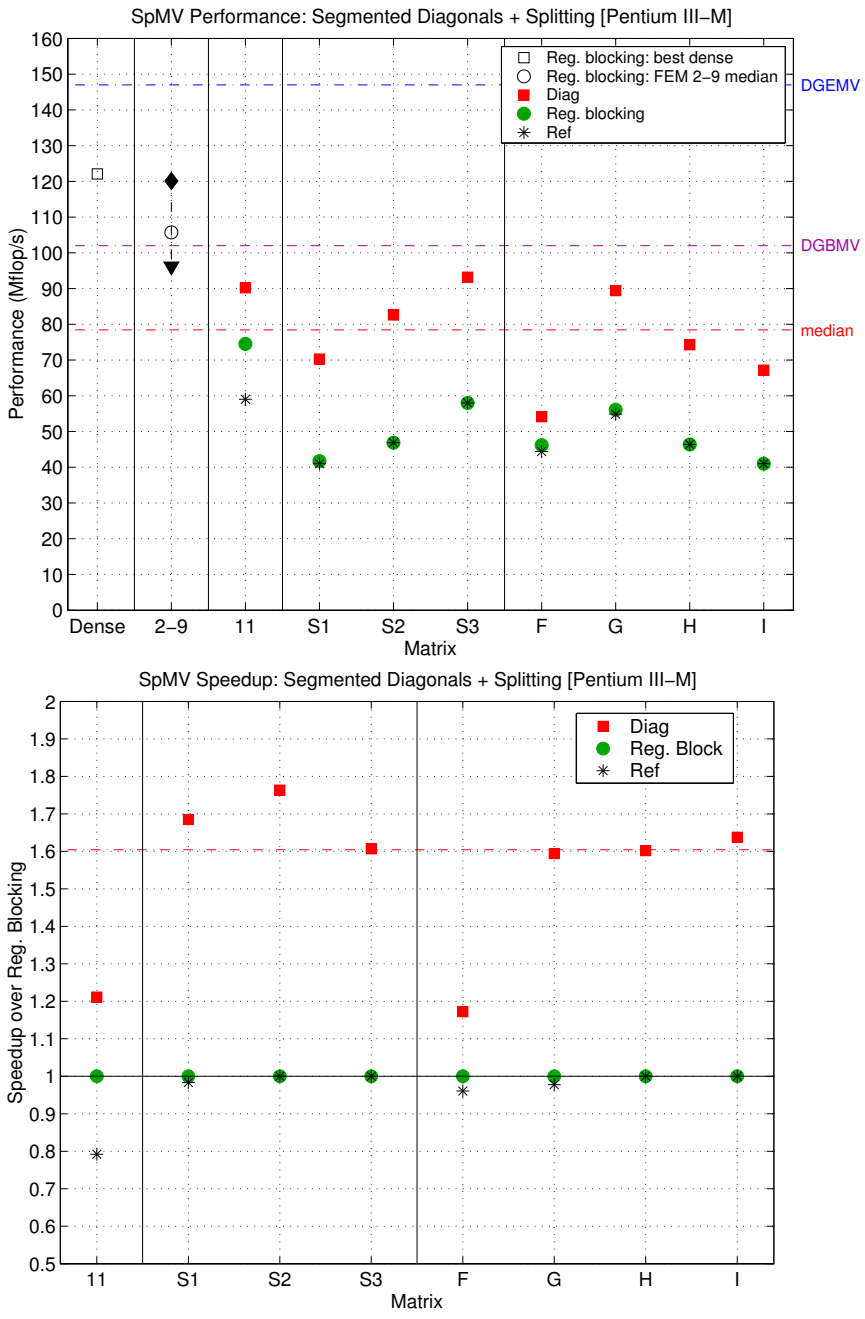

Figure 5.16: **Performance results on diagonal matrices: Pentium III-M.** (*Top*) Performance in Mflop/s of (*Bottom*) Speedup of the RSDIAG implementation over register blocking. This data is also tabulated in Table H.2.

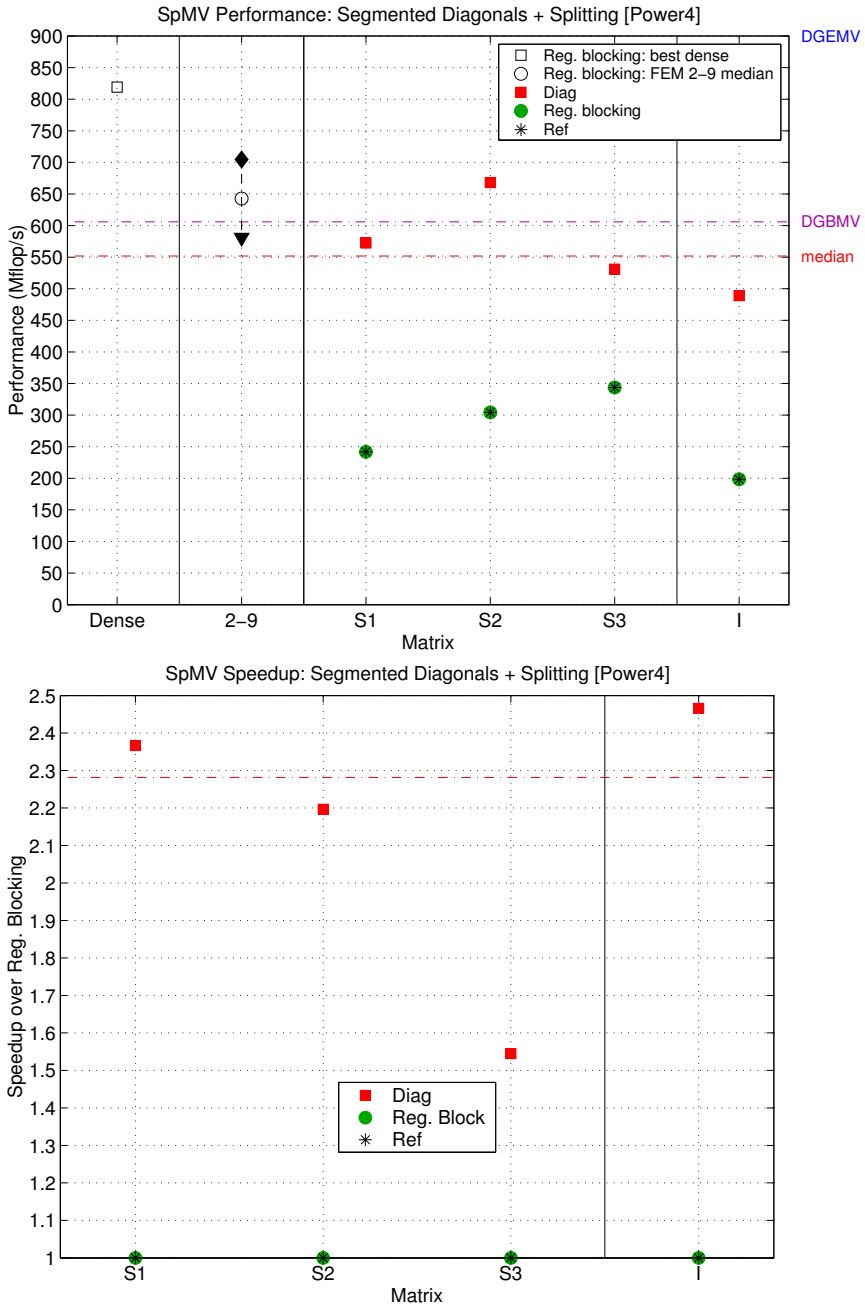

Figure 5.17: **Performance results on diagonal matrices: Power4.** (*Top*) Performance in Mflop/s of (*Bottom*) Speedup of the RSDIAG implementation over register blocking. This data is also tabulated in Table H.3.

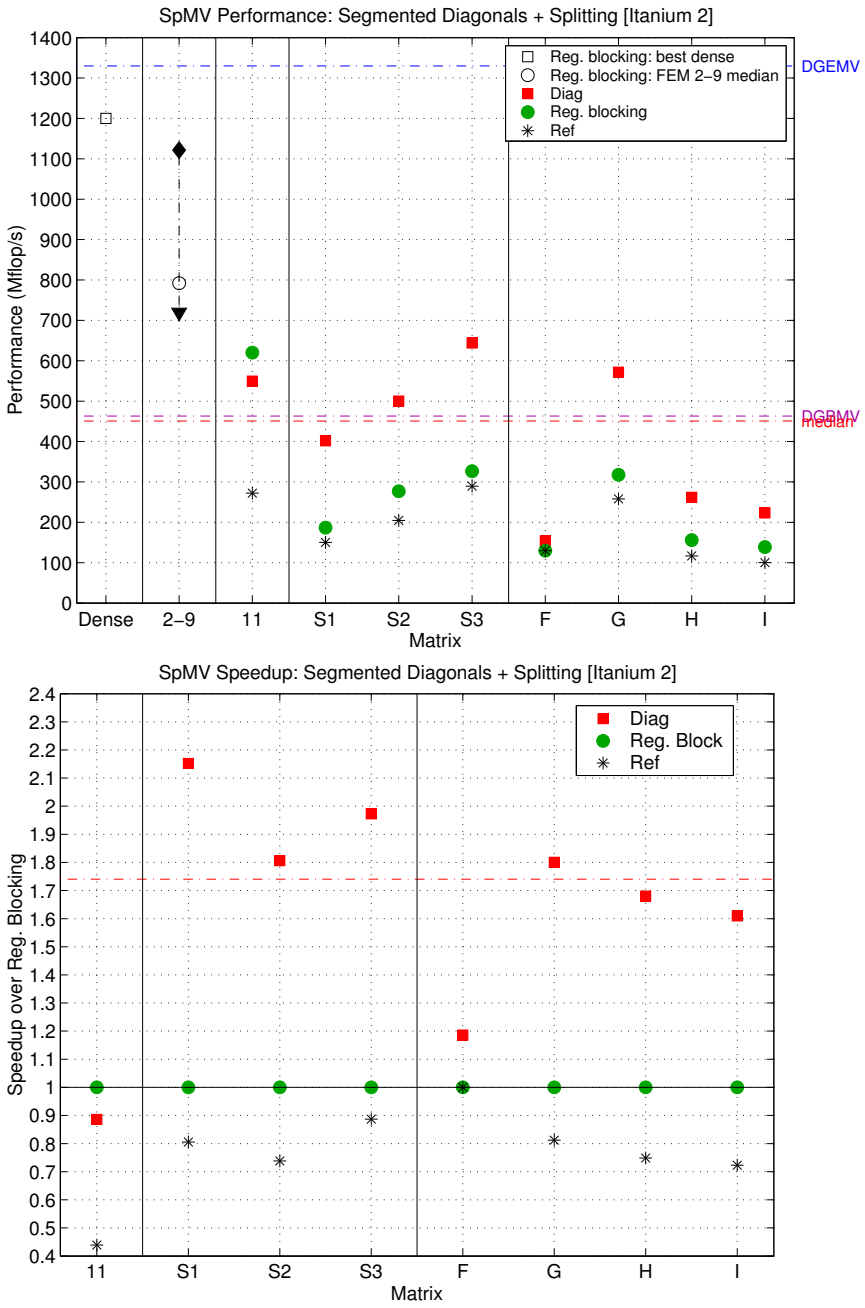

Figure 5.18: **Performance results on diagonal matrices: Itanium 2.** (*Top*) Performance in Mflop/s of (*Bottom*) Speedup of the RSDIAG implementation over register blocking. This data is also tabulated in Table H.4.

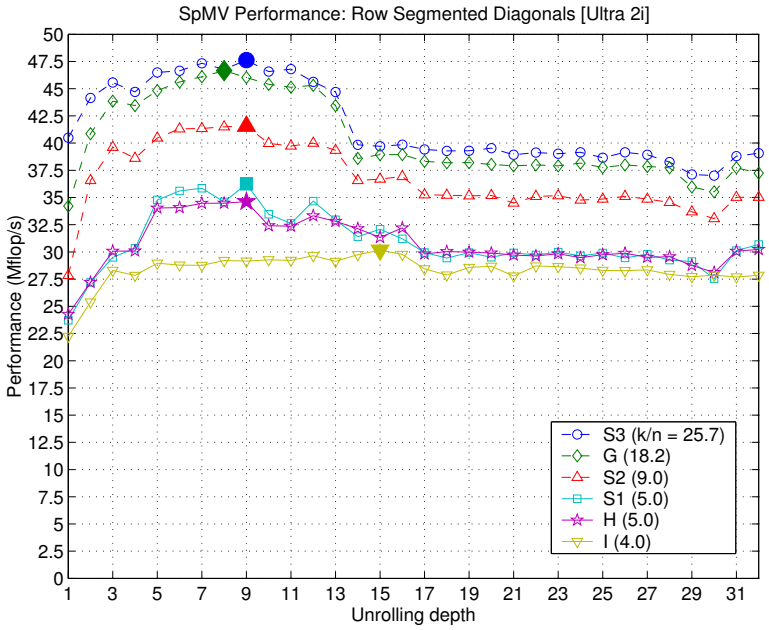

Figure 5.19: **Relationships among row segmented diagonal performance, unrolling depth u, and average number of non-zeros per row: Ultra 2i**. Each series represents a matrix. The legend shows the average number of non-zeros per row.

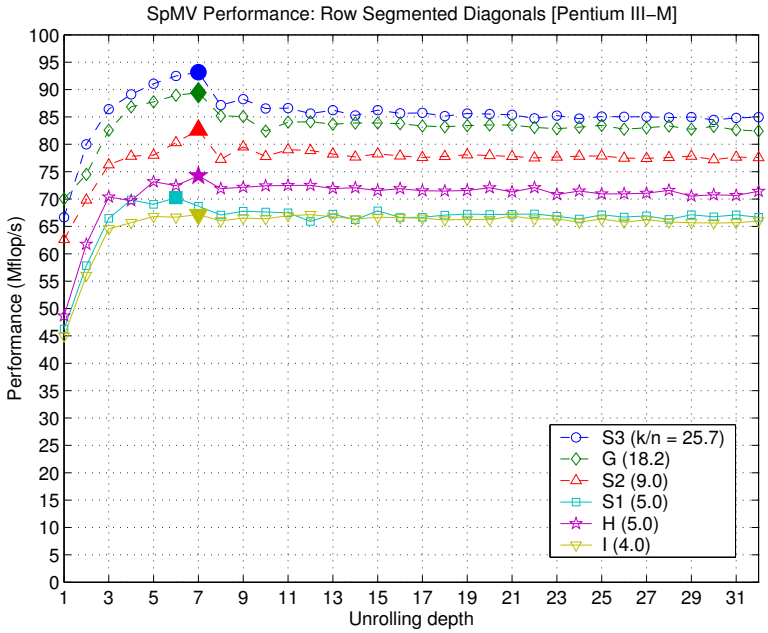

Figure 5.20: **Relationships among row segmented diagonal performance, unrolling depth u, and average number of non-zeros per row: Pentium III-M**. Each series represents a matrix. The legend shows the average number of non-zeros per row.

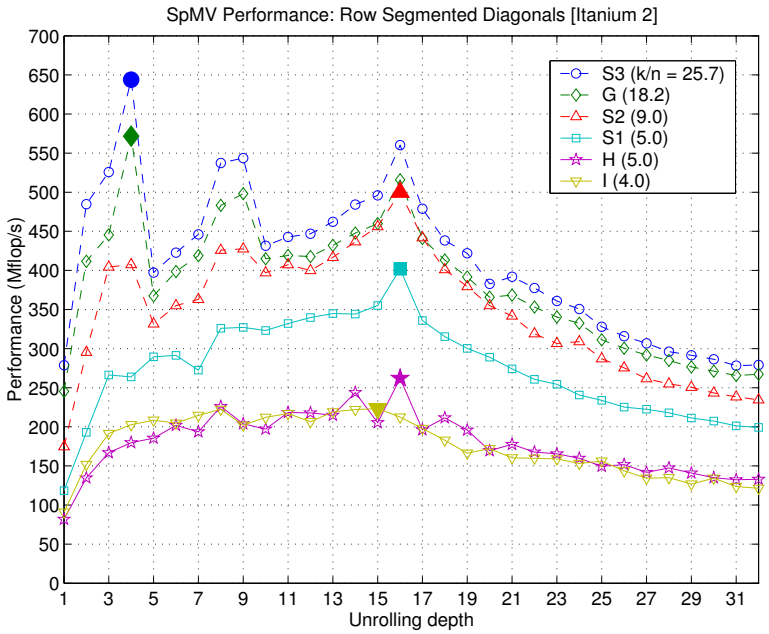

Figure 5.21: **Relationships among row segmented diagonal performance, unrolling depth u, and average number of non-zeros per row: Itanium 2.** Each series represents a matrix. The legend shows the average number of non-zeros per row.

number of non-zeros per row. All series are shown with hollow markers, and the best value of u on each curve shown by a solid marker. On the Pentium III, Matrices S1 and H are not strictly ordered by average number of diagonals per row, but otherwise the relationship persists. Nevertheless, the trends suggest both (1) a need to look more closely at architectural aspects affecting the performance on diagonal structures,[2] and (2) a possible heuristic for selecting the unrolling depth, possibly building on work demonstrated for dense matrix multiply [274]. We leave both possibilities to future work.

5.3 Summary and overview of additional techniques

The variable-block splitting and diagonal storage techniques presented in this chapter complement the body of existing peformance optimizations being considered for inclusion in the SPARSITY v2.0 system for SpMV. Below, we summarize these techniques, the maximum speedups over CSR and register blocking that we have observed, and notes regarding major

[2]For instance, one possible explanation is that we may be observing the cost of stores, since the number of store operations per element increases as the number of diagonals per row decreases.

unresolved issues. We provide pointers both to existing and upcoming reports that discuss these techniques in detail, and to related work external to this particular research project.

- **Register blocking, based on BCSR storage** (up to 4× speedups over CSR): A number of recent papers [165, 316, 164], as well as Chapters 3–4, validate and analyze this optimization in great depth. The largest pay-offs occur on matrices with abundant uniform block structure, such as Matrices 2–9, and to a lesser extent on Matrices 10–17.

- **Multiplication by multiple vectors** (7× over CSR, 2.5× over register blocked SpMV): Some applications and numerical algorithms require the sparse matrix-multiple vector multiply (SpMM) kernel $Y \leftarrow Y + A \cdot X$, where X and Y are dense matrices [164, 25]. We can reuse A in this kernel. When combining register blocking with unrolling across multiple vectors, Im, *et al.*, recently demonstrated up to 7× speedups for this kernel compared to a CSR implementation, and up to 2.5× speedups over register blocking without multiple vectors on the four platforms considered in this chapter [165]. Speedups appear to be possible across most matrices, not just those with block structure. The multiple vector optimization has also recently been combined with symmetry optimizations, discussed below [204].

  The major missing piece for an automatic tuning system is an efficient run-time tuning heuristic for selecting both the block size and the vector unrolling depth. It is possible that a simple extension to the single-vector heuristic of Chapter 3 will provide good tuning parameter predictions.

- **Cache blocking** (2.2× over CSR): Cache blocking, as described in implemented by Im [164] for SpMV, reorganizes a large sparse matrix into a collection of smaller, disjoint rectangular blocks to improve temporal access to elements of x. This technique helps to reduce misses to the source vector toward the lower bound of only cold start misses, as our bounds model of Chapter 4 assumes. The largest improvements occur on large, randomly structured matrices like linear programming Matrices 41–44, as well as matrices from latent semantic indexing applications [36]. We recently showed up to 2.2× speedups over CSR on the same platforms used in this chapter [165]. Currently, deciding when to cache block and how to select a cache block size remain unresolved. For partial answers, see forthcoming work by Nishtala, *et al.* [235].

Temam and Jalby propose an interesting and as-yet unexplored variation on cache blocking we refer to as *diagonal cache blocking* [294]. They show by a theoretical analysis in a simple cache model that reducing the bandwidth helps to minimize self-interferences misses. They further observe that blocking the matrix in "bands" achieves the same effect, though we are not aware of any empirical validation to date.

- **Symmetry** (symmetric register blocked SpMV is 2.8× faster than non-symmetric CSR, and 2.1× faster than non-symmetric register blocked SpMV; symmetric register blocked SpMM is 7.3× faster than CSR SpMV, and 2.6× over non-symmetric register blocked SpMM): Lee, *et al.*, study a register blocking scheme when A is symmetric (*i.e.*, $A = A^T$) [204]. Symmetry requires that we only store roughly half of the non-zero entries, and yields significant performance gains as well. In addition to performance optimizations, Lee, *et al.*, extend the performance bounds model of Chapter 4 to the symmetric register blocked case. In the single vector case, they find up to 2.8× speedups from a symmetric register blocked implementation relative to a CSR implementation, and 2.1× speedups relative to a non-symmetric register blocked implementation. In the multiple vector case, they find that combining symmetry, register blocking, and multiple vectors yields 7.3× speedups relative to a CSR implementation, and 2.6× relative to non-symmetric register blocking with multiple vectors. These results apply to Matrices 4, 6–10, 25, 27, 28, and 40 of the Sparsity benchmark suite, among many other application matrices. One remaining unresolved issue is how to select the tuning parameters automatically, possibly by a simple extension to the single-vector heuristic of Chapter 3.

  Besides extending the work on symmetry to related cases (*e.g.*, for Hermitian and skew Hermitian matrices, structurally but not numerically symmetric matrices), some matrices are "nearly" symmetric or structurally symmetric, meaning that filling in zeros for symmetry could also pay off.

- **Variable block splitting, based on UBCSR storage** (2.1× over CSR; 1.8× over register blocking; Section 5.1): Splitting for multiple block sizes has also been explored by Geus and Röllin [129] and Pinar and Heath [250]. Geus and Röllin explore up to 3-way splittings for a particular application matrix used in accelerator cavity design, but the splitting terms are still based on row-aligned BCSR format. (The last splitting term in their implementations is also fixed to be 1×1 (CSR), as in our work.) Pinar

and Heath restrict their attention to 2-way splittings where the first term is $1\times c$ format and the second in 1×1. The main distinctions of our work are the use of VBR as a convenient intermediate format, the relaxed row-alignment, and benchmarking on a wider class of matrices.

We view the lack of a heuristic for determining whether and how to split to be the major unresolved issue related to splitting.

- **Exploiting diagonal structure, based on RSDIAG storage** ($2\times$ over CSR; Section 5.2): The classical setting in which diagonal structure-centric formats like DIAG and jagged diagonal (JAD) format have been applied is on vector architectures [326, 237, 238]. Here, we show the potential pay-offs from careful application on superscalar cache-based microprocessors.

  Again, effective heuristics for deciding when and how to select the main matrix- and machine-specific tuning parameter (unrolling depth) remain unresolved. However, in the data of Section 5.2 we note that performance is a *relatively* smooth function of u and the number of non-zeros per row, compared to the way in which performance varies with block size, for instance.

- **Reordering to create dense blocks** ($1.5\times$ over CSR): Pinar and Heath proposed a method to reorder rows and columns of a sparse matrix to *create* dense rectangular block structure which might then be exploited by splitting [250]. Their formulation is based on the Traveling Salesman Problem. In the context of SPARSITY, Moon, *et al.*, have applied this idea to the SPARSITY benchmark suite, showing speedups over conventional register blocking of up to $1.5\times$ on Matrices 17, 20, 21, and 40. Heras, *et al.*, have also proposed TSP-based reordering schemes, with an emphasis on theoretical aspects of formulating the problem [157]. Open issues include when to apply TSP-based reordering, what TSP approximation heuristics are likely to work best, and what the run-time costs will be.

  Related to these reordering techniques are classical methods for reducing the matrix bandwidth or fill for numerical factorization [86, 186, 10, 263, 54, 127, 295, 300]. A number of researchers have pursued the use of bandwidth reducing orderings for SpMV as well, though it is unclear to what extent this method will pay-off in practice [166, 301, 62, 152]. However, Temam and Jalby have proven in a simple 1-level cache

model that reducing bandwidth helps to minimize self-interference misses, suggesting additional careful study may be fruitful [294].

Although pay-offs from individual techniques can be significant, the common challenge is deciding when to apply particular optimizations and how to choose the tuning parameters. Chapter 3 enhances the original SPARSITY v1.0 technique for selecting a register block size, and subsequent chapters successfully apply similar heuristics to sparse triangular solve (SpTS) and sparse $A^TA\cdot x$ (SpA^TA) kernels. However, heuristics for the other SpMV optimization techniques still need to be developed.

The class of matrices represented by Matrices 18–44 of the SPARSITY benchmark suite (Appendix B) largely remain difficult, with exceptions noted above. Our performance bounds analysis (Chapter 4) indicates that better low-level tuning of the CSR (*i.e.*, 1×1 register blocking) SpMV implementation may be possible. Recent work on low-level tuning of SpMV by unroll-and-jam (Mellor-Crummey, *et al.* [221]), software pipelining (Geus and Röllin [129]), and prefetching (Toledo [301]) are promising starting points.

Both this chapter and the earlier chapter reviewing register blocking (Chapter 3) assume the matrix has already been assembled on input. From this starting point, we take a "bottom-up" approach to improving performance by identifying canonical structures and then exploiting them for performance. The non-zero structure analysis tools developed by Bik and Wijshoff [44] and Knijnenburg and Wijshoff [192] complement this approach in that these tools provide a means by which to detect and extract non-zero patterns. However, it may also be possible to recover information about the original mesh geometry from the assembled matrix for applications in physical modeling [284]. Determining whether adopting this latter approach—or even using the unassembled matrix itself—will lead to better non-zero structure analyses is an opportunity for future work.

Chapter 6

Performance Tuning and Bounds for Sparse Triangular Solve

Contents

This chapter presents a sparse matrix data structure and tuning heuristics to improve the sparse triangular solve (SpTS) kernel, *i.e.*, computing x such that $Tx = y$, where T is a sparse triangular matrix and x, y are dense vectors. We confirm that the data structure selection and tuning methodology of the preceeding chapters, as applied to sparse

matrix-vector multiply (SpMV), extends to the SpTS kernel as well. We show performance improvements of up to 1.8× over a basic implementation.

SpTS is an important step in so-called *direct methods* for solving general sparse linear systems, $Ax = b$, where A is a sparse matrix. The first step of these methods factors A into the product $L \cdot U$ of a sparse lower triangular matrix L and sparse upper triangular matrix U. The vector x is then computed by (1) performing a *forward solve* $Ly = b$ to compute y, followed by (2) a *backward solve* $Ux = y$. Although the cost of factorization is typically much more expensive than the solve steps, many applications require tens or hundreds of solves, each corresponding to different a right-hand side b, while A—and hence, L and U—remain fixed.

The key observation behind our tuning technique for SpTS is that in practice, the factors L and U have a special dense structure owing to the factorization process. This structure consists of a large, dense triangle in the lower right-hand corner of the matrix; this *trailing triangle* can account for as much as 90% of the matrix non-zeros. Therefore, we consider both algorithmic and data structure reorganizations which partition the solve into a sparse phase and a dense phase. To the sparse phase, we adapt the *register blocking optimization*, previously proposed for SpMV to the SpTS kernel; to the dense phase, we make judicious use of highly tuned Basic Linear Algebra Subroutines (BLAS) routines by switching to a dense implementation (*switch-to-dense* optimization). We describe fully automatic hybrid off-line/on-line heuristics for selecting the key tuning parameters: the register block size and the point at which to use the dense algorithm (Section 6.1).

In addition, we develop upper and lower bounds on performance in the spirit of Chapter 4. We verify these models on a variety of hardware platforms and a set of triangular factors from applications (Table 6.1) using hardware counter data collected with the PAPI library [60]. We observe that, just as in the case of SpMV (Chapter 4), our optimized implementations can achieve 75% or more of these bounds. Like the findings for SpMV shown in Chapter 4, this observation suggests a limit on the improvements possible with additional low-level tuning.

In the remainder of this chapter, we restrict our attention to the solution of the lower triangular system, $Lx = y$, where L is an $n \times n$ sparse lower triangular matrix that does not necessarily have unit diagonal elements. For this problem, we refer to x as the *solution* vector, and y as the *right-hand side* (RHS) vector.

The material in this chapter appeared in a recent paper [319].

	Name	Application Area	n	Nnz in L	Dense Trailing Triangle n_2	Density	% Total Nnz
1	dense	Dense matrix	1000	500500	1000	100.0%	100.0%
2	memplus	Circuit simulation	17758	1976080	1978	97.7%	96.8%
3	wang4	Device simulation	26068	15137153	2810	95.0%	24.8%
4	ex11	Fluid flow	16614	9781925	2207	88.0%	22.0%
5	raefsky4	Structural mechanics	19779	12608863	2268	100.0%	20.4%
6	goodwin	Fluid mechanics	7320	984474	456	65.9%	6.97%
7	lhr10	Chemical processes	10672	368744	104	96.3%	1.43%

Table 6.1: **Triangular matrix benchmark suite**. The LU factorization of each matrix was computed using the sequential version of SuperLU 2.0 [94] and Matlab's column minimum degree ordering. The dimension n and number of non-zeros in the resulting lower triangular L factor is shown. We also show the dimension n_2 of the trailing triangle found by our switch-point heuristic (column 6), its density (column 7: fraction of the trailing triangle occupied by true non-zeros), and the fraction of all matrix non-zeros contained within the trailing triangle (column 8).

6.1 Optimization Techniques

The triangular matrices which arise in sparse Cholesky and LU factorization frequently have the kind of structure shown in Figure 6.1, spy plots of two examples of lower triangular factors. The lower right-most dense triangle of each matrix, which we call the *dense trailing triangle*, accounts for a significant fraction of the total number of non-zeros. In Figure 6.1 (*left*), the dimension of the entire factor is 17758 and the dimension of the trailing triangle is 2268; nevertheless, the trailing triangle accounts for 96% of all the non-zeros. Similarly, the trailing triangle of Figure 6.1 (*right*), contains approximately 20% of all matrix non-zeros. The remainder of the matrix (the *leading trapezoid*) appears to consist of many smaller dense blocks and triangles.

We exploit this structure by decomposing $Lx = y$ into sparse and dense parts:

$$\begin{pmatrix} L_1 & \\ L_2 & L_D \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{6.1}$$

where L_1 is a sparse $n_1 \times n_1$ lower-triangular matrix, L_2 is a sparse $n_2 \times n_1$ rectangular

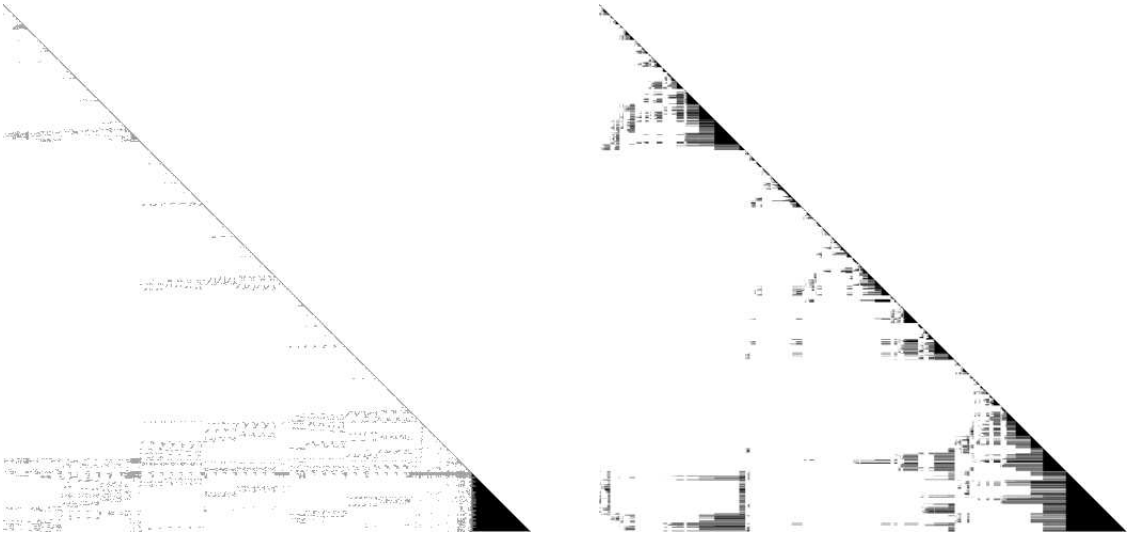

Figure 6.1: **Examples of sparse triangular matrices**. (*Left*) Matrix 2 (`memplus`) from Table 6.1 has a dimension of 17758. The dense trailing triangle, of size 1978, contains 96% of all the matrix non-zeros. (*Right*) Matrix 5 (`raefsky4`) from Table 6.1 has a dimension of 19779. The dense trailing triangle, of size 2268, accounts for 20% of all the matrix non-zeros.

matrix, and L_D is a dense $n_2 \times n_2$ trailing triangle. We solve for x_1 and x_2 in three steps:

$$L_1 x_1 = y_1 \tag{6.2}$$

$$\hat{y}_2 = y_2 - L_2 x_1 \tag{6.3}$$

$$L_D x_2 = \hat{y}_2 \tag{6.4}$$

Equation (6.2) is a SpTS, Equation (6.3) is a SpMV, and Equation (6.4) is a call to the tuned dense BLAS routine, TRSV. We refer to the implementation of Equation (6.4) by a call to TRSV as the *switch-to-dense* optimization. Although this process of splitting into sparse and dense components could be repeated for Equation (6.2), we do not consider this possibility here.

For reference, Figure 6.2 shows two common implementations in C of *dense* triangular solve: the row-oriented ("dot product") algorithm in Figure 6.2 (*top*), and the column-oriented ("axpy") algorithm in Figure 6.2 (*bottom*). The row-oriented algorithm is the basis for our register-blocked sparse algorithm; the column-oriented algorithm is essentially the reference implementation of the BLAS routine, TRSV, and its details are important in our analysis in Section 6.2.

```
   void dense_trisolve_dot( int n,
       const double* L, const double* y,
       double* x )
   {
     int i, j;
1    for( i = 0; i < n; i++ ) {
2      register double t = y[i];
3      for( j = 0; j < i; j++ )
4        t -= L[i+n*j]*x[j];
5      x[i] = t / L[i+n*i];
     }
   }

   void dense_trisolve_axpy( int n,
       const double* L, const double* y,
       double* x )
   {
     int i, j;
1    for( j = 0; j < n; j++ ) x[j] = 0;
2    for( j = 0; j < n; j++ ) {
3      register double t = (y[j] - x[j]) / L[j*n+j];
4      for( i = j+1; i < n; i++ )
5        x[i] = x[i] + L[i+n*j]*t;
6      x[j] = t;
     }
   }
```

Figure 6.2: **Dense triangular solve code (C)**. Reference implementations in C of (*top*) the row-oriented formulation, and (*bottom*) the column-oriented formulation of dense lower-triangular solve: $Lx = y$. In both routines, the matrix `L` is stored in unpacked column-major order (see Chapter 2). For simplicity, the stride is set to equal the matrix dimension, `n`, and the vectors are assumed to be unit-stride accessible.

6.1.1 Improving register reuse: register blocking

Recall from Chapters 2–4 that *register blocking* improves register reuse by reorganizing the matrix data structure into a sequence of "small" dense blocks, where the block sizes are chosen to keep small blocks of the solution and RHS vectors in registers [167]. In this chapter, we consider only square $b \times b$ block sizes. As before, we assume block compressed sparse row (BCSR) format for register blocking. The diagonal blocks are stored as full $b \times b$ blocks with explicit zeros above the diagonal, though no computation is performed using

these explicit zeros. As in the SpMV case, we fully unroll the $b \times b$ submatrix computations, reducing loop overheads and exposing scheduling opportunities to the compiler. An example of the 2×2 code appears in Figure 6.3. The body of the innermost `for` loop is very similar to the SpMV case shown in Figure 3.1, and the main difference is a subtraction instead of an addition. The other major difference in the SpTS BCSR code compared to the SpMV code is that the diagonal block is handled separately (line 5 of Figure 6.3).

Just as in the SpMV case, creating blocks may require filling in explicit zeros. Recall that we define the *fill ratio* to be the number of stored values (*i.e.*, including the explicit zeros) divided by the number of true (or "ideal") non-zeros. We may trade-off extra computation (*i.e.*, fill ratio > 1) for improved efficiency in the form of uniform code and memory access.

6.1.2 Using the dense BLAS: switch-to-dense

To support the switch-to-dense optimization, we reorganize the sparse matrix data structure for L into two parts: a dense submatrix for the trailing triangle L_D, and a sparse component for the leading trapezoid. We store the trailing triangle in dense, unpacked column-major format as specified by the interface to TRSV, and store the leading trapezoid in BCSR format as described above. We determine the column index at which to switch to the dense algorithm—the *switch-to-dense point* s (or simply, the *switch point*)—using the heuristic described below (Section 6.1.3).

6.1.3 Tuning parameter selection

In choosing values for the two tuning parameters—register block size b and switch point s—we first select the switch point, and then select the register block size.

Selecting the switch point

The switch point s is selected at run-time when the matrix is known. We choose s as follows, assuming the input matrix is stored in compressed sparse row (CSR) format format. Beginning at the diagonal element of the last row, we scan the bottom row until we reach *two consecutive* zero elements. The column index of this element marks the last column of

```
   void sparse_trisolve_BCSR_2x2( int n, const int* b_row_ptr,
       const int* b_col_ind, const double* b_values,
       const double* y, double* x )
   {
     int I, JJ;     assert( (n\%2) == 0 );

1    for( I = 0; I < n/2; I++) // loop over block rows
     {
2a     register double t0 = y[2*I];
2b     register double t1 = y[2*I+1];

3      for( JJ = b_row_ptr[I]; JJ < (b_row_ptr[I+1]-1); JJ++ )
       {
4a       int j0 = b_col_ind[ JJ ];
4b       register double x0 = x[j0];
4c       register double x1 = x[j0+1];

4d       t0 -= b_values[4*JJ+0] * x0;
4e       t1 -= b_values[4*JJ+1] * x1;

4f       t0 -= b_values[4*JJ+2] * x0;
4g       t1 -= b_values[4*JJ+3] * x1;
       }
5a     x[2*I] = t0 / b_values[4*JJ+0];
5b     x[2*I+1] = (t1 - (b_values[4*JJ+2]*x[2*I])) / b_values[4*JJ+3];
     }
   }
```

Figure 6.3: **SpTS implementation assuming 2×2 BCSR format**. An example of the 2×2 register blocked SpTS solve, assuming BCSR format. For simplicity, the dimension `n` is assumed to be a multiple of the block size in this example. Note that the matrix blocks are stored in row-major order, and the diagonal block is assumed (1) to be the last block in each row, and (2) to be stored as an unpacked (2×2) block. Lines are numbered as shown to illustrate the mapping between this implementation and the corresponding dense implementation of of Figure 6.2 (*top*).

the leading trapezoid.[1] Note that this method may select an s which causes additional fill-in of explicit zeros in the trailing triangle. As in the case of register blocking, tolerating some

[1]Detecting the no-fill switch point is much easier if compressed sparse column (CSC) format format is assumed. In fact, the dense trailing triangle can also be detected using symbolic structures (*e.g.*, the elimination tree) available during LU factorization. However, we do not assume access to such information. This assumption is consistent with the latest standardized Sparse Basic Linear Algebra Subroutines (SpBLAS) interface [49], earlier interfaces [267, 258], and parallel sparse BLAS libraries [116].

explicit fill can lead to some performance benefit. We are currently investigating a new selection procedure which evaluates the trade-off of gained efficiency versus fill to choose s.

Selecting the register block size

To select the register block size b, we adapt the SPARSITY v2.0 heuristic for SpMV (Chapter 3) to SpTS. There are 3 steps:

1. Collect a one-time *register profile* to characterize the platform. For SpTS, we evaluate the performance (Mflop/s) of the register blocked SpTS for all block sizes on a dense lower triangular matrix stored in BCSR format. These measurements are independent of the sparse matrix, and therefore only need to be made once per architecture.

2. When the matrix is known at run-time, estimate the fill for all block sizes. We can use the same fill estimator described in Chapter 3 to perform this step efficiently.

3. Select the block size b that maximizes

$$\text{Estimated Mflop/s} = \frac{\text{Mflop/s on dense matrix in BCSR for } b\times b \text{ blocking}}{\text{Estimated fill for } b\times b \text{ blocking}} \quad . \quad (6.5)$$

In principle, we could select different block sizes when executing the two sparse phases, Equation (6.2) and Equation (6.3); we only consider uniform block sizes here.

The costs of executing this heuristic are essentially identical to the costs described for SpMV in Chapter 3—approximately 10–30 executions of the reference implementation, where sampling the matrix accounts for less than 5 of those executions and the remaining cost is due to data structure conversion. Thus, the optimizations we propose are most suitable when SpTS must be performed many times.

6.2 Performance Bounds

Below, we adapt the bounds for SpMV described in Chapter 4 to SpTS. In particular, we assume the same latency-based model of execution time, which charges only for the cost of loads and stores, under the assumption that SpTS is memory bound. This assumption is valid because there are only 2 flops per matrix element, just as in the case of SpMV. We review the notation of the execution time model in Section 6.2.1.

The cost of loads and stores is, in turn, based on where data hits in the memory heirarchy, *i.e.*, the cost depends on where cache misses occur. It is the modeling of cache misses which is SpTS-specific. We describe our cache miss model in Section 6.2.2.

Refer to Chapter 4 for a review of the main assumptions and justification of our performance model.

6.2.1 Review of the latency-based execution time model

Our goal is to compute upper and lower bounds on performance. Let k_L be the number of non-zeros in the $n \times n$ sparse lower triangular matrix L. Triangular solve requires $2(k_L - n)$ multiplies and subtracts (2 flops per off-diagonal element), plus n divisions (1 division per diagonal element). Counting each division operation as 1 flop, the total number of flops is $2 \cdot k_L - n$. Thus, the performance P in Mflop/s is given by

$$P = \frac{(2 \cdot k_L - n)}{T} \cdot 10^{-6} \tag{6.6}$$

where T is the execution time of the solve in seconds. Note that in this definition, we do not count operations on explicitly filled in zeros as flops.

Let H_i be the number of hits at cache level i during the entire solve operation, and let M_i be the number of misses. Then, we use the same model of execution time presented in Chapter 4,

$$T = \sum_{i=1}^{\kappa-1} H_i \alpha_i + M_\kappa \alpha_{\text{mem}}, \tag{6.7}$$

where α_i is the access time (in seconds) at cache level i, κ is the level of the largest cache, and α_{mem} is the memory access time, and $\alpha_i \leq \alpha_{i+1}$. Note that Equation (6.7) is identical to Equation (4.3).

To obtain an upper bound on P, we need a lower bound on T. As discussed in Chapter 4, we use benchmarks and processor manuals to determine lower bounds on the access latencies, α_i. Moreover, we further bound T from below by obtaining lower bounds on each M_i. This fact follows from the observation that Equation (6.7) can be re-expressed in terms of loads, stores, and cache misses using $H_1 = \text{Loads} + \text{Stores} - M_1$, and $H_i = M_i - M_{i+1}$ for $i \geq 2$:

$$T = \alpha_1 \left(\text{Loads} + \text{Stores}\right) + \sum_{i=1}^{\kappa-1} (\alpha_{i+1} - \alpha_i) M_i + (\alpha_{\text{mem}} - \alpha_\kappa) M_\kappa \tag{6.8}$$

Since $\alpha_{i+1} - \alpha_i \geq 0$, minimizing M_i also minimizes T. Similarly, we can obtain a lower bound on P by computing an upper bound on each M_i. The bounds on M_i are SpTS-specific, and derived in Section 6.2.2.

6.2.2 Cache miss lower and upper bounds

In deriving cache misses for our optimized SpTS, we consider the sparse equations, Equations (6.2)–(6.3), separately from the dense solve, Equation (6.4).

We count the number of loads and stores required for Equation (6.2) as follows, assuming $b \times b$ register blocking. Let k be the total number of non-zeros in L_1 and L_2,[2] and let f_{rc} be the fill ratio after register blocking. Thus, kf_{rc} is the total number of stored values in L_1 and L_2. Then, the number of loads is

$$\begin{aligned} \text{Loads}_{\text{sparse}}(b) &= \underbrace{kf_{rc} + \frac{kf_{rc}}{rc} + \left\lceil \frac{m}{r} \right\rceil + 1}_{\text{matrix}} + \underbrace{\frac{kf_{rc}}{b}}_{\text{soln vec}} + \underbrace{n}_{\text{RHS}} \\ &= kf_{rc}\left(1 + \frac{1}{b^2} + \frac{1}{b}\right) + n + \left\lceil \frac{m}{r} \right\rceil + 1 \quad . \end{aligned} \tag{6.9}$$

We include terms for the matrix (all non-zeros, one column index per non-zero block, and $\lceil n/b \rceil + 1$ row pointers; see lines 3, 4a, and 4d–g in Figure 6.3), the solution vector (line 4b and 4c), and the RHS vector (line 2). The number of stores is $\text{Stores}_{\text{sparse}} = n$ (lines 5a and 5b in Figure 6.3).

To analyze the dense computation, Equation (6.4), we first assume a column-oriented ("axpy") algorithm for TRSV. We *model* the number of loads and stores required to execute Equation (6.4) as

$$\begin{aligned} \text{Loads}_{\text{dense}} &= \underbrace{\frac{n_2\,(n_2+1)}{2}}_{\text{matrix}} + \underbrace{\frac{n_2}{2}\left(\frac{n_2}{R} + 1\right)}_{\text{solution}} + \underbrace{n_2}_{\text{RHS}} \\ \text{Stores}_{\text{dense}} &= \underbrace{\frac{n_2}{2}\left(\frac{n_2}{R} + 1\right)}_{\text{solution}}, \end{aligned}$$

where the $1/R$ factors model register-level blocking in the dense code, assuming $R \times R$ register blocks.[3] In general, we do not know R if we are calling a proprietary vendor-supplied library; however, we can estimate R by examining load/store hardware counters when calling TRSV.

[2] For a dense matrix stored in sparse format, we would have $n_1 = k = 0$.

[3] The terms with R in them are derived by assuming R vector loads per register block. Assuming R divides n_2, there are a total of $\frac{n_2/R(n_2/R+1)}{2}$ blocks.

Next, we count the number of misses M_i, starting at the L1 cache. Let l_1 be the L1 line size, in doubles. We incur compulsory misses for every matrix line. The solution and RHS vector miss counts are more complicated. In the best case, these vectors fit into cache with no conflict misses; we incur only the $2n$ compulsory misses for the two vectors. Thus, a lower bound $M_{\text{lower}}^{(1)}$ on L1 misses is

$$M_{\text{lower}}^{(1)}(b) = \frac{1}{l_1}\left[kf_{rc}\left(1+\frac{1}{\gamma b^2}\right)+\frac{1}{\gamma}\left(\left\lceil\frac{m}{r}\right\rceil+1\right)+\left(2n+\frac{n_2\left(n_2+1\right)}{2}\right)\right]. \tag{6.10}$$

where the size of one double-precision value equals γ integers. The factor of $1/l_1$ accounts for the L1 line size. To compute $M_{\text{lower}}^{(i)}(b)$ at cache levels $i > 1$, we simply substitute the right line size. In the worst case, we miss on every access to a line of the solution vector; thus, a miss upper bound is

$$M_{\text{upper}}^{(1)}(b) = \frac{1}{l_1}\left[kf_{rc}\left(1+\frac{1}{\gamma b^2}\right)+\frac{1}{\gamma}\left(\left\lceil\frac{m}{r}\right\rceil+1\right)+\left(\frac{kf_{rc}}{b}+n+\text{Loads}_{\text{dense}}+\text{Stores}_{\text{dense}}\right)\right]. \tag{6.11}$$

Finally, we calculate an *upper bound* on performance P by substituting the lower bound on misses, Equation (6.10), into the expression for T, Equation (6.8). Similarly, we compute a *lower bound on performance* by substituting Equation (6.11) into Equation (6.8).

6.3 Performance Evaluation

We divide our analysis of SpTS into two parts. First, Section 6.3.1 validates our model of cache misses (Section 6.2) against actual measurements made with PAPI [60]. Second, we compare performance predicted by the bounds to actual measured performance in Section 6.3.2. Our experimental setup follows the methodology of Appendix B, though here we present results on a subset of four of the evaluation platforms: the Sun Ultra 2i, the Intel Pentium III, the Intel Itanium 1, and IBM Power3.

6.3.1 Validating the cache miss bounds

We used our heuristic procedure for selecting the switch point s. Keeping s fixed, we then performed an exhaustive search over all register block sizes up to $b = 5$, for all matrices and platforms, measuring execution time and cache hits and misses using PAPI. Figures 6.4–6.6 validate our bounds on misses, Equation (6.10) and Equation (6.11). In particular, for the largest cache sizes (L3 on Itanium, L2 on the other machines), the vector lengths are such

that the true miss counts are closer to Equation (6.10) than Equation (6.11), implying that conflict misses can be ignored.

6.3.2 Evaluating optimized SpTS performance

Figures 6.7–6.9 compare the observed performance of various implementations to the bounds derived in Section 6.2. In particular, we compare the following:

- **Analytic lower and upper performance bounds (Mflop/s)** (upper bound shown as dashed lines, lower bound as dash-dot lines):[4] We compute these bounds as discussed in Section 6.2 In particular, at each point we show the *best* bound over all b, with the switch point s fixed at the heuristic-selected value.

- **PAPI-based performance upper bound** (solid triangles): This bound was obtained by substituting measured cache hit and miss data from PAPI into Equation (6.7) and using the minimum memory latency for α_{mem}. This bound could be regarded as a more "realistic" bound than the analytic bound, since it assumes exact knowledge of misses.

- **Combined register blocking and switch-to-dense implementation** (solid circles): The register block size was again chosen exhaustively over all block sizes *after* the switch point s was chosen.

- **Switch-to-dense only** (hollow triangles): An implementation using only the switch-to-dense optimization (*i.e.*, without register blocking L_1 and L_2) at the same switch point s.

- **Register blocking only** (solid squares): The best implementation using only the register blocking optimization over all $1 \leq b \leq 5$.

- **Reference** (1×1) implementation (shown as asterisks):

The sizes n_2 of the trailing triangle determined by our switch point selection algorithm are shown for each matrix in Table 6.1. The heuristic does select a reasonable switch point—yielding true non-zero densities of 85% or higher in the trailing triangle—in all cases except

[4] In modeling the call to TRSV, we used the empirically estimated register block sizes of $R = 4$ on the Power3 and Itanium platforms, and $R = 18$ on the Ultra 2i platform. We used the vendor-supplied TRSV on the Power3 and Itanium. On the Ultra 2i, we used the ATLAS generated TRSV, which uses a recursive implementation [12] and 4×8 blocking at the base case.

Matrix 6 (`goodwin`). Also, although Figures 6.7–6.9 show the performance using the best register block size, the heuristics described in Section 6.1.3 chose the optimal block size in all cases on all platforms *except* for Matrix 2 (`memplus`) running on the Ultra 2i and Itanium. Nevertheless, the performance (Mflop/s) at the sizes chosen in these cases was within 6% of the best.

The main high-level observations are as follows:

- *The best implementations achieve speedups of up to* 1.8× *over the reference implementation, and between 75%–95% of the upper bound.* We conclude that additional performance improvements from low-level tuning will be limited, just as with SpMV (Chapter 4).
- *Most of the performance improvement comes from the switch-to-dense optimization*, with a generally relatively modest benefit from register blocking.

We elaborate on these points in the following discussion.

The best implementations achieve speedups of up to 1.8 over the reference implementation. Furthermore, they attain a significant fraction of the upper bound performance (Mflop/s). On the Ultra 2i, the implementations achieve 75% up to 85% of the upper bound performance; on the Itanium, 80–95%; and about 80–85% on the Power3. On the Itanium in particular, we observe performance that is very close to the estimated bounds. The vendor implementation of TRSV evidently exceeds our bounds. We are currently investigating this phenomenon. We know that the compiler (and, it is likely, the vendor TRSV) uses prefetching instructions. If done properly, we would expect this to invalidate our charging for the full latency cost in equation (6.7), allowing us to move data at rates approaching memory bandwidth instead.

In two cases—Matrix 5 (`raefsky4`) on the Ultra 2i and Itanium platforms—the combined effect of register blocking and the switch-to-dense call significantly improves on either optimization alone. On the Ultra 2i, register blocking alone achieves a speedup of 1.29, switch-to-dense achieves a speedup of 1.48, and the combined implementation yields a speedup of 1.76. On the Itanium, the register blocking-only speedup is 1.24, switch-to-dense-only is 1.51, and combined speedup is 1.81.

However, register blocking alone generally does not yield significant performance gains for the other matrices. In fact, on the Power3, register blocking has almost no effect,

whereas the switch-to-dense optimization performs very well. We observed that Matrices 3 (`wang4`), 4 (`ex11`), 6 (`goodwin`), and 7 (`lhr10`), none of which benefit from register blocking, all have register blocking fill ratios exceeding 1.35 when using the smallest non-unit block size, 2×2. The other matrices have fill ratios of less than 1.1 with up to 3×3 blocking. Two significant factors affecting the fill are (1) the choice of square block sizes and (2) imposition of a uniform grid. Non-square block sizes and the use of variable block sizes may be viable alternatives to the present scheme.

Furthermore, register blocking does not seem to work at all on the Power3. Recall that register blocking also did not yield performance close to upper bounds for SpMV on the Power3 (see Chapter 4). For SpTS, we see that the switch-to-dense optimization achieves much better performance, again suggesting the structural assumptions of register blocking do not hold for triangular solve.

Note that our upper bounds are computed with respect to our particular register blocking and switch-to-dense data structure. It is possible that other data structures (*e.g.*, those that remove the uniform block size assumption and therefore change the dependence of f_{rc} on b) could do better.

6.4 Related Work

Sparse triangular solve is a key component in many of the existing serial and parallel direct sparse factorization libraries (*e.g.*, SuperLU [94], MUMPS [11], UMFPACK [91], PSPASES [178], and SPOOLES [22], among others [171, 98]). These libraries have focused primarily on speeding up the factorization step, and employ sophisticated methods for creating dense structure. Efforts to speedup the triangular solve step in these software systems, among other studies [265, 264, 179, 144, 208, 273, 151, 9, 256], have focused on improving parallel scalability, whereas we address uniprocessor tuning exclusively here. As far as we know, our performance model of triangular solve is unique, though in fact it is essentially a modest adaptation of the same model for SpMV (Chapter 4).

Refer to the discussion of Chapter 5 for additional discussion of related work in sparse compilers.

6.5 Summary

The performance of our implementations approaches upper-bounds on a variety of architectures, suggesting that additional gains from low-level tuning (*e.g.*, instruction scheduling) will be limited. This observation confirms our findings for SpMV, and furthermore motivates the development of additional algorithmic techniques to improve reuse opportunities, for instance, via the use of multiple right-hand sides (Chapter 5) [231, 117, 167], and the use of higher-level kernels discussed in Chapter 7.

Register blocking with square blocks on a uniformly aligned grid appears to be too limiting to see appreciable performance benefits. Encouraged by the gains from the switch-to-dense algorithm, we expect variable blocking or more intelligently guided use of dense structures (*e.g.*, using elimination trees) to be promising future directions. Existing direct solvers make use of such higher-level information in their forward and backward substitution phases, and comparisons to these implementations is needed.

The success of the switch-to-dense optimization on our test problems may be particular to our choice of fill-reducing ordering. Other contexts which give rise to triangular matrices include (a) other ordering schemes, and (b) incomplete Cholesky and incomplete LU factorization for preconditioning. Relevant reviews of these contexts can be found elsewhere [71, 92].

At present, we treat selection of the dense triangle as a property of the matrix only, and not of the architecture. Whether there is any benefit to making this platform-dependent as well, *e.g.*, by off-line profiling of performance as a function of density, is an opportunity for future investigation.

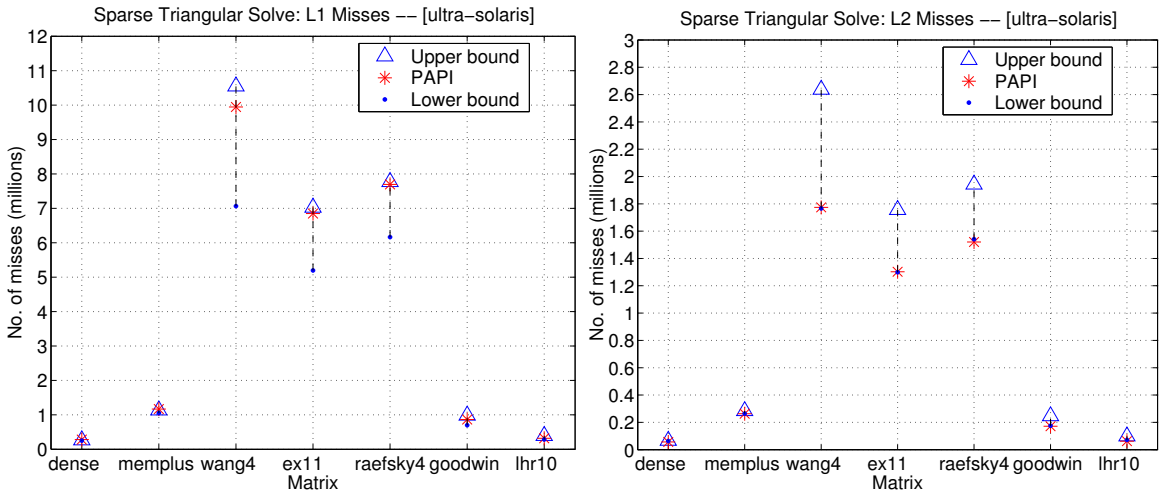

Figure 6.4: **SpTS miss model validation (Sun Ultra 2i)**. Our upper and lower bounds on L1 and L2 cache misses compared to PAPI measurements. The bounds match the data well. The true L2 misses match the lower bound well in the larger (L2) cache, suggesting the vector sizes are small enough that conflict misses play a relatively minor role.

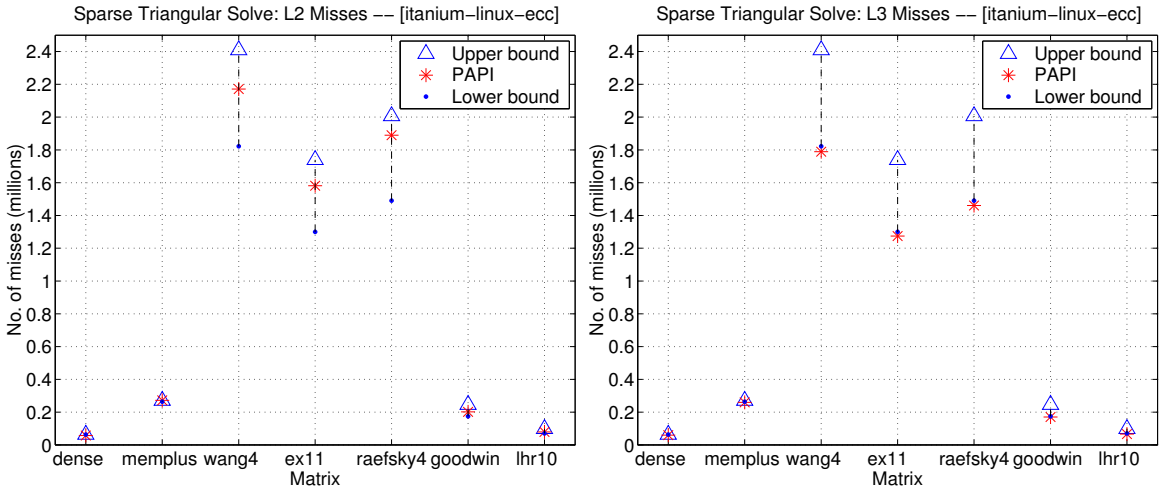

Figure 6.5: **SpTS miss model validation (Intel Itanium)**. Our upper and lower bounds on L1 and L2 cache misses compared to PAPI measurements. The bounds match the data well. As with Figure 6.4, Equation (6.10) is a good match to the measured misses for the larger (L3) cache.

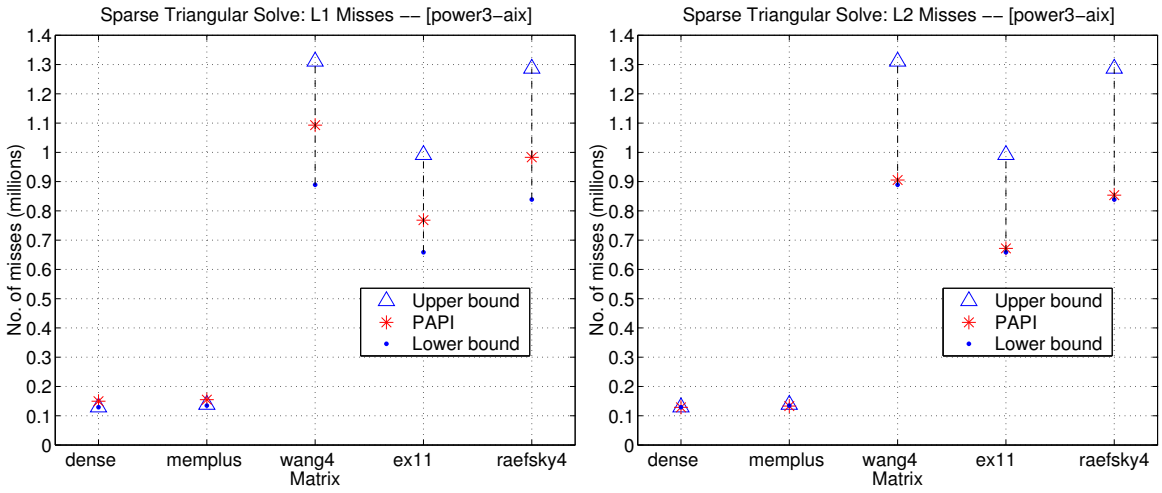

Figure 6.6: **SpTS miss model validation (IBM Power3)**. Our upper and lower bounds on L1 and L2 cache misses compared to PAPI measurements. Note that two matrices have been omitted since they fit approximately within the large (8 MB) L2 cache.

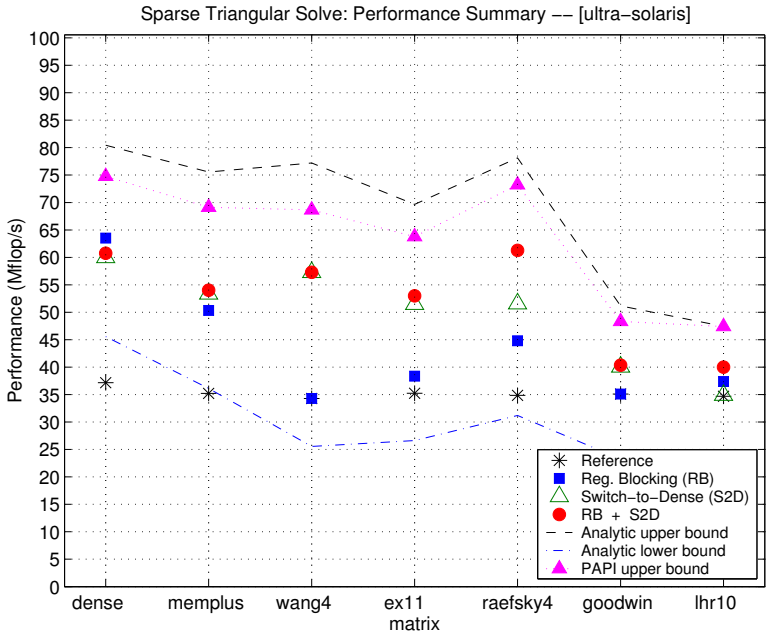

Figure 6.7: **Sparse triangular solve performance summary (Sun Ultra 2i)**. Performance (Mflop/s) shown for the seven items listed in Section 6.3.2. The best codes achieve 75–85% of the performance upper bound.

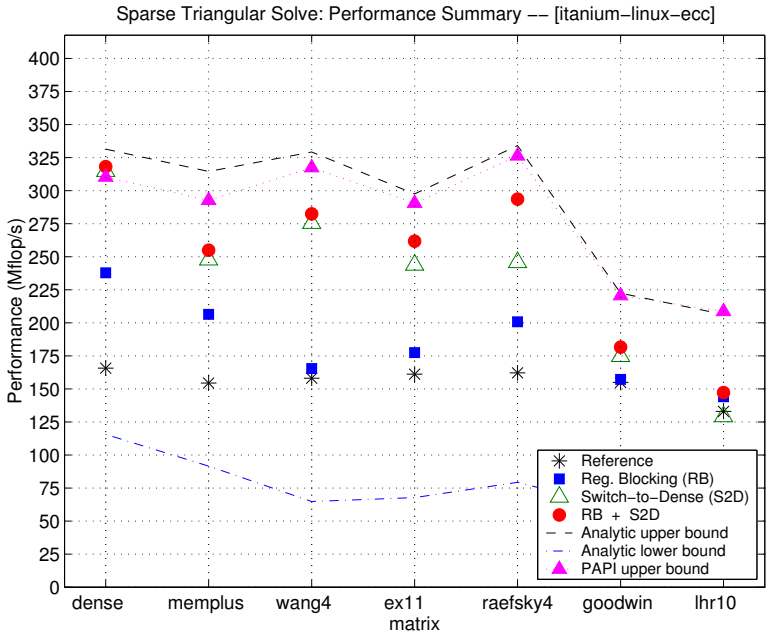

Figure 6.8: **Sparse triangular solve performance summary (Intel Itanium)**. Performance (Mflop/s) for the seven implementations listed in Section 6.3.2. The best implementations achieve 85–95% of the upper bound.

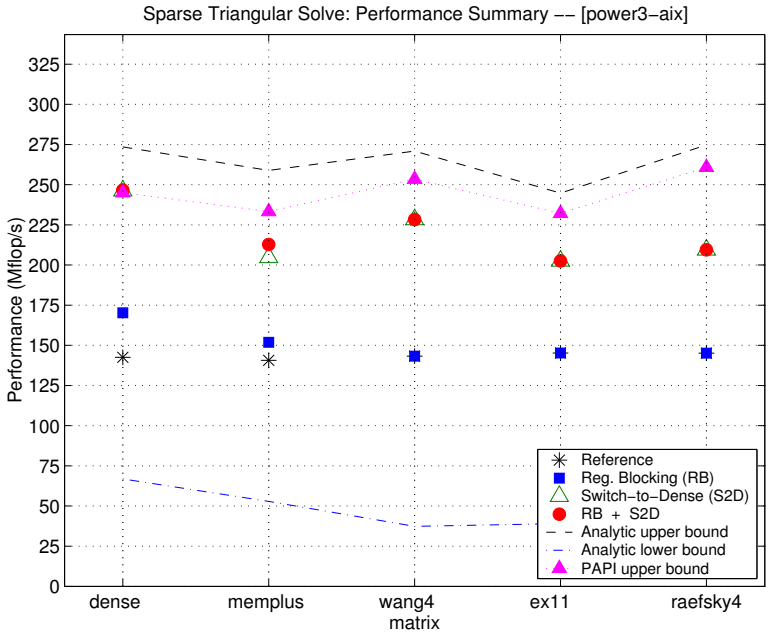

Figure 6.9: **Sparse triangular solve performance summary (IBM Power3)**. Performance (Mflop/s) of our SpTS implementations relative to various performance bounds (see Section 6.3.2). Performance improvements come mostly from switching-to-dense.

Chapter 7

Higher-Level Sparse Kernels

Contents

Among the fundamental limits on the performance of kernels like sparse matrix-vector multiply (SpMV) and sparse triangular solve (SpTS) is simply the time to read the matrix: the elements of A enjoy no temporal reuse when we treat these kernels as black box routines, cannot exploit multiple vectors, or cannot exploit knowledge about the matrix values (*e.g.*, symmetry). To achieve still higher performance, this chapter considers "higher-level" sparse kernels in which elements of the sparse matrix A can be reused. Our primary focus is on the kernel $y \leftarrow y + A^TA\cdot x$, or sparse $A^TA\cdot x$ (SpA^TA).[1] We also present preliminary findings when applying sparse powers of a matrix, *i.e.*, $y \leftarrow A^\rho\cdot x$, where the integer $\rho \geq 2$.

[1] We restrict our attention to SpA^TA here, though the same ideas apply to the computation of $AA^T\cdot x$.

For SpA^TA, we present a simple *cache interleaved* implementation in which we also apply the tuning ideas developed for SpMV in prior chapters. We show speedups between 1.5–4.2× over a reference implementation which computes $t \leftarrow A \cdot x$ and $y \leftarrow y + A^T \cdot t$ as separate steps, where A is stored in compressed sparse row (CSR) format. Furthermore, even if each of these steps is tuned by register-level blocking (Chapter 3) with an optimal choice of block size, our implementations are still up to 1.8× faster.

We adapt the performance upper bounds model of Chapter 4 to SpA^TA. We find that the performance of our implementations typically achieves 50–80% of the bound, a lower fraction than what we observe in the cases of SpMV and SpTS. This result suggests that future work could fruitfully apply automatic low-level tuning methods, in the spirit of automatic low-level tuning systems for dense linear algebra such as PHiPAC [46] and ATLAS [325], to improve further the performance of SpA^TA.

SpA^TA appears in a variety of problem contexts, including the inner-loop of interior point methods for mathematical programming problems [320], algorithms for computing the singular value decomposition [93], and Kleinberg's HITS algorithm for finding hubs and authorities in graphs [191], among others. Thus, our results will be immediately relevant to a number of important application domains.

We close this chapter by presenting preliminary results for another sparse kernel with potential opportunities to reuse elements of A: computing sparse $A^\rho \cdot x$. The basic optimization we apply is *serial sparse tiling*, proposed by Strout, *et al.*, in the case when A corresponds to application of a Gauss-Seidel smoothing operator [288]. Here, we review the method for general A, and demonstrate the potential speedups when the technique is combined with register blocking. Although these early results are encouraging, important questions about when (*i.e.*, on what matrices and platforms) and how best to apply and tune the method remain unresolved.

The material on SpA^TA originally appeared in a recent paper [317], and also summarizes the key findings of an extensive technical report [318].

7.1 Automatically Tuning $A^TA \cdot x$ for the Memory Hierarchy

We assume a baseline implementation of the sparse $A^TA \cdot x$ (SpA^TA) that first computes $t \leftarrow A \cdot x$ followed by $y \leftarrow y + A^T \cdot t$. For large matrices A, this implementation brings A through the memory hierarchy twice. However, we can compute $A^TA \cdot x$ by reading A from

main memory only once. Denote the rows of A by $a_1^T, a_2^T, \ldots, a_m^T$. Then, the operation $A^TA \cdot x$ can be expressed algorithmically as follows:

$$A^TA \cdot x = (a_1 \ldots a_m) \begin{pmatrix} a_1^T \\ \ldots \\ a_m^T \end{pmatrix} x = \sum_{i=1}^{m} a_i(a_i^T x). \tag{7.1}$$

That is, for each row a_i^T, we can compute the dot product $t_i = a_i^T x$, followed by an accumulation of the scaled vector $t_i a_i$ into y—thus, the row a_i^T is read from memory into cache to compute the dot product, assuming sufficient cache capacity, and then reused on the accumulate step. We refer to Equation (7.1) as the *cache interleaved* implementation of SpA^TA.

Moreover, we can take each a_i^T to be a *block of rows* instead of just a single row. Doing so allows us to apply cache interleaving on any of the block row-oriented formats described in preceeding chapters, such as the block compressed sparse row (BCSR) format used in register blocking as described in Chapter 3, or the row segmented diagonal (RSDIAG) format presented in Chapter 5. In this chapter, we only consider combining cache interleaving with register blocking to demonstrate the potential performance gains. The code for a cache interleaved, 2×2 register blocked implementation of sparse matrix-vector multiply (SpMV) appears in Figure 7.1.

The SPARSITY Version 2 heuristic for selecting the register block size, $r \times c$, can be adapted to SpA^TA in a straightforward way. The heuristic consists of 3 steps.

1. We collect a one-time *register profile* to characterize the platform. We evaluate the performance (Mflop/s) of the register blocked SpA^TA for all block sizes up to some limit on a dense matrix stored in BCSR format. These measurements are independent of the sparse matrix, and therefore only need to be made once per architecture.

2. When the matrix is known (in general, not until run-time), we estimate the *fill ratio* for all block sizes. Recall that the fill ratio is defined to be the number of stored non-zeros (including explicit zeros needed to pad the $r \times c$ BCSR data structure) divided by the number of true non-zeros. Refer to Chapter 3 for a detailed discussion of the trade-offs between fill, storage, and performance.

3. We select the block size $r \times c$ that maximizes

$$\text{Estimated Mflop/s} = \frac{\text{Mflop/s on a dense matrix in } r \times c \text{ BCSR}}{\text{Estimated fill ratio for } r \times c \text{ blocking}} \quad . \tag{7.2}$$

For a discussion of the overheads of executing the heuristic and converting the matrix to BCSR, see Chapter 3.

7.2 Upper Bounds on $A^TA \cdot x$ Performance

Our bounds for the cache-optimized, register blocked implementations of SpA^TA (as described in Section 7.1) are based on bounds developed for SpMV in Chapter 4. To derive upper bounds, we make the following guiding assumptions:

1. SpA^TA is memory bound since most of the time is spent streaming through matrix data. Thus, we bound time from below by considering only the cost of *memory* operations. Furthermore, we assume write-back caches (true of the platforms considered in this dissertation) and sufficient store buffer capacity so that we can consider only loads and ignore the cost of stores.

2. Our model of execution time assigns an empirically derived costs to accesses at each level of the memory hierarchy. Refer to Section 4.2.1 for more information on how we obtain these *effective cache access latencies*.

3. As shown below in Equation (7.5), we further bound time from below by computing a lower bound on cache misses. Our bound considers only compulsory and capacity misses, and *ignores conflict misses*. (Recall that for SpMV, capacity misses were also ignored.) We account for cache capacity and line size but assume full associativity.

4. We do not consider the cost of TLB misses. Since operations like SpA^TA, SpMV, and sparse triangular solve (SpTS) essentially spend most of their time streaming through the matrix using stride 1 accesses, there are always very few TLB misses. (We have verified this experimentally using hardware counters.)

We use the notation of Chapter 4. Let the total time of SpA^TA be T seconds. Then, the performance P in Mflop/s is

$$P = \frac{4k}{T} \times 10^{-6} \tag{7.3}$$

where k is the number of non-zeros in the $m \times n$ sparse matrix A, *excluding* explicitly filled in zeros.[2] To get an *upper bound on performance*, we need a *lower bound* on T. We present

[2] That is, T is a function of the machine architecture and data structure, so we can fairly compare different values of P for fixed A and machine.

```
    void spmv_bcsr_2x2_ata( int mb, const int* ptr, const int* ind,
                            const double* val,
                            const double* x, double* y, double* t )
    {
        int i;

        /* for each block row i of A */
 1      for( i = 0; i < mb; i++, t += 2 )
        {
            int j;
 2          register double t0 = 0, t1 = 0;
 3          const int* ind_t = ind;
 4          const double* val_t = val;

            /* compute (block row of A) times x */
 5          for( j = ptr[i]; j < ptr[i+1]; j++, ind_t++, val_t += 2*2 )
            {
 6              t0 += val_t[0*2+0] * x[ind_t[0]+0];
 7              t1 += val_t[1*2+0] * x[ind_t[0]+0];
 8              t0 += val_t[0*2+1] * x[ind_t[0]+1];
 9              t1 += val_t[1*2+1] * x[ind_t[0]+1];
            }

10          t[0] = t0;
11          t[1] = t1;

            /* compute y <-- (block row of A)^T times t */
12          for( j = ptr[i]; j < ptr[i+1]; j++, ind++, val += 2*2 )
            {
13              double* yp = y + ind[0];
14              register double y0 = 0, y1 = 0;

15              y0 += val[0*2+0] * t0;
16              y1 += val[0*2+1] * t0;
17              y0 += val[1*2+0] * t1;
18              y1 += val[1*2+1] * t1;

19              yp[0] += y0;
20              yp[1] += y1;
            }
        }
    }
```

Figure 7.1: **Cache-optimized, 2×2 sparse $A^TA \cdot x$ implementation**. Here, A is stored in 2×2 BCSR format, where A has `2*mb` rows.

our lower bound on T, which incorporates Assumptions 1 and 2, in Section 7.2.1, below. Our expression for T in turn uses lower bounds on cache misses (Assumption 3) described in Section 7.2.2.

7.2.1 A latency-based execution time model

We model execution time by counting only the cost of memory accesses. Consider a machine with κ cache levels, where the access latency to the L_i cache is α_i seconds, and the memory access latency is α_{mem}. Suppose SpA^TA executes H_i cache accesses (or cache hits) and M_i cache misses at each level i, and that the total number of loads is Loads. We charge α_i for each access to cache level i; thus, the execution time T, ignoring the cost of non-memory operations, is

$$T = \sum_{i=1}^{\kappa} \alpha_i H_i + \alpha_{\text{mem}} M_\kappa \tag{7.4}$$

$$= \alpha_1 \text{Loads} + \sum_{i=1}^{\kappa-1} (\alpha_{i+1} - \alpha_i) M_i + \alpha_{\text{mem}} M_\kappa \tag{7.5}$$

where Equations (7.4) and (7.5) are equivalent since $H_1 = \text{Loads} - M_1$ and $H_i = M_{i-1} - M_i$ for $2 \leq i \leq \kappa$. According to Equation (7.5), we can minimize T by minimizing M_i, assuming $\alpha_{i+1} \geq \alpha_i$. In Section 7.2.2, we give expressions for Loads, M_i to evaluate Equation (7.5).

7.2.2 A lower bound on cache misses

Following Equation (7.5), we obtain a lower bound on M_i for SpA^TA by counting compulsory and capacity misses but ignoring conflict misses. Our bound is a function of the cache configuration and matrix data structure.

Let C_i be the size of each cache i in double-precision words, and let l_i be the line size, in doubles, with $C_1 \leq \ldots \leq C_\kappa$, and $l_1 \leq \ldots \leq l_\kappa$. Suppose γ integer indices use the same storage as 1 double.[3] To get lower bounds, assume full associativity and complete user-control over how data is placed in cache.

Recall the notation of Chapter 3 for describing the $r \times c$ BCSR data structure for the $m \times n$ sparse matrix A which has k non-zeros. For simplicity, assume r divides m and c divides n. Let K_{rc} be the number of $r \times c$ blocks, and $f_{rc} = \frac{K_{rc} \cdot rc}{k}$ be the fill ratio. Let

[3] For all the machines in this study, we use 32-bit integers; thus, $\gamma = 2$.

$\hat{k} = \hat{k}(r,c) = K_{rc} \cdot rc$ be the number of *stored values*, *i.e.*, including fill. Then, the total number of loads is $\text{Loads} = \text{Loads}_A + \text{Loads}_x + \text{Loads}_y$, where

$$\text{Loads}_A = 2\left(\hat{k} + \frac{\hat{k}}{rc}\right) + \frac{m}{r} \qquad \text{Loads}_x = \frac{\hat{k}}{r} \qquad \text{Loads}_y = \frac{\hat{k}}{r}. \tag{7.6}$$

Loads_A contains terms for the values, block column indices, and row pointers, and the factor of 2 accounts for reading A twice: once to compute $A \cdot x$, and once for A^T times the result (see Figure 7.1, lines 6–9 and 15–18). The number of row pointers is really $\frac{m}{r} + 1$, which we approximate by $\frac{m}{r}$ here under the reasonable assumption that $\frac{m}{r} \gg 1$. Loads_x and Loads_y are the total number of loads required to read x and y, where we load c elements of each vector for each of the $\frac{\hat{k}}{rc}$ blocks (Figure 7.1, lines 6–9 and 15–18).

We must account for the amount of data, or *working set*, required to multiply by a block row and its transpose in order to model capacity misses correctly. For the moment, assume that all block rows have the same number of $r \times c$ blocks; then, each block row has $\frac{\hat{k}}{rc} \times \frac{r}{m} = \frac{\hat{k}}{cm}$ blocks. We define the *matrix working set*, $\hat{W}$, to be the size of matrix data for a block row:

$$\hat{W} = \frac{\hat{k}}{m} r + \frac{1}{\gamma}\frac{\hat{k}}{cm} + \frac{1}{\gamma}$$

The total size of the matrix data in doubles is $\frac{m}{r}\hat{W}$. Similarly, we define the *vector working set*, $\hat{V}$, to be the size of the corresponding vector elements for x and y:

$$\hat{V} = 2\frac{\hat{k}}{m}$$

i.e., there are $\frac{\hat{k}}{m}$ non-zeros per row, each of which corresponds to a vector element to be reused within a block row; the factor of 2 counts both x and y elements.

The following is a lower bound on the L_i cache misses, $M_{\text{lower}}^{(i)} \leq M_i$:

$$M_{\text{lower}}^{(i)} = \frac{1}{l_i}\left[\frac{m}{r}\hat{W} + 2n + \frac{m}{r} \cdot \max\{\hat{W} + \hat{V} - C_i, 0\}\right] \quad . \tag{7.9}$$

We derive this lower bound in detail in Appendix I.1.

To see that Equation (7.9) is reasonable, consider two limiting cases, assuming $l_i = 1$ for simplicity. First, when the entire working set fits in cache, $\hat{W} + \hat{V} \leq C_i$ and Equation (7.9) simplifies to just the compulsory misses, $\frac{m}{r}\hat{W} + 2n$. Second, when the working set is much greater than the size of the cache, or $\hat{W} + \hat{V} \gg C_i$, then all accesses miss: $M_{\text{lower}}^{(i)} \approx 2\frac{m}{r}\hat{W} + 2n + \frac{m}{r}\hat{V}$. This expression includes 2 reads of the matrix ($2\frac{m}{r}\hat{W}$) and a miss on every vector access.

The factor of $\frac{1}{l_i}$ in Equation (7.9) optimistically assumes we will incur only 1 miss per cache line in the best case. To mitigate the effect of this assumption, we could refine these bounds by taking $\hat{W}$ and $\hat{V}$ to be functions of the non-zero structure of each block row, though we do not do so here.

7.3 Experimental Results and Analysis

Below, we present an experimental validation of the cache miss bounds model described in Section 7.2.2, and an experimental evaluation of our cache-optimized, register-blocked implementations of SpA^TA with respect to the upper bounds described in Section 7.2.1. These experiments were conducted following the methodology outlined in Appendix B, on 44 matrices and the following 4 platforms: Ultra 2i, Pentium III, Power3, and Itanium 1. (On each platform, matrices small relative to the size of the largest cache have been omitted to avoid reporting inflated performance results.) Actual cache misses were measured using the PAPI hardware counter library v2.3 [60].

To execute the SPARSITY Version 2 heuristic for SpA^TA, we used the register profiles shown in Figures 7.2–7.3. This benchmarking data, the one-time machine characterization used in step 1 of the heuristic (Section 7.1), shows the performance of cache-optimized, $r\times c$ register blocked SpA^TA for a dense matrix stored in sparse format. Block sizes up to 8×8 are shown. As with similar data for SpMV in Chapter 3, we see a dramatic variation in performance as a function of the platform.

7.3.1 Validation of the cache miss model

Figures 7.4–7.5 compares the load and cache miss counts given by our model, Equations (7.6)–(7.9), to those observed using PAPI. We measured the performance (Mflop/s) for all block sizes to determine empirically the best block size, $r_{\text{opt}}\times c_{\text{opt}}$, for each matrix and platform. Figures 7.4–7.5 show, at the matrix- and machine-dependent block size $r_{\text{opt}}\times c_{\text{opt}}$, the following:

- The ratio of measured load operations to the loads predicted by Equation (7.6) (shown as solid squares).
- The ratio of measured L_1, L_2, and L_3 cache misses to the lower bound, Equation (7.9) (shown as circles, asterisks, and ×s, respectively).

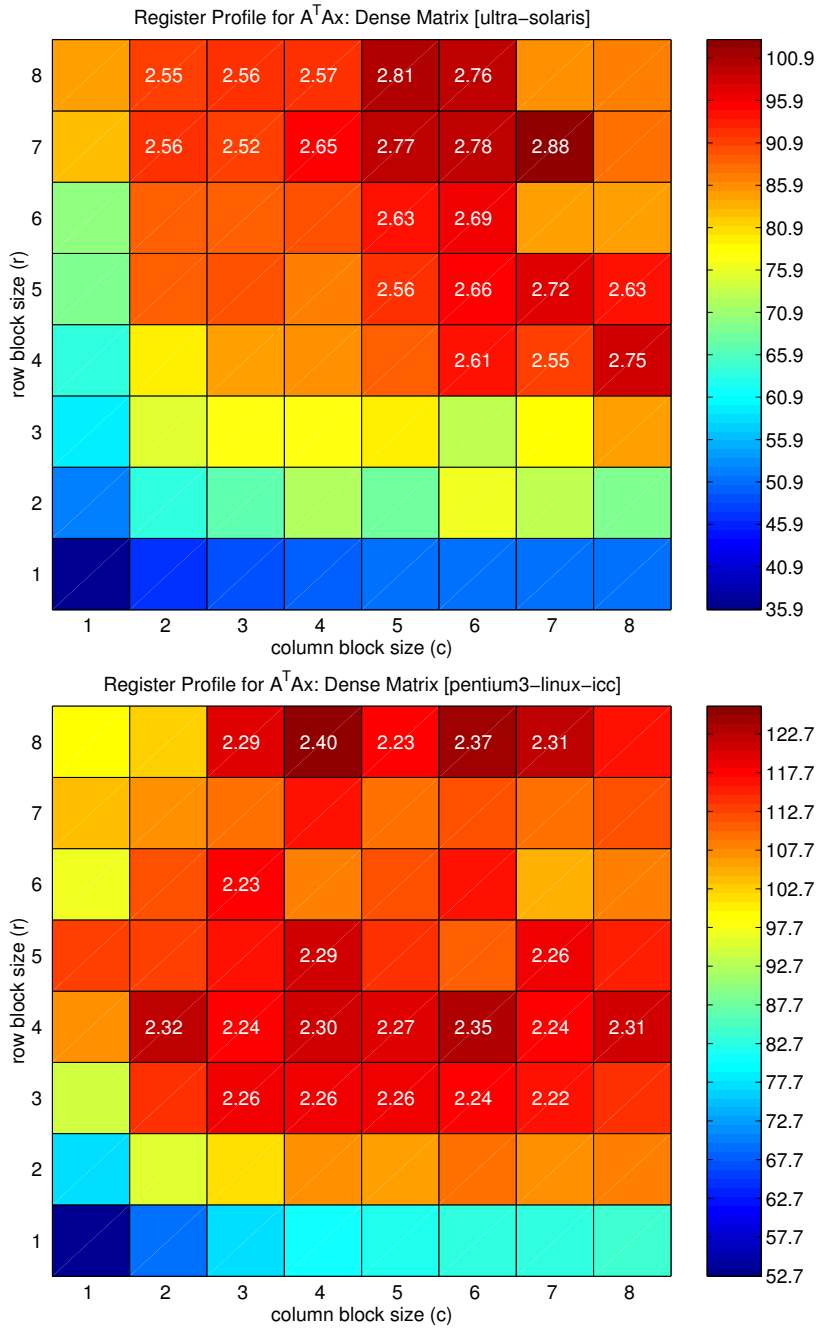

Figure 7.2: **Cache-optimized, register blocked $A^TA{\cdot}x$ performance profiles (off-line benchmarks) capture machine-dependent structure: Ultra 2i and Pentium III**. We show the performance of the cache-optimized, register blocked code on a dense matrix stored in sparse $r{\times}c$ format, for all $r{\times}c$ up to 8×8. Each square is an implementation, shaded by its performance (Mflop/s) and labeled by its speedup over the unblocked (1×1), cache-optimized code. (*Top*) Profile for the Ultra 2i. (*Bottom*) Pentium III.

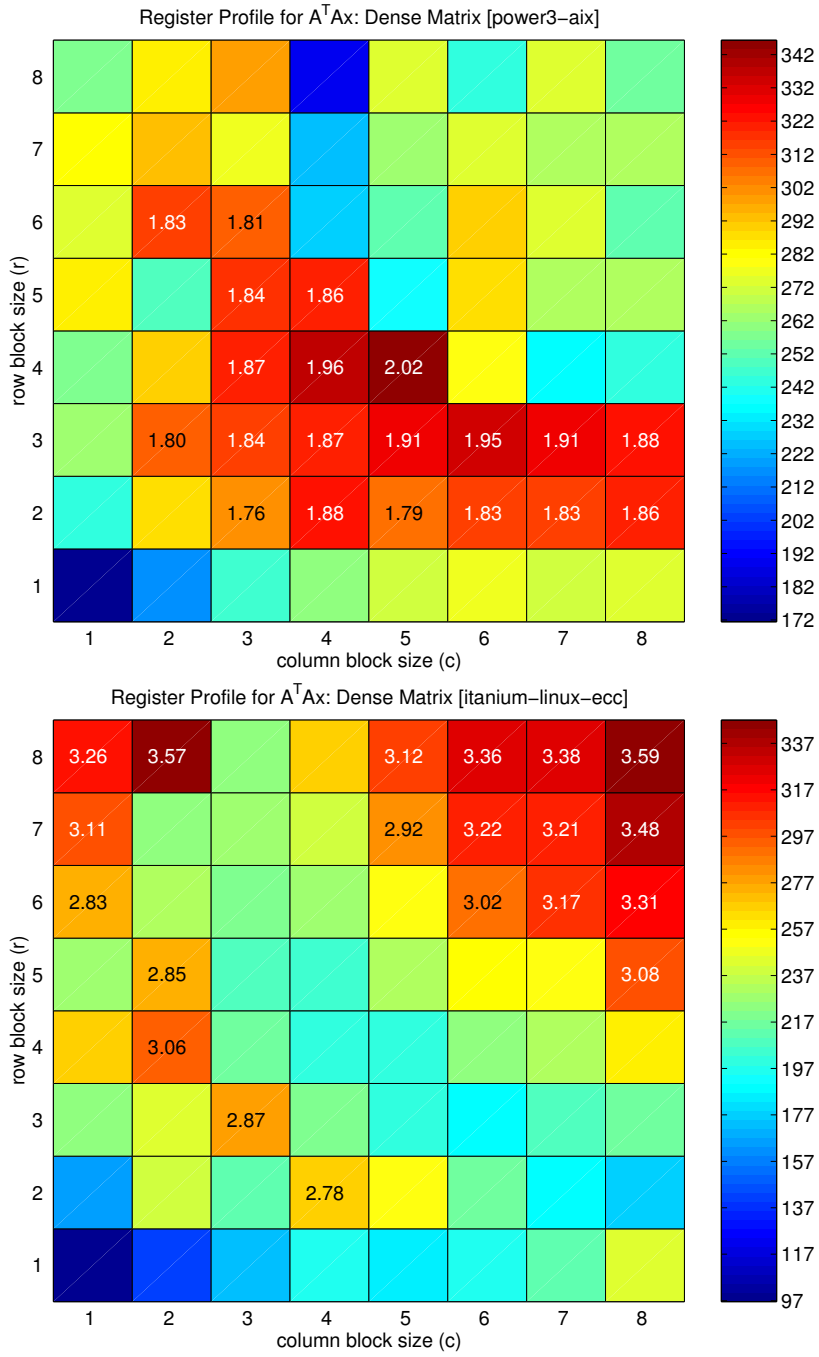

Figure 7.3: **Cache-optimized, register blocked $A^TA \cdot x$ performance profiles (off-line benchmarks) capture machine-dependent structure: Power3 and Itanium 1**. We show the performance of the cache-optimized, register blocked code on a dense matrix stored in sparse $r \times c$ format, for all $r \times c$ up to 8×8. Each square is an implementation, shaded by its performance (Mflop/s) and labeled by its speedup over the unblocked (1×1), cache-optimized code. (*Top*) Profile for the Power3. (*Bottom*) Itanium 1.

Furthermore, for each category (*i.e.*, loads, L_i misses), we show the median ratio as a dashed horizontal line. Since our model is indeed a lower bound, all ratios are at least 1; if our model exactly predicted reality, then all ratios would equal 1. We observe the following:

1. L_2 and L_3 cache miss counts tend to be very accurate: the observed counts are typically within 5–10% of the lower bound, indicating that the cache capacities are sufficient to justify ignoring conflict misses at these levels.

2. The ratio of observed L_1 miss counts to the model is relatively high on the Ultra 2i (median ratio of 1.34×) and the Pentium III (1.23×), compared to the Power3 (1.16×) and Itanium (1.00×). One explanation is the lack of L_1 cache capacity, which causes more misses than predicted by our model. Though we account for capacity misses, we use a lower bound which assumes full associativity. (The L_1 cache on the Ultra 2i is direct-mapped, and 2-way on the Pentium III.) On the Itanium, although the L_1 size is the same as that on the Ultra 2i, less capacity is needed relative to the Ultra 2i because only integer data is cached in L_1. (Our bounds for Itanium account for this aspect of the cache architecture.)

3. On the Pentium III and Itanium, the observed load counts are high relative to the model. On the Pentium III, separate load and store counters were not available, so stores are included in the counts. Manually accounting for these stores yields the expected number of loads to within 10% when spilling does not occur (not shown). A secondary reason for high load counts on the Pentium III is that spilling occurs with a few of the implementations (as confirmed by inspection of the assembly code).

   On the Itanium, prefetch instructions (inserted by the compiler) are counted as loads by the hardware counter for load instructions. (By contrast, prefetches are also inserted by the IBM compiler, but are not counted as loads.)

4. On matrices 15 and 40–44 (linear programming), observed miss counts (particularly L_1 misses) tend to be much higher than for the other matrices. These matrices tend to have a much more random distribution of non-zeros than the others, and therefore our assumption of being able to exploit spatial locality fully (the $\frac{1}{l_i}$ factor in Equation (7.9)) does not hold. Thus, we expect the upper bound to be optimistic for these matrices.

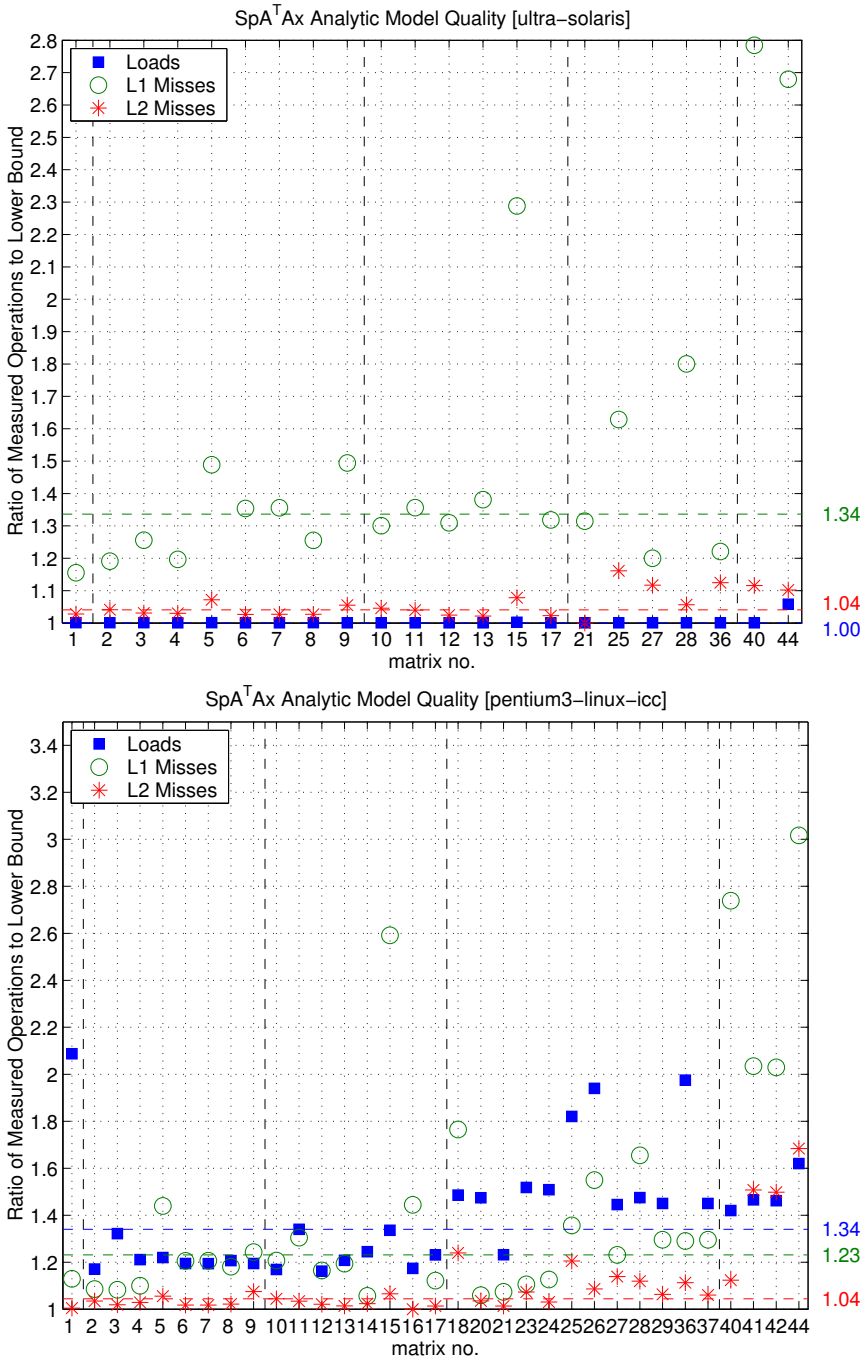

Figure 7.4: **Cache miss model validation: Ultra 2i and Pentium III**. We show the ratio (y-axis) of measured loads and cache misses to the counts predicted by our lower bound model, Equations (7.6)–(7.9), for each matrix (x-axis). (*Top*) Ultra 2i. (*Bottom*) Pentium III. The median of the ratios is shown as a dotted horizontal line, with its value labeled to the right of each plot.

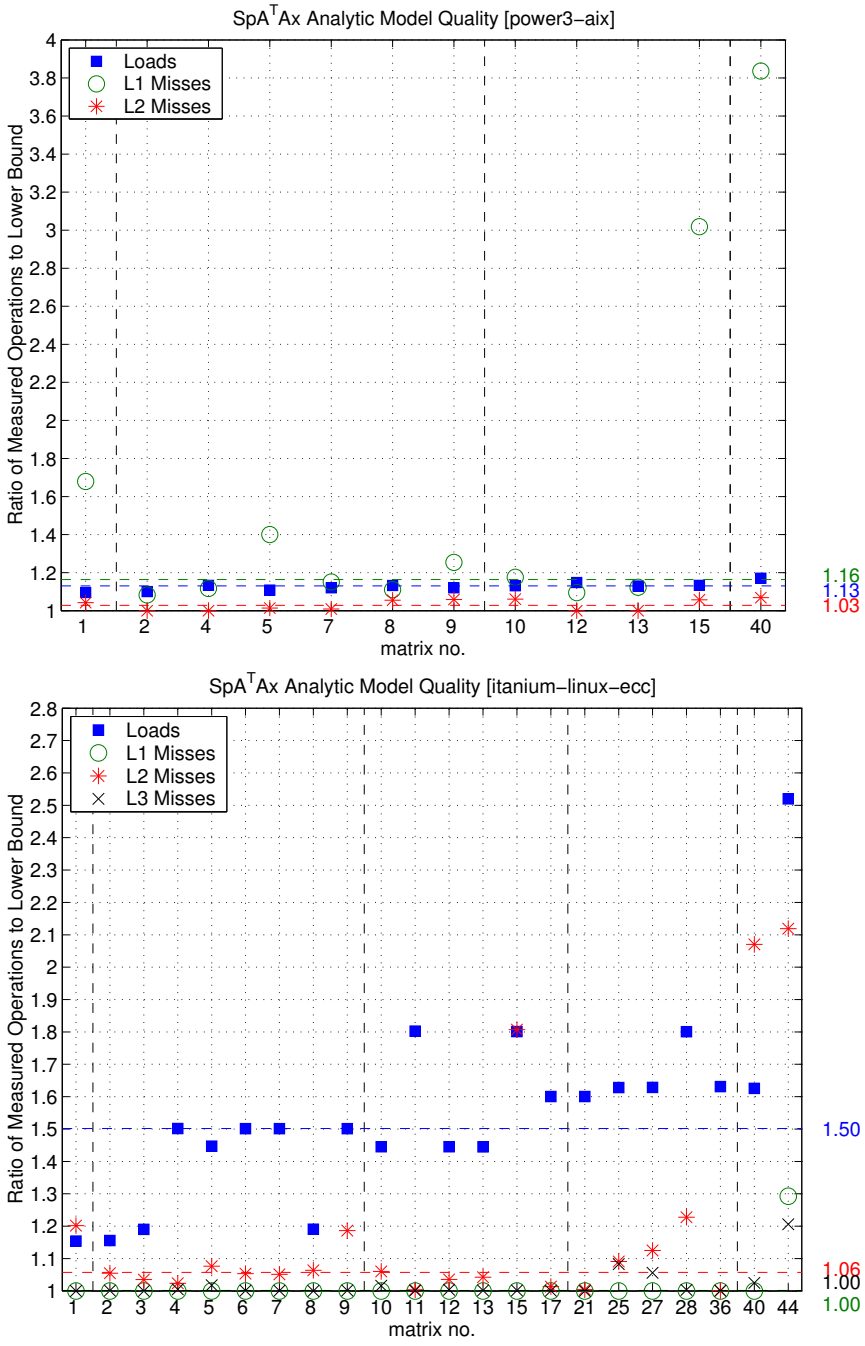

Figure 7.5: **Cache miss model validation: Power3 and Itanium 1**. We show the ratio (y-axis) of measured loads and cache misses to the counts predicted by our lower bound model, Equations (7.6)–(7.9), for each matrix (x-axis). (*Top*) Power3. (*Bottom*) Itanium 1. The Itanium has three levels of cache. The median of the ratios is shown as a dotted horizontal line, with its value labeled to the right of each plot.

In summary, we claim that the data show our lower bound cache miss estimates are reasonable, and that we are able to account for discrepancies based on both our modeling assumptions and our knowledge of each architecture.

7.3.2 Performance evaluation of our $A^TA \cdot x$ implementations

Figures 7.6–7.9 summarize the results of our performance evaluation results. We compare the performance (Mflop/s; y-axis) of the following for each matrix (x-axis):

- **Upper bound**, or analytic upper bound (shown as a solid line): This line shows the fastest (highest) value of our performance upper bound, Equations (7.3)–(7.9), over all $r \times c$ block sizes up to 8×8. We denote the block size shown by $r_{\text{up}} \times c_{\text{up}}$. To evaluate the our performance bounds, we use the cache parameters shown in Table 4.1 (see also Appendix B).

- **PAPI upper bound** (shown by triangles): The "PAPI upper bound" is also an upper bound, except that we substitute true loads and misses as measured by PAPI for Loads and M_i in Equation (7.5). In some sense, the PAPI bound is the true bound since misses are "modeled" exactly; the gap between the PAPI bound and the upper bound indicates how well Equations (7.6)–(7.9) reflect reality. The data points shown are for the same block size $r_{\text{up}} \times c_{\text{up}}$ used in the analytic upper bound.

  The block sizes ($r_{\text{up}} \times c_{\text{up}}$) used in the analytic and PAPI upper bounds are not necessarily the same as those used in Section 7.3.1. Nevertheless, the observations of Section 7.3.1 are qualitatively the same. We chose to use the best model bound in order to show the best possible performance expected, assuming ideal scheduling.

- **Best cache optimized, register blocked** implementation (squares): We implemented the optimization described in Section 7.1. These points show the best observed performance over all block sizes up to 8×8. We denote the block size shown by $r_{\text{opt}} \times c_{\text{opt}}$, which may differ from $r_{\text{up}} \times c_{\text{up}}$.

- **Heuristic cache optimized, register blocked** implementation (solid circles): These points show the performance of the cache optimized implementation using a register block size, $r_{\text{h}} \times c_{\text{h}}$, chosen by the heuristic.

- **Register blocking only** (diamonds and arrows): This implementation computes $t \leftarrow A \cdot x$ and $y \leftarrow A^T \cdot t$ as separate steps but with register blocking. The same block size, $r_{\text{reg}} \times c_{\text{reg}}$, is used in both steps, and the best performance over all block sizes up to 8×8 is shown.

  We also indicate the performance of each individual step using a blue arrow. The lowest point on the arrow (blue small solid dot) indicates the performance of just the transpose part ($A^T \cdot t$). The highest point on the arrow (blue small upward pointing solid triangle) shows the performance of just the non-transposed (or "normal") part ($A \cdot x$). In both cases, we use $2k$ flops. The transpose component was always slower than the normal component.

- **Cache optimization only** (shown by asterisks): This code implements the algorithmically cache optimized version of $\text{Sp}A^TA$ shown in Equation (7.1), but without any register-level blocking (*i.e.*, with $r = c = 1$).

- **Reference** implementation (×'s): The reference computes $t = Ax$ and $y = A^Tt$ as separate steps, with no register-level blocking.

Appendix I.2 show the values of $r_{\text{opt}} \times c_{\text{opt}}$, $r_{\text{h}} \times c_{\text{h}}$, and $r_{\text{reg}} \times c_{\text{reg}}$ used in Figures 7.6–7.9. We draw the following 5 high-level conclusions based on Figures 7.6–7.9.

1. *The cache optimization leads to uniformly good performance improvements.* Applying the cache optimization, even without register blocking, leads to speedups ranging from up to 1.2× on the Itanium and Power3 platforms, to just over 1.6× on the Ultra 2i and Pentium III platforms. This can be seen by comparing **Cache optimization only** to **Reference** in each plot. The speedups do not vary significantly across matrices, suggesting that this optimization is always worth trying.

2. *Register blocking and the cache optimization can be combined to good effect.* When the algorithmic cache blocking and register blocking are combined, we observe speedups from 1.2× up to 4.2× over the reference code. Furthermore, comparing the best combined implementation to register blocking only, we see speedups of up to 1.8×.

   The effect of combining the register blocking and the cache optimization is synergistic: the observed, combined speedup is *at least* the product (the register blocking only speedup) × (the cache-optimization only speedup), when $r_{\text{reg}} \times c_{\text{reg}}$ and $r_{\text{opt}} \times c_{\text{opt}}$

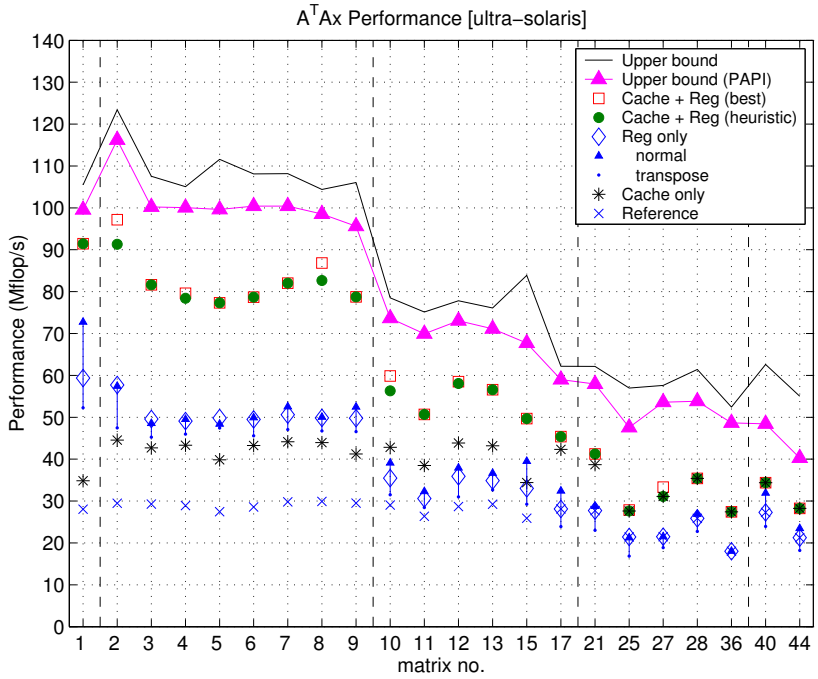

Figure 7.6: $A^TA \cdot x$ **performance on the Sun Ultra 2i platform.** A speedup version of this plot appears in Appendix I.3.

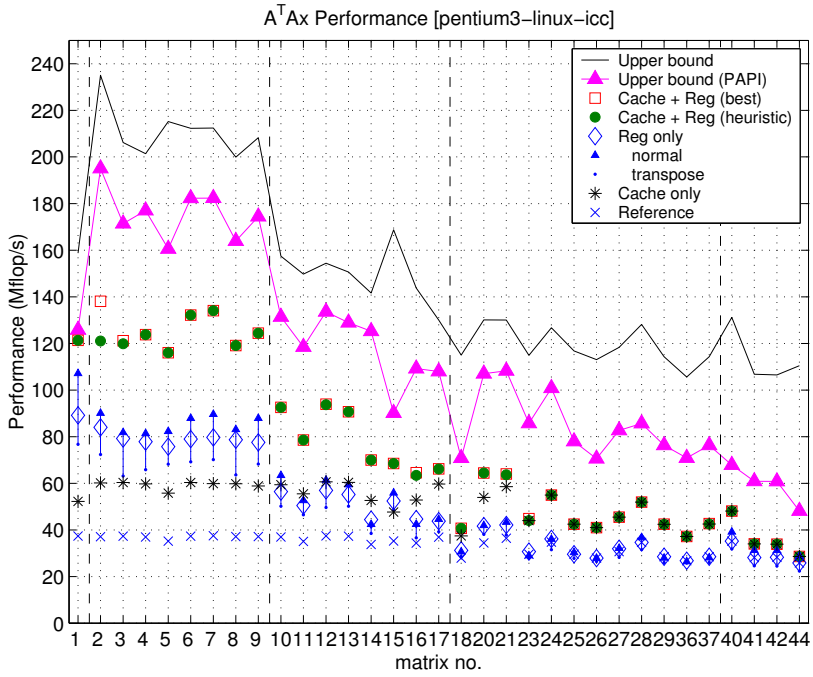

Figure 7.7: $A^TA \cdot x$ **performance on the Intel Pentium III platform.** A speedup version of this plot appears in Appendix I.3.

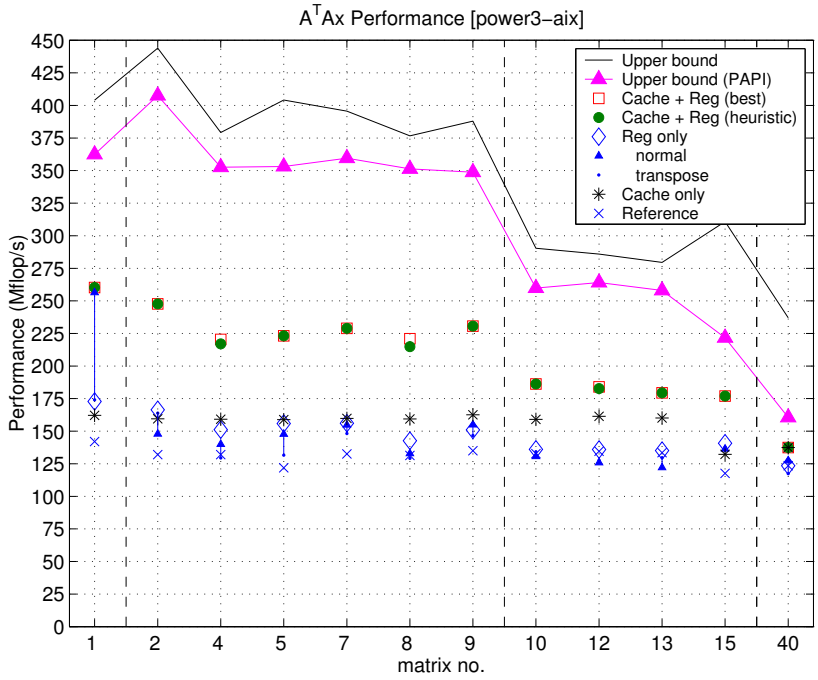

Figure 7.8: $A^TA \cdot x$ **performance on the IBM Power3 platform**. A speedup version of this plot appears in Appendix I.3.

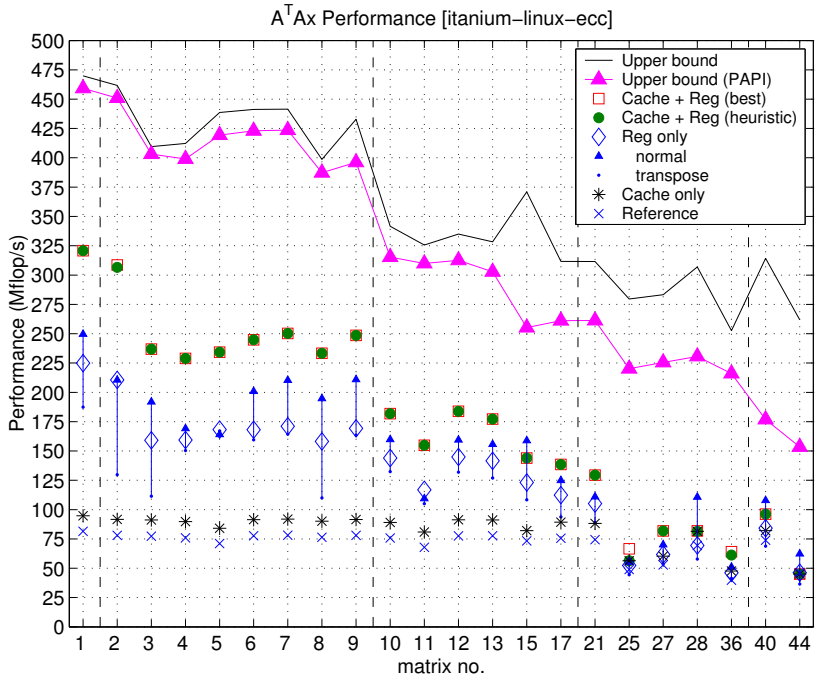

Figure 7.9: $A^TA \cdot x$ **performance on the Intel Itanium platform**. A speedup version of this plot appears in Appendix I.3.

match. Indeed, the combined speedup is greater than this ratio on the Ultra 2i, Power3, and Itanium platforms. In Appendix I.3, we show speedup versions of Figures 7.6–7.9 in order to make the claim of synergy explicit. Since cache interleaving places the matrix data in cache for the transpose multiply phase, one possible explanation for the synergistic effect is that the compilers on these three platforms schedule instructions for in-cache workloads better than out-of-cache workloads.

3. *Our heuristic always chooses a near-optimal block size.* Indeed, the performance of the block size selected by the heuristic is within 10% of the exhaustive best in all but four instances—in those cases, the heuristic performance is within 15% of the best.

   In Appendix I.2, we summarize this data in detail, showing the optimal block sizes for SpA^TA, both with and without the cache and register blocking optimizations. We also consider the case in which we use the optimal register blocking only block size, $r_{\text{reg}} \times c_{\text{reg}}$, with the cache optimization. On all platforms except the Power3, we find a number of cases in which the choice of $r_{\text{reg}} \times c_{\text{reg}}$ with the cache optimization is more than 10% worse than choosing the $r_{\text{opt}} \times c_{\text{opt}}$ block size predicted by our heuristic. Therefore, using a SpA^TA-specific heuristic leads to more robust block size selection.

4. *Our implementations are within 20–30% of the PAPI upper bound for FEM matrices, but within only about 40–50% on other matrices.* The gap between actual performance and the upper bound is larger than what we observed previously for SpMV and SpTS [316, 319]. This result suggests that a larger pay-off is expected from low-level tuning by, for instance, applying tuning techniques used in systems such as ATLAS/PHiPAC to further improve performance.

5. *Our analytic model of misses is accurate for FEM matrices, but less accurate for the others.* For the FEM matrices 1–17, the PAPI upper bound is typically within 10–15% of the analytic upper bound, indicating that our analytic model of misses is accurate in these cases. For the matrices 18–44, the gap between the analytic upper bound and the PAPI upper bound increases with increasing matrix number because our cache miss lower bounds assume maximum spatial locality in the accesses to x, indicated by the factor of $\frac{1}{l_i}$ in Equation (7.9). We discuss this effect in Section 7.3.1. FEM matrices have naturally dense block structure and can benefit from spatial locality; matrices with more random structure (*e.g.*, linear programming matrices 40–44) cannot. In

principle, we can refine our lower bounds to account for this by a more detailed examination of the non-zero structure.

The gap between the analytic and PAPI upper bounds is larger (as a fraction of the analytic upper bound) on the Pentium III than on the other three platforms. As discussed in Section 7.3.1, this is due to two factors: (1) we did not have separate counters for load and store operations, so we are charging for stores as well in the PAPI upper bound, and (2) in some cases, the limited number of registers on the Pentium III (8 registers) led to spilling in some implementations (confirmed by inspection of the load operation counts and inspection of the assembly code).

The interested reader may find additional, detailed discussion of these results in a recent technical report [318].

7.4 Matrix Powers: $A^{\rho} \cdot x$

The kernel $y \leftarrow A^{\rho} \cdot x$, which appears in simple iterative algorithms like the power method for computing eigenvalues [93], also has opportunities to reuse elements of A. Strout, *et al.*, proposed a *serial sparse tiling* algorithm for the case when $A^{\rho} \cdot x$ is the application of the Gauss-Seidel smoothing operator to x [288]. We review this algorithm for arbitrary A, describe a simple tiled compressed sparse row (TCSR) format data structure for storing A, and present the $A^{\rho} \cdot x$ kernel using this data structure (Section 7.4.1). We discuss some preliminary proof-of-principle experiments on various classes of matrices in Section 7.4.2. We find that encouraging speedups are possible, though important questions—such as deciding when and how to tile—remain unresolved.

7.4.1 Basic serial sparse tiling algorithm

The serial sparse tiling algorithm tiles $A^{\rho} \cdot x$ by partitioning a dependency graph representing the computation. Consider the case when A is a 7×7 tridiagonal matrix and $\rho = 2$. Let $t \leftarrow Ax$ and $y \leftarrow At$. Figure 7.10 shows a symbolic dependency graph of this computation, where the leftmost column of nodes represents the elements of x, the middle column represents t, and the rightmost column represents the y. The edges indicate the dependency structure, where each edge (v, w), labeled by (i, j), represents multiplication by the (i, j) element of A, *i.e.*, $w \leftarrow w + a_{i,j} \cdot v$. For example, to compute the final values y_0 and y_1, shaded in red,

requires all the matrix and vector elements (edges and nodes) that are also shaded red. A subset S of elements in y defines a *tile*, which is the collection of nodes and edges obtained by tracing backwards in the graph starting the nodes in S to find all paths that reach the nodes representing x. Figure 7.10 shows 3 such tiles, shaded red, purple, and cyan. The tiling is *serial* because strict adherence to the dependencies shown in the graph requires that the red tile be execute before the purple tile, and the purple tile before the cyan tile.

In the example of Figure 7.10, elements of A (edges) that are reused within the same tile are shown by solid lines; the remaining edges (used across tiles) are shown by dashed lines. Assuming no reuse between executions of different tiles, sufficient cache capacity, and no conflicts, the minimum number of elements of A reused by executing the computation in legal tile order is 14 out of 19 possible in this example. Since the tiles are executed in order, we might also expect that with sufficient cache capacity, edges shared by two adjacent tiles may also be reused—for example, element $(2, 2)$ is used in both the red and the purple tiles, and there is a chance that $(2, 2)$ will still be in cache by the time it is needed in the purple tile.

Strout's serial sparse tiling algorithm can be described for general A as follows:

1. Given A and ρ, compute the dependency graph G.

2. Partition the elements of y into τ sets, and compute the corresponding τ tiles based on G. Let $T^{(i)}$ be the set of all nodes belonging to the i^{th} tile $(0 \leq i < \tau)$. Assume the tiles are numbered according to some legal ordering, where $T^{(i)}$ must be executed before $T^{(i+1)}$. (This ordering can be determined, for instance, by topologically sorting the directed graph representing dependencies between tiles.)

3. Carry out the computation $y \leftarrow A^{\rho} \cdot x$ by evaluating each tile in order.

The original paper on serial sparse tiling [288] uses the Metis graph partitioner to perform step 2 [186]. Below, we describe a simple data structure to hold the tile information, and then express step 3 assuming this data structure.

Let A be an $n \times n$ matrix stored in compressed sparse row (CSR) format (see Chapter 2). We store the tiles in two integer arrays, without modifying the data structure that holds A. Consider the computation $t^{(\rho)} \leftarrow A^{\rho} \cdot t^{(0)}$, where we introduce temporary vectors and $t^{(r)} \leftarrow A \cdot t^{(r-1)}$ for $1 \leq r < \rho$. The following pseudo-code shows how to construct these two arrays, `row_ind` and `tile_ptr`, given the tiled graph of $t^{(\rho)} \leftarrow A^{\rho} \cdot t^{(0)}$.

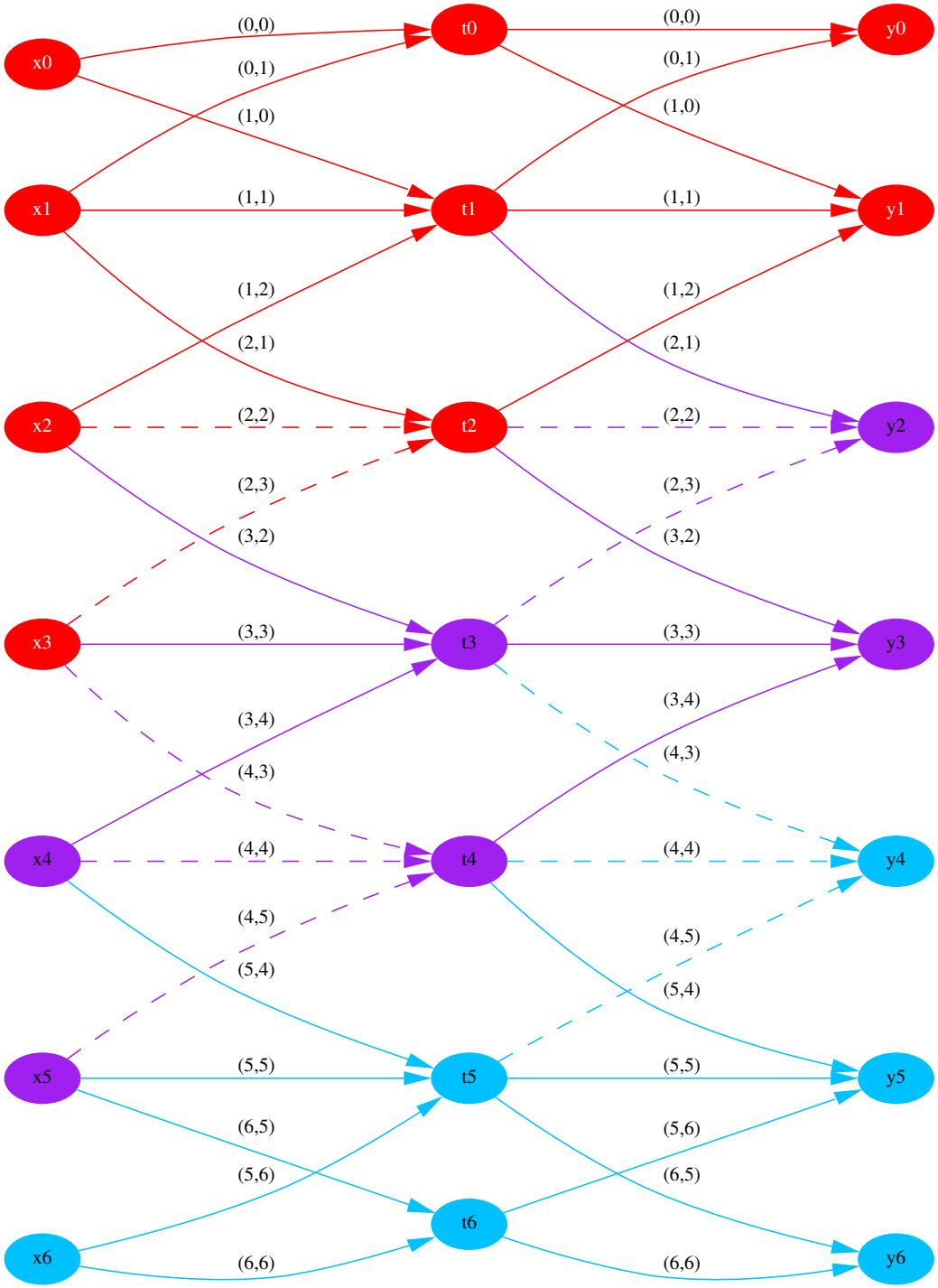

Figure 7.10: **Serial sparse tiling applied to $y \leftarrow A^2 \cdot x$ where A is tridiagonal**. We show the graph of the computation $t \leftarrow A \cdot x$, $y \leftarrow A \cdot t$, where A is a 7×7 tridiagonal matrix. Nodes represent elements x, t, and y. Each edge (v, w), labeled by (i, j), represents the update $w \leftarrow w + a_{i,j} \cdot v$. We show 3 sparse tiles, where all matrix and vector elements needed to evaluate a tile are shown in the same color. Solid edges show elements of A which are reused within the same tile. The minimum number of elements of A reused, assuming sufficient cache capacity and no conflicts, is 14 out of a possible 19 elements.

The array `row_ind`, of length $n \cdot \rho$, stores the indices of each temporary vector, listed in lexicographic order by tile and iteration r. The array `tile_ptr`, of length $\tau \cdot \rho + 1$, holds the starting offsets in `row_ind` of each tile and iteration.

```
    Algorithm CreateTilePointers(T^(0), ..., T^(τ−1))
1       tile_ptr[0 ... τρ + 1] ← 0 /* Array of length τ · ρ + 1, initialized to 0 */
2       row_ind ← [] /* empty array */
3       for p = 0 to τ − 1 do /* for each tile */
4           for r = 1 to ρ do /* for each "power" */
5               Let s be the number of elements of t^(r) contained in T^(p).
6               Append these s elements onto row_ind.
7               Record the next starting offset in row_ind, i.e.,
                    tile_ptr[p · ρ + r] ← tile_ptr[p · ρ + r − 1] + s
8       return tile_ptr, row_ind
```

For the example of Figure 7.10, Algorithm **CreateTilePointers** returns

$$\texttt{row\_ind} = [0,1,2,0,1,3,4,2,3,5,6,4,5,6] \quad , \quad \texttt{tile\_ptr}\,[0,3,5,7,9,11,14]$$

Following the notation of Chapter 2, we denote the arrays comprising the CSR data structure by `ptr` (row pointers), `ind` (column indices), and `val` (non-zero values). The tiled computation of $t^{(\rho)} \leftarrow A^{\rho} \cdot t^{(0)}$ can then be expressed as follows:

```
    type val : int[k]
    type ind : int[k]
    type ptr : int[n + 1]
    type row_ind : int[n · ρ]
    type tile_ptr : int[τ · ρ + 1]
1   Initialize temporary vectors, t^(r) ← 0 for 1 ≤ r ≤ ρ
2   for p = 0 to τ − 1 do /* for each tile */
3       for r = 1 to ρ do /* for each iteration */
4           for s = tile_ptr[p · ρ + r − 1] to tile_ptr[p · ρ + r] do
5               i ← row_ind[s] /* row index */
6               for l = ptr[i] to ptr[i + 1] do
7                   j ← ind[l] /* column index */
8                   t_i^(r) ← t_i^(r) + val[l] · t_j^(r−1)
```

Lines 4–8 are essentially SpMV in CSR format on a subset of the rows of A. The data structure and algorithm can be extended straightforwardly to a tiled blocked compressed sparse row (TBCSR) format, where BCSR is used as the base format. In the blocked case, line 8 above may be unrolled, just as in the usual implementation of register blocking (Section 3.1).

When the above multiplication routine completes, the temporary vectors contain the intermediate powers. Certain numerical algorithms (*e.g.*, Arnoldi and Lanczos algorithms for eigenproblems) require these vectors [93].

7.4.2 Preliminary results

To verify that speedups are possible and that cache misses are reduced, we implemented and tested the tiled SpMV scheme with register blocking on the Ultra 2i and Pentium III. These results are "preliminary" in that the relatively limited number of experiments is sufficient to demonstrate the feasibility of the sparse tiling technique, but leave a number of issues unresolved—namely, how and when to tile, as well as on what architectures we might expect to benefit from tiling.

For $A^{\rho}\cdot x$ with $\rho \geq 2$, "speedups" are measured as the performance in Mflop/s of $A^{\rho}\cdot x$ compared to the performance in Mflop/s of $A\cdot x$. The Mflop/s rates are measured in turn using the original number of non-zeros in A, ignoring fill as in the preceeding chapters. Thus, a speedup of 2 for a sparse tiled implementation of $A^{\rho}\cdot x$ means that executing $A^{\rho}\cdot x$ takes $\frac{1}{2}$ the time as ρ separate calls to $A\cdot x$.

We present results for two experiments which can be summarized as follows:

1. On the class of stencil matrices discussed in Chapter 5, we observe good speedups over register blocking when tiling and register blocking are combined: on the Ultra 2i for $A^2\cdot x$, over 2× compared to the reference implementation, and nearly 1.5× faster than register blocking without tiling. In looking at cache misses, we find that as the block size increases, the number of misses for $A^{\rho}\cdot x$ is asymptotically reduced by a factor of ρ. Furthermore, we observe that cache misses under tiling are relatively insensitive to the main tiling tuning parameter—the number of tiles τ—once the median number of matrix elements per tile begins to fit into cache.

   Speedups on the Pentium III are also reasonably good (for example, up to 2.3× in the best case for $A^2\cdot x$), but as we discuss below, speedup as a function of block size

is qualitatively somewhat different from the Ultra 2i. We do not yet fully understand the precise reasons for this behavior, implying that additional work is needed to understand when and how to apply tiling more robustly across platforms.

2. We evaluate sparse tiling performance on the SPARSITY matrix benchmark suite. In selecting the tuning parameters ($r \times c$ and τ), we use the results of the stencil experiments as a guide. We confirm the performance improvements on Ultra 2i, particularly on matrices from finite element method (FEM) applications where we observe speedups over register blocking alone of nearly 1.6× for $A^2 \cdot x$, 1.85× for $A^3 \cdot x$, and 1.9× for $A^4 \cdot x$.

   Results on the Pentium III are somewhat mixed. Although maximum speedups over blocking but not tiling can be good, median speedups on all classes of matrices tend to be low (1.2× and less). As in the experiment on stencils, this observation points to a need to understand more clearly the aspects of sparse tiling which are platform/architecture-specific.

Results on stencil matrices

We consider serial sparse tiling performance on the following sequence of 9 stencil matrices (see Chapter 5):

- Tridiagonal: A larger example of the matrix shown in Figure 7.10.
- 2-D, 5-point stencil
- 2-D, 9-point stencil
- Blocked 2-D, 9-point stencils: A sequence of 5 blocked matrices, obtained by replacing individual non-zeros in the 2-D, 9-point stencil matrix by $b \times b$ blocks. We use $b \in \{2, 3, 4, 6, 8\}$. Most rows have $9b$ non-zeros.
- 3-D, 27-point stencil

We use these matrices in this proof-of-principle experiment because it is reasonable to tile the computation of $y \leftarrow A^\rho \cdot x$ by simply grouping equal-sized consecutive subsets of

the elements of y, as shown in the example of Figure 7.10. Experimenting with various partitioning and reordering schemes is a good opportunity for future work.[4]

Figures 7.11–7.12 summarize the speedup results and observed cache misses on the Ultra 2i and Pentium III. We compare the following implementations:

- **Tiled and blocked implementations** of $A^2 \cdot x$ (red solid dots), $A^3 \cdot x$ (green solid triangles), and $A^4 \cdot x$ (blue solid diamonds): The number of tiles τ and block size are chosen by exhaustively searching over all power-of-2 tile sizes up to and including the maximum possible value for τ *and* all block sizes that divide the natural block size.

  Even though $A^3 \cdot x$ can also be executed as $A^2 \cdot x$ followed by $A \cdot x$, and $A^4 \cdot x$ can be executed by two calls to $A^2 \cdot x$ or one call to $A^3 \cdot x$ followed by a call to $A \cdot x$, we only show results for tiling the entire graph of the computation $A^{\rho} \cdot x$. Choosing decompositions given a fixed value of ρ is an opportunity for future work.

- **Register blocking** (hollow purple squares): An implementation of SpMV using BCSR format. For the blocked, 2-D stencil matrices, the block size is chosen by exhaustive search over all possible block sizes that divide the natural block size b. For the remaining matrices, we simply show the reference performance.

- **Reference** (black asterisks): An implementation of SpMV using CSR format.

This data, at the block size and tile size parameters yielding the best observed performance, also appear in Table 7.1 for the Ultra 2i and in Table 7.2 for the Pentium III.

The Ultra 2i demonstrates the considerable potential of serial sparse tiling, particularly when combined with register blocking (Figure 7.11 (*top*)). For matrices without blocking, the speedups are relatively modest at less than 1.55× even for $A^4 \cdot x$, with diminishing returns in the performance gains for $A^{\rho} \cdot x$ as ρ increases. Indeed, for the 3-D 27-point stencil, there are no speedups. With blocking, however, the results are more encouraging: $A^2 \cdot x$ runs up to 2.6× faster, $A^3 \cdot x$ up to 3× faster, and $A^4 \cdot x$ up to 3.2×.

To verify the extent to which cache misses are reduced, we show the number of cache misses seen by each implementation as a fraction of the number of cache misses observed for register blocking only on the Ultra 2i in Figure 7.11 (*bottom*). (Each data

[4] As discussed in Chapter 5, the stencil matrices are dominated by a diagonal structure that eliminates the need for most of the indices (*e.g.*, using our RSDIAG data structure). We do not consider diagonal data structures here since we are primarily interested in the effect of reusing elements of A in CSR and BCSR formats.

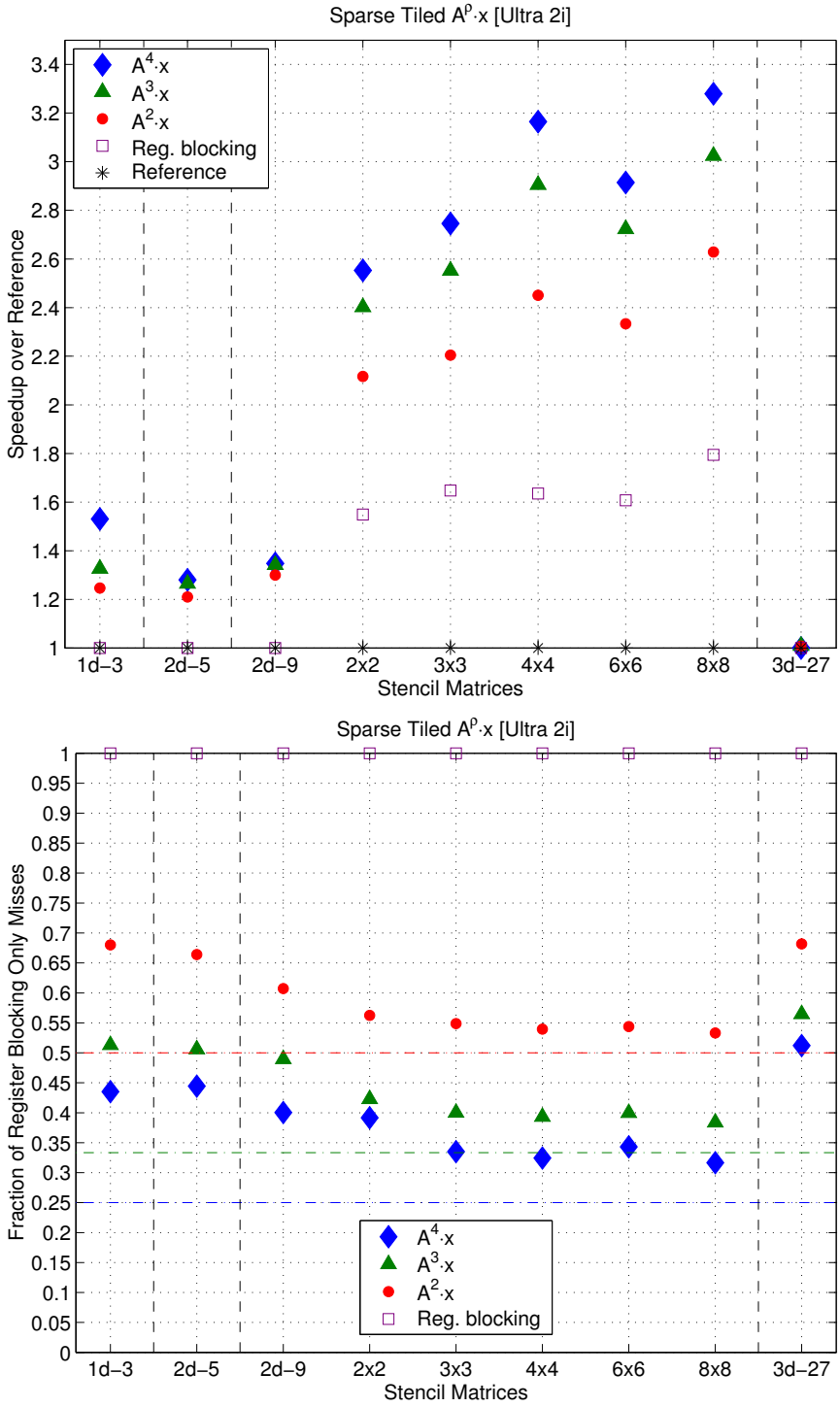

Figure 7.11: **Speedups and cache miss reduction for serial sparse tiled $A^{\rho} \cdot x$ on stencil matrices: Ultra 2i**. The reference is an unblocked, untiled SpMV using CSR format. We compare register blocking only (purple hollow squares), and combined register blocking + serial sparse tiling for $A^2 \cdot x$ (red solid dots), $A^3 \cdot x$ (green solid triangles), and $A^4 \cdot x$ (blue solid diamonds), to the reference. For block sizes and number of tiles used, see Table 7.1. (*Top*) Speedup over the reference implementation. (*Bottom*) Number of L_2 cache misses observed for each implementation, as a fraction of the number of misses observed for the register blocked but untiled code.

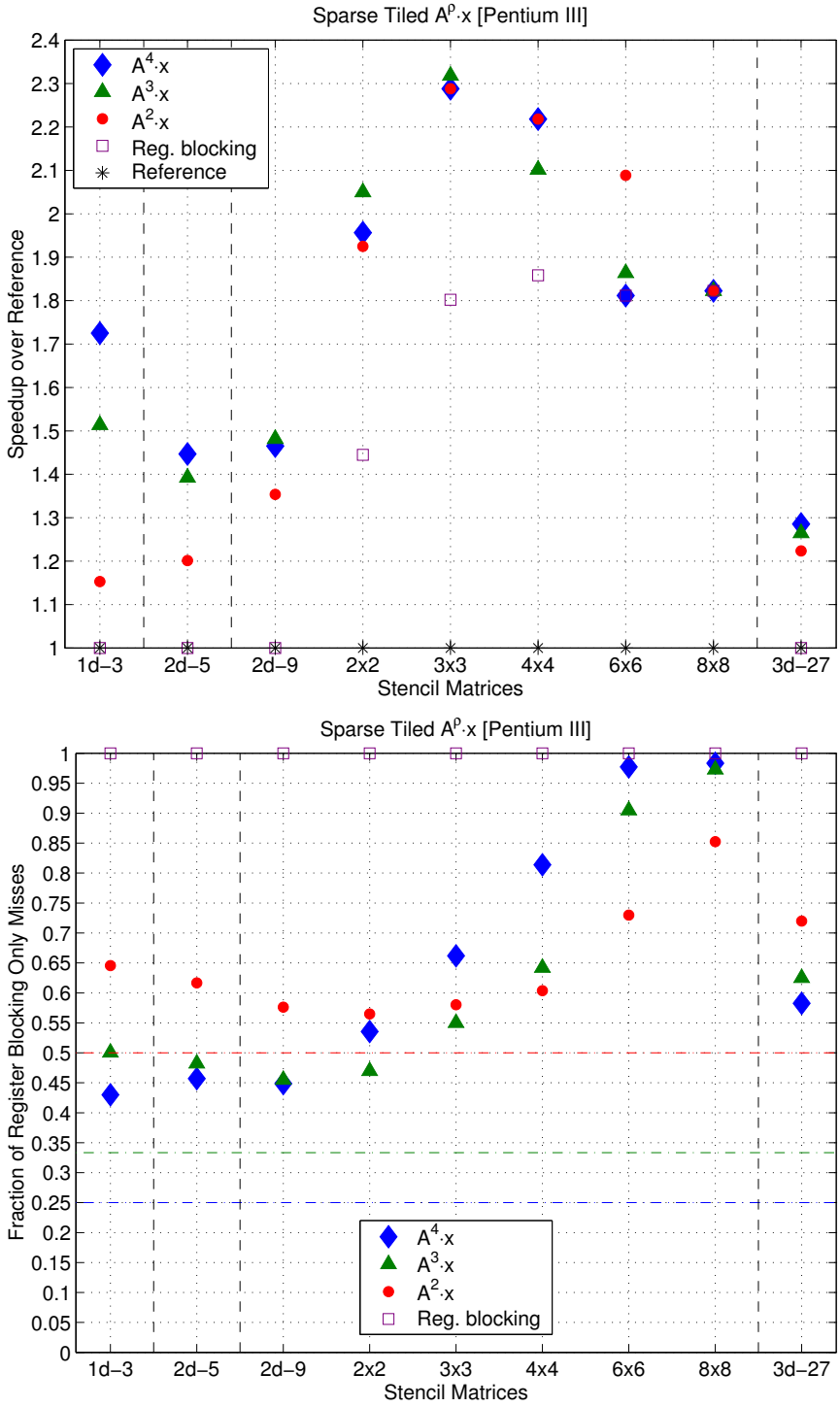

Figure 7.12: **Speedups and cache miss reduction for serial sparse tiled $A^{\rho} \cdot x$ on stencil matrices: Pentium III**. The reference is an unblocked, untiled SpMV using CSR format. We compare register blocking only (purple hollow squares), and combined register blocking + serial sparse tiling for $A^2 \cdot x$ (red solid dots), $A^3 \cdot x$ (green solid triangles), and $A^4 \cdot x$ (blue solid diamonds), to the reference. For block sizes and number of tiles used, see Table 7.2. (*Top*) Speedup over the reference implementation. (*Bottom*) Number of L_2 cache misses observed for each implementation, as a fraction of the number of misses observed for the register blocked but untiled code.

point uses the same tuning parameters as the implementation shown in Figure 7.11 (*top*), also listed in Table 7.1.) In the best case, we expect this fraction to approach $\frac{1}{\rho}$ for a tiled implementation of $A^{\rho} \cdot x$, shown by horizontal dashed lines. Indeed, cache misses approach these limits asymptotically for the 2-D 9-point stencils as the block size increases. Even for the 3-D 27-point stencil matrix, which saw no improvement in performance, the reduction in cache misses indicates that tiling is at least having the desired effect.

On the Pentium III (Figure 7.12), there is a comparable range of speedups for $A^2 \cdot x$ but qualitatively different speedup behavior when the block size increases. In addition, further improvements for $A^3 \cdot x$ and $A^4 \cdot x$ are modest relative to the base improvement for $A^2 \cdot x$ and register blocking. The L_2 misses shown in Figure 7.12 (*bottom*) show that as the block size increases beyond 3×3, the blocked and tiled implementations exhibit an *increase* in the relative numbers of misses. Although we do not fully understand this phenomenon at present, the qualitative difference in behavior between the two machines suggests the importance of platform-specific tuning with respect to sparse tiling.

Although we used exhaustive search to choose the number of tiles τ in this experiment on stencils, we find that the overall reduction in cache misses is relatively insensitive to τ once each tile roughly fits into cache. We define what we mean by the "size" of a tile, and further discuss this observation below.

We define the *tile size* as follows. Suppose we tile $A^{\rho} \cdot x$ and then execute the tiled implementation. We "assign" each matrix element to the first tile which uses it. The *tile size* of a given tile is the number of bytes needed to store all the non-zero matrix values and indices that are assigned to the tile. In the example of Figure 7.10, the red tile has a size of 8 doubles + 8 integers, the purple tile 6 doubles + 6 integers, and the cyan tile 5 doubles + 5 integers. The sum of all tile sizes equals the size of the CSR data structure (ignoring row pointers). In the case of a blocked matrix, there is as usual only 1 index per block instead of 1 per non-zero. As τ increases, we can reasonably expect the average tile size to decrease.

Figure 7.13 shows the number of L_2 misses for $A^2 \cdot x$, $A^3 \cdot x$, and $A^4 \cdot x$ as τ increases for the 2-D 9-point stencil matrix with 8×8 blocks. We show data for the the Ultra 2i in Figure 7.13 (*top*), and on the Pentium III in Figure 7.13 (*bottom*). The register block size is fixed at 8×8 on the Ultra 2i and 4×2 on the Pentium III. The y-axis of each plot shows L_2 misses for the tiled and blocked code relative to L_2 misses for the untiled but blocked code. The x-axis (log scale) shows the median tile size as a fraction of the L_2 cache size,

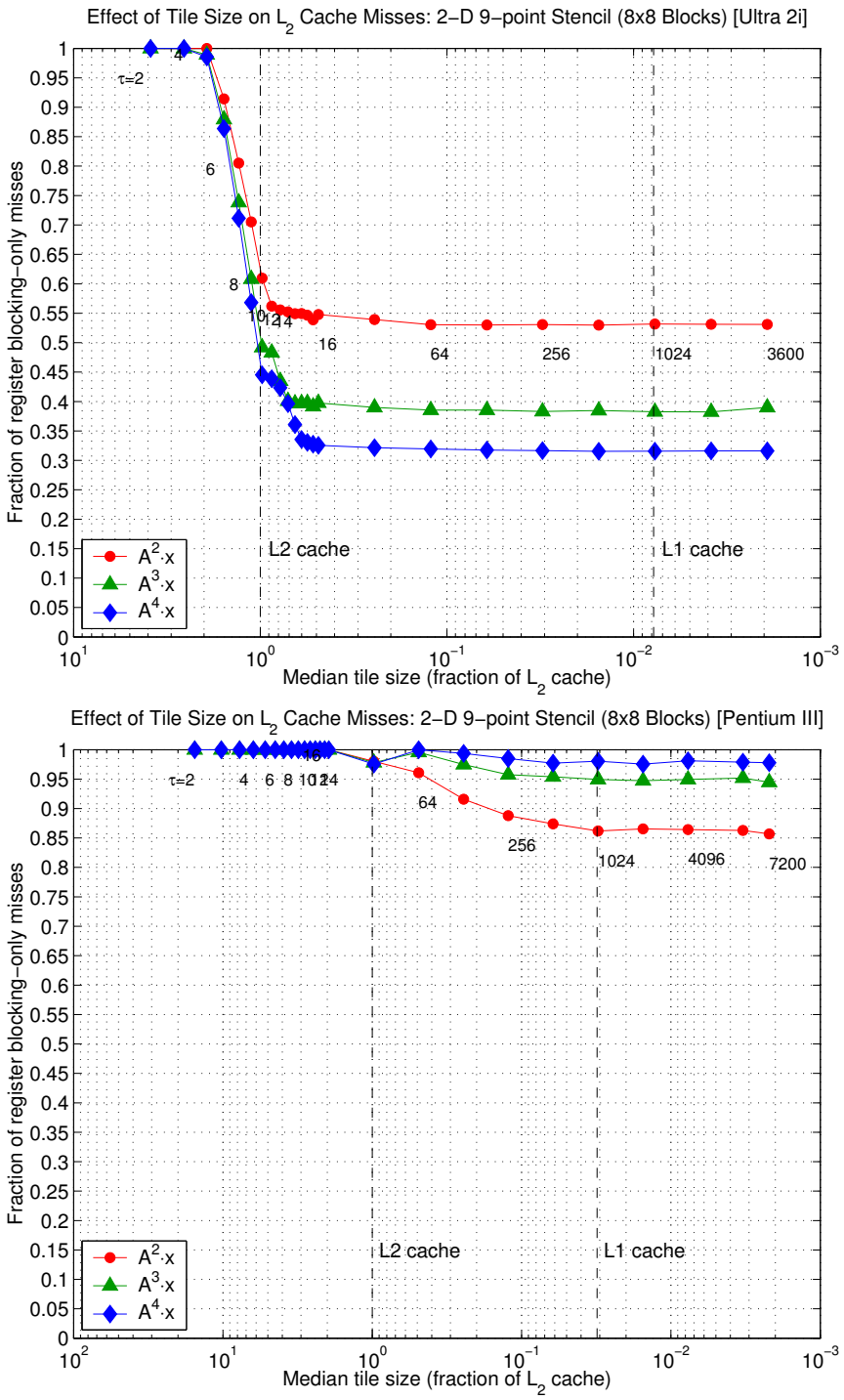

Figure 7.13: **Effect of tile size on L_2 cache misses: 2-D 9-point stencil (8x8 blocking), Ultra 2i (*top*) and Pentium III (*bottom*).** The y-axis shows the number of L_2 misses (as a fraction of misses observed for the untiled register blocked code) for each serial sparse tiled implementation of $A^2 \cdot x$, $A^3 \cdot x$, and $A^4 \cdot x$. The x-axis (log-scale) shows the median size of a tile as a fraction of the L_2 cache size. A transition occurs in the number of L_2 misses as the median size of a tile approaches the L_2 cache size on the Ultra 2i. A similar transition occurs between the L_1 and L_2 boundaries on the Pentium III.

where the median is taken over all tiles. We label a few points by their corresponding value of τ; points at the same x-location have the same value of τ. Vertical lines mark the L_2 and L_1 cache boundaries (2 MB and 16 KB, respectively, on the Ultra 2i, and 512 KB and 16 KB on the Pentium III).

The fraction of misses for the tiled and blocked $A^{\rho} \cdot x$ code transitions from the maximum value of 1 toward the asymptotic limit of $\frac{1}{\rho}$ on the Ultra 2i, once the median tile size falls below twice the L_2 cache size. A roughly similar transition occurs on the Pentium III as well, though the relative number of misses is much higher, being minimized at .85 when $\rho = 2$, and actually higher when ρ is 3 or 4. In addition, the misses do not really bottom-out when $\rho = 2$ until the L_1 boundary is reached. This fact may reflect differences in the relative capacities between the L_1 and L_2 caches on the two machines, though more careful modeling and analysis is needed to make strong conclusions. Regardless of these differences, choosing τ to be the maximum possible value effectively minimizes misses on either platform, at least given the stencil matrix structure and the simple partitioning scheme that groups consecutive equal sized sets of elements of y when building tiles.

Results on the Sparsity benchmark suite

We demonstrate that speedups are possible on other application matrices using the sparse tiling technique, though we do not resolve the important questions of when (*i.e.*, on what matrices and platforms) to apply it. This section presents results from an experiment in which we applied sparse tiling of $A^2 \cdot x$, $A^3 \cdot x$, and $A^4 \cdot x$ for the SPARSITY benchmark matrices (Appendix B) on the Ultra 2i and Pentium III.

As with the SpA^TA experiments, we exclude matrices which fit in the largest machine cache. (In addition, we also exclude the non-square matrices 41–44.) Recall that these matrices can be roughly categorized into three groups: FEM matrices 2–9 which are dominated by a single block size and uniform alignment, FEM matrices 10–17 which have multiple "natural" block sizes and/or non-uniform alignment, and matrices 18–44 from assorted applications that tend not have natural dense rectangular block structure. In these experiments, we create the tiles by grouping consecutive elements of the destination vector (as in the stencil matrix experiments), we fix the block size to be the same as the best block size for SpMV (see Chapter 3), and we choose τ to be the maximum possible value as suggested by the results of the previous section.

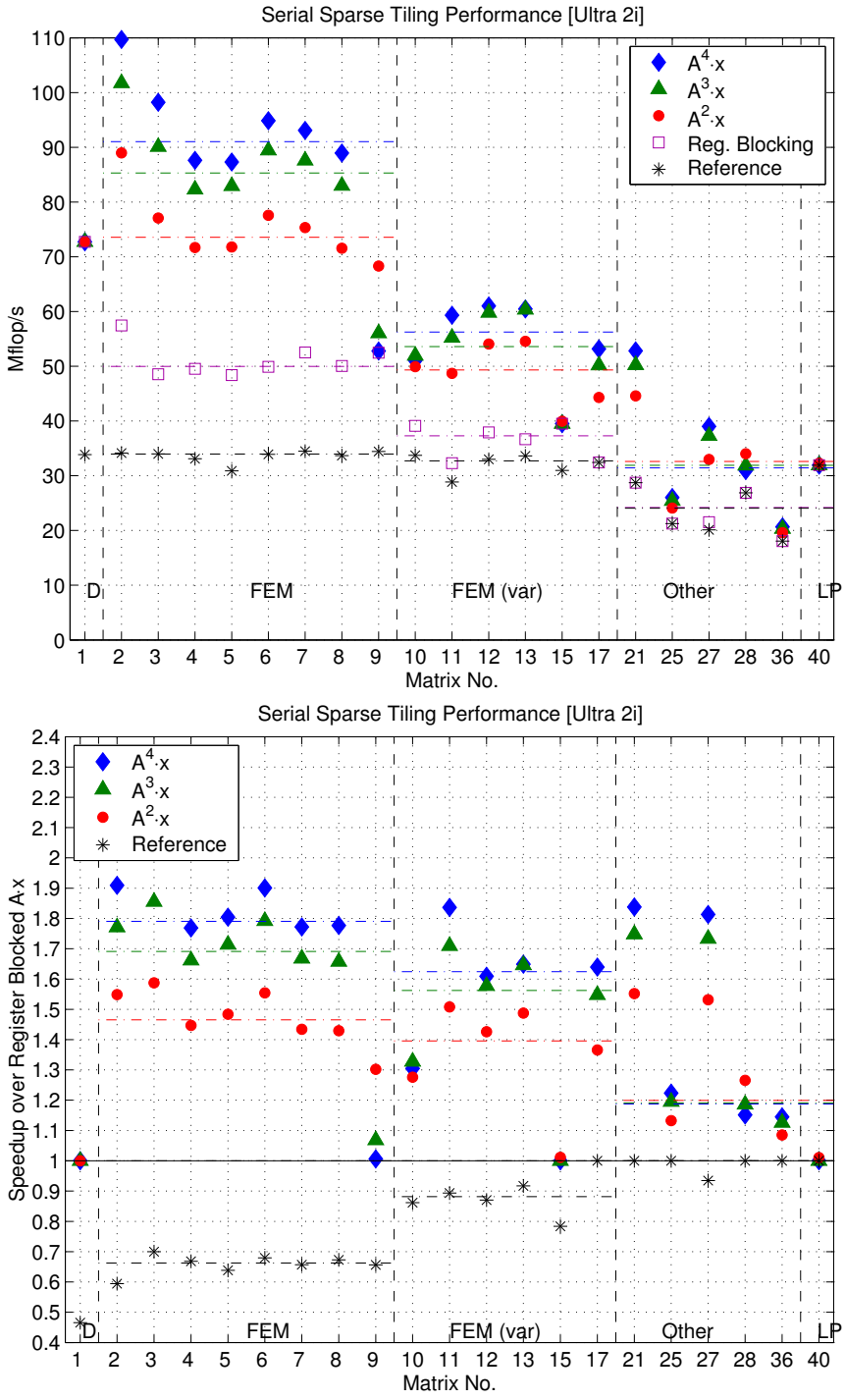

Figure 7.14: **Serial sparse tiling performance on the Sparsity matrix benchmark suite: Ultra 2i**. For each of the three primary matrix groups, we show the median speedup for each implementation (reference, register blocking but untiled, tiled $A^2 \cdot x$, tiled $A^3 \cdot x$, and tiled $A^4 \cdot x$) by a dash-dot horizontal line whose color matches the corresponding marker. This data is also tabulated in Table J.1.

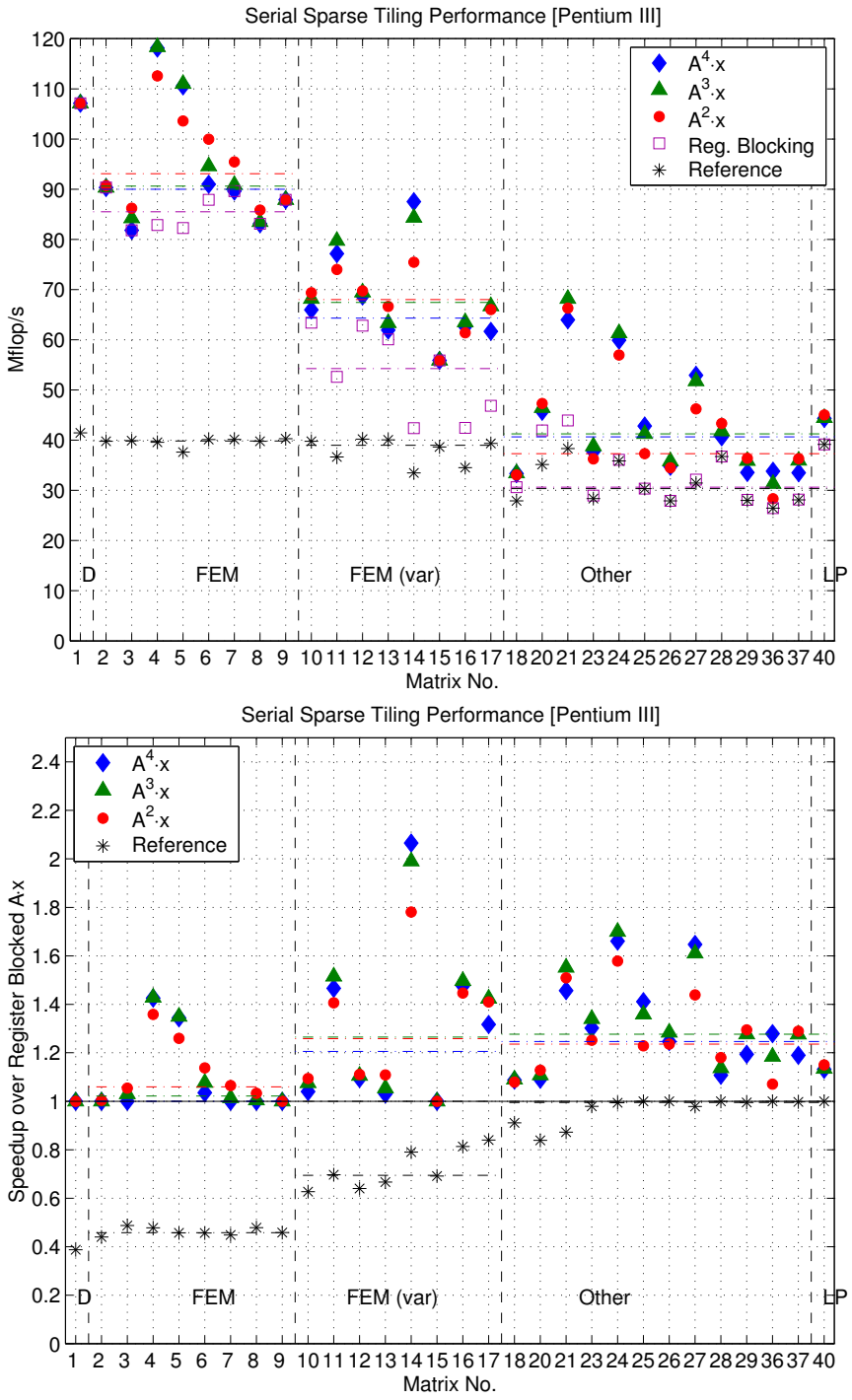

Figure 7.15: **Serial sparse tiling performance on the Sparsity matrix benchmark suite: Pentium III**. For each of the three primary matrix groups, we show the median speedup for each implementation (reference, register blocking but untiled, tiled $A^2 \cdot x$, tiled $A^3 \cdot x$, and tiled $A^4 \cdot x$) by a dash-dot horizontal line whose color matches the corresponding marker. This data is also tabulated in Table J.2.

The Ultra 2i data confirm that tiling using the simple partitioning scheme and combining tiling with register blocking can yield good speedups relative to register blocking without tiling, as shown in Figure 7.14 (*top*). The same data appear in Figure 7.14 (*bottom*) as speedups relative to the register blocked SpMV code. For each of the three matrix categories and $A^{\rho} \cdot x$ kernel, we show the median performance and median speedup by dashed-dot horizontal lines within the matrix category with a color that matches the kernel. Improvements due to tiling over register blocking only are largest on FEM matrices 2–9 (median of 1.45× for $\rho = 2$, nearly 1.7× for $\rho = 3$, and 1.8× for $\rho = 4$) and smallest on matrices 18–40 (1.2× independent of ρ). Although the latter class of matrices remain difficult, the former show considerable potential improvements on top of register blocking.

Results on the Pentium III are mixed. Though we observe appreciable maximum speedups—up to 1.8× when $\rho = 2$, and up to 2× or more when $\rho = 3$ or 4—median speedups in all classes of matrices ranges from none (FEM 2–9) to 1.2×. Indeed, the FEM 2–9 results run counter to what we observe on the Ultra 2i for the same class of matrices. In addition, observe that performance of $A^2 \cdot x$ can even be somewhat faster than fully tiled $A^3 \cdot x$ and $A^4 \cdot x$, suggesting that a practical implementation may wish to consider decompositions of a given power into optimal subproblems.[5] In short, the Pentium III data raises important questions about on what matrices and platforms we can expect sparse tiling to be profitable.

7.5 Summary

Our study of performance of SpMV and SpTS kernels relative to an upper bounds model indicated that still greater performance improvements would need to come from kernels with inherently more reuse of the matrix. This chapter considers SpA^TA and serial sparse tiled implementations of sparse $A^{\rho} \cdot x$ as two possibilities.

The speedups of up to 4.2× that we have observed for SpA^TA, when compared to reference CSR implementations that apply A and A^T as separate steps, indicate that there is tremendous potential to boost performance in applications dominated by this kernel. Even compared to register blocking without the cache optimization, performance of our implementations are still up to 1.8×. The implementation of our heuristic and its accuracy in

[5] A similar problem arises in computing the fast Fourier transform by the Cooley-Tukey algorithm, where in the one-dimensional case an input problem of size N may be decomposed into p subproblems of size q, where $N = p \cdot q$ [83]. The FFTW system approaches this problem of choosing the best decomposition using a dynamic programming approach [123].

choosing a block size helps to validate the approach to tuning parameter selection originally proposed in SPARSITY[164], and refined here in Chapter 3. A similar kernel from which we might expect improvements is simultaneous application of A and A^T, *i.e.*, simultaneous evaluation of $y \leftarrow A \cdot x$ and $z \leftarrow A^T \cdot w$ [30]. Owing to the fairly uniform improvements from SpA^TA on the evaluation platforms of this chapter, we advocate the inclusion of these kernels in future sparse matrix libraries.

Our upper bounds for SpA^TA indicate that there is a more room for improvement using low-level tuning techniques than with prior work on SpMV and SpTS. Applying automated search techniques to improve scheduling, as developed in ATLAS [325] and PHiPAC [46], is a natural extension of this work. An additional opportunity for future work is to implement our suggested refinements to the bounds that make explicit use of matrix non-zero structure (*e.g.*, making the working set size block row structure dependent, and accounting for the degree of actual spatial locality in source vector accesses). Such a refined model could be used to study how performance varies with architectural parameters, in the spirit of Chapter 4 and the SpMV modeling work by Temam and Jalby [294].

The preliminary results on tiling for sparse $A^\rho \cdot x$, inspired by recent work by Strout [288], are encouraging though limited in that the important questions of when and how best to apply the technique remain unresolved. We hope our experiments serve as a useful starting point for future work. For instance, we see in Figure 7.13 that tiling leads to the expected asymptotic reduction in cache misses on one architecture but not another. Understanding why could be resolved by better characterizing the relationship between the tiled graph structure and machine-specific details like the cache configuration.

Another higher-level sparse kernel is the sparse triple product, or RAR^T where A and R are sparse matrices. The triple product is a bottleneck in the multigrid solvers [105, 106, 4, 175, 277], for instance. There has been some work on the general problem of multiplying sparse matrices [146, 56, 79, 290], including recent work in a large-scale quantum chemistry application that calls matrix-multiply kernels automatically generated and tuned by PHiPAC for particular block sizes [68, 46]. This latter example suggests that there exists a potential opportunity to apply tuning ideas to the sparse triple product kernel.

Matrix	Reference Mflop/s	Register Blocked Mflop/s	Sparse Tiled $A^2 \cdot x$ Mflop/s	Sparse Tiled $A^3 \cdot x$ Mflop/s	Sparse Tiled $A^4 \cdot x$ Mflop/s
Tridiagonal $n = 100000$ $k = 299998$	18	—	22 1×1 $\tau = 32$	24 1×1 256	27 1×1 1024
2D 5-pt stencil $n = 388129$ $k = 1938153$	20	—	24 1×1 $\tau = 128$	25 1×1 1024	25 1×1 1024
2D 9-pt stencil $n = 250000$ $k = 2244004$	21	—	28 1×1 $\tau = 256$	29 1×1 2048	29 1×1 2048
2D 9-pt, 2x2 $n = 125000$ $k = 2238016$	27	42 2×2	57 2×2 $\tau = 512$	65 2×2 512	69 2×2 1024
2D 9-pt, 3x3 $n = 76800$ $k = 2056356$	31	52 3×3	69 3×3 $\tau = 1024$	80 3×3 512	86 3×3 256
2D 9-pt, 4x4 $n = 57600$ $k = 2050624$	32	52 4×2	77 4×4 $\tau = 512$	92 4×4 512	100 4×4 1024
2D 9-pt, 6x6 $n = 38400$ $k = 2039184$	34	54 3×3	78 3×3 $\tau = 256$	91 3×3 512	98 3×3 512
2D 9-pt, 8x8 $n = 28800$ $k = 2027776$	35	62 8×8	91 8×8 $\tau = 256$	105 8×8 256	114 8×4 512
3D 27-pt $n = 74088$ $k = 1906624$	30	—	30 1×1 $\tau = 512$	31 1×1 16384	30 1×1 16384

Table 7.1: **Proof-of-principle results for serial sparse tiled $A^{\rho} \cdot x$ on stencil matrices: Ultra 2i platform**. DGEMV performance is 59 Mflop/s, and peak is 667 Mflop/s. (See Appendix B for more configuration details). We show the dimension n and number of non-zeros k for each matrix (column 1), the number of tiles used (column 2), the reference performance based on untiled CSR (column 3), tiled performance (column 4), and the ratio of L_2 misses under tiling to untiled misses (column 5). For reference we show performance in our row-segmented diagonal format (see Chapter 5), with an unrolling depth of 7 (column 6). Speedups over the reference are shown in square brackets.

Matrix	Reference Mflop/s	Register Blocked Mflop/s	Sparse Tiled $A^2 \cdot x$ Mflop/s	Sparse Tiled $A^3 \cdot x$ Mflop/s	Sparse Tiled $A^4 \cdot x$ Mflop/s
Tridiagonal $n = 100000$ $k = 299998$	25	—	29 1×1 $\tau = 64$	38 1×1 1024	43 1×1 2048
2D 5-pt stencil $n = 388129$ $k = 1938153$	31	—	37 1×1 $\tau = 512$	43 1×1 4096	45 1×1 4096
2D 9-pt stencil $n = 250000$ $k = 2244004$	38	—	51 1×1 $\tau = 1024$	56 1×1 4096	55 1×1 16384
2D 9-pt, 2x2 $n = 125000$ $k = 2238016$	45	65 2×2	86 2×2 $\tau = 2048$	92 2×2 2048	88 2×2 8192
2D 9-pt, 3x3 $n = 76800$ $k = 2056356$	48	86 3×3	110 3×3 $\tau = 4096$	111 3×3 2048	110 3×3 4096
2D 9-pt, 4x4 $n = 57600$ $k = 2050624$	49	91 4×2	109 4×2 $\tau = 4096$	103 4×2 4096	109 4×2 4096
2D 9-pt, 6x6 $n = 38400$ $k = 2039184$	51	93 3×3	107 3×3 $\tau = 8192$	96 3×3 256	93 3×3 8192
2D 9-pt, 8x8 $n = 28800$ $k = 2027776$	53	97 4×8	97 4×2 $\tau = 8192$	97 4×2 512	97 4×2 8192
3D 27-pt $n = 74088$ $k = 1906624$	46	—	56 1×1 $\tau = 1024$	58 1×1 8192	59 1×1 4096

Table 7.2: **Proof-of-principle results for serial sparse tiled $A^\rho \cdot x$ on stencil matrices: Pentium III platform**. DGEMV performance is 58 Mflop/s, and peak is 500 Mflop/s. (See Appendix B for more configuration details). We show the dimension n and number of non-zeros k for each matrix (column 1), the number of tiles used (column 2), the reference performance based on untiled CSR (column 3), tiled performance (column 4), and the ratio of L_2 misses under tiling to untiled misses (column 5). For reference we show performance in our row-segmented diagonal format (see Chapter 5), with an unrolling depth of 7 (column 6). Speedups over the reference are shown in square brackets.

Chapter 8

Library Design and Implementation

Contents

The steps and costs associated with tuning have implications for the design of sparse matrix libraries. This chapter proposes a set of automatic tuning extensions to the recent Sparse Basic Linear Algebra Subroutines (SpBLAS) interface [108, 50, 49]. In short, we propose

1. to extend the SpBLAS to include new kernels based on the results of Chapter 7: sparse $A\&A^T$ (Sp$A\&A^T$), sparse $A^TA\cdot x$ (SpA^TA), and sparse $AA^T\cdot x$ (SpAA^T),

2. to add one routine per kernel to enable kernel-specific tuning, and

3. and to add two additional routines that help make the tuning process transparent and, to some degree, controllable by the user.

Our proposed changes are fully upward compatible with the existing SpBLAS interface, and complement existing approaches to sparse kernel implementation (Section 8.4).

Our summary (Section 8.2) of the SpBLAS interface itself is based on material from three sources: Chapter 3 of the recent BLAS standard revision [49], the SpBLAS interface review paper by Duff, Heroux, and Pozo [108], and the reference implementation by Duff and Vömel [109].

8.1 Rationale and Design Goals

This dissertation shows that tuning depends on the kernel, platform, and matrix. Each of these critically affects performance but may be known at different stages of application development and execution:

- **Kernel:** Which kernels will be needed for a particular application? The preceeding chapters show that each kernel has its own implementation space. The kernels needed for a particular application will most likely be known at compile-time, but precise calling sequences and workloads may not be known until run-time.

- **Platform:** On what architecture or microarchitecture will the application run? What compilers and other libraries are available? These details are primarily known at "install-time" and affect the construction of kernels.

- **Matrix:** What matrix properties can we exploit, such as block structures, diagonal structures, and symmetry? In general, the matrix is not known until run-time, but certain structural properties—and therefore, matrix data reorganization strategies—may be saved and reused across different runs.

Our search-based approach to tuning addresses these questions in two distinct stages, namely, by (1) platform-dependent benchmarking at install-time, and (2) kernel- and matrix-dependent search at both compile- and run-time. However, run-time tuning has associated costs and is therefore appropriate only in certain (but common) contexts. A library implementation of the Sparse Basic Linear Algebra Subroutines (SpBLAS) interface, with the appropriate tuning extensions, allows us to exploit all of the above information about the underlying platform at both install-time and run-time, while still exposing the steps and costs of tuning to the user. An additional advantage of the library approach is that it

complements other approaches (Section 8.4). Indeed, integration with existing systems is a possible and likely avenue for future work.

The high-level design goals of our library implementation can be summarized as follows. Underlying these goals is the desire to make the tuning process relatively transparent to a user of the library.

- **Provide a "building-block" interface:** The SpBLAS defines a relatively compact interface consisting of just a few basic operations. These primitives serve as useful building blocks in applications or higher-level kernels (*e.g.*, linear systems solvers or eigensolvers).

- **Providing new kernels:** This dissertation demonstrates significant speedups for kernels not currently included in the SpBLAS standard, including sparse $A\&A^T$ (Sp$A\&A^T$) and sparse $A^TA\cdot x$ (SpA^TA)/sparse $AA^T\cdot x$ (SpAA^T).

- **Upward compatibility with SpBLAS:** We seek extensions that would *not* require source code modifications to existing SpBLAS applications. Instead, such applications would simply need to re-link with a tuning-enabled SpBLAS implementation. Interfaces for the new kernels should follow the SpBLAS interface conventions.

- **User inspection and control of the tuning process:** The interface should provide mechanisms by which to examine summaries of which transformations and optimizations were applied during tuning. In addition, we provide functionality to allow a user (1) to save tuning decisions to apply on future problems (*a la* FFTW's *wisdom* [123]), (2) to control the cost of tuning by giving hints about the workload and specify memory usage constraints, and (3) to circumvent the automated tuning process by manually specifying what optimizations to apply.

Implicit in these goals is the notion that the target user of our library is interested in tuning details, though it is certainly possible to use our proposed interface without any specific knowledge about tuning.

8.2 Overview of the Sparse BLAS interface

The SpBLAS interface is functionally similar to the dense BLAS, with one major distinction. Like the dense BLAS, the SpBLAS defines a machine-independent interface to "low-level"

linear algebra kernels like dot products, vector scaling, matrix-vector multiply, matrix multiply, and triangular solve. What distinguishes the SpBLAS from its dense counterpart is that a matrix in the SpBLAS is represented by a generic handle rather than a specific array data structure as in the dense BLAS. As a consequence, the underlying SpBLAS library implementation is responsible for choosing a data structure in which to store the matrix, and this data structure is completely hidden from the user.

The basic pattern by which users are expected to call SpBLAS routines consists of six distinct steps:

1. **Create a matrix handle:** The user signals to the SpBLAS implementation that a new matrix is to be allocated and defined.

2. **Assert matrix properties:** The user may optionally specify hints about the non-zero structure, *e.g.*, to indicate that the matrix is symmetric, has blocks, etc.

3. **Insert matrix entries:** The user makes a sequence of calls to insert non-zeros into the matrix data structure. These non-zeros may be inserted one at a time, or a row/column/block at a time. A special clique-insertion routine is provided for finite element method applications (see below).

4. **Signal the end of matrix creation:** When all non-zeros have been inserted, the user signals the end of the matrix creation process. At this point, the handle and matrix data become logically immutable.

5. **Call operations on the handle:** Any of the basic kernels may be called on the handle. These include sparse matrix-vector multiply (SpMV), sparse triangular solve (SpTS), sparse matrix-multiple vector multiply (SpMM), and sparse triangular solve with multiple right-hand sides (SpTSM).

6. **Destroy the handle:** The SpBLAS provides a matrix deallocation routine. The handle becomes invalid once this routine is called.

These steps imply that, from the user's perspective, there is some state information associated with a matrix handle. Indeed, the SpBLAS standard provides a means by which the current state may be queried (see below).

Figure 8.1 shows sample C code which constructs a 3×3 lower triangular matrix and calls SpMV. In the C interface to the SpBLAS, all routines are prefixed by `BLAS_us`,

$$A = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ .5 & 0 & 1 \end{pmatrix}$$

```
blas_sparse_matrix A_handle;
double x[3] = { ... }, y[3] = { ... } ;
/* Create A */
A_handle = BLAS_duscr_begin( 3, 3 ) ;
BLAS_ussp( A_handle, blas_unit_diag ) ;
BLAS_ussp( A_handle, blas_lower_triangular ) ;
BLAS_duscr_insert_entry( A_handle, 1, 0, -2 ) ;
BLAS_duscr_insert_entry( A_handle, 2, 0, .5 ) ;
BLAS_duscr_end( A_handle ) ;
/* Compute y ← y + Ax */
BLAS_dusmv( blas_no_trans, -3.0, A_handle, x, 1, y, 1 ) ;
/* Deallocate A */
BLAS_usds( A_handle ) ;
```

Figure 8.1: **SpBLAS calling sequence example**. (*Left*) A 3×3 lower triangular matrix. (*Right*) Sample SpBLAS calling sequence that constructs A and calls SpMV. This example uses point-insertion routines to insert individual non-zeros in the strictly lower triangular portion of the matrix, and specifies ones on the diagonal using the matrix property hints. In the call to `BLAS_dusmv`, the constant `1` values indicate that consecutive elements of the vectors `x` and `y` should be accessed with unit stride.

where the `us` stands for "unstructured sparse." (Bindings to all routines are available in both C and Fortran versions.) The example uses property assertions to declare the matrix to be strictly lower triangular and to have a unit diagonal (via calls to `BLAS_ussp`), and then uses point-entry insertion routines to specify the values in the strictly lower triangle. Below, we discuss the implications of the SpBLAS interface on any library implementation, with a particular emphasis on issues related to performance and memory usage.

The type `blas_sparse_matrix` is specified in the standard to be equivalent to a C `int`. On most platforms, a handle is therefore represented by a 32-bit integer which may not be compatible with a pointer type. Therefore, the library implementation is responsible for associating the handle with actual matrix data. In a multithreaded environment, care must be taken to ensure that the SpBLAS creation and non-zero insertions are thread-safe.

By design, the library cannot know the total size of the matrix (*e.g.*, number of non-zeros) when the handle is created. The library implementation is therefore responsible for managing memory associated with matrix construction and for making assembly as

efficiently as possible, since the user cannot know these costs up-front.

Properties serve as hints to the library implementation, to help decide how best to store the matrix. We list a few of the possible properties in Table 8.1. (For a complete list, refer to the complete SpBLAS standard [50].) All properties must be specified before the first non-zero insertion. The results are undefined if incompatible properties are specified (*i.e.*, both the lower triangular and upper triangular properties are set). Furthermore, once insertion has begun, any insertion that violates an asserted property will fail. In the example of Figure 8.1, specifying the unit-diagonal property means that the implementation could potentially save storage of the diagonal.

The SpBLAS also provides a routine `BLAS_usgp` to query properties ("get properties") of the handle, such as the dimensions, number of non-zeros, type, whether the matrix is symmetric, upper or lower triangular, and so on. Furthermore, the current state of the handle may be queried as well. The SpBLAS defines a number of constants to indicate the state: `blas_new_handle` is the initial state after the call to `BLAS_uscr_begin`, `blas_open_handle` after the first non-zero entry has been inserted but before the call to `BLAS_uscr_end`, `blas_valid_handle` after `BLAS_uscr_end` has completed successfully, and `blas_void_handle` after deallocation of the handle or if the handle is otherwise not a valid handle.

Although Figure 8.1 inserts each non-zero individually, the SpBLAS standard defines a variety of other insertion methods. These include insertion of an entire row or column simultaneously and insertion of a contiguous $r \times c$ block of non-zeros. In addition, the standard defines a "clique-insertion" routine. A clique is represented by a two-dimensional $r \times c$ array `val` along with an integer array `row_ind` of length r and and array `col_ind` of length c. Entry `val[i,j]` is inserted into position (`row_ind[i]`, `col_ind[j]`) of the matrix. The library implementation must support any combination of these insertion routines. One consequence of this flexibility is that the implementation must be careful with regard to dynamic memory allocation, in light of the fact that the user cannot provide the implementation with a hint about memory usage (*e.g.*, by specifying the total number of non-zeros to pre-allocate).[1]

[1]Regarding insertion of non-zeros, we remark on a number of details. First, the library implementation is free to interpret insertions of explicit zeros as either structural non-zeros or true zeros that are not stored. In addition, if a non-zero is repeatedly inserted, the user may specify (via a property) whether these values are to be summed or the last value taken to be the non-zero value. Finally, note that users may also specify whether indices should be interpreted as zero-based or one-based indices. The default is language-binding

Property Name	Description
blas_non_unit_diag	Non-zero diagonal entries are stored (default)
blas_unit_diag	Diagonal entries not stored and assumed to be 1
blas_no_repeated_indices	Indices are unique (default)
blas_repeated_indices	Repeated indices are summed on insertion
blas_lower_symmetric	Matrix is symmetric
blas_upper_symmetric	
blas_lower_hermitian	Matrix is Hermitian
blas_upper_hermitian	
blas_lower_triangular	Matrix is lower triangular
blas_upper_triangular	Matrix is upper triangular
blas_irregular	Assume no regular structure
blas_regular	Structure comes from a regular grid
blas_block_irregular	Assume blocks occur but otherwise no regular structure
blas_block_regular	Structure comes from a regular grid
blas_unassembled	Matrix is best represented by a sum of cliques

Table 8.1: **SpBLAS properties**. This table shows a subset of the possible structural properties that a user may assert. The last five properties are intended to be structural hints only and do not affect program correctness.

Once all the non-zeros have been inserted, the handle and associated matrix data become logically immutable at the call to `BLAS_duscr_end( · )`. Thus, even if a new matrix differs from an existing matrix only in the non-zero values, it is not possible to reuse the structure, and the new matrix must be constructed from scratch. There are no routines that allow querying of non-zero entries.

After matrix creation is complete, the handle may be used by any of the routines where a matrix handle is expected. Thus, however the matrix is represented internally, the library implementation must ensure correct operation for any kernel called on that handle.

We summarize the available SpBLAS operations in Table 8.2. All operations expect one sparse operand and one or more dense operands, *i.e.*, there are no operations on only sparse operands. The SpBLAS operations are classified as Level 1, 2, and 3 routines, just as is done in the dense BLAS. Each class indicates the level of data reuse. The Level 1 routines operate on vectors and have no inherent reuse. The Level 2 routines operate on a single matrix and a vector and exhibit reuse opportunities only in the vector accesses. The Level 3 routines operate on a sparse matrix and a dense matrix (*i.e.*, multiple vectors), and

specific (0-based for C, and 1-based for Fortran).

Level	Description	Mathematical Operation	SpBLAS Routine
1	Dot product	$\gamma \leftarrow s^T \cdot y$	`usdot`
	Vector scale ("axpy")	$y \leftarrow y + \alpha \cdot s$	`usaxpy`
	Gather	$s \leftarrow y\|_s$	`usga`
	Gather and zero	$s \leftarrow y\|_s; y\|_s \leftarrow 0$	`usgz`
	Scatter	$y\|_s \leftarrow s$	`ussc`
2	Matrix-vector multiply	$y \leftarrow y + \alpha \cdot \mathrm{op}(A) \cdot x$	`usmv`
	Triangular solve	$x \leftarrow \alpha \cdot \mathrm{op}(L)^{-1} \cdot x$	`ussv`
3	Matrix-multiple vector multiply	$Y \leftarrow Y + \alpha \cdot \mathrm{op}(A) \cdot X$	`usmm`
	Multiple-vector triangular solve	$X \leftarrow \alpha \cdot \mathrm{op}(L)^{-1} \cdot X$	`ussm`

Table 8.2: **SpBLAS computational kernels**. Greek letters ($\alpha, \beta, \ldots$) denote scalar variables, s denotes a sparse vector, x, y denote dense vectors, X, Y denote dense matrices (*i.e.*, multiple dense vectors), A denotes a sparse matrix, and L denotes a sparse triangular (lower or upper) matrix. The function $\mathrm{op}(A)$ indicates that either A, A^T, or A^H are supported by the interface. The notation $y|_s$ denotes the entries of y at the same indices given by the sparse vector s.

exhibit matrix-level reuse. This classification exposes the potential computational efficiency of each kernel to the user. The library implementation should strive to meet the user's performance expectations.

Errors may occur while allocating memory or inserting non-zeros that violate asserted matrix properties. The standard specifies that each routine must return an error value so that the application may try to detect and recover from errors.

8.3 Tuning Extensions

There are at least two possible entry points for tuning in the current SpBLAS interface. One possibility is to tune during the call to `uscr_end`. At this point, all of the properties have been set and the non-zeros inserted. The disadvantage of tuning in this call is that we may not be able to tune for a particular kernel since we do not know what kernel(s) will be used. A second possibility would be to tune at the first call to the kernel (*e.g.*, at the call to `usmv` or `ussv`). However, we do not know how often the kernel will be called with a given handle, and therefore the implementation cannot judge whether the cost of tuning will be amortized over many uses. Thus, although neither of these two entry-points would require changing the current BLAS interface, they are not ideal in light of their respective disadvantages.

We propose the following extensions to the SpBLAS to overcome the limitations of tuning within the existing interface.

- **Interfaces for new kernels:** Chapter 7 demonstrates significant speedups for Sp$A\&A^T$ and SpA^TA/SpAA^T kernels. Their use in a number of applications warrants their consideration as part of the SpBLAS standard.

- **One "tune" routine per kernel:** For each kernel supported by the standard, we propose the addition of one tuning routine per kernel. Each tuning routine takes a given matrix handle and specification of a workload as input, and produces a new handle as output. The input handle must be in the state `blas_valid_handle`. Logically, the new handle refers to a copy of the input matrix that has been stored internally using a data structure specialized to the associated kernel. The new handle has the same semantics as any other handle. The input handle (and the matrix it represents) remains unchanged. The purpose of the workload specification is to allow the user to control indirectly the resources (time and memory) to be used during the tuning process. By requiring the user both to call a tuning routine explicitly and to specify a workload estimate, we expose the tuning step and cost.

  To maintain upward compatibility with the current SpBLAS interface, the user is not required to call this routine to obtain a correctly running program.

- **Handle tuning save and restore:** We propose the addition of routines that would allow a SpBLAS user to save and restore tuning-related information for an assembled handle, whether or not that handle has been tuned, to a file. The intent is to enable (1) saving and loading profiling or usage information associated with a handle, and (2) recording and recalling any tuning transformations that may have been applied. The precise file format is implementation-specific, though we suggest human-readable formats to promote transparency (*i.e.*, user inspection and modification) of the tuning process. (An alternative to saving to a file is saving to a string or some other descriptive data structure.)

The proposed routines are summarized in Table 8.3. We present the precise interfaces and, where applicable, suggested implementation notes below. We use notation for names and types (in particular, precisions) following the conventions outlined for the SpBLAS. In particular, where `X` appears in a routine name, a one-letter code denoting a data type

Class	Description	Mathematical Operation	Routine
New Kernels	Sp$A\&A^T$ Apply A, A^T to 1 vector each	$y \leftarrow y + \alpha \cdot A \cdot x$; $z \leftarrow z + \beta \cdot A^T \cdot w$	`usa_atv`
	Multiple vector Sp$A\&A^T$ Apply A, A^T to 1 matrix each	$Y \leftarrow Y + \alpha \cdot A \cdot X$; $Z \leftarrow Z + \beta \cdot A^T \cdot W$	`usa_atm`
	SpA^TA, SpAA^T Apply A^TA or AA^T to 1 vector	$y \leftarrow y + \alpha \cdot A^TA \cdot x$ $z \leftarrow z + \beta \cdot A \cdot x$; *or* $y \leftarrow y + \alpha \cdot AA^T \cdot x$ $z \leftarrow z + \beta \cdot A^T \cdot x$;	`usatav`
	Multiple vector SpA^TA, SpAA^T Apply A^TA or AA^T to a matrix	$Y \leftarrow Y + \alpha \cdot A^TA \cdot X$; $Z \leftarrow Z + \beta \cdot A \cdot X$ *or* $Y \leftarrow Y + \alpha \cdot AA^T \cdot X$; $Z \leftarrow Z + \beta \cdot A^T \cdot X$	`usatam`
Kernel-specific tuning	SpMV		`usmv_tune`
	SpTS		`ussv_tune`
	SpMM		`usmm_tune`
	SpTSM		`ussm_tune`
	Sp$A\&A^T$		`usa_atv_tune`
	Multiple vector Sp$A\&A^T$		`usa_atm_tune`
	SpA^TA, SpAA^T		`usatav_tune`
	Multiple vector SpA^TA, SpAA^T		`usatam_tune`
Save and Restore	Save handle profiling/tuning data to a file		`ustuneinfo_save`
	Load handle profiling/tuning data from a file and apply		`ustuneinfo_apply`

Table 8.3: **Proposed SpBLAS extensions to support tuning**. We propose three classes of new routines: (1) new kernels, (2) kernel-specific tuning routines, and (3) routines to save/edit/restore tuning or profiling information. The notation follows Table 8.2.

should be specified. Examples include `s` for single-precision, `d` for double-precision, `c` for single-precision complex, and `z` for double-precision complex. (The Fortran 95 reference implementation defines a fifth type for integers [109].) In addition, argument types must match the routine data type, and we use the following generic names to represent the corresponding type: `SCALAR_IN` denotes a scalar input type and `ARRAY` denotes an array type.

```
-----------------------------------------------------------------------
int BLAS_Xusa_atv( SCALAR_IN alpha,  SCALAR_IN beta,
        blas_sparse_matrix A,
        const ARRAY x ,  int incx,  ARRAY y ,  int incy,
        ARRAY z ,  int incz,  const ARRAY w ,  int incw );
```

Implements simultaneous multiplication of $y \leftarrow y + \alpha \cdot A \cdot x$ and $z \leftarrow z + \beta \cdot A^T \cdot w$, where x, y, z, and w are vectors.

```
-----------------------------------------------------------------------
int BLAS_Xusa_atm( enum blas_order_type order,  SCALAR_IN alpha,  SCALAR_IN beta,
        blas_sparse_matrix A,
        const ARRAY X ,  int ldX,  ARRAY Y ,  int ldY,
        const ARRAY Z ,  int ldZ,  ARRAY W ,  int ldW );
```

Multiple-vector version of BLAS_Xusa_atv.

Figure 8.2: **Proposed SpBLAS interfaces for sparse $A\&A^T$.**

8.3.1 Interfaces for sparse $A\&A^T$, $A^TA\cdot x$, and $AA^T\cdot x$

Our proposed interfaces for the Sp$A\&A^T$, SpA^TA, and SpAA^T kernels mimic the conventions of the existing SpMV and SpTS kernels. The Sp$A\&A^T$ interfaces for the single and multiple vector cases are shown in Figure 8.2. In the case of the SpA^TA/SpAA^T kernel, we propose a single routine with a parameter of type `enum blas_ata_type` whose values, `blas_ata` or `blas_aat`, specify which kernel is desired. The corresponding single and multiple vector interfaces are shown in Figure 8.3. Note that the first parameter to the multiple vector routines is of type `enum blas_order_type`; its value indicates whether the multiple vectors are stored as a matrix in column major (`blas_colmajor`) or row major (`blas_rowmajor`) order.

Discussion

The interfaces have been designed to be largely consistent with existing SpBLAS level 2 and level 3 kernels, and are thus largely self-explanatory. The return codes follow the convention

```
-------------------------------------------------------------------
int BLAS_Xusatav( enum blas_ata_type kernel,  SCALAR_IN alpha,  SCALAR_IN beta,
        blas_sparse_matrix A,  const ARRAY x ,  int incx,
        ARRAY y ,  int incy,  ARRAY z ,  int incz );
```

When `kernel` is `blas_ata`, this routine implements $y \leftarrow y + \alpha \cdot A^T A \cdot x$; $z \leftarrow z + \beta \cdot A \cdot x$. When `kernel` is `blas_aat`, this routine implements $y \leftarrow y + \alpha \cdot AA^T \cdot x$; $z \leftarrow z + \beta \cdot A^T \cdot x$. When $\beta = 0$, then z is left unchanged, *i.e.*, the z vector should be *ignored* in this case. In other words, by distinguishing between zero and non-zero values of β, the user can control whether or not the intermediate product (Ax or $A^T x$) is stored.

```
-------------------------------------------------------------------
int BLAS_Xusatam( enum blas_order_type order,  enum blas_ata_type kernel,
        SCALAR_IN alpha,  SCALAR_IN beta,  blas_sparse_matrix A,
        const ARRAY X ,  int ldX,  ARRAY Y ,  int ldY,  ARRAY Z ,  int ldZ );
```

Multiple vector version of `BLAS_Xusatav`.

Figure 8.3: **Proposed SpBLAS interfaces for sparse $A^T A \cdot x$ and $AA^T \cdot x$.**

outlined for `usmv`, `ussv`, and so on: the routines return a 0 only on success.

We clarify and emphasize one important aspect of behavior for the Sp$A^T A$/SpAA^T kernels: whether or not a vector is supplied to hold an intermediate product. For instance, suppose the user requires the Sp$A^T A$ kernel, but does *not* need the intermediate product Ax. In the interface, this product is accumulated into the vector z according to $z \leftarrow z + \beta Ax$. The dimensions of A may be such that the length of z (or equivalently, the number of rows of A) is very large compared to the number of non-zeros, and further that the user does not want to store z. Then, we define the behavior of β exactly equal to 0 to perform *no* accesses to the vector z. The BLAS standard defines a constant numerical zero against which β can be tested for this case. This behavior allows the user to pass in an unallocated (NULL) or otherwise invalid vector in place of z, thereby avoiding the corresponding storage.

8.3.2 Kernel-specific tune routines

For each kernel, we propose adding a corresponding tuning routine, as shown in Figures 8.4–8.5. The general form of these routine interfaces is

```
A_tuned_handle = BLAS_XusKER_tune (A_handle, num_calls, max_mem, <opts> );
```

where KER specifies the sparse kernel (*e.g.*, mv for SpMV, mm for SpMM). The input arguments are as follows.

- A_handle: any handle to a sparse matrix in the state blas_valid_handle. That is, A_handle should be in the same state in which any handle would be following successful completion of uscr_end.

- num_calls: an integer indicating the number of times the user expects to call the corresponding kernel on the same matrix. The tuning routine uses the value as a hint as to how much time to spend tuning (*i.e.*, doing a "run-time search" and possible data structure conversion). In addition, we propose four special (negative) predefined constants if the number of iterations is unknown or the user does not have any guesses: blas_tune_aggressive (routine can spend as much time for tuning as desired), blas_tune_moderate (routine should spend less time for tuning than the aggressive setting), blas_tune_conservative (routine should only spend "a little bit" of time for tuning), and blas_tune_none (routine should spend no time for tuning). The interpretation of num_calls is implementation-specific, although a reasonable guide might be that tuning should not cost much more than about num_calls untuned executions of the kernel. As discussed in Chapter 3, the cost of aggressive tuning can be roughly 40 SpMV operations.

- max_mem: an integer suggesting the maximum amount of memory the tuning routine should use to store the tuned matrix, in multiples of the matrix size. Recall from Chapter 5 that in some cases it will pay-off to store significantly more memory than the size of the original matrix—in the case of fill, we observe instances where increasing the total storage by more than 25% can nevertheless yield significant speedups. To prevent the user/application from being "surprised" by significant memory usage, we propose the inclusion of this parameter to guide the tuning routine as to how much memory should be used. If max_mem ≤ 0, then no restrictions are placed on the routine's memory usage.

- `<opts>:` a placeholder for additional kernel-specific arguments. Our interfaces primarily use run-time parameters here (*e.g.*, the constant α, the expected values of `incx, incy`, the `enum blas_ata_type kernel` flag for the SpA^TA kernel, ...). Although this particular interface requires a user to keep track of multiple handles, memory usage can be controlled by the library implementation by tracking or tying these handles to a single underlying stored matrix.

The tune routine returns `A_tuned_handle`, a new handle to a matrix in the `blas_valid_handle` state. That is, the state of `A_tuned_handle` is equivalent to that of a handle after the call to `uscr_end`. `A_tuned_handle` may be used anywhere a handle is expected. However, while the user may expect *correct* behavior when using `A_tuned_handle`, she should not expect good performance except in calls to the corresponding kernel with the same `<opts>` specified.

The purpose in returning a new handle is to expose the potential cost in memory to the user. Indeed, the user may free `A_handle` after `A_tuned_handle` has been created. Also, note that the user may call the tune routine on a tuned handle, possibly with different values of `<opts>`. The behavior of such a call is at the discretion of the implementation.

Discussion

Although this proposed tuning interface meets the stated goals of our interface (Section 8.1), there are a number of drawbacks to the library approach. One is that each new kernel requires defining new interfaces and the corresponding tuning routines. Since the SpBLAS is a standard, this aspect of the library approach may be appropriate if only to prevent the standard from growing unmanageably. However, a user's most important kernel is her kernel, so the library approach is limited in instances where the "right" kernel is not available in the library.

Users may also view the strict black-box interface as a disadvantage. For instance, Chapter 5 shows that reordering rows and columns of the matrix can effectively create exploitable dense structure for SpMV. However, to preserve the semantics of the existing interface to the `usmv` routine, the destination vector must be permuted accordingly on entry and again on exit. Depending on the application (*e.g.*, for certain kinds of linear solvers, or eigensolvers in which only eigenvalues are needed), it may be possible to permute only once at the beginning of a sequence of SpMV operations and once again at the end, thus amortizing the cost of applying the permutation. However, to do so a user might require

```
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xusmv_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        enum blas_trans_type transa,  SCALAR_IN alpha,  int incx,  int incy );
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xussv_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        enum blas_trans_type transa,  SCALAR_IN alpha,  int incx );
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xusa_atv_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        SCALAR_IN alpha,  SCALAR_IN beta,
        int incx,  int incy,  int incz,  int incw );
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xusatav_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        enum blas_ata_type kernel,  SCALAR_IN alpha,  SCALAR_IN beta,
        int incx,  int incy,  int incz );
```

Figure 8.4: **Proposed tuning interfaces for the Level 2 SpBLAS routines**. Tuning interfaces for SpMV, SpTS, SpA&A^T, and SpA^TA/SpAA^T.

access to the permutation itself, which is not possible in the current interface.[2]

8.3.3 Handle profile save and restore

To work toward our stated goal of making the tuning process transparent, we propose a mechanism by which concise descriptions of the tuning transformations applied to a given matrix (and on a given platform) may be saved to a file. We refer to this description as a *tuning descriptor*. A saved descriptor may be re-applied later during an application run or in a subsequent application run, possibly to a different matrix. Moreover, if the library implementation chooses a documented (and preferably human-readable) format,

[2]There are a number of possible solutions, including the definition of a kernel for $y \leftarrow A^k x$, or explicit routines to query for and apply the permutations.

```
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xusmm_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        enum blas_order_type order,  enum blas_trans_type transa,
        SCALAR_IN alpha,  int ldX,  int ldY );
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xussm_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        enum blas_order_type order,  enum blas_trans_type transa,
        SCALAR_IN alpha,  int ldX );
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xusa_atm_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        enum blas_order_type order,  SCALAR_IN alpha,  SCALAR_IN beta,
        int ldX,  int ldY,  int ldZ,  int ldW );
-----------------------------------------------------------------
blas_sparse_matrix BLAS_Xusatam_tune(
        blas_sparse_matrix A_handle,  int num_calls,  int max_mem,
        enum blas_order_type order,  enum blas_ata_type kernel,
        SCALAR_IN alpha,  SCALAR_IN beta,
        int ldX,  int ldY,  int ldZ );
```

Figure 8.5: **Proposed tuning interfaces for the Level 3 SpBLAS routines**. Tuning interfaces for SpMV, SpTS, SpA&A^T, and SpA^TA/SpAA^T.

these descriptions could be viewed or even edited by the user.

The tuning transformations presented in this dissertation can all be described concisely. Two informal examples of tuning descriptors might be, (1) "store A in 2×3 block compressed sparse row format," and (2) "split A into the sum $A_1 + A_2 + A_3$, where A_1 is stored in diagonal format using 3 iteration unrolling, A_2 is stored in symmetric 4×4 block format, and A_3 contains all remaining non-zeros in CSR format."[3] These examples, though

[3]In the latter example, these transformations can be further refined to include more detailed splitting criterion, so that the transformation is exactly reproducible on the same matrix.

optimized for a particular matrix, are described in such a way so as to be applicable to (though not necessarily optimal for) a different matrix. In the remainder of this section, we describe our proposed interface for enabling a save and restore functionality, and make a number of recommendations for implementers.

Proposed interfaces

We propose two routines, one to save a descriptor and one to read and apply a descriptor. The interface to the save routine is the following:

```
int BLAS_ustuneinfo_save(blas_sparse_matrix A_handle, const char* outfilename );
```

where, `A_handle` is a handle to any matrix in the `blas_valid_handle` state, and `outfilename` is a valid filename for output. Note that `A_handle` may or may not have been generated by a call to a tuning routine, as we discuss below. This routine overwrites the output file with the tuning descriptor of the handle. The descriptor is entirely specific to the particular library implementation. This routine returns 0 on success, and an error code consistent with the error returns of other SpBLAS routines (refer to the standard for details). In addition, we propose the addition of two error constants so that the implementation can determine the cause of failure: `blas_error_no_file` when the file does not exist, or `blas_error_parse_error` if the descriptor was malformed.

The companion routine to read a descriptor from a file, and apply it to an existing handle is defined as follows:

```
blas_sparse_matrix BLAS_ustuneinfo_apply(blas_sparse_matrix A_handle,
        const char* infilename );
```

This routine reads a tuning descriptor from a file, and returns a new matrix handle in the `blas_valid_handle` state. The new handle corresponds to a matrix representation in which the transformations specified in the file have been applied to a matrix given by `A_handle`. `A_handle` must also be in the `blas_valid_handle` state on the call to `BLAS_ustuneinfo_apply`. This routine returns a non-zero value on error.

Discussion and Recommendations

The user should expect the following behavior from these routines. Let `A_tuned_handle` be the handle generated by a call to a tuning routine on the handle `A_handle`, and suppose we

have saved the tuning descriptor for `A_tuned_handle`. Then, a call to `BLAS_ustuneinfo_apply` on `A_handle` and the same descriptor will return a new handle whose performance is the same as the performance of `A_tuned_handle`.

However, it is difficult to define or specify precisely the semantics or behavior of `BLAS_ustuneinfo_save` and `BLAS_ustuneinfo_apply` because tuning is a matrix and machine specific—and furthermore, library implementation-specific—process. Furthermore, we should not expect the descriptors themselves to be portable across machines.[4] Therefore, we recommend leaving the precise behavior of these routines up to the implementation, with the expected behavior as described above.

Our proposed interface does open the possibility of other kinds information gathering and tuning. Note that the only restriction on an input handle `A_handle` to either of these routines is that it be in the valid state. Therefore, a handle not created by a call to a tuning routine may still be "saved." One instance in which one could imagine using this feature is in profiling the usage of a matrix handle. For example, the library implementation could keep statistics on how often certain kernels are called on a particular matrix, and this information could be stored in a file. A subsequent application run or call to "apply" could use this additional profiling information to tune.

This example raises the issue of how much information can or should be saved. This issue is difficult to resolve precisely at present because the space of optimizations is still being developed, and the information needed will be optimization and machine/vendor specific.

Although we have proposed using files to communicate tuning information, the portability and feasibility of doing so, particularly in parallel and distributed environments, may be problematic. We emphasize that the proposed extensions are a starting point for additional discussion.

Given files to save the tuning information, we strongly recommend that implementors choose a human-readable (*i.e.*, text) and easily parsable format for the descriptors. Doing so in principle allows users to inspect the transformations chosen for a particular matrix. Furthermore, an ambitious user may choose to edit (by hand or otherwise) a descriptor before calling the restore/apply routine to experiment with other tuning styles, since tuning is necessarily a heuristic process.

[4] For instance, a particular machine might include additional information in the descriptor to specify that a some machine-dependent instruction sequence be used.

8.4 Complementary Approaches

There are a number of complementary approaches to a library implementation. One is to implement a library using a language with generic programming constructs such as templates in C++ [230]. This approach has been adopted Blitz++ [309] and the Matrix Template Library (MTL) [278] to build generic libraries in C++ that mimic dense BLAS functionality. The use of templates faciliates the generation of large numbers of library routines with relatively small amount of code, and flexibly handles issues of producing libraries that can handle different precisions. Sophisticated use of templates furthermore allows some limited optimization, such as unrolling. In some cases, loop-fusion like transformations have been implemented using templates [309]. However, this approach lacks an explicit mechanism for dealing with run-time search. Furthermore, the template mechanism for code generation can put enormous stress (in terms of memory and execution time) on the compiler.[5]

Another approach which extends the generic programming idea is compiler-based sparse code generation via restructuring compilers, pursued by Bik [41, 42, 44], Stodghill, *et al.* [287, 5, 215, 214], and Pugh and Shpeisman [254, 172]. These are clean, general approaches to code generation: the user expresses separately both the kernels (as dense code with random access to matrix elements) and a formal specification of a desired sparse data structure; a restructuring compiler combines the two descriptions to produce a sparse implementation. In addition, since any kernel can in principle be expressed, this overcomes a library approach in which all possible kernels must be pre-defined. Nevertheless, we view this technology as complementary to the overall library approach: while sparse compilers could be used to provide the underlying implementations of sparse primitives, they do not explicitly make use of matrix structural information available, in general, only at run-time.[6]

A third approach is to extend an existing library or system. There are a number of application-level libraries (*e.g.*, PETSc [27, 26], among others [128, 267, 258, 154]) and high-level application tools (*e.g.*, MATLAB [296, 132], Octave [111], approaches that apply compiler analyses and transformations to MATLAB code [8, 222]) that provide high-level sparse kernel support. Integration with these systems has a number of advantages, including

[5] This concern is "practical" in nature and could be overcome through better compiler front-end technology. Another minor but related concern is the lack of consistency in how well aspects of the template mechanism are supported, making portability an issue.

[6] Technically, Bik's sparse compiler does use matrix non-zero structure information [44], but is restricted in the following two senses: (1) it assumes that the matrix is available at "compile-time," and (2) it supports a limited number of fixed data structures.

the ability to hide data structure details and the tuning process from the user, and the large potential user base. However, our goal is to provide building blocks in the spirit of the BLAS with the steps and costs of tuning exposed. This model of development has been very successful with other numerical libraries, examples of which include the integration of ATLAS and FFTW tuning systems into the commercial MATLAB system. Thus, it should be possible to integrate a SpBLAS library into an existing system as well.

8.5 Summary

Although an original motivation for the SpBLAS design was to allow matrix and hardware vendor-specific tuning of sparse kernels, our analysis shows that additional mechanisms are needed to support tuning in the style proposed by dissertation. Our specific proposal adds kernel-specific tuning routines (one per supported kernel). In addition, we propose new functionality that allows saving and restoring the tuning descriptors—or even other profiling information—associated with a given handle.

In addition to our tuning proposals, we propose the addition of the Sp$A\&A^T$ and SpA^TA/SpAA^T kernels to the standard. These kernels would enable more efficient implementations of certain iterative linear solvers, eigensolvers, and interior-point algorithms.

Some of the drawbacks of a general library approach are discussed in Section 8.3, for which complementary approaches to sparse kernel generation exist (Section 8.4). We emphasize machine and matrix specific tuning as critical to achieving high performance. An important question is to what extent such tuning, particularly run-time tuning, can be integrated with these other approaches.

There are no mechanisms in the SpBLAS standard for modifying the non-zero values of a matrix. Their omission is understandable since it is difficult to guarantee efficient methods for randomly accessing non-zeros for most sparse formats. Nevertheless, such a facility would allow reuse of the *structure* of the sparse matrix even if the values change. This situation arises, for example, in computing the LU factorization of a matrix where the triangular factors L and U are reused. As it stands, the matrix must be completely re-built from scratch. Nevertheless, at least our tuning descriptor save and restore facility enables a possibly cheaper tuning step. We feel this issue warrants further thought for subsequent revisions of the SpBLAS standard.

Chapter 9

Statistical Approaches to Search

Contents

The unifying idea behind the automatic tuning systems that have inspired our work is choosing an implementation by empirical search where static models appear inadequate.

The process of search consists of augmenting heuristic models of candidate implementations with data collected from actual program runs, and then evaluating these models to choose an implementation. In this chapter, we consider the use of *statistical models*. A statistical approach instantiates models based only on the experimental data, and not (necessarily) on the specifics of any particular kernel. The goal of this approach is to bring general, automatable techniques to bear on the common problem of how to perform a search.

Recall that the general methodology adopted by recent automatic tuning systems can be summarized as follows (see also Chapter 1). First, rather than code a given kernel by hand for each computing platform of interest, these systems contain parameterized code generators that (a) encapsulate possible tuning strategies for the kernel, and (b) output an implementation, usually in a high-level language (like C or Fortran) in order to leverage existing compiler instruction-scheduling technology. By "tuning strategies" we mean that the generators can output implementations which vary by machine characteristics (*e.g.*, different instruction mixes and schedules), optimization techniques (*e.g.*, loop unrolling, cache blocking, the use of alternative data structures), run-time data (*e.g.*, problem size), and kernel-specific transformations and algorithmic variants. Second, these systems tune for a particular platform by *searching* the space of implementations defined by the generator. Typical tuning systems search using a combination of heuristic performance modeling and empirical evaluation (*i.e.*, actually running code for particular implementations). In many cases it is possible to perform the potentially lengthy search process only once per platform. However, even the cost of more frequent compile-time, run-time, or hybrid compile-time/run-time searches can often be amortized over many uses of the kernel.

This chapter begins by arguing that searching is an important and effective means by which to achieve near-peak performance. Indeed, search-based methods have proliferated in a variety of computing contexts, including applications, compilers, and run-time systems, as we discuss in Section 9.4. We show the difficulty of identifying the best implementation empirically, even within a space of reasonable implementations for the well-studied kernel, dense matrix multiply (Section 9.1). Our choice of matrix multiply is motivated by the enormous research effort in developing static models for choosing an implementation. Just as in the sparse matrix-vector multiply (SpMV) example of Section 1.3, we show that the performance behavior of even dense matrix multiply can be a surprising function of tuning parameters. The data we present motivate searching and suggest the necessity of exhaustive search to *guarantee* the best possible implementation.

However, exhaustive searches are frequently infeasiable, and moreover, performance can depend critically on input data that may only be known at run-time. We address these two search-related problems using statistical modeling techniques. Specifically, we propose solutions to the *early stopping problem* and the problem of *run-time implementation selection*. Our techniques are designed to complement existing methods developed for these problems.

The early stopping problem arises when it is not possible to perform an exhaustive search (Section 9.2). Existing tuning systems use a combination of kernel-specific, heuristic performance modeling and empirical search techniques to avoid exhaustive search. We present a complementary technique based on a simple statistical analysis of the data gathered while a search is on-going. Our method allows us to perform only a partial search while still providing an estimate on the performance of the best implementation found.

The run-time selection problem for computational kernels was first posed by Rice [261], and again more recently by Brewer [55] (Section 9.3). Informally, suppose we are given a small number of implementations, each of which is fastest on some class of inputs. We assume we do not know the classes precisely ahead of time, but that we are allowed to collect a sample of performance of the implementations on a subset of all possible inputs. We then address the problem of automatically constructing a set of decision rules which can be applied at run-time to select the best implementation on any given input. We formulate the problem as a statistical classification task and illustrate the variety of models and techniques that can be applied within our framework.

Our analyses are based on data collected from an existing tuning system, PHiPAC [46, 47]. PHiPAC generates highly-tuned, BLAS compatible dense matrix multiply implementations, and a more detailed overview of PHiPAC appears in Section 9.1. Our use of PHiPAC is primarily to supply sample performance data on which we can demonstrate the statistical methods of this chapter. (For complete implementations of the BLAS, we recommend the use of either the ATLAS tuning system or any number of existing hand-tuned libraries when available. In particular, ATLAS improves on PHiPAC ideas, extending their applicability to the entire BLAS standard [325, 324].)

The ideas and timing of this work are indicative of a general trend in the use of search-based methods at various stages of application development. We review the diverse body of related research projects in Section 9.4. Although this dissertation has adopted the perspective of tuning specific computational kernels, for which it is possible to obtain a

significant fraction of peak performance by exploiting all kernel properties, the larger body of related work seeks to apply the idea of searching more generally: within the compiler, within the run-time system, and within specialized applications or problem-solving environments. Collectively, these studies imply a variety of general software architectures for empirical search-based tuning.

The material in this chapter recently appeared in several papers [313, 315, 314].

9.1 Revisiting The Case for Search: Dense Matrix Multiply

We motivate the need for empirical search methods using matrix multiply performance data as a case study. We show that, within a particular space of performance optimization (tuning) parameters, (1) performance can be a surprisingly complex function of the parameters, (2) performance behavior in these spaces varies markedly from architecture to architecture, and (3) the very best implementation in this space can be hard to find. Taken together, these observations suggest that a purely static modeling approach will be insufficient to find the best choice of parameters.

9.1.1 Factors influencing matrix multiply performance

We briefly review the classical optimization strategies for matrix multiply, and make a number of observations that justify some of the assumptions of this chapter (Section 9.1.2 in particular). Roughly speaking, the optimization techniques fall into two broad categories: (1) cache- and TLB-level optimizations, such as cache tiling (blocking) and copy optimization (*e.g.*, as described by Lam [201] or by Goto with respect to TLB considerations [134]), and (2) register-level and instruction-level optimizations, such as register-level tiling, loop unrolling, software pipelining, and prefetching. Our argument motivating search is based on the surprisingly complex performance behavior observed within the space of register- and instruction-level optimizations, so it is important to understand what role such optimizations play in overall performance.

For cache optimizations, a variety of sophisticated static models have been developed for kernels like matrix multiply to help understand cache behavior, to predict optimal tile sizes, and to transform loops to improve temporal locality [118, 130, 201, 330, 70, 219, 80, 66, 67]. Some of these models are expensive to evaluate due to the complexity of accurately modeling interactions between the processor and various levels of the memory hierarchy

[227].[1] Moreover, the pay-off due to tiling, though significant, may ultimately account for only a fraction of performance improvement in a well-tuned code. Recently, Parello, *et al.*, showed that cache-level optimizations accounted for 12–20% of the possible performance improvement in a well-tuned dense matrix multiply implementation on an Alpha 21264 processor based machine, and the remainder of the performance improvement came from register- and instruction-level optimizations [245].

To give some additional intuition for how these two classes of optimizations contribute to overall performance, consider the following experiment comparing matrix multiply performance for a sequence of $n \times n$ matrices. Figure 9.1 shows examples of the cumulative contribution to performance (Mflop/s) for matrix multiply implementations in which (1) only cache tiling and copy optimization have been applied, shown by solid squares, and (2) applying the register-level tiling, software pipelining, and prefetching have been applied in conjunction with these cache optimizations, shown by triangles. These implementations were generated with PHiPAC, discussed below in more detail (Section 9.1.2). In addition, we show the performance of a reference implementation consisting of 3 nested loops coded in C and compiled with full optimizations using a vendor compiler (solid line), and a hand-tuned implementation provided by the hardware vendor (solid circles). The platform used in Figure 9.1 (*top*) is a workstation based on a 333 MHz Sun Ultra 2i processor with a 2 MB L2 cache and the Sun v6 C compiler, and in Figure 9.1 (*bottom*) is an 800 MHz Intel Mobile Pentium III processor with a 256 KB L2 cache and the Intel C compiler. On the Pentium III, we also show the performance of the hand-tuned, assembly-coded library by Goto [134], shown by asterisks.

On the Ultra 2i, the cache-only implementation is 17× faster than the reference implementation for large n, but only 42% as fast as the automatically generated implementation with both cache- and register-level optimizations. On the Pentium III, the cache-only implementation is 3.9× faster than the reference, and about 55–60% of of the register and cache optimized code. Furthermore, the PHiPAC-generated code matches or closely approaches that of the hand-tuned codes. On the Pentium III, the PHiPAC routine is within 5–10% of the performance of the assembly-coded routine by Goto at large n [134]. Thus, while cache-level optimizations significantly increase performance over the reference im-

[1] Indeed, in general it is even hard to approximate the optimal placement of data in memory so as to minimize cache misses. Recently, Petrank and Rawitz have shown the problem of optimal cache-conscious data placement to be in the same hardness class as the minimum coloring and maximum clique problems [246].

plementation, applying them together with register- and instruction-level optimizations is critical to approaching the performance of hand-tuned code.

These observations are an important part of our argument below (Section 9.1.2) motivating empirical search-based methods. First, we focus exclusively on performance in the space of register- and instruction-level optimizations on in-cache matrix workloads. The justification is that this class of optimizations is essential to achieving high-performance. Even if we extend the estimate by Parello, *et al.*—specifically, from the observation that 12–20% of overall performance is due to cache-level optimizations, to 12–60% based on Figure 9.1—there is still a considerable margin for further performance improvements from register- and instruction-level optimizations. Second, we explore this space using the PHiPAC generator. Since PHiPAC-generated code can achieve good performance in practice, we claim this generator is a reasonable one to use.

9.1.2 A needle in a haystack: the need for search

To show the necessity of search-based methods, we examine performance within the space of register-tiled implementations. The automatically generated implementations of Figure 9.1 were created using the parameterized code generator provided by the PHiPAC matrix multiply tuning system [46, 47]. (Although PHiPAC is no longer actively maintained, here the PHiPAC generator has been modified to include some software pipelining styles and prefetching options developed for the ATLAS system [325].) This generator implements register- and instruction-level optimizations including (1) register tiling where non-square tile sizes are allowed, (2) loop unrolling, and (3) a choice of software pipelining strategies and insertion of prefetch instructions. The output of the generator is an implementation in either C or Fortran in which the register-tiled code fragment is fully unrolled; thus, the system relies on an existing compiler to perform the instruction scheduling.

PHiPAC searches the combinatorially large space defined by possible optimizations in building its implementation. To limit search time, machine parameters (such as the number of registers available and cache sizes) are used to restrict tile sizes. In spite of this and other search-space pruning heuristics, searches can generally take many hours or even a day depending on the user-selectable thoroughness of the search. Nevertheless, as we suggest in Figure 9.1, performance can be comparable to hand-tuned implementations.

Consider the following experiment in which we fixed a particular software pipelin-

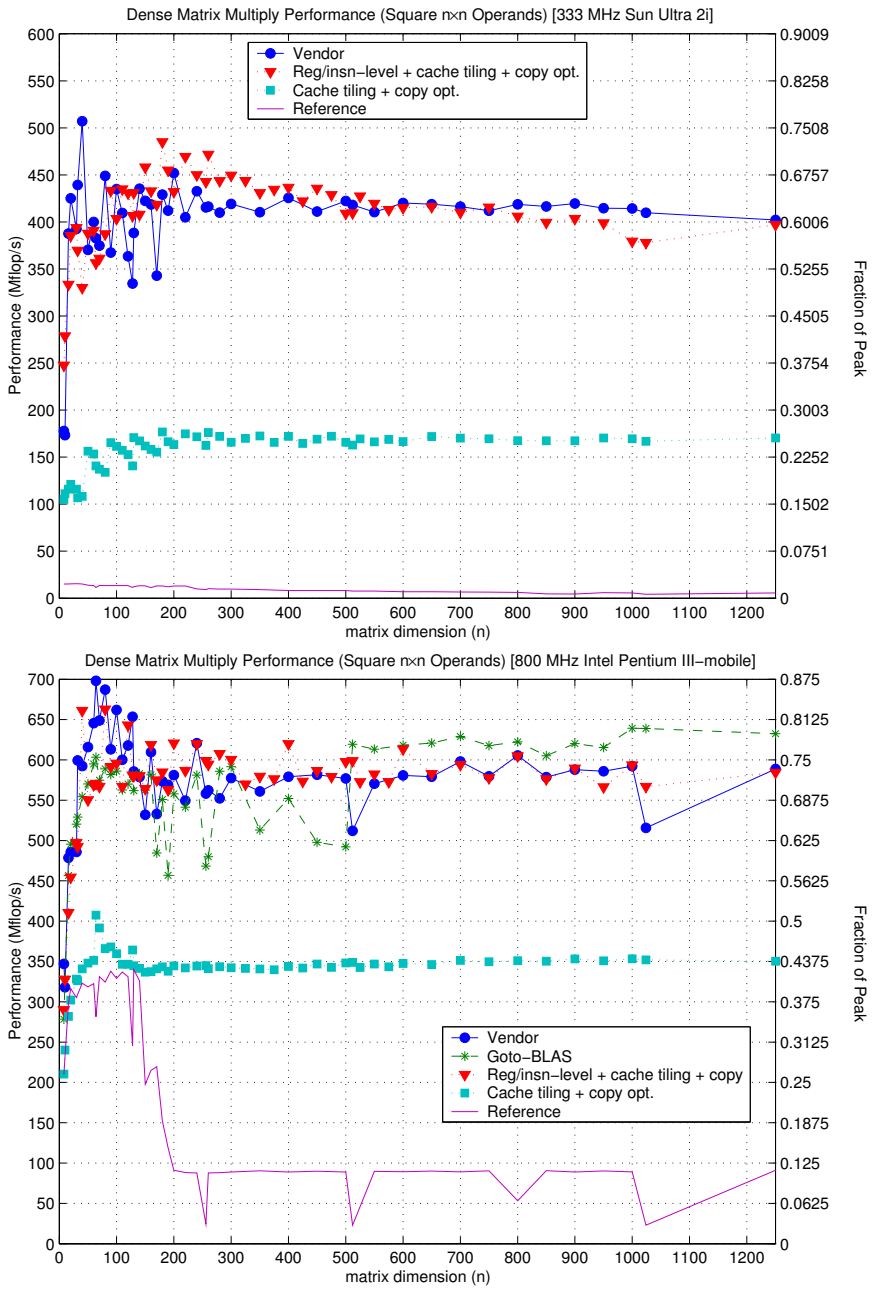

Figure 9.1: **Contributions from cache- and register-level optimizations to dense matrix multiply performance**. Performance (Mflop/s) of $n \times n$ matrix multiply for a workstation based on the Sun Ultra 2i processor (*top*) and an 800 MHz Mobile Pentium III processor (*bottom*). The theoretical peaks are 667 Mflop/s and 800 Mflop/s, respectively. We include values of n that are powers of 2. Although copy optimization (shown by cyan squares) improves performance significantly compared to the reference (purple solid line), register and instruction level optimizations (red triangles) are critical to approaching the performance of hand-tuned code.

ing strategy and explored the space of possible register tile sizes on 11 different platforms. As it happens, this space is three-dimensional and we index it by integer triplets (m_0, k_0, n_0).[2] Using heuristics based on the maximum number of registers available, this space was pruned to contain between 500 and 10000 reasonable implementations per platform.

Figure 9.2 (*top*) shows what fraction of implementations (y-axis) achieved at least a given fraction of machine peak (x-axis), on a workload in which all matrix operands fit within the largest available cache. On two machines, a relatively large fraction of implementations achieve close to machine peak: 10% of implementations on the Power2/133 and 3% on the Itanium 2/900 are within 90% of machine peak. By contrast, only 1.7% on a uniprocessor Cray T3E node, 0.2% on the Pentium III-M/800, and fewer than 4% on a Sun Ultra 2i/333 achieved more than 80% of machine peak. And on a majority of the platforms, fewer than 1% of implemenations were within 5% of the best. Worse still, nearly 30% of implementations on the Cray T3E ran at less than 15% of machine peak. Two important ideas emerge from these observations: (1) different machines can display widely different characteristics, making generalization of search properties across them difficult, and (2) finding the very best implementations is akin to finding a "needle in a haystack."

The latter difficulty is illustrated in Figure 9.2 (*bottom*), which shows a 2-D slice ($k_0 = 1$) of the 3-D tile space on the Ultra 2i/333. The plot is color coded from dark blue=66 Mflop/s to red=615 Mflop/s, and the lone red square at $(m_0 = 2, n_0 = 3)$ was the fastest. The black region in the upper-right of Figure 9.2 (*bottom*) was pruned (*i.e.*, not searched) based on the number of registers. We see that performance is not a smooth function of algorithmic details as we might have expected. Accurate sampling, interpolation, or other modeling of this space is difficult. Like Figure 9.2 (*top*), this motivates empirical search.

9.2 A Statistical Early Stopping Criterion

Although an exhaustive search can guarantee finding the best implementation within the space of implementations considered, such searches can be demanding, requiring dedicated machine time for long periods. If we assume that search will be performed only once per platform, then an exhaustive search may be justified. However, users today are more frequently running tuning systems themselves, or may wish to build kernels that are customized

[2] By dimensional constraints on the operation $C \leftarrow AB$, we choose an $m_0 \times k_0$ tile for the A operand, a $k_0 \times n_0$ tile for the B operand, and a $m_0 \times n_0$ tile for the C operand.

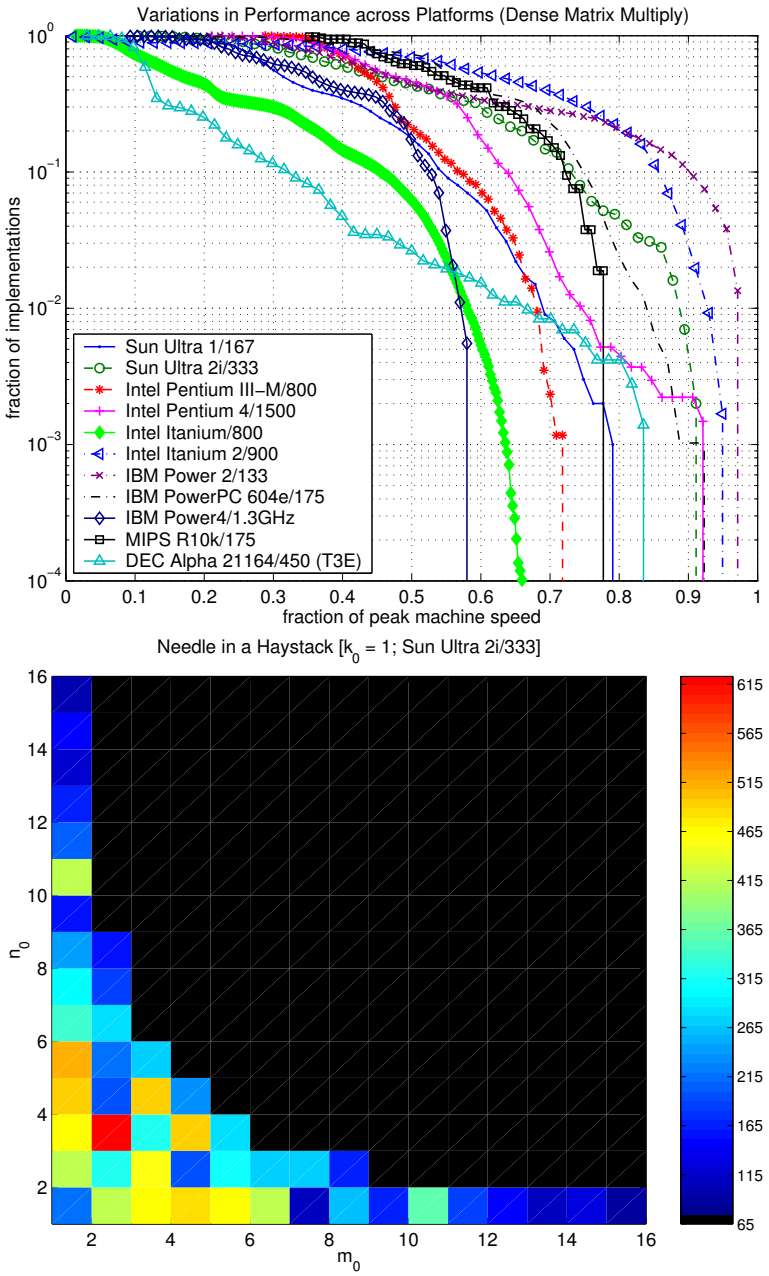

Figure 9.2: **A needle in a haystack**. (*Top*) The fraction of implementations (y-axis) attaining at least a given level of peak machine speed (x-axis) on six platforms. The distributions of performance vary dramatically across platforms. (*Bottom*) A 2-D slice of the 3-D register tile space on the Sun Ultra 2i/333 platform. Each square represents an implementation ($m_0 \times 1 \times n_0$ tile) shaded by performance (color-scale in Mflop/s). The fastest occurs at ($m_0 = 2, n_0 = 3$), having achieved 615 Mflop/s out of a 667 Mflop/s peak. The dark region extending to the upper-right has been pruned from the search. Finding the optimal point in these highly irregular spaces can be like looking for a "needle in a haystack."

for their particular application or non-standard hardware configuration. Furthermore, the notion of run-time searching, as pursued in dynamic optimization systems (Section 9.4) demand extensive search-space pruning.

Thus far, tuning systems have sought to prune the search spaces using heuristics and performance models specific to their code generators. Here, we consider a complementary method for stopping a search early based only on performance data gathered during the search. In particular, Figure 9.2 (*top*), described in the previous section, suggests that even when we cannot otherwise model the space, we do have access to the statistical distribution of performance. On-line estimation of this distribution is the key idea behind the following early stopping criterion. This criterion allows a user to specify that the search should stop when the probability that the performance of the best implementation observed is approximately within some fraction of the best possible within the space.

9.2.1 A formal model and stopping criterion

The following is a formal model of the search process. Suppose there are N possible implementations. When we generate implementation i, we measure its performance x_i. Assume that each x_i is normalized so that $\max_i x_i = 1$. (We discuss the issue of normalization further in Section 9.2.1.) Define the space of implementations as $S = \{x_1, \ldots, x_N\}$. Let X be a random variable corresponding to the value of an element drawn uniformly at random from S, and let $n(x)$ be the number of elements of S less than or equal to x. Then X has a cumulative distribution function (cdf) $F(x) = Pr[X \leq x] = n(x)/N$. At time t, where t is an integer between 1 and N inclusive, suppose we generate an implementation at random *without* replacement. Let X_t be a random variable corresponding to the observed performance, and furthermore let $M_t = \max_{1 \leq i \leq t} X_i$ be the random variable corresponding to the maximum observed performance up to t.

We can now ask the following question at each time t: what is the probability that M_t is at least $1 - \epsilon$, where ϵ is chosen by the user or library developer based on performance requirements? When this probability exceeds some desired threshold $1 - \alpha$, also specified by the user, then we stop the search. Formally, this stopping criterion can be expressed by

$$Pr[M_t > 1 - \epsilon] \quad > \quad 1 - \alpha$$

or, equivalently,

$$Pr[M_t \leq 1 - \epsilon] \quad < \quad \alpha \quad . \tag{9.1}$$

Let $G_t(x) = Pr[M_t \leq x]$ be the cdf for M_t. We refer to $G_t(x)$ as the *max-distribution*. Given $F(x)$, the max-distribution—and thus the left-hand side of Equation (9.1)—can be computed exactly as we show below in Section 9.2.1. However, since $F(x)$ cannot be known until an entire search has been completed, we must approximate the max-distribution. We use the standard approximation for $F(x)$ based on the current sampled performance data up to time t—the so-called empirical cdf (ecdf) for X. Section 9.2.1 presents our early stopping procedure based on these ideas, and discusses the issues that arise in practice.

Computing the max-distribution exactly and approximately

We explicitly compute the max-distribution as follows. First, observe that

$$G_t(x) = Pr[M_t \leq x] \quad = \quad Pr[X_1 \leq x, X_2 \leq x, \ldots, X_t \leq x].$$

Recall that the search proceeds by choosing implementations uniformly at random without replacement. We can look at the calculation of the max-distribution as a counting problem. At time t, there are $\binom{N}{t}$ ways to have selected t implementations. Of these, the number of ways to choose t implementations, all with performance at most x, is $\binom{n(x)}{t}$, provided $n(x) \geq t$. To cover $n(x) < t$, let $\binom{a}{b} = 0$ when $a < b$ for notational ease. Thus,

$$G_t(x) \quad = \quad \frac{\binom{n(x)}{t}}{\binom{N}{t}} = \frac{\binom{N \cdot F(x)}{t}}{\binom{N}{t}} \tag{9.2}$$

where the latter equality follows from the definition of $F(x)$.

We cannot evaluate the max-distribution after $t < N$ samples because of its dependence on $F(x)$. However, we can use the t observed samples to approximate $F(x)$ using the empirical cdf (ecdf) $\hat{F}_t(x)$ based on the t samples:

$$\hat{F}_t(x) = \frac{\hat{n}_t(x)}{t} \tag{9.3}$$

where $\hat{n}_t(x)$ is the number of observed samples that are at most x at time t. We can now approximate $G_t(x)$ by the following $\hat{G}_t(x)$:

$$\hat{G}_t(x) \quad = \quad \frac{\binom{\lceil N \cdot \hat{F}_t(x) \rceil}{t}}{\binom{N}{t}} \tag{9.4}$$

The ceiling ensures that we evaluate the binomial coefficient in the numerator using an integer. Thus, our empirical stopping criterion, which approximates the "true" stopping criterion shown in Equation (9.1), is

$$\hat{G}_t(x) \quad \leq \quad \alpha \qquad . \tag{9.5}$$

Implementing an early stopping procedure

A search with our early stopping criterion proceeds as follows. First, a user or library designer specifies the search tolerance parameters ϵ and α. Then at each time t, the automated search system carries out the following steps:

1. Compute $\hat{F}_t(1-\epsilon)$, Equation (9.3), using *rescaled* samples as described below.
2. Compute $\hat{G}_t(1-\epsilon)$, Equation (9.4).
3. If the empirical criterion, Equation (9.5), is satisified, then terminate the search.

Note that the ecdf $\hat{F}_t(x)$ models $F(x)$, making no assumptions about how performance varies with respect to the implementation tuning parameters. Thus, unlike gradient descent methods, this model can be used in situations where performance is an irregular function of tuning parameters, such as the example shown in Figure 9.2 (*bottom*).

There are two additional practical issues to address. First, due to inherent variance in the estimate $\hat{F}_t(x)$, it may be problematic to evaluate empirical stopping criterion, Equation (9.5), at every time t. Instead, we wait until t exceeds some minimum number of samples, $t_{\min}$, and then evaluate the stopping criterion at periodic intervals. For the experiments in this study, we use $t_{\min} = .02N$, and re-evaluate the stopping criterion at every $.01N$ samples, following a rule-of-thumb regarding ecdf approximation [40].

Second, we need a reasonable way to scale performance so that it lies between 0 and 1. Scaling by theoretical machine peak speed is not appropriate for all kernels, and a true upper bound on performance may be difficult to estimate. We choose to *rescale* the samples at each time t by the *current* maximum. That is, if $\{s_1, \ldots, s_t\}$ are the observed values of performance up to time t, and $m_t = \max_{1 \le k \le t} s_k$, then we construct the ecdf $\hat{F}_t(x)$ using the values $\{s_k/m_t\}$. This rescaling procedure tends to overestimate the fraction of samples near the maximum, meaning the stopping condition will be satisfied earlier than when it would have been satisfied had we known the true distribution $F(x)$. Furthermore, we would expect that by stopping earlier than the true condition indicates, we will tend to find implementations whose performance is less than $1-\epsilon$. Nevertheless, as we show in Section 9.2.2, in practice this rescaling procedure appears to be sufficient to characterize the shape of the distributions, meaning that for an appropriate range of α values, we still

tend to find implementations with performance greater than $1 - \epsilon$.[3]

There are distributions for which we would not expect good results. For instance, consider a distribution in which 1 implementation has performance equal to 1, and the remaining $N - 1$ implementations have performance equal to $\frac{1}{2}$, where $N \gg 1$. After the first $t_{\min}$ samples, under our rescaling policy, all samples will be renormalized to 1 and the ecdf $\hat{F}_t(1 - \epsilon)$ will evaluate to zero for any $\epsilon > 0$. Thus, the stopping condition will be immediately satisfied, but the realized performance will be $\frac{1}{2}$. This artificial example might seem unrepresentative of distributions arising in practice (as we verify in Section 9.2.2), but it is important to note the potential pitfalls.

9.2.2 Results and discussion using PHiPAC data

We applied the above model to the register tile space data for the platforms shown in Figure 9.2 (*top*). On each platform, we simulated 300 searches using a random permutation of the exhaustive search data collected for Figure 9.2 (*top*). For various values of ϵ and α, we measured (1) the average stopping time over all searches, and (2) the average proximity in performance of the implementation found to the best found by exhaustive search.

Figures 9.3–9.6 show the results for the Intel Itanium 2, Alpha 21164 (Cray T3E node), Sun Ultra 2i, and Intel Mobile Pentium III platforms, respectively. The top half of Figures 9.3–9.6 show the average stopping time as a fraction of the search space size for various values of ϵ and α. That is, each plot shows at what value of t/N the empirical stopping criterion, Equation (9.5), was satisfied.

Since our rescaling procedure will tend to overestimate the fraction of implementations near the maximum (as discussed in Section 9.2.1), we must check that the performance of the implementation chosen is indeed close to (if not well within) the specified tolerance ϵ when α is "small," and moreover what constitutes a small α. Therefore, the bottom half of Figures 9.3–9.6 shows the average proximity to the best performance when the search stopped. More specifically, for each (ϵ, α) we show $1 - \bar{M}_t$, where $\bar{M}_t$ is the average observed maximum at the time t when Equation (9.5) was satisfied. (Note that $\bar{M}_t$ is the "true" performance where the maximum performance is taken to be 1.)

[3] We conjecture, based on some preliminary experimental evidence, that it may be possible to extend the known theoretical bounds on the quality of ecdf approximation due to Kolmogorov and Smirnov [48, 236, 195] to the case where samples are rescaled in this way. Such an extension would provide theoretical grounds that this rescaling procedure is reasonable.

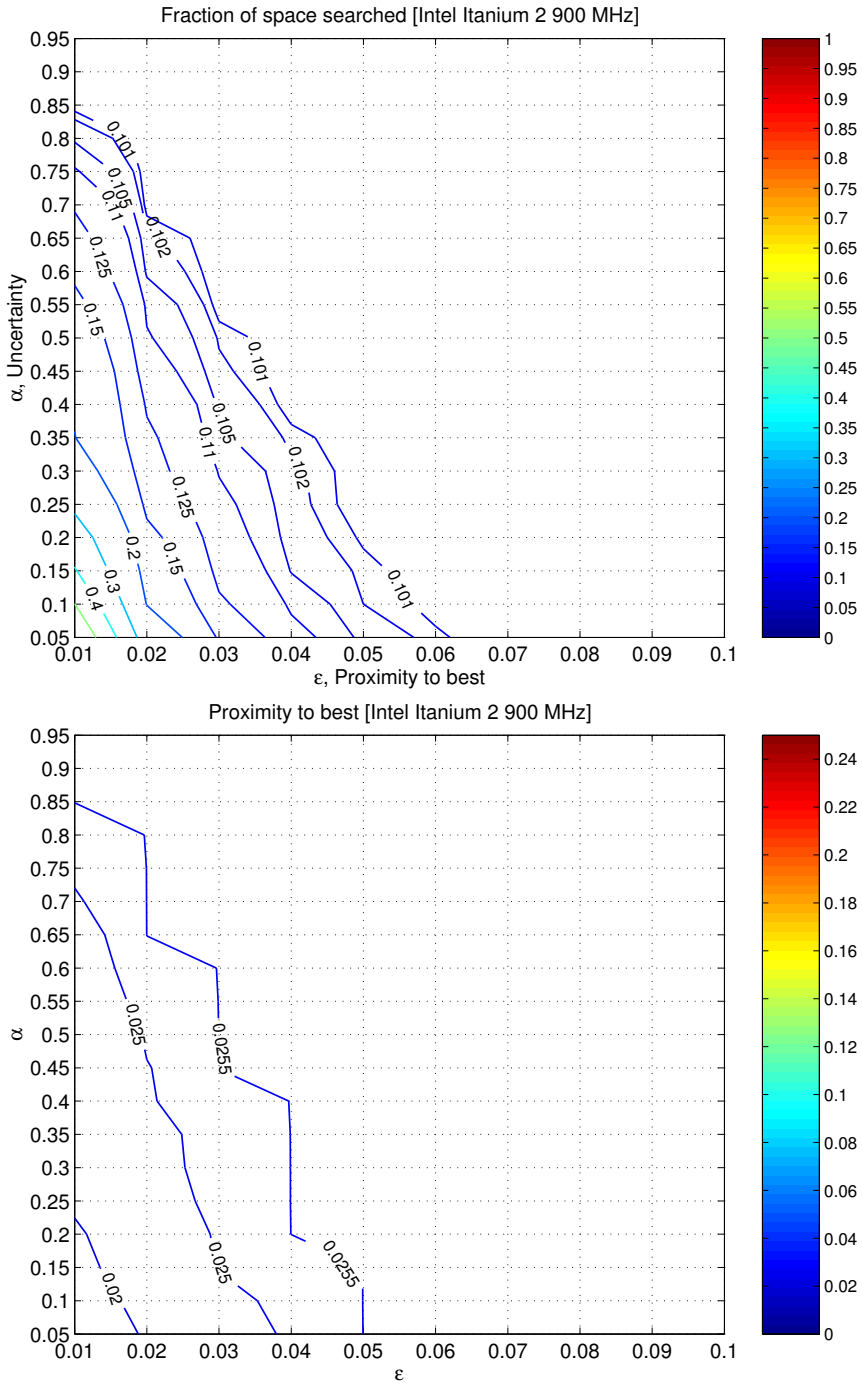

Figure 9.3: **Stopping time and performance of the implementation found: Intel Itanium 2/900**. Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the Intel Itanium 2/900 platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

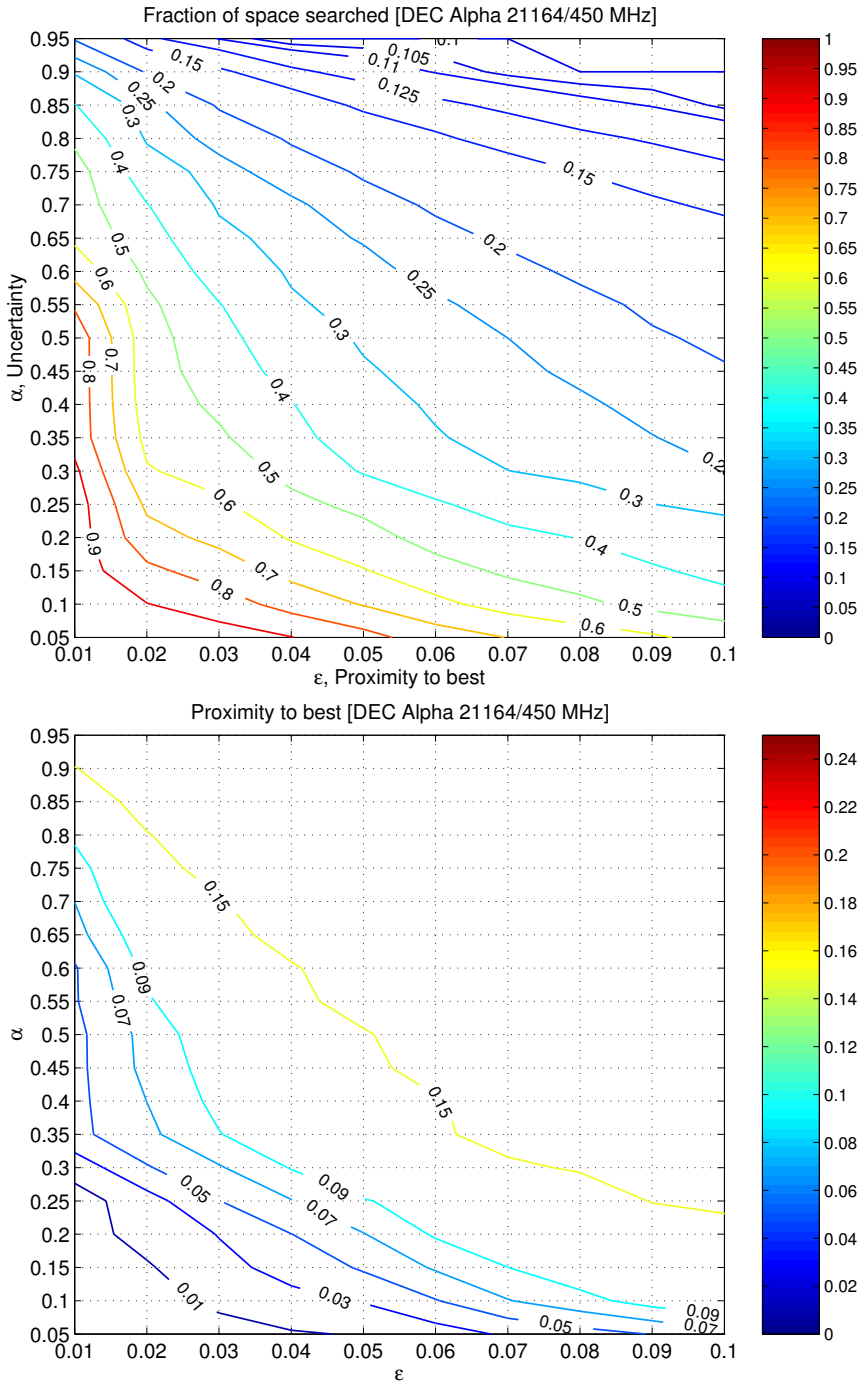

Figure 9.4: **Stopping time and performance of the implementation found: DEC Alpha 21164/450 (Cray T3E node)**. Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the DEC Alpha 21164/450 (Cray T3E node) platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

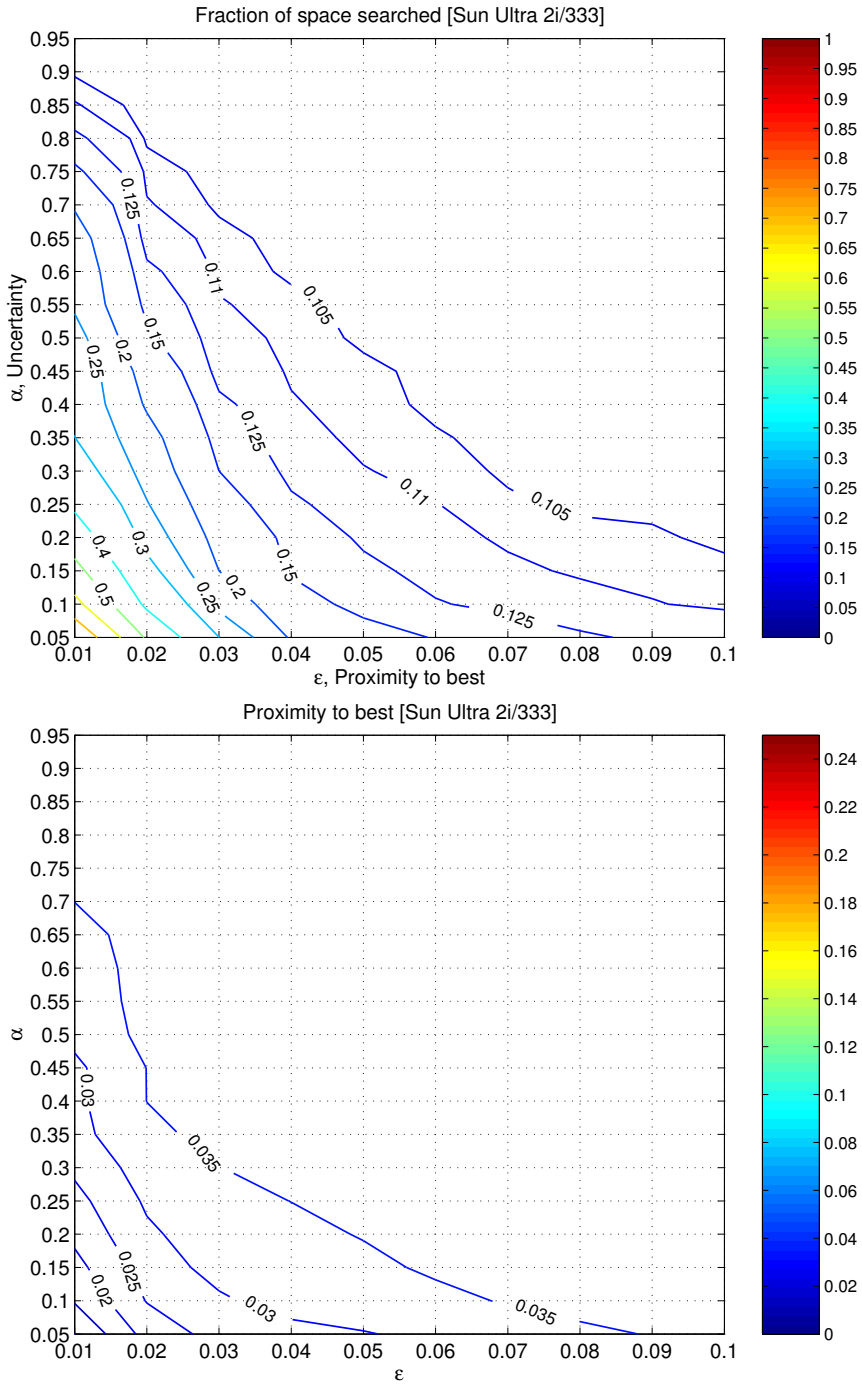

Figure 9.5: **Stopping time and performance of the implementation found: Sun Ultra 2i/333**. Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the Sun Ultra 2i/333 platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

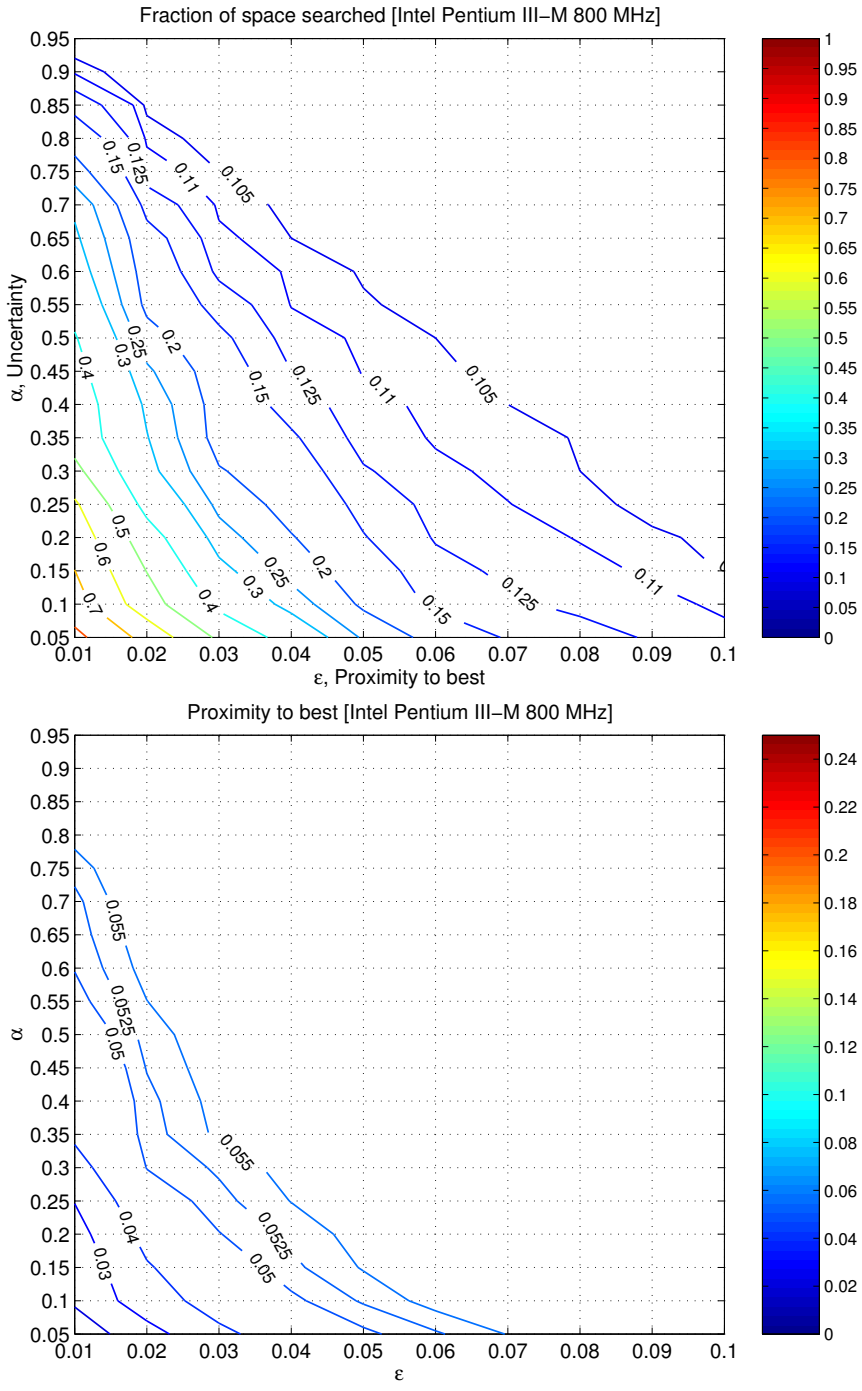

Figure 9.6: **Stopping time and performance of the implementation found: Intel Mobile Pentium III/800**. Average stopping time (*top*), as a fraction of the total search space, and proximity to the best performance (*bottom*), as the difference between normalized performance scores, on the Intel Mobile Pentium III/800 platform as functions of the tolerance parameters ϵ (x-axis) and α (y-axis).

Suppose the user selects $\epsilon = .05$ and $\alpha = .1$, and then begins the search. These particular parameter values can be interpreted as the request, "stop the search when we find an implementation within 5% of the best with less than 10% uncertainty." Were this search conducted on the Itanium 2, for which many samples exhibit performance near the best within the space, we would observe that the search ends after sampling just under 10.2% of the full space on average (Figure 9.3 (*top*)), having found an implementation whose performance was within 2.55% of the best (Figure 9.3 (*bottom*)). Note that we requested an implementation within 5% ($\epsilon = .05$), and indeed the distribution of performance on the Itanium 2 is such that we could do even slightly better (2.55%) on average.

The Alpha 21164 T3E node is a difficult platform on which to stop searches early. According to Figure 9.2 (*top*), the Alpha 21164 distribution has a relatively long tail, meaning very few implementations are fast. At $\epsilon = .05$ and $\alpha = .1$, Figure 9.4 (*top*) shows that indeed we must sample about 70% of the full space. Still, we do find an implementation within about 3% of the best on average. Indeed, for $\epsilon = .05$, we will find implementations within 5% of the best for all $\alpha \lesssim .15$.

On the Ultra 2i (Figure 9.5), the search ends after sampling about 14% of the space, having found an implementation between 3–3.5% of the best, again at $\epsilon = .05, \alpha = .1$. On the Pentium III (Figure 9.6), the search ends after just under 20%, having found an implementation within 5.25% of the best.

The differing stopping times across all four platforms show that the model does indeed adapt to the characteristics of the implementations and the underlying machine. Furthermore, the size of the space searched can be reduced considerably, without requiring any assumptions about how performance varies within the space. Moreover, these examples suggest that the approximation $\hat{F}_t(x)$ to the true distribution $F(x)$ is a reasonable one in practice, judging by the proximity of the performance of the implementation selected compared to $1 - \epsilon$ when $\alpha \lesssim .15$.

There are many other possible combinatorial search algorithms, including simulated annealing and the use of genetic algorithms, among others. We review the application of these techniques to related search-based systems in Section 9.4. In prior work, we have experimented with search methods including random, ordered, best-first, and simulated annealing [46]. The OCEANS project [190] has also reported on a quantitative comparison of these methods and others applied to a search-based compilation system. In these two instances, random search was comparable to and easier to implement than competing tech-

niques. Our stopping condition adds user-interpretable bounds (ϵ and α) to the random method, while preserving the simplicity of the random method's implementation.

In addition, the idea of user-interpretable bounds allows a search system to provide feedback to the user in other search contexts. For example, if the user wishes to specify a maximum search time (*e.g.*, "stop searching after 1 hour"), the estimate of the probability $Pr[M_t > 1 - \epsilon]$ could be computed for various values of ϵ at the end of the search and reported to the user. A user could stop and resume searches, using these estimates to gauge the likely difficulty of tuning on her particular architecture.

Finally, the stopping condition as we have presented complements existing pruning techniques: a random search with our stopping criterion can always be applied to any space after pruning by other heuristics or methods.

9.3 Statistical Classifiers for Run-time Selection

The previous sections assume that a single optimal implementation exists. For some applications, however, several implementations may be "optimal" depending on the run-time inputs. In this section, we consider the run-time implementation selection problem [261, 55]: how can we automatically build decision rules to select the best implementation for a given input? Below, we treat this problem as a statistical classification task. We show how the problem might be tackled from this perspective by applying three types of statistical models to a matrix multiply example. In this example, given the dimensions of the input matrices, we must choose at run-time one implementation from among three, where each of the three implementations has been tuned for matrices that fit in different levels of cache.

9.3.1 A formal framework

We can pose the selection problem as the following classification task. Suppose we are given

1. a set of m "good" implementations of an algorithm, $A = \{a_1, \ldots, a_m\}$ which all give the same output when presented with the same input,

2. a set of n samples $S_0 = \{s_1, s_2, \ldots, s_n\}$ from the space S of all possible inputs (*i.e.*, $S_0 \subseteq S$), where each s_i is a d-dimensional real vector, and

3. the execution time $T(a, s)$ of algorithm a on input s, where $a \in A$ and $s \in S$.

Our goal is to find a decision function $f(s)$ that maps an input s to the best implementation in A, *i.e.*, $f : S \rightarrow A$. The idea is to construct $f(s)$ using the performance of the implementations in A on a sample of the inputs S_0. We refer to S_0 as the *training set*, and we refer to the execution time data $T(a, s)$ for $a \in A, s \in S_0$ as the *training data*. In geometric terms, we would like to partition the input space by implementation, as shown in Figure 9.7 (*left*). This partitioning would occur at compile (or "build") time. At run-time, the user calls a single routine which, when given an input s, evaluates $f(s)$ to select an implementation.

The decision function f models the relative performance of the implementations in A. Here, we consider three types of statistical models that trade-off classification accuracy against the cost of building f and the cost of executing f at run-time. Roughly speaking, we can summarize these models as follows:

1. *Parametric data modeling*: We can build a parametric statistical model of the execution time data directly. For each implementation, we posit a parameterized model of execution time and use the training data to estimate the parameters of this model (*e.g.*, by linear regression for a linear model). At run-time, we simply evaluate the models to predict the execution time of each implementation. This method has been explored in prior work on run-time selection by Brewer [55]. Because we choose the model of execution time, we can control the cost of evaluating f by varying the complexity of the model (*i.e.*, the number of model parameters).

2. *Parametric geometric modeling*: Rather than model the execution time directly, we can also model the shape of the partitions in the input space parametrically, by, say, assuming that the *boundaries* between partitions can be described concisely by parameterized functions. For example, if the input space is two-dimensional, we might posit that each boundary is a straight line which can of course be described concisely by specifying its slope and intercept. Our task is to estimate the parameters (*e.g.*, slope and intercept) of all boundaries using the training data. Such a model might be appropriate if a sufficiently accurate model of execution time is not known but the boundaries can be modeled. Like parametric data modeling methods, we can control the cost of evaluating f by our choice of functions that represent the boundaries.

3. *Nonparametric geometric modeling*: Rather than assume that the partition boundaries have a particular shape, we can also construct implicit models of the boundaries

in terms of the actual data points. In statistical terms, this type of representation of the boundaries is called nonparametric. Here, we use the *support vector method* to construct just such a nonparametric model [308]. The advantage of the nonparametric approach is that we do not have to make any explicit assumptions about the input distributions, running times, or geometry of the partitions. However, we will need to store at least some subset of the data points which make up the implicit boundary representation. Thus, the reduction in assumptions comes at the price of more expensive evaluation and storage of f compared to a parametric method.

(This categorization of models implies a fourth method: nonparametric data modeling. Such models are certainly possible, for example, by the use of support vector regression to construct a nonparametric model of the data [281]. We do not consider these models here.)

To illustrate the classification framework, we apply the above three models to a matrix multiply example. Consider the operation $C \leftarrow C + AB$, where A, B, and C are dense matrices of size $M \times K$, $K \times N$, and $M \times N$, respectively, as shown in Figure 9.7 (*right*). These three parameters make the input space S three-dimensional. In PHiPAC, it is possible to generate different implementations tuned on different matrix workloads [47]. Essentially, this involves conducting a search where the size of the matrices on which the implementations are benchmarked is specified so that the matrices fit within a particular cache level. For instance, we could have three implementations, one tuned for the matrix sizes that fit approximately within L1 cache, those that fit within L2, and all larger sizes.

We compare the accuracy of the above modeling methods using two metrics. First, we use the average misclassification rate, *i.e.*, the fraction of test samples mispredicted. We always choose the *test set* S' to exclude the training data S_0, that is, $S' \subseteq (S - S_0)$. However, if the performance difference between two implementations is small, a misprediction may still be acceptable. Thus, our second comparison metric is the slow-down of the predicted implementation relative to the true best. That is, for each point in the test set, we compute the relative slow-down $\frac{t_{\text{selected}}}{t_{\text{best}}} - 1$, where t_{selected} and t_{best} are the execution times of the predicted and best algorithms for a given input, respectively. For a given modeling technique, we consider the distribution of slow-downs for points in the test set.

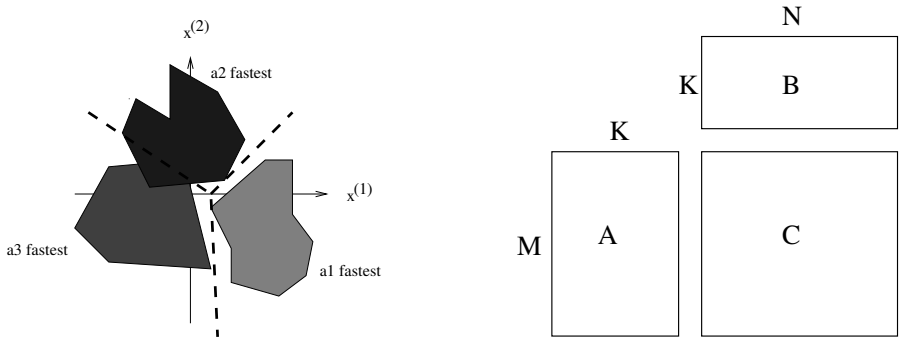

Figure 9.7: **Illustration of the run-time implementation selection problem**. (*Left*) Geometric interpetation of the run-time selection problem: A hypothetical 2-D input space in which one of three algorithms runs fastest in some region of the space. Our goal is to partition the input space by algorithm. (*Right*) The matrix multiply operation $C \leftarrow C+AB$ is specified by three dimensions, M, K, and N.

9.3.2 Parametric data model: linear regression modeling

In our first approach, proposed by Brewer [55], we postulate a parametric model for the running time of each implementation off-line, and then choose the fastest implementation based on the execution time predicted by the models at run-time. For instance, matrix multiply on $N \times N$ matrices might have a running time for implementation a of the form

$$T_a(N) = \beta_3 N^3 + \beta_2 N^2 + \beta_1 N + \beta_0.$$

where we can use standard regression techniques to determine the coefficients β_k, given the running times on some sample inputs S_0. The decision function is just $f(s) = \text{argmin}_{a \in A} T_a(s)$.

A strength of this approach is that the models, and thus the accuracy and cost of a prediction, can be as simple or as complicated as desired. For example, for matrices of more general sizes, (M, K, N), we might hypothesize a model $T_a(M, K, N)$ with linear coefficients and the terms MKN, MK, KN, MN, M, K, N, and 1:

$$T_a(N) = \beta_7 MKN + \beta_6 MK + \beta_5 KN + \beta_4 MN + \beta_3 M + \beta_2 K + \beta_1 N + \beta_0. \qquad (9.6)$$

We can even eliminate terms whose coefficients are "small" to reduce the run-time cost of generating a prediction. For matrix multiply, a simple model of this form could even be automatically derived by an analysis of the 3-nested loops structure. However, in general it might be difficult to determine a sufficiently precise parametric form that captures the interaction effects between the processor and all levels of the memory hierarchy. Moreover,

for other more complicated kernels or algorithms—having, say, more complicated control flow like recursion or conditional branches—such a model may be more difficult to derive.

9.3.3 Parametric geometric model: separating hyperplanes

One geometric approach is to first assume that there are some number of boundaries, each described parametrically, that divide the implementations, and then find best-fit boundaries with respect to an appropriate cost function.

Formally, associate with each implementation a a weight function $w_{\theta_a}(s)$, parameterized by θ_a, which returns a value between 0 and 1 for some input value s. Furthermore, let the weights satisfy the property, $\sum_{a \in A} w_{\theta_a}(s) = 1$. Our decision function selects the algorithm with the highest weight on input s, $f(s) = \text{argmax}_{a \in A} \{w_{\theta_a}(s)\}$. We can compute the parameters $\theta_{a_1}, \ldots, \theta_{a_m}$ (and thus, the weights) so as to minimize the the following weighted execution time over the training set:

$$C(\theta_{a_1}, \ldots, \theta_{a_m}) = \frac{1}{|S_0|} \sum_{s \in S_0} \sum_{a \in A} w_{\theta_a}(s) \cdot T(a, s). \tag{9.7}$$

If we view $w_{\theta_a}(s)$ as a probability of selecting algorithm a on input s, then C is a measure of the expected execution time if we first choose an input uniformly at random from S_0, and then choose an implementation with the probabilities given by the weights on input s.

In this formulation, inputs s with large execution times $T(a, s)$ will tend to dominate the optimization. Thus, if all inputs are considered to be equally important, it may be desirable to use some form of normalized execution time. We defer a more detailed discussion of this issue to Section 9.3.5.

Of the many possible choices for $w_{\theta_a}(\cdot)$, we choose the *logistic* function,

$$w_{\theta_a}(s) = \frac{\exp\left(\theta_a^T s + \theta_{a,0}\right)}{\sum_{b \in A} \exp\left(\theta_b^T s + \theta_{b,0}\right)} \tag{9.8}$$

where θ_a has the same dimensions as s, $\theta_{a,0}$ is an additional parameter to estimate. The denominator ensures that $\sum_{a \in A} w_{\theta_a}(s) = 1$. Although there is some statistical motivation for choosing the logistic function [177], in this case it also turns out that the derivatives of the weights are particularly easy to compute. Thus, we can estimate θ_a and $\theta_{a,0}$ by minimizing Equation (9.7) numerically using Newton's method.

A nice property of the weight function is that f is cheap to evaluate at run-time: the linear form $\theta_a^T s + \theta_{a,0}$ costs $O(d)$ operations to evaluate, where d is the dimension of

the space. The primary disadvantage of this approach is that the same linear form makes this formulation equivalent to asking for hyperplane boundaries to partition the space. Hyperplanes may not be a good way to separate the input space as we shall see below. Of course, other forms are certainly possible, but positing their precise form *a priori* might not be obvious, and more complicated forms could also complicate the numerical optimization.

9.3.4 Nonparametric geometric model: support vectors

Techniques exist to model the partition boundaries nonparametrically. The support vector (SV) method is one way to construct just such a nonparametric model, given a labeled sample of points in the space [308].

Specifically, each training sample $s_i \in S_0$ is given a label $l_i \in A$ to indicate which implementation was fastest on input s_i. That is, the training points are assigned to classes by implementation. The SV method then computes a partitioning by selecting a subset of training points that best represents the location of the boundaries, where by "best" we mean that the minimum geometric distance between classes is maximized.[4] The resulting decision function $f(s)$ is essentially a linear combination of terms with the factor $K(s_i, s)$, where only s_i in the selected subset are used, and K is some symmetric positive definite function. Ideally, K is chosen to suit the data, but there are also a variety of "standard" choices for K as well. We refer the reader to the description by Vapnik for more details on the theory and implementation of the method [308].

The SV method is regarded as a state-of-the-art method for the task of statistical classification on many kinds of data, and we include it in our discussion as a kind of practical upper-bound on prediction accuracy. However, the time to compute $f(s)$ is up to a factor of $|S_0|$ greater than that of the other methods since some fraction of the training points must be retained to evaluate f. Thus, evaluation of $f(s)$ is possibly much more expensive to calculate at run-time than either of the other two methods.

9.3.5 Results and discussion with PHiPAC data

We offer a brief comparison of the three methods on the matrix multiply example described in Section 9.3.1, using PHiPAC to generate the implementations on a Sun Ultra 1/170 workstation with a 16 KB L1 cache and a 512 KB L2 cache.

[4] Formally, this is known as the *optimal margin* criterion [308].

Experimental setup

To evaluate the prediction accuracy of the three run-time selection algorithms, we conducted the following experiment. First, we built three matrix multiply implementations using PHiPAC: (a) one with only register-level tiling, (b) one with register + L1 tiling, and (c) one with register, L1, and L2 tiling. We considered the performance of these implementations within a 2-D cross-section of the full 3-D input space in which $M = N$ and $1 \leq M, K, N \leq 800$. We selected disjoint subsets of points in this space, where each subset contained 1936 points chosen at random.[5] Then we further divided each subset into 500 testing points and 1436 training points. We trained and tested the three statistical models (details below), measuring the prediction accuracy on each test set.

In Figure 9.8, we show an example of a 500-point testing set from this space where each point is color-coded by the implementation which ran fastest. The implementation which was fastest on the majority of inputs is the default implementation generated by PHiPAC containing full filing optimizations, and is shown by a blue "`x`". Thus, a useful reference is a *baseline predictor* which always chooses this implementation: the misclassification rate of this predictor was 24%. The implementation using only register-tiling makes up the central "banana-shaped" region in the center of Figure 9.8, shown by a red "`o`". The register and L1 tiled implementation, shown by a green asterisk (`*`), was fastest on a minority of points in the lower left-hand corner of the space. Observe that the space has complicated boundaries, and is not strictly cleanly separable.

The three statistical models were implemented as follows.

- We implemented the linear least squares regression method as described in Section 9.3.2, Equation (9.6). Since the least squares fit is based on choosing the fit parameters to minimize the total square error between the execution time data and the model predictions, errors in the larger problem sizes will contribute more significantly to the total squared error than smaller sizes, and therefore tend to dominate the fit. This could be adjusted by using weighted least squares methods, or by normalizing execution time differently. We do not pursue these variations here.

- For the separating hyperplane method outlined in Section 9.3.3, we built a model using 6 hyperplanes in order to try to better capture the central region in which the

[5] The points were chosen from a distribution with a bias toward small sizes.

register-only implementation was fastest. Furthermore, we replaced the execution time $T(a, s)$ in Equation (9.7) by a "binary" execution time $\hat{T}(a, s)$ such that $\hat{T}(a, s) = 0$ if a was the fastest on input s, and otherwise $\hat{T}(a, s) = 1$. (We also compared this binary scheme to a variety of other notions of execution time, including normalizing each $T(a, s)$ by MKN to put all execution time data on a similar scale. However, we found the binary notion of time gave the best results in terms of the average misclassification rate on this particular data set.)

- For the support vector method of Section 9.3.4, we used Platt's *sequential minimal optimization* algorithm with a Gaussian kernel for the function $K(\cdot, \cdot)$ [251]. In Platt's algorithm, we set the tuning parameter $C = 100$ [251]. We built multiclass classifiers from ensembles of binary classifiers, as described by Vapnik [308].

Below, we report on the overall misclassification rate for each model as the average over all of the 10 test sets.

Results and discussion

Figures 9.9–9.11 show qualitative examples of the predictions made by the three models on a sample test set. The regression method captures the boundaries roughly but does not correctly model one of the implementations (upper-left of Figure 9.9). The separating hyperplane method is a poor qualitative fit to the data. The SV method appears to produce the best predictions. Quantatively, the misclassification rates, averaged over the 10 test sets, were 34% for the regression predictor, 31% for the separating hyperplanes predictor, 12% for the SV predictor. Only the SV predictor significantly outperformed the baseline predictor.

However, misclassification rate seems too strict a measure of prediction performance, since we may be willing to tolerate some penalties to obtain a fast prediction. Therefore, we also show the distribution of slow-downs due to mispredictions in Figure 9.12. Each curve depicts this distribution for one of the four predictors. The distribution shown is for one of the 10 trials which yielded the lowest misclassification rate. Slow-down appears on the x-axis, and the fraction of predictions on all 1936 points (including both testing and training points) exceeding a given slow-down is shown on the y-axis.

Consider the baseline predictor (solid blue line with '+' markers). Only 5–6% of predictions led to slow-downs of more than 5%, and that only about 0.4% of predictions led to slow-downs of more than 10%. Noting the discretization, evidently only 1 out of

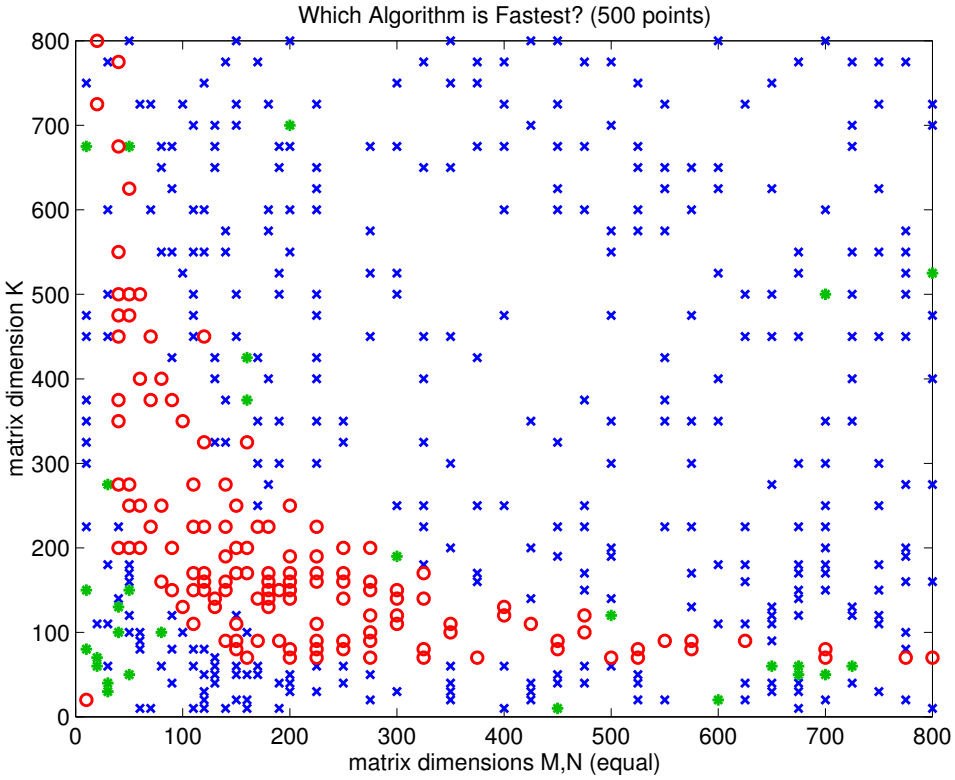

Figure 9.8: **Classification truth map: points in the input space marked by the fastest implementation**. A "truth map" showing the regions in which particular implementations are fastest. The points shown represent a 500-point sample of a 2-D slice (specifically, $M = N$) of the input space. An implementation with only register tiling is shown with a red `o`; one with L1 and register tiling is shown with a green `*`; one with register, L1, and L2 tiling is shown with a blue `x`. The baseline predictor always chooses the blue algorithm. The average misclassification rate for this baseline predictor is 24.5%.

the 1936 cases led to a slow-down of more than 47%, with no implementations being between 18–47% slower. These data indicate that the baseline predictor performs fairly well, and that furthermore the performance of the three tuned implementations is fairly similar. Therefore, we do not expect to improve upon the baseline predictor by much. This hypothesis is borne out by observing the slow-down distributions of the separating hyperplane and regression predictors (green circles and red 'x' markers, respectively), neither of which improves significantly (if at all) over the baseline.

However, we also see that for slow-downs of up to 5% (and, to a lesser extent, up to 10%), the support vector predictor (cyan '*' markers) shows a significant improvement over the baseline predictor. It is possible that this difference would be significant in some

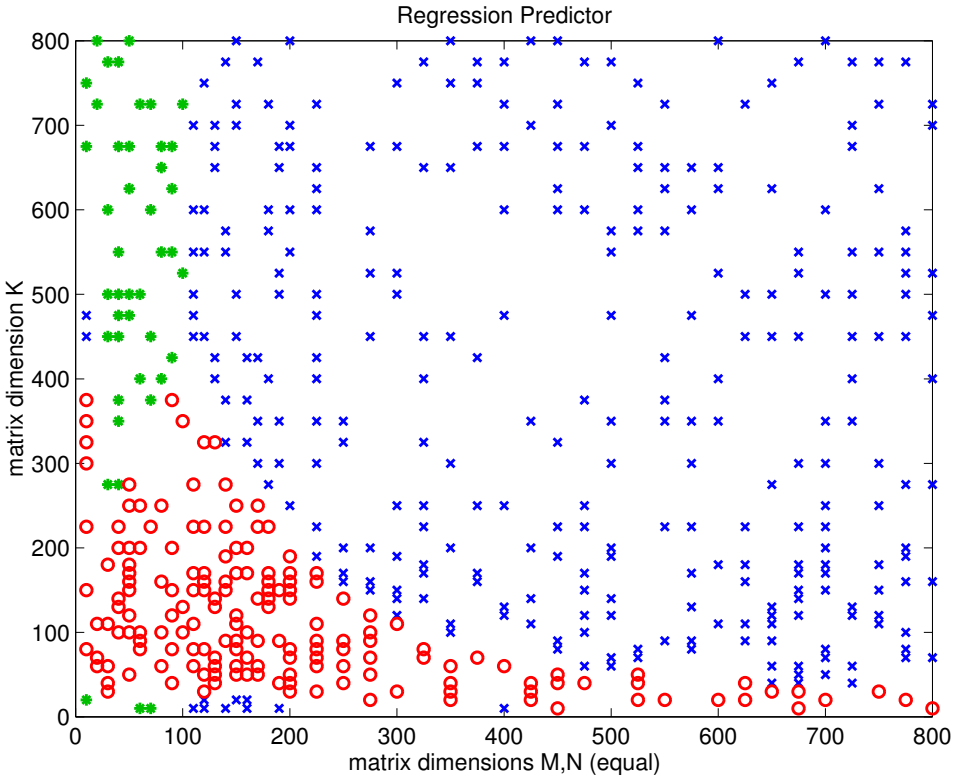

Figure 9.9: **Classification example: regression predictor**. Sample classification results for the regression predictor on the same 500-point sample shown in Figure 9.8. The average misclassification rate for this predictor was 34%.

applications with very strict performance requirements, thereby justifying the use of the more complex statistical model. Furthermore, had the differences in execution time between implementations been larger, the support vector predictor would have appeared even more attractive.

There are a number of cross-over points in Figure 9.12. For instance, comparing the regression and separating hyperplanes methods, we see that even though the overall misclassification rate for the separating hyperplanes predictor is lower than the regression predictor, the tail of the distribution for the regression predictor becomes much smaller. A similar cross-over exists between the baseline and support vector predictors. These cross-overs suggest the possibility of hybrid schemes that combine predictors or take different actions on inputs in the "tails" of these distributions, provided these inputs could somehow be identified or otherwise isolated.

In terms of prediction times (*i.e.*, the time to evaluate $f(s)$), both the regression

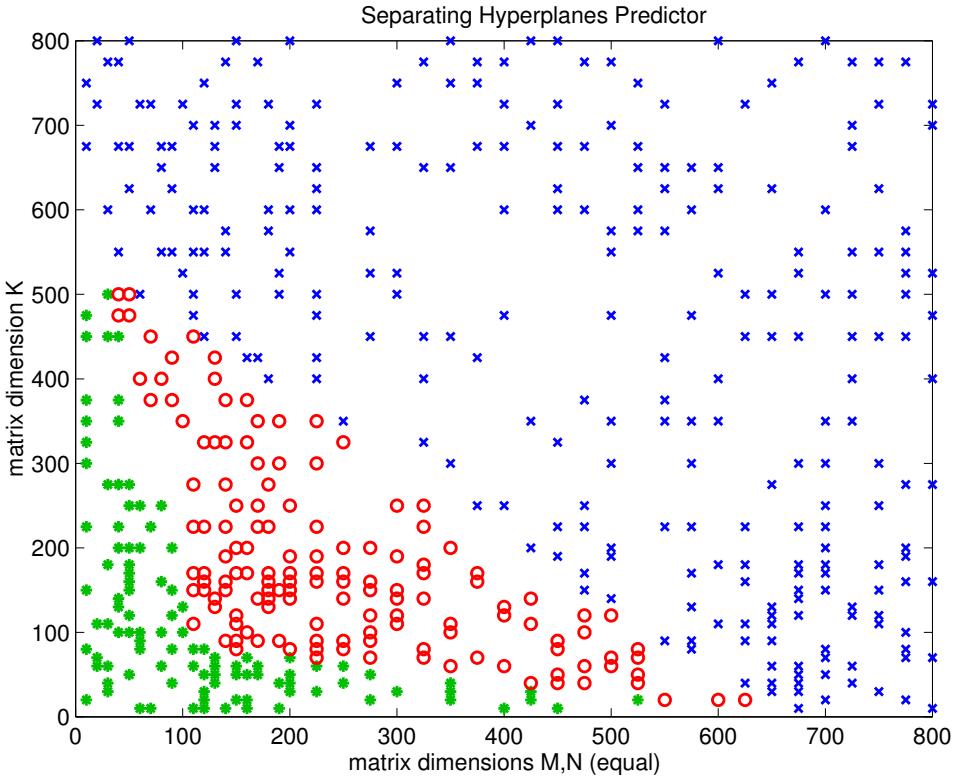

Figure 9.10: **Classification example: separating hyperplanes predictor**. Sample classification results for the separating hyperplanes predictor on the same 500-point sample shown in Figure 9.8. The average misclassification rate for this predictor was 31%.

and separating hyperplane methods lead to reasonably fast predictors. Prediction times were roughly equivalent to the execution time of a 3×3 matrix multiply. By contrast, the prediction cost of the SVM is about a 64×64 matrix multiply, which would prohibit its use when small sizes occur often. Again, it may be possible to reduce this run-time overhead by a simple conditional test of the input dimensions, or perhaps a hybrid predictor.

However, this analysis is not intended to be definitive. For instance, we cannot fairly report on specific training costs due to differences in the implementations in our experimental setting.[6] Also, matrix multiply is only one possible application, and we see that it does not stress all of the strengths and weaknesses of the three methods. Furthermore, a user or application might care about only a particular region of the full input-space which is different from the one used in our example. Instead, our primary aim is simply to

[6]In particular, the hyperplane and regression methods were written in Matlab, while the SMO support vector training code was written in C.

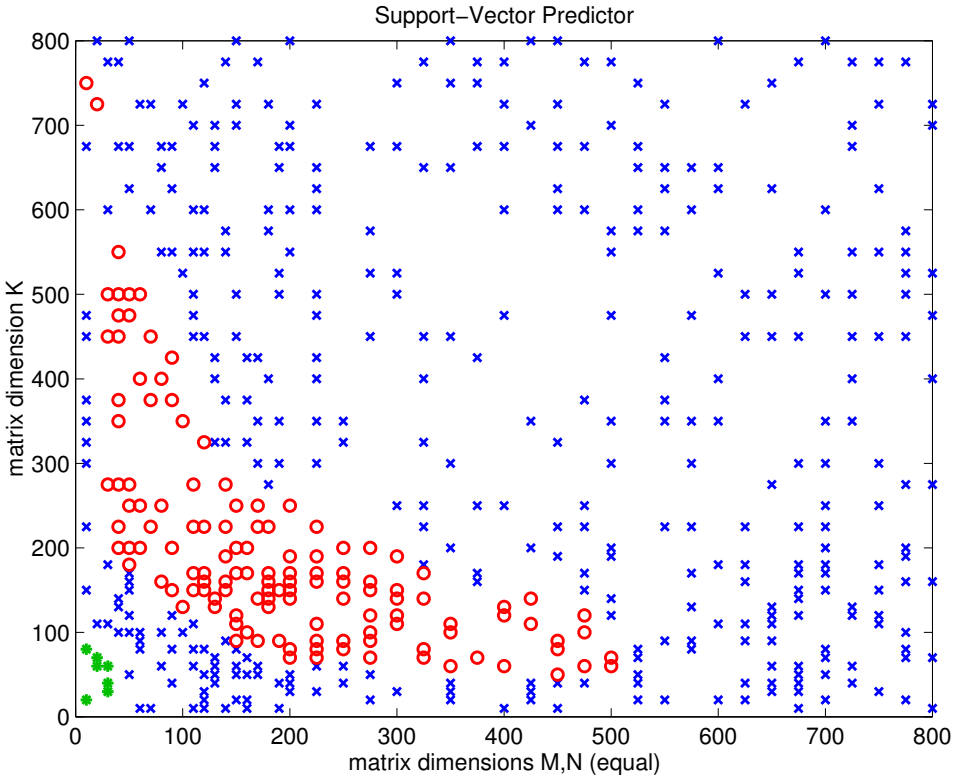

Figure 9.11: **Classification example: support vector predictor**. Sample classification results for the support vector predictor on the same 500-point sample shown in Figure 9.8. The average misclassification rate for this predictor was 12%.

present the general framework and illustrate the issues on actual data. Moreover, there are many possible models; the examples presented here offer a flavor of the role that statistical modeling of performance data can play.

9.4 A Survey of Empirical Search-Based Approaches to Code Generation

There has been a flurry of research activity in the use of empirical search-based approaches to platform-specific code generation and tuning. The primary motivation, as we demonstrate here for matrix multiply, is the difficulty of instantiating purely static models that predict performance with sufficient accuracy to decide among possible code and data structure transformations. Augmenting such models with observed performance appears to yield viable and promising ways to make these decisions.

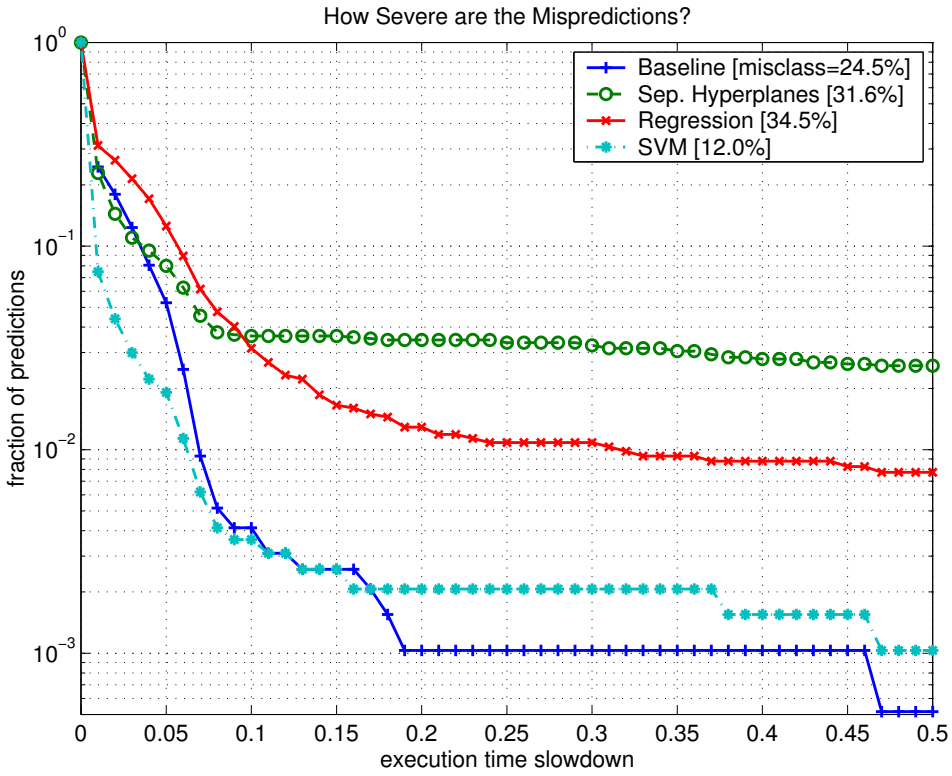

Figure 9.12: **Classification errors: distribution of slow-downs**. Each line corresponds to the distribution of slow-downs due to mispredictions on a 1936 point sample for a particular predictor. A point on a given line indicates what fraction of predictions (y-axis) resulted in more than a particular slow-down (x-axis). Note the logarithmic scale on the y-axis.

In our review of the diverse body of related work, we note how each study or project addresses the following high-level questions:

1. **What is the unit of optimization?** In a recent position paper on *feedback-directed optimization*, Smith argues that a useful way to classify dynamic optimization methods is by the size and semantics of the piece of the program being optimized [280]. Traditional static compilation applies optimizations in "units" which following programming language conventions, *e.g.*, within a basic block, within a loop nest, within a procedure, or within a module. By contrast, dynamic (run-time) techniques optimize across units relevant to run-time behavior, *e.g.*, along a sequence of consecutively executed basic blocks (a *trace* or *path*).

   Following this classification, we divide the related work on empirical search-based tun-

ing primarily into two high-level categories: *kernel-centric* tuning and *compiler-centric* tuning. This dissertation adopts the kernel-centric perspective in which the unit of optimization is the kernel itself. The code generator—and hence, the implementation space—is specific to the kernel. One would expect that a generator specialized to a particular kernel might best exploit mathematical structure or other structure in the data (possibly known only at run-time) relevant to performance. As we discuss below, this approach has been very successful in the domains of linear algebra and signal processing, where understanding problem-specific structure leads to new, tunable algorithms and data structures.

In the compiler-centric view, the implementation space is defined by the space of possible compiler transformations that can be applied to any program expressed in a general-purpose programming language. In fact, the usual suite of optimizations for matrix multiply can all be expressed as compiler transformations on the standard 3-nested loop implementation, and thus it is possible in principle for a compiler to generate the same high-performance implementation that can be generated by hand. However, what makes a specialized generator useful in this instance is that the expert who writes the generator identifies the precise transformations which are hypothesized to be most relevant to improving performance. Moreover, we could not reasonably expect a general purpose compiler to know about all of the possible mathematical transformations or alternative algorithms and data structures for a given kernel—it is precisely these kinds of transformations that have yielded the highest performance for other important computational kernels like the discrete Fourier transform (DFT) or operations on sparse matrices.

We view these approaches as complementary, since hybrid approaches are also possible. For instance, here we consider the use of a matrix multiply-specific generator that ouputs C or Fortran code, thus leaving aspects of the code generation task (namely, scheduling) to the compiler. What these approaches share is that their respective implementation spaces can be very large and difficult to model. It is the challenge of *choosing* an implementation that motivates empirical search-based tuning.

2. **How should the implementation space be searched?** Empirical search-based approaches typically choose implementations by some combination of *modeling* and *experimentation* (*i.e.*, actually running the code) to predict performance and thereby

choose implementations. Section 9.1 argues that performance can be a complex function of algorithmic parameters, and therefore may be difficult to model accurately using only static models in practice. This chapter explores the use of statistical models, constructed from empirical data, to model performance within the space of implementations. The related work demonstrates that a variety of additional kinds of models are possible. For instance, one idea that has been explored in several projects is the use of evolutionary (genetic) algorithms to model and search the space of implementations.

3. **When to search?** The process of searching an implementation space could happen at any time, whether it be strictly off-line (*e.g.*, once per architecture or once per application), strictly at run-time, or in some combination. The cost of an off-line search can presumably be amortized over many uses, while a run-time search can maximally use the information only available at run-time. Again, hybrid approaches are common in practice.

   The question of when to search has implications for software system support. For instance, a strictly off-line approach requires only that a user make calls to a special library or a special search-based compiler. Searching at run-time could also be hidden in a library call, but might also require changes to the run-time system to support dynamic code generation or dynamic instrumentation or trap-handling to support certain types of profiling. This survey mentions a number of examples.

Our survey summarizes how various projects and studies have approached these questions, with a primary emphasis on the kernel-centric vs. compiler-centric approaches, though again these we see these two viewpoints as complementary.[7] Collectively, these questions imply a variety of possible software architectures for generating code adapted to a particular hardware platform and run-time workload.

9.4.1 Kernel-centric empirical search-based tuning

Typical kernel-centric tuning systems contain specialized code generators that exploit specific mathematical properties of the kernel or properties of the data. The target performance goal of these systems is to achieve the performance of hand-tuned code. Most research has

[7] The focus is on software, though the idea of search has been applied to hardware design space exploration for field programmable gate arrays (FPGAs) [283].

focused on tuning in the domains of dense and sparse linear algebra, and signal processing. In these areas, there is a rich mathematical structure relevant to performance to exploit. We review recent developments in these and other areas below. (For alternative views of some of this work, we refer the reader to recent position papers on the notion of *active libraries* [310] and self-adapting numerical software [103].)

Dense and sparse linear algebra

Dense matrix multiply is among the most important of the computational kernels in dense linear algebra both because a large fraction (say, 75% or more) of peak speed can be achieved on most machines with proper tuning, and also because many other dense kernels can be expressed as calls to matrix multiply [181]. The prototype PHiPAC system was an early system for generating automatically tuned implementations of this kernel with cache tiling, register tiling, and a variety of unrolling and software pipelining options [46]. The notion of automatically generating tiled matrix multiply implementations from a concise specification with the possibility of searching the space of tile sizes for matrix multiply also appeared in early work by McCalpin and Smotherman [218]. The ATLAS project has since extended the applicability of the PHiPAC prototype to all of the other dense matrix kernels that constitute the BLAS [325]. These systems contain specalized, kernel-specific code generators, as discussed in Section 9.1. Furthermore, most of the search process can be performed completely off-line, once per machine architecture. The final output of these systems is a library implementing the BLAS against which a user can link her application.

One promising avenue of research relates to the construction of sophisticated new generators. Veldhuizen [309], Siek and Lumsdaine [278], and Renard and Pommier [259] have developed C++ language-based techniques for cleanly expressing dense linear algebra kernels. More recent work by Gunnels, *et al.*, in the FLAME project demonstrates the feasibility of systematic derivation of algorithmic variations for a variety of dense matrix kernels [142]. These variants would be suitable implementations a_i in our run-time selection framework (Section 9.3). In addition, FLAME provides a new methodology by which one can cleanly generate implementations of kernels that exploit caches. However, these implementations still rely on highly-tuned "inner" matrix multiply code, which in turn requires register- and instruction-level tuning. Therefore, we view all of these approaches to code generation as complementing empirical search-based register- and instruction-level tuning.

Another complementary research area is the study of so-called *cache-oblivious* algorithms, which to claim eliminate the need for cache-level tuning to some extent for a number of computational kernels. Like traditional tiling techniques [159, 275], cache-oblivious algorithms for matrix multiply, LU factorization, and QR factorization have been shown to asymptotically minimize data movement among various levels of the memory hierarchy, under certain cache modeling assumptions [302, 124, 12, 119, 120, 147].[8] Unlike tiling, cache-oblivious algorithms do not make explicit reference to a "tile size" tuning parameter, and thus appear to eliminate the need to search for optimal cache tile sizes either by modeling or by empirical search. Furthermore, language-level support now exists both to convert loop-nests to recursion automatically [334] and also to convert linear array data structures and indexing to recursive formats [329]. However, we note in Section 9.1 that at least for matrix multiply, cache-level optimizations account for only a part (perhaps 12–60%, depending on the platform) of the total performance improvement possible, and therefore complements additional register- and instruction-level tuning. The nature of performance in these spaces, as shown in Figure 9.2, together with recent results showing that even carefully constructed models of the register- and instruction-level implementation space can mispredict [335], imply that empirical search is still necessary for tuning.[9]

For matrix multiply, algorithmic variations that require fewer than $O(n^3)$ flops for $n \times n$ matrices, such as Strassen's algorithm, are certainly beyond the kind of transformations we expect general purpose compilers to be able to derive. Furthermore, like cache-oblivious algorithms, practical and highly efficient implementations of Strassen's algorithm still depend on highly-tuned base-case implementations in which register- and instruction-level tuning is critical [162, 299].

The BLAS-tuning ideas have been applied to higher-level, parallel dense linear algebra libraries. In the context of cluster computing in the Grid, Chen, *et al.*, have designed a self-tuning version of the LAPACK library for Clusters (LFC) [73, 72]. LFC preserves LAPACK's serial library interface, and decides at run-time whether and how to parallelize a call to a dense linear solve routine, based on the current cluster load. In a similar spirit, Liniker, *et al.*, have applied the idea of run-time selection to the selection of *data layout* in their distributed parallel version of the BLAS library [210, 33]. Their library,

[8] Cache-oblivious algorithms have been developed for a variety of other contexts as well, such as fast tree, priority queue, and graph algorithms [17, 35, 57].

[9] Indeed, recent work has qualitatively confirmed the need and importance of fast "base case" implementations in recursive implementations [123, 125, 240].

called DESOBLAS, is based on the idea of delayed evaluation: all calls to DESOBLAS library routines return immediately, and are not executed until either a result is explicitly accessed by the user or the user forces an evaluation of all unexecuted calls. At evaluation time, DESOBLAS uses information about the entire sequence of operations that need to be performed to make decisions about how to distribute data. Both LFC and DESOBLAS adopt the library interface approach, but defer optimization until run-time.

Kernels arising in sparse linear algebra, such as sparse matrix-vector multiply, complicate tuning compared to their dense counterparts because performance depends on the non-zero structure of the sparse matrix. For sparse kernels, the user must choose a data structure that minimizes storage of the matrix while still allowing efficient mapping of the kernel to the target architecture. Worse still, the matrix structure may not be known until run-time. Prototype systems exist which allow a user to specify separately both the kernel and the data structure, while a specialized generator (or restructuring compiler) combines the two specifications to generate an actual implementation [43, 254, 287]. At present, such systems do not explicitly address the register- and instruction-level tuning issues, nor do they adequately address the run-time problem of choosing a data structure given a sparse matrix. Automatic tuning with respect to these low-level tuning and data structure selection issues have been taken up by recent work on the SPARSITY system [167, 316].

Digital signal processing

Recent interest in automatic tuning of digital signal processing (DSP) applications is driven both by the rich mathematical structure of DSP kernels and by the variety of target hardware platforms. One of the best-studied kernels, the discrete Fourier transform (DFT), admits derivation of many fast Fourier transform (FFT) algorithms. The fast algorithms require significantly fewer flops than a naïve DFT implementation, but since different algorithms have different memory access patterns, strictly minimizing flops does not necessarily minimize execution time. The problem of tuning is further complicated by the fact that the target architectures for DSP kernels range widely from general purpose microprocessors and vector architectures to special-purpose DSP chips.

FFTW was the first tuning system for various flavors of the discrete Fourier transform (DFT) [123]. FFTW is notable for its use of a high-level, symbolic representation of the FFT algorithm, as well as its run-time search which saves and uses performance history

information. Search boils down to selecting the best fully-unrolled base case implementations, or equivalently, the base cases with the best instruction scheduling. The search process occurs only at run-time because that is when the problem size is assumed to be known. There have since been additional efforts in signal processing which build on the FFTW ideas. The SPIRAL system is built on top of a symbolic algebra system, allows users to enter customized transforms in an interpreted environment using a high-level tensor notation, and uses a novel search method based on genetic algorithms [255, 279]. The performance of the implementations generated by these systems is largely comparable both to one another and to vendor-supplied routines. One distinction between the two systems is that SPIRAL's search is off-line, and carried out for a specific kernel of a given size, whereas FFTW chooses the algorithm at run-time. The most recent FFT tuning system has been the UHFFT system, which is essentially an alternative implementation of FFTW that includes a different implementation of the code generator [225]. In all three systems, the output of the code generator is either C or Fortran code, and the user interface to a tuned routine is via a library or subroutine call.

Other kernel domains

One immediate extension of the work in dense linear algebra is to extend tuning ideas to calculations in finite fields. Dumas, *et al.*, are investigating the use of ATLAS and ATLAS-like techniques to tune their finite field linear algebra subroutine (FFLAS) library [110].

In the area of parallel distributed communications, Vadhiyar, *et al.*, propose techniques to tune automatically the Message Passing Interface (MPI) collective operations [306]. The most efficient implementations of these kernels, which include "broadcast," "scatter/gather," and "reduce," depend on characteristics of the network hardware. Like its tuning system predecessors in dense linear algebra, this prototype for MPI kernels targets the implementation of a standard library interface. Achieved performance meets or exceeds that of vendor-supplied implementations on several platforms. The search for an optimal implementation is conducted entirely off-line, using heuristics to prune the space and a benchmarking workload that stresses message size and number of participating processors, among other features.

Empirical search-based tuning systems for sorting have shown some promise. Recent work by Arge, *et al.*, demonstrate that algorithms which minimize cache misses under

simple but reasonable cache models lead to sorting implementations which are suboptimal in practice [18]. They furthermore stress the importance of register- and instruction-level tuning, and use all of these ideas to propose a new sorting algorithm space with machine-dependent tuning parameters. A preliminary study by Darcy shows that even for the well-studied quicksort algorithm, an extensive implementation space exists and exhibits distributions of performance like those shown in Figure 9.2 (*top*) [89]. Lagoudakis and Littman have shown how the selection problem for sorting can be tackled using statistical methods not considered here, namely, by reinforcement learning techniques [200]. Most recently, Li, *et al.*, have synthesized similar ideas and produced an self-tunable sorting library [209]. Together, these studies suggest the applicability of search-based methods to non-numerical computational kernels.

Recently, Baumgartner, *et al.*, have proposed a system to generate entire parallel applications for a class of quantum chemistry computations [32, 78]. Like SPIRAL, this system provides a way for chemists to specify their computation in a high-level notation, and carries out a symbolic search to determine a memory and flop efficient implementation. The authors note that the best implementation depends ultimately on machine-specific parameters. Some heuristics tied to machine parameters (*e.g.*, available memory) guide search.

Dolan and Moré have identified empirical distributions of performance as a mechanism for comparing various mathematical optimization solvers [99]. Specifically, the distributions estimate the probability that the performance of a given solver will be within a given factor of the best performance of all solvers considered. Their data was collected using the online optimization server, NEOS, in which users submit optimization jobs to be executed on NEOS-hosted computational servers. The primary aim of their study was to propose a new "metric" (namely, the distributions themselves) as a way of comparing different optimization solvers. However, what these distributions also show is that Grid-like computing environments can be used to generate a considerable amount of performance data, possibly to be exploited in run-time selection contexts as described in Section 9.3.

A key problem in the run-time selection framework we present in Section 9.3 is the classical statistical learning problem of *feature selection*. In our case, features are the attributes that define the input space. The matrix multiply example assumes the input matrix dimensions constitute the best features. Can features be identified automatically in a general setting? A number of recent projects have proposed methods, in the context of

performance analysis and algorithm selection, which we view as possible solutions. Santiago, *et al.*, apply the statistical experimental design methods to program tuning [272]. These methods essentially provide a systematic way to analyze how much hypothesized factors contribute to performance. The most significant contributors identified could constitute suitable features for classification. A different approach has been to codify expert knowledge in the form of a database, recommender, or expert system in particular domains, such as a partial differential equation (PDE) solver [211, 257, 160], or a molecular dynamics simulation [194]. In both cases, each algorithmic variation is categorized by manually identified features which would be suitable for statistical modeling.

Note that what is common to most of the preceeding projects is a library-based approach, whether tuning occurs off-line or at run-time. The Active Harmony project seeks to provide a general API and run-time system that supports run-time selection and run-time parameter tuning in the setting of the Grid [305]. This work, though in its early stages, highlights the need for search in new computational environments.

9.4.2 Compiler-centric empirical search-based tuning

The idea of using data gathered during program execution to aid compilation has previously appeared in the compiler literature under the broad term *feedback-directed optimization* (FDO). A recent survey and position paper by Smith reviewed developments in subareas of FDO including profile-guided compilation (Section 40) and dynamic optimization (Section 42) [280]. FDO methods are applied to a variety of program representations: source code in a general-purpose high-level language (*e.g.*, C or Java), compiler intermediate form, or even a binary executable. These representations enable transformations to improve performance on general applications, either off-line or at run-time. Binary representations enable optimizations on applications that have shipped or on applications that are delivered as mobile code. The underlying philosophy of FDO is the notion that optimization without reference to actual program behavior is insufficient to generate optimal or near-optimal code.

In our view, the developments in FDO join renewed efforts in superoptimizers (Section 9.4.2) and the new notion of self-tuning compilers (Section 40) in an important trend in compilation systems toward the use of empirically-derived models of the underlying machines and programs.

Superoptimizers

Massalin coined the term *superoptimizer* for his exhaustive search-based instruction generator [213]. Given a short program, represented as a sequence of (six or so) machine language instructions, the superoptimizer exhaustively searched all possible equivalent instruction sequences for a shorter (and equivalently at the time, faster) program. Though extremely expensive compared to the usual cost of compilation, the intent of the system was to "superoptimize" particular bottlenecks off-line. The overall approach represents a noble effort to generate truly "optimal" code.[10]

Joshi, *et al.*, substitute exhaustive search in Massalin's superoptimizer with an automated theorem prover in their Denali superoptimizer [180]. One can think of the prover as acting as a modeler of program performance. Given a sequence of expressions in a C-like notation, Denali uses the automated prover to generate a machine instruction sequence that is provably the fastest implementation possible. However, to make such a proof-based code generation system practical, Denali's authors necessarily had to assume (a) a certain model of the machine (*e.g.*, multiple issue with pipeline dependencies specified but fixed instruction latencies), and (b) a particular class of acceptable constructive proofs (*i.e.*, matching proofs). Nevertheless, Denali is able to generate extremely good code for short instruction sequences (roughly 16 instructions in a day's worth of time) representing ALU-bound operations on the Alpha EV6. As the Denali authors note, it might be possible to apply their approach more broadly by refining the instruction latency estimates, particularly for memory operations, with measured data from actual runs—again suggesting a combined modeling and empirical search approach.

Profile-guided compilation and iterative compilation

The idea behind *profile-guided compilation* (PGC) is to carry out compiler transformations using information gathered during actual execution runs [193, 135]. Compilers can instrument code to gather execution frequency statistics at the level of subroutines, basic blocks, or paths. On subsequent compiles, these statistics can be used to enable more aggressive use of "classical" compiler optimizations (*e.g.*, constant propagation, copy propagation, common subexpression elimination, dead code removal, loop invariant code removal, loop induction variable elimination, global variable migration) along frequent execution paths

[10] A refinement of the original superoptimizer, based on `gcc`, is also available [136].

[69, 28]. The PGC approach has been extended to help guide prefetch instruction placement on x86 architectures [29]. PGC can be viewed as a form of empirical search in which the implementation space is implicitly defined to be the space of all possible compiler transformations over all inputs, and the user (programmer) directs the search by repeatedly compiling and executing the program.

The search process of PGC can be automated by replacing the user-driven compile/execute sequence with a compiler-driven one. The term *iterative compilation* has been coined to refer to such a compiler process [190, 307]. Users annotate their program source with a list of which transformations—*e.g.*, loop unrolling, tiling, software pipelining—should be tried on a particular segment of code, along with any relevant parametric ranges (*e.g.*, a range of loop unrolling depths). The compiler then benchmarks the code fragment under the specified transformations. In a similar vein, Pike and Hilfinger built tile-size search using simulated annealing into the Titanium compiler, with application to a multigrid solver [249]. The Genetic Algorithm Parallelisation System (GAPS) by Nisbet addressed the problem of compile-time selection of an optimal sequence of serial and parallel loop transformations for scientific applications [233]. GAPS uses a genetic algorithms approach to direct search over the space of possible transformations, with the initial population seeded by a transformation chosen by "conventional" compiler techniques. The costs in all of these examples are significantly longer compile cycles (*i.e.*, including the costs of running the executable and re-optimizing), but the approach is "off-line" since the costs are incurred before the application ships. Furthermore, the compile-time costs can be reduced by restricting the iterative compilation process to only known application bottlenecks. In short, what all of these iterative compilation examples demonstrate is the utility of a search-based approach for tuning general codes that requires minimal user intervention.

Self-tuning compilers

We use the term *self-tuning compiler* to refer to recent work in which the compiler itself—*e.g.*, the compiler's internal models for selecting transformations, or the optimization phase ordering—is adapted to the machine architecture. The goal of this class of methods is to avoid significantly increasing compile-times (as occurs in iterative compilation) while still adapting the generated code to the underlying architecture.

Mitchell, *et al.*, proposed a scheme in which models of various types of memory

access patterns are measured for a given machine when the compiler is installed [226]. At analysis time, memory references within loop nests are decomposed and modeled by functions of these canonical patterns. An execution time model is then automatically derived. Instantiating and comparing these models allows the compiler to compare different transformations of the loop nest. Though the predicted execution times are not always accurate in an absolute sense, the early experimental evidence suggests that they may be sufficiently accurate to predict the relative ranking of candidate loop transformations.

The Meta Optimization project proposes automatic tuning of the compiler's internal *priority* (or *cost*) functions [285]. The compiler uses these functions to choose a code generation action based on known characteristics of the program. For example, in deciding whether or not to prefetch a particular memory reference within a loop, the compiler evaluates a binary priority function that considers the current loop trip count estimates, cache parameters, and estimated prefetch latency,[11] among other factors. The precise function is usually tuned by the compiler writer. In the Meta Optimization scheme, the compiler implementer specifies these factors, their ranges, and a hypothesized form of the function, and Meta Optimization uses a genetic programming approach to determine (*i.e.*, to evolve) a better form for the function. The candidate functions are evaluated on a benchmark or suite of benchmark programs to choose one. Thus, priority functions can be tuned once for all applications, or for a particular application or class of applications.

In addition to internal models, another aspect of the compiler subject to heuristics and tuning is the optimization phase ordering, *i.e.*, the order in which optimizations are applied. Although this ordering is usually fixed through experimentation by a compiler writer, Cooper, *et al.*, have proposed the notion of an adaptive compiler which experimentally determines the ordering for a given machine [85, 84]. Their compiler uses genetic algorithms to search the space of possible transformation orders. Each transformation order is evaluated against some metric (*e.g.*, execution time or code size) on a pre-defined set of benchmark programs.

Nisbet has taken a similar genetic programming approach to construction of a self-tuning compiler for parallel applications [234].

The Liberty compiler research group has proposed an automated scheme to organize the space of optimization configurations into a small decision tree that can be quickly

[11] The minimum time between the prefetch and its corresponding load.

traversed at compile-time [304]. Roughly speaking, their study starts with the Intel IA-64 compiler and identifies the equivalent of k internal binary flags that control optimization. This defines a space of possible configurations of size 2^k. This space is systematically pruned, and a final, significantly smaller set of configurations are selected.[12] (In a traditional compiler implementation, a compiler writer would manually choose just 1 such configuration based on intuition and experimentation.) The final configurations are organized into a decision tree. At compile-time, this tree is traversed and each configuration visited is applied to the code. The effect of the configuration is predicted by a static model, and used to decide which paths to traverse and what final configuration to select. This work combines the model-tuning of the other self-tuning compiler projects and the idea of iterative compilation (except that in this instance, performance is predicted by a static model instead of by running the code.)

Dynamic (run-time) optimization

Dynamic optimization refers to the idea of applying compiler optimizations and code generation at run-time. Just-in-time (JIT) compilation, particularly for Java-based programs, is one well-known example. Among the central problems in dynamic optimization are automatically deciding what part of an application to optimize, and how to reduce the run-time cost of optimization. Here, we concentrate on summarizing the work in which empirical search-based modeling is particularly relevant. We refer the reader to Smith's survey [280] and related work on dynamic compilation software architectures [206, 61] for additional references on specific run-time code generation techniques [82, 252, 137].

Given a target fragment of code at run-time, the Jalapeño JIT compiler for Java decides what level of optimization to apply based on an empirically derived cost-benefit model [19, 63]. This model weighs the expected pay-off from a given optimization level, given an estimate of the frequency of future execution, against the expected cost of optimizing. Profiling helps to identify the program hotspots and cost estimates, and evaluation of the cost-benefit model is a form of empirical-model based search.

Two recent projects have proposed allowing the compiler to generate multiple ver-

[12] In the original work's experiment, not all flags considered are binary. Nevertheless, the size of the original space is equivalent to the case when $k = 19$. The final number of configurations selected is 12. Also note that the paper proposes a technique for pruning the space which may be a variant of a common statistical method known as fractional factorial design (FFD) [332]. FFD has been applied to the automatic selection of compiler flags [75].

sions of a code fragment (*e.g.*, loop body, procedure), enabling run-time search and selection for general programs [97, 312]. Diniz and Rinard coined the term *dynamic feedback* for the technique used in their parallelizing compiler for C++ [97]. For a particular synchronization optimization, they generate multiple versions of the relevant portion of code, each of which has been optimized with a different level of aggressiveness. The generated program alternates between sampling and production phases. During sampling, the program executes and times each of the versions. Thus, the sampling phase is essentially an instance of empirical search. During the (typically much longer) production phase, the best version detected during sampling executes. The length of each phase must be carefully selected to minimize the overall overhead of the approach. The program continues the sampling and production cycle, thus dynamically adjusting the optimization policies to suit the current application context. The dynamic feedback approach has been revisited and generalized in the ADAPT project, an extension of the Polaris parallelizing compiler [312]. The ADAPT framework provides more generalized mechanisms for "optimization writers" to specify how variants are generated, and how they may be heuristically pruned at run-time. In contast to the assumed model of run-time selection in Section 9.3, where the statistical models are generated off-line, in this dynamic feedback approach the models themselves must be generated at run-time during the sampling phase.

Kistler and Franz propose a sophisticated system architecture, built on top of the Oberon System 3 environment, for performing *continuous program optimization* [189]. They take a "whole systems" view in which the compiler, the dynamic loader, and the operating system all participate in the code generation process. The compiler generates an executable in an intermediate binary representation. When the application is launched, this binary is translated into machine language, with minimal or no optimizations. The program is periodically sampled to collect profile data (such as frequency, time, or hardware counter statistics). A separate thread periodically examines the profile data to identify either bottlenecks or changes in application behavior that might warrant re-optimization, and generates a list of candidate procedures to optimize. An empirical cost-benefit analysis is used to decide which, if any, of these candidates should be re-optimized. The code image for re-optimized procedures is replaced on the fly with the new image, provided it is not currently executing. For the particular dynamic optimizations they consider in their prototype—trace-based instruction rescheduling and data reorganization—off-line search-based optimization still outperforms continuous re-optimization for BLAS routines. Nevertheless,

their idea applies more generally and with some success on other irregular, non-numerical routines with dynamic (linked) data structures. However, the cost of continuous profiling and re-optimization are such that much of the benefit can be realized only for very long running programs, if at all.

9.5 Summary

For existing automatic tuning systems which follow the two-step "generate-and-search" methodology, the results of this chapter draw attention to the process of searching itself as an interesting and challenging area for research. We advocate statistical methods to address some of the challenges which arise. Our survey of related work (Section 9.4) indicates that the use of empirical search-based tuning is widespread, and furthermore suggests that the methods proposed herein will be relevant in a number of contexts besides kernel-centric tuning systems.

Among the current automatic tuning challenges is pruning the enormous implementation spaces. Existing tuning systems use problem-specific heuristics and performance models; our statistical model for stopping a search early is a complementary technique. It has the nice properties of (1) making very few assumptions about the performance of the implementations, (2) incorporating performance feedback data, and (3) providing users with a meaningful way to control the search procedure (namely, via probabilistic thresholds).

Another challenge is finding efficient ways to select implementations at run-time when several known implementations are available. Our aim has been to discuss a possible framework, using sampling and statistical classification, for attacking this problem in the context of automatic tuning systems. Other approaches are being explored for implementing "poly-algorithms" for a variety of domains [194, 39, 145].

Many other modeling techniques remain to be explored. For instance, the early stopping problem can be posed as a similar problem which has been treated extensively in the statistical literature under the theory of optimal stopping [76, 114, 115]. Problems treated in this theory can incorporate the cost of the search itself [45]. Such cost-incorporating techniques would be especially useful if we wished to perform searches not just at build-time, as we consider here, but at run-time—for instance, in the case of a just-in-time or other dynamic compilation system.

In the case of run-time selection, we make implicit geometric assumptions about

inputs to the kernels being points in some continuous space. However, inputs could also be binary flags or other arbitrary discrete labels. This can be handled in the same way as in the traditional classification settings, namely, either by finding mappings from the discrete spaces into continuous (feature) spaces, or by using statistical models with discrete probability distributions (*e.g.*, using graphical models [121]).

Although matrix multiply represents only one in many possible families of applications, our survey reveals that search-based methods have demonstrated their utility for other kernels in scientific application domains like the discrete Fourier transform (DFT) and sparse matrix-vector multiply (SpMV). These other computational kernels differ from matrix multiply in that they have less computation per datum ($O(logn)$ flops per signal element in the case of the DFT, and 2 flops per matrix element in the case of SpMV), as well as additional memory indirection (in the case of SpMV). Moreover, search-based tuning has shown promise for non-numerical kernels such as sorting or parallel distributed collective communications (Section 9.4). The effectiveness of search in all of these examples suggests that a search-based methodology applies more generally.

In short, this work connects high performance software engineering with statistical modeling ideas. The idea of searching is being incorporated into a variety of software systems at the level of applications, compilers, and run-time systems, as our survey in Section 9.4 shows. This further emphasizes the relevance of search beyond specialized tuning systems.

Chapter 10

Conclusions and Future Directions

Contents

This dissertation advocates an empirical search-based approach for automatic tuning of sparse matrix kernels. This work is driven by historical trends suggesting that untuned sparse kernel performance is worsening over time and that tuning is important to maintaining Moore's law-like scaling, coupled with advances in the development of automatic tuning systems for related domains (*e.g.*, linear algebra and signal processing). The strength of the search-based approach is strongly supported by both our results and the considerable recent interest in applying empirical search to performance optimization, as reviewed in our extensive survey (Section 9.4).

Below, we summarize our main results for sparse kernels (Section 10.1), review the high-level themes and ideas (Section 10.2), and sketch future research areas that illustrate the breadth and depth of challenges in automatic performance tuning (Section 10.3).

10.1 Main Results for Sparse Kernels

We summarize the kinds of maximum performance improvements that can be expected by the methods proposed both in this dissertation and in the larger BeBOP research project [3] of which this dissertation is a part. Speedups are listed relative to compressed sparse row (CSR) format implementations, except where noted. (For more detailed summaries, see Section 5.3, Section 6.5, Section 7.5.)

- **sparse matrix-vector multiply (SpMV)** :
  - *Register blocking*: up to 4× speedups and 75% or more of performance upper bounds (Chapters 3–4) [165, 316, 164].
  - *Multiplication by multiple vectors*: 7× speedups, ignoring symmetry [165, 164].
  - *Cache blocking*: 2.2× [165, 164].
  - *Symmetry*: 2.8× for SpMV and 7.3× for sparse matrix-multiple vector multiply (SpMM), or 2.6× relative to non-symmetric register blocking with multiple vectors [204].
  - *Variable blocking and splitting, based on variable block row (VBR) format and unaligned block compressed sparse row (UBCSR) format* : 2.1× over CSR, or 1.8× over register blocking (Section 5.1).
  - *Diagonals using row segmented diagonal (RSDIAG) format* : 2× (Section 5.2).
  - *TSP-based reordering to create dense blocks*: 1.5× [228].
- **sparse triangular solve (SpTS)** , with *register blocking and the switch-to-dense optimizations*: up to 1.8× speedups, and 75% or more of performance upper bounds (Chapter 6) [319].
- **sparse $A^TA\cdot x$ (SpA^TA)** , with *register blocking and cache interleaving*: up to 4.2× over CSR, 1.8× over register blocking only, and 50–80% of the performance upper bound (Chapter 7) [317, 318].
- **sparse** $A^\rho\cdot x$, with *serial sparse tiling*: up to 2× over to CSR or 1.5× over a register blocked implementations without tiling (Chapter 7).

10.2 Summary of High-Level Themes

At a very high-level, the underlying themes and philosophy of this dissertation can be summarized as follows.

- **"Kernel-centric" optimization**: As discussed in Section 9.4, we focus on optimization at the unit of a kernel, which we treat as a black box and for which we apply as many application or domain-specific concepts (such as the pattern of a sparse matrix) as possible to improve performance. In contrast, traditional static and dynamic compiler approaches optimize at the level of basic blocks, loop nests, procedures, modules, and traces or paths (sequences of basic blocks executed at run-time). Aggressive use of knowledge about matrix non-zero patterns leads to a variety of considerable pay-offs for SpMV, SpTS, SpA^TA, and sparse $A^{\rho} \cdot x$, as summarized in Section 5.3, Section 6.5, and Section 7.5.

  This dissertation focuses purely on non-zero patterns, ignoring the non-zero values (except in the case of symmetry). For a given sparse matrix—or more generally, for a given application or problem—there is a potentially much deeper mathematical structure that can be exploited for performance.

  Whether and how to extend the optimization techniques to more general settings would appear to be a drawback of the kernel-level optimization approach.

- **Performance bounds modeling**: The goals of performance bounds modeling are (1) to evaluate the quality of the generated code, identifying when more aggressive low-level is likely to pay-off, and (2) to gain insights into how kernel performance interacts with architectural parameters. As an example of meeting goal (1), bounds lead to the conclusion that in the case of SpA^TA, additional low-level tuning is likely to pay-off. As an example of addressing goal (2), we suggest the use of strictly increasing cache line sizes in multi-level memory hierarchies for streaming applications.

- **Empirical search-based optimization**: We adopt and improve upon the specific approach advocated by SPARSITY [164] in which search is conducted in two phases. The first phase is an off-line benchmarking phase that characterizes the performance of possible implementations on the given machine in a manner independent of the user's specific problem. The second is a run-time "search" consisting of (a) estimating the relevant matrix structural properties, followed by (b) evaluating a heuristic

model that combines the estimated properties and benchmarking data to select an implementation. This approach works well for choosing tuning parameters for SpMV (Chapter 3), SpTS (Chapter 6), and SpA^TA (Chapter 7).

- **Statistical performance models**: Simply put, the process of search generates data on which we can base and build a model. Such models characterize performance in some way, and we can imagine making optimization decisions based on evaluating these models, as we demonstrate in Chapter 9.

10.3 Future Directions

We envision a variety of ways in which to build on the work and ideas in this dissertation. The summaries at the end of individual chapters discuss additional specific technical opportunities.

10.3.1 Composing code generators and search spaces

Our basic model of a code generator for kernels is that we call the generator specifying values for tuning parameters, and the output is an implementation at those tuning parameter values. To extend a tuning system to generate new kernels beyond an existing set of predefined kernels, a desirable property of the code generators is that they be in some sense "composable," *i.e.*, we can build new code generators either by extending or composing existing generators. For example, we might build a generator with its own tuning parameters just for dot products, and build higher-level generators for a matrix-vector multiply which use (or call) the dot product generator.

If generators are composable, then search spaces should be composable, too. For example, suppose that the kernel $t \leftarrow A^T \cdot x, y \leftarrow A \cdot t$ is not supported in an existing sparse kernel tuning system. Mathematically, the locality-sensitive version discussed in Chapter 7 proceeds as follows: for each row a_i^T, first compute the dot product $t_i \leftarrow a_i^T \cdot x$, followed by a vector scale $t_i \cdot a_i$. Thus, we can in principle build a generator which emits the loop construct for iteration over the rows of A, and within the loop call the built-in generators for the dot-product and vector scale. The tuning parameters for the new generator are the cross-product of the parameters of the component generators.

The choice of how to search the new space is a separate issue. We could rely on

known tuning parameters for the individual operations, and simply reuse them for the new generator. Alternatively, since the dot-product and vector scale occur in a new context, we can search for entirely new parameters for the two subcomponents.

We are not restricted to inheriting only the tuning space of the component pieces—in the act of composition, we can add tuning parameters as well. For the locality-sensitive $A^TA\cdot x$ kernel, instead of multiplying by 1 row of A at a time, we can multiply by a block of rows, where the block size is a new tuning parameter.

This general notion of implementation/search space composability is likely to be an important idea in more general tuning systems.

10.3.2 Optimizing beyond kernels, and tuning for applications

We have made considerable progress by focusing on optimization at the level of kernels. However, the performance bounds motivated us in part to consider higher-level kernels such as SpA^TA and sparse $A^\rho\cdot x$. A natural next step is to consider higher-level algorithms. A recent example is a study on the effect of combined register and multiple-vector blocking on the block Lanczos algorithm for solving eigenproblems [161], and the use of multiple vectors in the design of iterative block linear solvers [25, 24].

At the level of applications, SpMV is having an impact in a variety of "new" domains beyond the traditional scientific and engineering applications. A prominent example is the Google PageRank algorithm, where the matrix essentially represents the connectivity graph of the web [242, 148, 58, 202, 303]. The structure of this matrix is very special, and there have been a number of early characterizations of the structure of this matrix [184, 150, 183, 220, 243]. Can this structure be exploited to compute PageRanks more quickly? For instance, PageRank is based on the power method for computing the dominant eigenvector of a matrix, and therefore the kernel $A^\rho\cdot x$ may in principle be applied. Recent analyses suggest that this eigenvector is relatively stable to perturbations in the connectivity matrix for the PageRank problem [232, 187]. Can this property be exploited to further improve $A^\rho\cdot x$ performance by judiciously dropping edges in the tiled representation? Moreover, if it is known that an optimization like SpMM can run 2–7× faster, will this permit new PageRank-like algorithms for specific search contexts [149, 174], or enable the use of alternative numerical algorithms in the spirit of recent experiments [185, 16]?

10.3.3 Systematic data structure selection

We identify deciding when and how to apply specific optimizations for SpMV (Section 5.3) as a current challenge. Even if each optimization had a good heuristic for selecting tuning parameters, how would these heuristics interact? What combinations of optimizations are likely to lead to the largest improvements in performance? This problem is much like the combinatorially hard problem of selecting compiler transformations, and is likely to benefit from lessons learned in compiler construction [199].

10.3.4 Tuning for other architectures and emerging environments

The focus of this dissertation is tuning sparse kernels on platforms based on cache-based superscalar microprocessors. Other important classes of machines include vector architectures [237] and simultaneous multithreaded processors. Current work on tuning SpMV for vector architectures are typically based on formats like jagged diagonal (JAD) format which we found to be especially ill-suited to many cache-based superscalar micros. Thus, an entirely different implementation space may be needed.

A related problem is specifically generating and tuning in the parallel setting. Recently, Kudo, *et al.*, reported on preliminary experiments in tuning parallel SpMV based on Sparsity ideas [198]. They generate MPI versions of SpMV with different strategies based on sending packed vector messages or using block gathers to communicate elements of x distributed across processors.

Adaptability of libraries and software is particularly critical in emerging grid environments. An important question moving forward is how to provide general software system support for automatic tuning to general applications that run in these environments. Current work on applying empirical search methods at every stage of a software system (including compilers, operating systems, and in the run-time environment), and in particular the recent work by Tăpus, *et al.*, on providing library-based support for carrying out searches (the Active Harmony system) [305] or Krintz on binary annotations [196], are all promising directions.

A broad generalization of automatic tuning is recent work on tuning of "whole systems." Recently, Parekh, *et al.*, have looked at designing self-tuning controllers for server environments based on statistical modeling and control theory [244]. This area is particularly challenging due to the difficulties of characterizing dynamic workloads and

modeling the interactions of many complex system components. Recently, Petrini, *et al.*, showed factor of 2 performance improvements on a hydrodynamics application running on a large-scale parallel system (ASCI Q)—this improvement came solely from understanding and altering system software dynamics, requiring no changes to the application code [247]. Looking at automatic tuning at the level of complete hardware/software systems is an exciting current challenge.

10.3.5 Cryptokernels

Our survey of Section 9.4 cites advances in automatic tuning of computational kernels in the domains of linear algebra, signal processing, parallel distributed collective communications primitives, and sorting. Some of these systems show convincingly that a deep knowledge of mathematical structure leads to significant improvements in performance.

Another area in which mathematical structure is likely to play a role is in the area of cryptography. Examples of cryptographic operations (kernels) include encryption, decryption, key generation, inverse modulo a prime, and repeated squaring. There are a number of challenges:

- *Integer-instruction intensive workloads*: The typical instruction mixes are dominated by integer operations on a variety of word sizes.
- *A variety of architectures*: Basic kernels like encryption and decryption need to be implemented on diverse hardware platforms, from 8-bit "smart cards" to high-end workstations.
- *Multiobjective performance optimization*: The metrics of performance of interest include not just execution time, but also storage and power consumption.

As part of the most recent Advanced Encryption Standard (AES) revision sponsored by the National Institute of Standards and Technology (NIST), researchers and practioners were invited to propose new encryption standards and to tune candidate implementations on a variety of architectures [2, 31, 77, 205, 271, 276, 321, 322, 331]. More recently, Bhaskar, *et al.*, have shown how to exploit properties of Galois fields to express varying levels of bit- and word-level parallelism, and then map operations to the integer SIMD instruction sets (*e.g.*, AltiVec, Sun VIS, Intel SSE) available in many modern microprocessors [38]. Together, this

body of work suggests that an automatic tuning system for cryptographic kernels is likely to be a fruitful short-term research opportunity.

10.3.6 Learning models of kernel and applications

Chapter 9 argues that when it is difficult to derive simple analytical models of performance, it may nevertheless be possible to construct *statistical models*. In an empirical search-based system, these are natural models to try to build because the process of search can generate a significant amount of data. It is highly likely that the structure of the models themselves can be automatically derived from high-level specifications [141, 143], or even static analysis [96], and subsequently fit to data.

Performance models are only one example of the type of model we might build. We speculate that recent research on using the data collected from traces could benefit, too. For example, one could imagine building a statistical model of memory reference patterns, based on the memory address traces. Recent work on memory analysis tools are beginning to look at the problem of producing more compact representations of large traces to understand performance (*e.g.*, the SIGMA tool [95], POEMS [59], as well as other trace compaction and mining methods [6, 311, 87, 212]). Oly and Reed apply statistical analysis to predict and prefetch I/O requests in scientific applications [241]. Bringing the full power of statistical modeling to bear on these and related problems seems a promising and exciting area for new research.

Bibliography

[1] PARASOL Project test matrices, 1999. `www.parallab.uib.no/parasol/data.html`.

[2] Advanced Encryption Standard, December 2001. `csrc.nist.gov/CryptoToolkit/aes`.

[3] Berkeley Benchmarking and OPtimization (BeBOP) Project, 2004. `bebop.cs.berkeley.edu`.

[4] M. F. Adams. *Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 1998.

[5] N. Ahmed, N. Mateev, K. Pingali, and P. Stodghill. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of Supercomputing 2000*, Dallas, TX, November 2000.

[6] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[7] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, San Francisco, CA, USA, 2002.

[8] G. Almási and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[9] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal on Scientific Computing*, 14(2):446–460, March 1993.

[10] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[11] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[12] B. S. Andersen, F. Gustavson, A. Karaivanov, J. Wasniewski, and P. Y. Yalamov. LAWRA–Linear Algebra With Recursive Algorithms. In *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, Kazimierz Dolny, Poland, September 1999.

[13] B. S. Andersen, F. G. Gustavson, and J. Wasniewski. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software*, 27(2):214–244, June 2001.

[14] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide: Third Edition.* SIAM, Philadelphia, PA, USA, 1999. `www.netlib.org/lapack/lug`.

[15] S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo. *RS/6000 Scientific and Technical Computing: Power3 Introduction and Tuning.* International Business Machines, Austin, TX, USA, 1998. `www.redbooks.ibm.com`.

[16] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin. Pagerank computation and the structure of the web: Experiments and algorithms. In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, HI, USA, May 2002.

[17] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 268–276, Montreal, Quebec, Canada, 2002. ACM Press.

[18] L. Arge, J. Chase, J. S. Vitter, and R. Wickremesinghe. Efficient sorting using registers and caches. *ACM Journal on Experimental Algorithmics*, 6:1–18, 2001.

[19] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM: The controller's analytical model. In *MICRO-33: Third ACM Workshop on Feedback-Directed Dynamic Optimization*, Monterey, CA, USA, December 2000.

[20] K. Asanović. The IPM WWW home page. `http://www.icsi.berkeley.edu/~krste/IPM.html`.

[21] K. Asanović. The RPRF WWW home page. `http://www.icsi.berkeley.edu/~krste/RPRF.html`.

[22] C. Ashcraft and R. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[23] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS parallel benchmark results. Technical report, NASA Ames Research Center, Moffett Field, CA, USA, October 1994.

[24] A. Baker, J. Dennis, and E. R. Jessup. Toward memory-efficient linear solvers. In J. Palma, J. Dongarra, V. Hernández, and A. A. Sousa, editors, *Proceedings of the 5th International Conference on High Performance Computing for Computational Science (VECPAR)*, volume 2565 of *LNCS*, pages 315–327, Porto, Portugal, June 2002. Springer.

[25] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted GMRES. Technical Report CU-CS-045-03, University of Colorado, Dept. of Computer Science, January 2003.

[26] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc User's Manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2002. www.mcs.anl.gov/petsc.

[27] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[28] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, Paris, France, December 1996.

[29] R. Barnes. Feedback-directed data cache optimizations for the x86. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture, Second Workshop on Feedback-Directed Optimization*, Haifa, Israel, November 1999.

[30] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, USA, 1994.

[31] L. E. Bassham. Efficiency testing of ANSI C implementations of Round 2 candidate algorithms for the Advanced Encryption Standard. In *Proceedings of the 3rd AES Candidate Conference*, New York, NY, USA, April 2000.

[32] G. Baumgartner, D. E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C.-C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Saddayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[33] O. Beckmann and P. H. J. Kelley. Runtime interprocedural data placement optimization for lazy parallel libraries. In *EuroPar*, LNCS. Springer, August 1997.

[34] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir. *The Power4 Processor: Introduction and Tuning Guide*. International Business Machines, Austin, TX, USA, 2001. `www.redbooks.ibm.com`.

[35] M. A. Bender, E. D. Demaine, and M. Farrach-Colton. Cache-oblivious B-Trees. In *IEEE Symposium on Foundations of Computer Science*, pages 399–409, 2000.

[36] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[37] K. Beyls and E. H. D'Hollander. Compile-time cache hint generation for EPIC architectures. In *MICRO-35: Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers*, Istanbul, Turkey, November 2002.

[38] R. Bhaskar, P. K. Dubey, V. Kumar, A. Rudra, and A. Sharma. Efficient Galois field arithmetic on SIMD architectures. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, San Diego, CA, USA, June 2003.

[39] S. Bhowmick, P. Raghavan, and K. Teranishi. A combinatorial scheme for developing efficient composite solvers. In *Proceedings of the International Conference on Computational Science*, volume 2330 of *LNCS*, pages 325–334, Amsterdam, The Netherlands, April 2002. Springer.

[40] P. J. Bickel and K. A. Doksum. *Mathematical Statistics: Basic Ideas and Selected Topics*. Holden-Day, Inc., San Francisco, CA, 1977.

[41] A. J. C. Bik. *Compiler Support for Sparse Matrix Codes*. PhD thesis, Leiden University, 1996.

[42] A. J. C. Bik, P. J. H. Birkhaus, P. M. W. Knijnenburg, and H. A. G. Wijshoff. The automatic generation of sparse primitives. *ACM TOMS*, 24(2):190–225, July 1998.

[43] A. J. C. Bik and H. A. G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31(1):14–24, 1995.

[44] A. J. C. Bik and H. A. G. Wijshoff. Automatic nonzero structure analysis. *SIAM Journal on Computing*, 28(5):1576–1587, 1999.

[45] S. Bikhchandani and S. Sharma. Optimal search with learning. *Journal of Economic Dynamics and Control*, 20:339–359, 1996.

[46] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC.

[47] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C. Chin. The PHiPAC v1.0 matrix-multiply distribution. Technical Report UCB/CSD-98-1020, University of California, Berkeley, October 1998.

[48] Z. W. Birnbaum. Numerical tabulation of the distribution of kolmogorov's statistic for finite sample size. *Journal of the American Statistical Association*, 47:425–441, September 1952.

[49] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, and J. W. von Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) standard: BLAS Technical Forum, 2001. `www.netlib.org/blas/blast-forum`.

[50] S. L. Blackford, J. W. Demmel, J. Dongarra, I. S. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[51] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computations on vector multiprocessors. Technical Report CMU-CS-93-173, Department of Computer Science, Carnegie Mellon Unversity, August 1993.

[52] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, March 1998.

[53] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London, 1997. Chapman and Hall. `math.nist.gov/MatrixMarket`.

[54] I. Brainman and S. Toledo. Nested-dissection orderings for sparse LU with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 23(4):998–1012, May 2002.

[55] E. Brewer. High-level optimization via automated statistical modeling. In *Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, USA, July 1995.

[56] P. Briggs. Sparse matrix multiplication. *SIGPLAN Notices*, 31(11):33–37, November 1996.

[57] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[58] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th International World Wide Web Conference (WWW9)*, 2000. `http://www9.org/w9cdrom/160/160.html`.

[59] J. C. Browne, E. Berger, and A. Dube. Compositional development of performance models in POEMS. *The International Journal of High Performance Computing Applications*, 14(4):283–291, Winter 2000.

[60] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, November 2000.

[61] D. Bruening, T. Garnett, and S. Amarsinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, San Francisco, CA, USA, March 2003.

[62] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. Technical report, Numerical Analysis Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, 1995.

[63] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM Java Grande Conference*, pages 129–141, San Francisco, CA, USA, June 1999.

[64] X. Cai, Y. Saad, and M. Sosonkina. Parallel iterative methods in modern physical applications. In *Proceedings of the International Conference on Computational Science*, volume 2330 of *LNCS*, pages 345–354, Amsterdam, The Netherlands, April 2002. Springer.

[65] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1998.

[66] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing*, pages 114–124, 1992.

[67] L. Carter, J. Ferrante, S. F. Hummel, B. Alpern, and K.-S. Gatlin. Hierarchical tiling: A methodology for high performance. Technical Report UCSD//CS96-508, University of California, San Diego, 1996.

[68] M. Challacombe. A general parallel sparse-blocked matrix multiply for linear scaling scf theory. *Computer Physics Communications*, 128:93, 2000.

[69] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software–Practice & Experience*, 21(12):1301–1321, December 1991.

[70] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297, Snowbird, UT, USA, June 2001.

[71] T.-Y. Chen. *Preconditioning sparse matrices for computing eigenvalues and solving linear systems of equations*. PhD thesis, Computer Science Division, University of California, Berkeley, Berkeley, CA, USA, 2001.

[72] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and LAPACK for clusters. Technical Report UT-CS-03-499, University of Tennessee, January 2003.

[73] Z. Chen, J. J. Dongarra, P. Luszczek, and K. Roche. Self-adapting software for numerical linear algebra library routines on clusters. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Proceedings of the 3rd International Conference on Computational Science*, volume 2659 of *LNCS*, pages 665–672, Melbourne, Australia, June 2003. Springer.

[74] Y. Choi, A. Knies, G. Vedaraman, J. Williamson, and I. Esmer. Design and experience: Using the Intel Itanium2 processor. In *MICRO-35: Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers*, Istanbul, Turkey, November 2002.

[75] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Second Workshop on Feedback-Directed Optimization*, Haifa, Israel, November 1999.

[76] Y. S. Chow, H. Robbins, and D. Siegmund. *Great Expectations: The Theory of Optimal Stopping*. Houghton-Mifflin, Boston, 1971.

[77] C. Clapp. Instruction-level parallelism in AES candidates. In *Proceedings of the 2nd AES Candidate Conference*, Rome, Italy, March 1999.

[78] D. Cociorva, G. Baumgartner, C.-C. Lam, P. Sadayappan, J. Ramanujam, D. E. Bernholdt, and R. Harrison. Space-time trade-off optimization for a class of electronic structure calculations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[79] E. Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, 1999.

[80] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[81] J.-F. Collard and D. Lavery. Optimizations to prevent cache penalties for the Intel Itanium 2. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 105–114, San Francisco, CA, USA, March 2003.

[82] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noy, S. Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. In *ACM Computer Surveys, Symposium on Partial Evaluation*, volume 30, September 1998.

[83] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[84] K. D. Cooper, T. J. Harvey, D. Subramanian, and L. Torczon. Compilation order matters. Technical Report –, Rice University, Houston, TX, USA, January 2002.

[85] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, 2002.

[86] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the ACM National Conference*, 1969.

[87] C. da Lu and D. A. Reed. Compact application signatures for parallel and distributed scientific computing. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[88] A. da Silva, J. Pfaendtner, J. Guo, M. Sienkiewicz, and S. E. Cohn. Assessing the effects of data selection with DAO's physical-space statistical analysis system. In *Proceedings of International Symposium on Assimilation of Observations*, Tokyo, Japan, March 1995.

[89] J. D. Darcy. Finding a fast quicksort implementation for java, Winter 2002. `www.sonic.net/ jddarcy/Research/cs339-quicksort.pdf`.

[90] T. Davis. UF Sparse Matrix Collection. www.cise.ufl.edu/research/sparse/matrices.

[91] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 19(1):140–158, 1997.

[92] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. Technical report, Dept. of Computer and Information Science and Engineering, University of Florida, Gainsville, FL, USA, October 2000.

[93] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[94] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[95] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[96] P. Diniz. A compiler approach to performance prediction using empirical modeling. In *Proceedings of the ICCS Workshop on Performance Evaluation, Modeling, and Anal-*

ysis of Scientific Applications on Large-Scale Systems, Melbourne, Australia, June 2003.

[97] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[98] F. Dobrian, G. Kumfert, and A. Pothen. Object-oriented desig for sparse direct solvers. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, volume 1505 of *LNCS*, pages 207–214, Santa Fe, NM, USA, December 1998.

[99] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[100] J. Dongarra. Top 500 supercomputers. `www.top500.org`.

[101] J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[102] J. Dongarra, J. D. Croz, I. Duff, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[103] J. Dongarra and V. Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science*, Melbourne, Australia, June 2003.

[104] J. Dongarra, V. Eijkhout, and P. Luszczek. Recursive approach in sparse matrix LU factorization. *Scientific Programming*, 9(1):51–60, 2001.

[105] C. C. Douglas. Caching in with multigrid algorithms: Problems in two dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.

[106] C. C. Douglas, J. Hu, M. Kowarschik, U. Rude, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.

[107] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, Oxford, 1986.

[108] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software*, 28(2):239–267, June 2002.

[109] I. S. Duff and C. Vömel. Algorithm 818: A reference model implementation of the sparse BLAS in Fortran 95. *ACM Transactions on Mathematical Software*, 28(2):268–283, June 2002.

[110] J.-G. Dumas, T. Gautier, and C. Pernet. Finite Field Linear Algebra Subroutines. In *Proceedings of the International Symposium on Symbolic and Algebraic Computations*, Lille, France, July 2002.

[111] J. W. Eaton. Octave, 2003. `www.octave.org`.

[112] L. Enyou and C. Thomborson. Data cache parameter measurements. In *Proceedings of the International Conference on Computer Design*, pages 376–383, October 1998.

[113] A. Ertl. The memory wall fallacy, 1996. `www.complang.tuwien.ac.at/anton/memory-wall.html`.

[114] T. S. Ferguson. Who solved the secretary problem? *Statistical Science*, 3:282–289, August 1989.

[115] T. S. Ferguson. *Optimal Stopping and Applications*. (unpublished manuscript), UCLA course notes: `www.math.ucla.edu/~tom`, 2000.

[116] S. Filippone and M. Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26(4):527–550, December 2000.

[117] B. B. Fraguela, R. Doallo, and E. L. Zapata. Memory hierarchy performance prediction for sparse blocked algorithms. *Parallel Processing Letters*, 9(3), March 1999.

[118] B. B. Fraguela, R. Doallo, and E. L. Zapta. Automatic analytic modeling for the estimation of cache misses. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 221–231, Newport Beach, CA, USA, October 1999.

[119] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, July 1997.

[120] J. D. Frens and D. S. Wise. QR factorization with morton-ordered quadtree matrices for memory re-use and parallelism. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[121] B. Frey. *Graphical Models for Machine Learning and Digital Communications*. MIT Press, Boston, 1998.

[122] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[123] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Seattle, Washington, May 1998.

[124] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache–oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, New York, NY, October 1999.

[125] K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Proceedings of Supercomputing*, Portland, OR, USA, November 1999.

[126] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In *Proceedings of the 2nd European Conference on Parallel Processing (Euro-Par'96), Lyon, France*, volume 1123,1124 of *Lecture Notes in Computer Science*, pages 128–135. Springer-Verlag, Berlin, Germany, 1996.

[127] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, April 1973.

[128] A. George and J. W. H. Liu. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software*, 5(2):139–162, June 1979.

[129] R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Proceedings of the International Conference on Parallel Computing (ParCo)*, pages 308–315. Imperial College Press, 1999.

[130] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[131] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. Technical Report CSL-91-04, Xerox PARC, July 1991.

[132] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[133] M. Gilli, E. Kellezi, and G. Pauletto. Solving finite difference schemes arising in trivariate option pricing. *Journal of Economic Dynamics and Control*, 26, 2002.

[134] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, University of Texas at Austin, November 2002.

[135] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982.

[136] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *SIGPLAN Notices*, 27(7):341–352, July 1992.

[137] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Journal of Theoretical Computer Science*, 248(1–2):147–199, October 2000.

[138] R. G. Grimes, D. R. Kincaid, and D. M. Young. ITPACK 2.0 User's Guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas, Austin, TX, USA, August 1979.

[139] W. D. Gropp, D. K. Kasushik, D. E. Keyes, and B. F. Smith. Towards realistic bounds for implicit CFD codes. In *Proceedings of Parallel Computational Fluid Dynamics*, pages 241–248, 1999.

[140] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Latency, bandwidth, and concurrent issue limitations in high-performance CFD. In *Proceedings of the First MIT Conference on Computational Fluid and Solid Mechanics*, Cambridge, MA, USA, June 2001.

[141] J. Gunnels and R. van de Geijn. Formal methods for high-performance linear algebra algorithms. In *Working Conference on Software Architectures for Scientific Com puting Applications*, Ottawa, Ontario, Canada, October 2000.

[142] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4), December 2001.

[143] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn. A family of high-performance matrix multiplication algorithms. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 51–60, San Francisco, CA, May 2001. Springer.

[144] A. Gupta and V. Kumar. Parallel algorithms for forward elimination and backward substitution in direct solution of sparse linear systems. In *Supercomputing*, San Diego, CA, 1995.

[145] R. Gupta and R. Bodik. Adaptive loop transformations for scientific programs. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, San Antonio, TX, USA, October 1995.

[146] F. G. Gustavson. Two fast algorithms for sparse matrices: multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.

[147] E. J. Haunschmid, C. W. Ueberhuber, and P. Wurzinger. Cache oblivious high performance algorithms for matrix multiplication. Technical Report AURORA TR2002-08, Vienna University of Technology, 2002.

[148] T. Haveliwala. Efficient computation of PageRank. Technical Report 1999-31, Stanford University, 2000.

[149] T. Haveliwala. Topic-sensitive PageRank. In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, HI, USA, May 2002.

[150] T. H. Haveliwala and S. D. Kamvar. The second eigenvalue of the Google matrix, March 2003. (unpublished manuscript).

[151] M. T. Heath and P. Raghavan. The performance of parallel sparse triangular solution. In S. Schreiber, M. T. Heath, and A. Ranade, editors, *Proceedings of the IMA Workshop for Algorithms for Parallel Processing*, volume 105, pages 289–306, Minneapolis, MN, 1998. Springer-Verlag.

[152] G. Heber, R. Biswas, and G. R. Rao. Self-avoiding walks over adaptive unstructured grids. *Concurrency: Practice and Experience*, 12(2–3):85–109, 2000.

[153] G. Heber, A. J. Dolgert, M. Alt, K. A. Mazurkiewicz, and L. Stringer. Fracture mechanics on the intel itanium architecture: A case study. In *Workshop on EPIC Architectures and Compiler Technology (ACM MICRO 34)*, Austin, TX, December 2001.

[154] B.-O. Heimsund. JMP: A sparse matrix library in Java, 2003. `http://www.mi.uib.no/~bjornoh/jmp`.

[155] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[156] G. M. Henry. Flexible, high-performance matrix multiply via a self-modifying runtime code. Technical Report TR-2001-46, University of Texas at Austin, December 2001.

[157] D. B. Heras, V. B. Perez, J. C. C. Dominguez, and F. F. Rivera. Modeling and improving locality for irregular problems: sparse matrix-vector product on cache memories as a case study. In *HPCN Europe*, pages 201–210, 1999.

[158] Hewlett-Packard. HP's mathematical software library (mlib), 2002. www.hp.com.

[159] J. W. Hong and H. T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 326–333, May 1981.

[160] E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, and C. E. Houstis. PYTHIA-II: A knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software*, 26(2):277–253, June 2000.

[161] C. Hsu. Effects of block size on the block Lanczos algorithm, June 2003. Senior Honors Thesis.

[162] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of Supercomputing*, August 1996.

[163] IBM. ESSL Guide and Reference, 2001. `www-1.ibm.com/servers/eserver/pseries/library/sp_bo`

[164] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, May 2000.

[165] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 2004. (*t*o appear).

[166] E.-J. Im and K. A. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, USA, March 1999.

[167] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 127–136, San Francisco, CA, May 2001. Springer.

[168] Intel. Intel Itanium Processor Reference Manual for Software Optimization, November 2001.

[169] Intel. Intel math kernel library, version 6.0, 2003. `www.intel.com/software/products/mkl`.

[170] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. In *Proceedings of the International Conference on Computational Science*, volume 2330 of *LNCS*, pages 335–344, Amsterdam, The Netherlands, April 2002. Springer.

[171] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. In *Proceedings of the International Conference on Computational Science*, number II in LNCS, pages 335–344, April 2002.

[172] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of the International Scientific Computing in Object-Oriented Parallel Environments*, Marina del Rey, CA, USA, December 1997.

[173] W. Jalby and C. Lemuet. Exploring and optimizing Itanium2 caches performance for scientific code. In *MICRO-35: Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers*, Istanbul, Turkey, November 2002.

[174] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungary, May 2003.

[175] G. Jin and J. Mellor-Crummey. Experiences tuning SMG98—a semicoarsening multigrid benchmark based on the *hypre* library. In *Proceedings of the International Conference on Supercomputing*, pages 305–314, New York, NY, USA, June 2002.

[176] M. T. Jones and P. E. Plassman. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20:753–773, 1994.

[177] M. I. Jordan. Why the logistic function? Technical Report 9503, MIT, 1995.

[178] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and F. Gustavson. PSPASES: An efficient and scalable parallel sparse direct solver. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[179] M. V. Joshi, , A. G. G. Karypis, and V. Kumar. A high performance two dimensional scalable parallel algorithm for solving sparse triangular systems. In *Proceedings of the 4th International Conference on High Performance Computing*, Cambridge, MA, December 1997.

[180] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. Technical Report 171, Compaq SRC, August 2001.

[181] B. Kagstrom, P. Ling, and C. V. Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software*, 24(3):268–302, 1998.

[182] B. Kagstrom and C. V. Loan. GEMM-based Level-3 BLAS. Technical Report CTC91-TR47, Dept. of Computer Science, Cornell University, December 1989.

[183] S. D. Kamvar and T. H. Haveliwala. Exploiting the block structure of the web for computing PageRank, March 2003.

[184] S. D. Kamvar, T. H. Haveliwala, and G. H. Golub. Adaptive methods for the computation of PageRank. In *Proceedings of the International Conference on Numerical Solution of Markov Chains*, Urbana, IL, USA, September 2003.

[185] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating PageRank computations. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungary, May 2003.

[186] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minneapolis, MN, USA, 1995.

[187] S. J. Kirkland. Conditioning properties of the stationary distribution for a Markov chain. *Electronic Journal of Linear Algebra*, 10:1–15, 2003.

[188] L. B. Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3–4):144–149, December 2002.

[189] T. Kistler and M. Franz. Continuous program optimization: a case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, July 2003.

[190] T. Kisuki, P. M. Knijnenburg, M. F. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proceedings of the 8th International Workshop on Compilers for Parallel Computers*, pages 35–44, Aussois, France, 2000.

[191] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[192] P. Knijnenburg and H. A. G. Wijshoff. On improving data locality in sparse matrix computations. Technical Report 94-15, Dept. of Computer Science, Leiden University, Leiden, The Netherlands, 1994.

[193] D. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1(2):105–133, April–June 1971.

[194] A. N. Ko and J. A. Izaguirre. MDSimAid: Automatic optimization of fast electrostatics algorithms for molecular simulations. In *Proceedings of the International Conference on Computational Science*, LNCS, Melbourne, Australia, 2003. Springer.

[195] A. N. Kolmogorov. Confidence limits for an unknown distribution function. *Annals of Mathematical Statistics*, 12:461–463, 1941.

[196] C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 69–78, San Francisco, CA, USA, March 2003.

[197] U. Kster. Benchmarks: Sparse matrix vector multiplication, 2001.

[198] M. Kudo, H. Kuroda, and Y. Kanada. Parallel blocked sparse matrix-vector multiplication with auto-tuning system. In *Proceedings of the International Conference on Computational Science*, Melbourne, Australia, June 2003.

[199] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2003.

[200] M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 511–518, Stanford, CA, June 2000.

[201] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[202] A. N. Langville and C. D. Meyer. A survey of eigenvector methods of web information retrieval, February 2003. (submitted to *SIAM Review*).

[203] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[204] B. C. Lee, R. Vuduc, J. W. Demmel, K. A. Yelick, M. deLorimier, and L. Zhong. Performance optimizations and bounds for sparse symmetric matrix-multiple vector multiply. Technical Report UCB/CSD-03-1297, University of California, Berkeley, Berkeley, CA, USA, November 2003.

[205] L. Leibrock. Exploratory candidate algorithm performance characteristics in commercial Symmetric Multiprocessing (SMP) environments for the Advanced Encryption Standard (AES). In *Proceedings of the 2nd AES Candidate Conference*, Rome, Italy, March 1999.

[206] M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report TR-490, Dept. of Computer Science, Indiana University, September 1997.

[207] C. Leopold. Tight bounds on capacity misses for 3D stencil codes. In *Proceedings of the International Conference on Computational Science*, volume 2329 of *LNCS*, pages 843–852, Amsterdam, The Netherlands, April 2002. Springer.

[208] G. Li and T. F. Coleman. A parallel triangular solver for a distributed-memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, pages 485–502, May 1998.

[209] X. Li, M. J. Garzarn, and D. Padua. A memory hierarchy conscious and self-tunable sorting library. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO) with Special Emphasis on Feedback-Directed and Runtime Optimization*, Palo Alto, CA, USA, March 2004.

[210] P. Liniker, O. Beckmann, and P. H. J. Kelly. Delayed evaluation, self-optimising software components as a programming model. In *Euro-Par*, Paderborn, Germany, August 2002.

[211] M. Lucks and I. Gladwell. Automated selection of mathematical software. *ACM Transactions on Mathematical Software*, 18(1):11–34, March 1992.

[212] J. Marathe, F. Mueller, T. Mohan, S. A. McKee, B. R. de Supinski, and A. Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 289–300, San Francisco, CA, USA, March 2003.

[213] H. Massalin. Superoptimizer–a look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Palo Alto, CA, USA, 1987.

[214] N. Mateev, K. Pingali, and P. Stodghill. The Bernoulli Generic Matrix Library. Technical Report TR-2000-1808, Cornell University, 2000.

[215] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *International Conference on Supercomputing*, 2000.

[216] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *Newsletter of the IEEE Technical Committee on Computer Architecture*, December 1995. `http://tab.computer.org/tcca/NEWS/DEC95/DEC95.HTM`.

[217] J. D. McCalpin. STREAM: Measuring sustainable memory bandwidth in high performance computers, 1995. `http://www.cs.virginia.edu/stream`.

[218] J. D. McCalpin and M. Smotherman. Automatic benchmark generation for cache optimization of matrix algorithms. In R. Geist and S. Junkins, editors, *Proceedings of the 33rd Annual Southeast Conference*, pages 195–204. ACM, March 1995.

[219] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[220] G. Meghabghab. Google's web page ranking applied to different topological web graph structures. *Journal of the American Society for Information Science*, 52(9):736–747, 2001.

[221] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix vector multiply using unroll-and-jam. In *Proceedings of the Los Alamos Computer Science Institute Third Annual Symposium*, Santa Fe, NM, USA, October 2002.

[222] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, Austin, TX, October 1999.

[223] K. Minami and H. Okuda. Performance optimization of GeoFEM on various computer architectures. Technical Report GeoFEM 2001-006, Research Organization for Information Science and Technology (RIST), Tokyo, Japan, october 2001.

[224] D. Mirković and S. L. Johnsson. Automatic performance tuning in the UHFFT library. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 71–80, San Francisco, CA, May 2001. Springer.

[225] D. Mirkovic, R. Mahasoom, and L. Johnsson. An adaptive software library for fast fourier transforms. In *Proceedings of the International Conference on Supercomputing*, pages 215–224, Sante Fe, NM, May 2000.

[226] N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 81–96, San Francisco, CA, May 2001. Springer.

[227] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[228] H. J. Moon, R. Vuduc, J. W. Demmel, and K. A. Yelick. Matrix splitting and reordering for sparse matrix-vector multiply. Technical Report (*to appear*), University of California, Berkeley, Berkeley, CA, USA, 2004.

[229] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[230] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software: Practice and Experience*, 24:632–642, 1994.

[231] J. J. Navarro, E. García, J. L. Larriba-Pey, and T. Juan. Algorithms for sparse matrix computations on high-performance workstations. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 301–308, Philadelpha, PA, USA, May 1996.

[232] A. Y. Ng, A. X. Zheng, and M. I. Jordan. Link analysis, eigenvectors, and stability. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, Seattle, WA, USA, August 2001.

[233] A. Nisbet. GAPS: Iterative feedback directed parallelization using genetic algorithms. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, Paris, France, June 1998.

[234] A. P. Nisbet. Towards retargettable compilers—feedback-directed compilation using genetic algorithms. In *Proceedings of Compilers for Parallel Computers*, Edinburgh, Scotland, June 2001.

[235] R. Nishtala, R. Vuduc, J. W. Demmel, and K. A. Yelick. Performance optimizations and bounds for sparse symmetric matrix-multiple vector multiply. Technical Report (*to appear*), University of California, Berkeley, Berkeley, CA, USA, 2004.

[236] G. E. Noether. Note on the kolmogorov statistic in the discrete case. *Metrika*, 7:115–116, 1963.

[237] H. Okuda, K. Nakajima, M. Iizuka, L. Chen, and H. Nakamura. Parallel finite element analysis platform for the Earth Simulator: GeoFEM. In P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, editors, *Proceedings of the 3rd International Conference on Computational Science Workshop on Computational Earthquake Physics and Solid Earth System Simulation*, volume III of *LNCS 2659*, pages 773–780, Melbourne, Australia, June 2003. Springer.

[238] H. Okuda, K. Nakajima, M. Iizuka, L. Chen, and H. Nakamura. Parallel finite element analysis platform for the Earth Simulator: GeoFEM. Technical Report 03-001, University of Tokyo, January 2003.

[239] L. Oliker, J. Carter, J. Shalf, D. Skinner, S. Ethier, R. Biswas, J. Djomehri, and R. V. der Wijngaart. Evaluation of cache-based superscalar and cacheless vector

architectures for scientific computations. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, 2003. (*to appear*).

[240] J. H. Olsen and S. C. Skov. Cache-oblivious algorithms in practice. Master's thesis, University of Copenhagen, Copehagen, Denmark, 2002.

[241] J. Oly and D. A. Reed. Markov model prediction of I/O requests for scientific applications. In *Proceedings of the International Conference on Supercomputing*, pages 147–155, New York, NY, USA, June 2002.

[242] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford University, November 1999.

[243] G. Pandurangan, P. Raghavan, and E. Upfal. Using PageRank to characterize web structure. In *Proceedings of the 8th Annual International Computing and Combinatorics Conference*, 2002.

[244] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. S. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[245] D. Parello, O. Temam, and J.-M. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance—matrix multiply revisited. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[246] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 101–112, Portland, OR, USA, January 2002. ACM Press.

[247] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of Supercomputing*, Phoenix, AZ, USA, November 2003.

[248] B. Philippe, Y. Saad, and W. J. Stewart. Numerical methods in Markov chain modelling. *Operations Research*, 40(6):1156–1179, 1992.

[249] G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[250] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing*, 1999.

[251] J. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods — Support Vector Learning*, Jan 1999.

[252] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.

[253] U. W. Pooch and A. Nieder. A survey of indexing techniques for sparse matrices. *Computing Surveys*, 5(2):109–133, June 1973.

[254] W. Pugh and T. Shpeisman. Generation of efficient code for sparse matrix computations. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, August 1998.

[255] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast automatic generation of DSP algorithms. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 97–106, San Francisco, CA, May 2001. Springer.

[256] P. Raghavan. Efficient parallel triangular solution using selective inversion. *Parallel Processing Letters*, 8(1):29–40, 1998.

[257] N. Ramakrishnan and R. E. Valdés-Pérez. Note on generalization in experimental algorithmics. *ACM Transactions on Mathematical Software*, 26(4):568–580, December 2000.

[258] K. Remington and R. Pozo. NIST Sparse BLAS: User's Guide. Technical report, NIST, 1996. `gams.nist.gov/spblas`.

[259] Y. Renard and J. Pommier. GMM++: A generic template matrix C++ library, September 2003. `www.gmm.insa-tlse.fr/getfem/gmm.html`.

[260] J. A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, Belmont, CA, 2nd edition, 1995.

[261] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[262] J. R. Rice and R. F. Boisvert. *Solving elliptic problems using ELLPACK*. Springer, New York, NY, USA, 1985.

[263] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. *Graph Theory and Computing*, pages 183–217, 1973.

[264] E. Rothberg. Alternatives for solving sparse triangular systems on distributed-memory multiprocessors. *Parallel Computing*, 21(7):1121–1136, 1995.

[265] E. Rothberg and A. Gupta. Parallel ICCG on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck. *Parallel Computing*, 18(7):719–741, July 1992.

[266] Y. Saad. Kyrlov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1228, November 1989.

[267] Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.

[268] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20th Century. Technical Report umsi-99-152, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, USA, 1999.

[269] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, University of California, Berkeley, February 1992.

[270] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[271] F. Sano, M. Koike, S. Kawamura, and M. Shiba. Performance evaluation of AES finalists on the high-end Smart Card. In *Proceedings of the 3rd AES Candidate Conference*, New York, NY, USA, April 2000.

[272] N. G. Santiago, D. T. Rover, and D. Rodriguez. A statistical approach for the analysis of the relation between low-level performance information, the code, and the environment. In *Proceedings of the ICPP 4th Workshop on High Performance Scientfic and Engineering Computing with Applications*, pages 282–289, Vancouver, British Columbia, Canada, August 2002.

[273] E. E. Santos. Solving triangular linear systems in parallel using substitution. In *Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 553–560, San Antonio, TX, October 1995.

[274] V. Sarkar. Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 29(5), October 2001.

[275] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In D.-Z. Du and M. Li, editors, *Computing and Combinatorics*, volume 959 of *LNCS*, pages 270–281, 1995.

[276] B. Schneier and D. Whiting. A performance comparison of the five AES finalists. In *Proceedings of the 3rd AES Candidate Conference*, New York, NY, USA, April 2000.

[277] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 107–116, San Francisco, CA, May 2001. Springer.

[278] J. G. Siek and A. Lumsdaine. A rational approach to portable high performance: the Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (fast) library. In *Proceedings of ECOOP*, 1998.

[279] B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proc. of the 17th Int'l Conf. on Machine Learning*, 2000.

[280] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*, Boston, MA, USA, January 2000.

[281] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. Technical Report NC2-TR-1998-030, European Community ESPRIT Working Group in Neural and Computational Learning Theory, 1998. `www.neurocolt.com`.

[282] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, December 2001.

[283] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[284] A. Stathopoulos and S.-H. Teng. Recovering mesh geometry from a stiffness matrix. *Numerical Algorithms*, 30(3–4):303–322, August 2002.

[285] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2003.

[286] W. J. Stewart. MARCA Models Home Page, 1995. `http://www.csc.ncsu.edu/faculty/WStewart/MARCA_Models/MARCA_Models.html`.

[287] P. Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, August 1997.

[288] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 137–146, San Francisco, CA, May 2001. Springer.

[289] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2003.

[290] P. D. Sulatycke and K. Ghosh. Caching-efficient multithreaded fast multiplication of sparse matrices. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, Orlando, FL, USA, March–April 1998.

[291] Sun-Microsystems. UltraSPARC IIi: User's Manual, 1999.

[292] Sun-Microsystems. Sun Performance Library User's Guide for Fortran and C, 2002. `docs.sun.com/db/doc/806-7995`.

[293] D. K. Tafti. GenIDLEST – A scalable parallel computational tool for simulating complex turbulent flows. In *Proceedings of the ASME International Mechanical Engineering Congress and Exposition*, New York, NY, USA, November 2001. `mycroft.ncsa.uiuc.edu/www-0/projects/GenIDLEST`.

[294] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing '92*, 1992.

[295] S.-H. Teng. Fast nested dissection on finite element meshes. *SIAM Journal on Matrix Analysis and Applications*, 18(3):552–565, 1997.

[296] I. The MathWorks. Matlab, 2003. `www.mathworks.com`.

[297] J. W. Thomas. Inlining of mathematical functions in HP-UX for Itanium 2. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 135–144, San Francisco, CA, USA, March 2003.

[298] C. Thomborson and Y. Yu. Measuring data cache and TLB parameters under Linux. In *Proceedings of the Symposium on Performance Evaluation of Computer and Telecommunication Systems*, Vancouver, British Columbia, July 2000. Society for Computer Simulation International.

[299] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of Supercomputing '98*, Orlando, FL, November 1998.

[300] W. F. Tinney and J. W. Walker. Direct solution of sparse network equations by optimally ordered triangular factorization. In *Proceedings of IEEE*, volume 55, pages 1801–1809, November 1967.

[301] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[302] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[303] J. Tomlin. A new paradigm for ranking pages on the world wide web. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungary, May 2003.

[304] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 204–215, San Francisco, CA, USA, March 2003.

[305] C. Tăpus, I.-H. Chung, and J. K. Hollingsworth. Active Harmony: Towards automated performance tuning. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[306] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective operations. In *Proceedings of Supercomputing 2000*, Dallas, TX, November 2000.

[307] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline-unrolling trade-offs. In *Proceedings of the 4th International Workshop on Compilers for Embedded Systems*, St. Goar, Germany, September 1999.

[308] V. N. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, Inc., 1998.

[309] T. Veldhuizen. Arrays in Blitz++. In *Proceedings of ISCOPE*, volume 1505 of *LNCS*. Springer-Verlag, 1998.

[310] T. L. Veldhuizen and D. Gannon. Active Libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Philadelphia, PA, USA, 1998. SIAM.

[311] J. S. Vetter and A. Yoo. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[312] M. J. Voss and R. Eigenmann. ADAPT: Automated De-coupled Adaptive Program Transforma tion. In *Proceedings of the International Conference on Parallel Process ing*, Toronto, Canada, August 2000.

[313] R. Vuduc, J. Demmel, and J. Bilmes. Statistical modeling of feedback data in an automatic tuning system. In *MICRO-33: Third ACM Workshop on Feedback-Directed Dynamic Optimization*, Monterey, CA, December 2000.

[314] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. *International Journal of High Performance Computing Applications*, 2004. (*to* appear).

[315] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for automatic performance tuning. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 117–126, San Francisco, CA, May 2001. Springer.

[316] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[317] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick. Memory hierarchy optimizations and bounds for sparse A^TAx. In *Proceedings of the ICCS Workshop on Parallel Linear Algebra*, volume LNCS, Melbourne, Australia, June 2003. Springer.

[318] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick. Memory hierarchy optimizations and bounds for sparse A^TAx. Technical Report UCB/CSD-03-1232, University of California, Berkeley, Berkeley, CA, USA, February 2003.

[319] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries*, New York, USA, June 2002.

[320] W. Wang and D. P. O'Leary. Adaptive use of iterative methods in interior point methods for linear programming. Technical Report UMIACS-95-111, University of Maryland at College Park, College Park, MD, USA, 1995.

[321] B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware performance simulations of Round 2 Advanced Encryption Standard algorithms. In *Proceedings of the 3rd AES Candidate Conference*, New York, NY, USA, April 2000.

[322] R. Weiss and N. Binkert. A comparison of AES candidates on the Alpha 21264. In *Proceedings of the 3rd AES Candidate Conference*, New York, NY, USA, April 2000.

[323] G. Wellein, G. Hager, A. Basermann, and H. Fehske. Fast sparse matrix-vector multiplication for teraflop/s computers. In J. Palma, J. Dongarra, V. Hernández, and A. A. Sousa, editors, *Proceedings of the 5th International Conference on High Performance Computing for Computational Science (VECPAR)*, volume 2565 of *LNCS*, pages 302–314, Porto, Portugal, June 2002. Springer.

[324] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing*, Orlando, FL, 1998.

[325] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[326] J. B. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings of the International Conference on High-Performance C omputing*, 1997.

[327] D. J. Wilkinson and S. K. H. Yeung. A sparse matrix approach to bayesian computation in large linear models. *Computational Statistics and Data Analysis*, (to appear), 2002.

[328] K. M. Wilson and K. Olukotun. High-bandwidth on-chip cache design. *IEEE Transactions on Computers*, 50(4):292–307, April 2001.

[329] D. S. Wise, J. D. Frens, Y. Gu, and G. A. Alexander. Language support for Morton-order matrices. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 24–33, Snowbird, UT, USA, 2001. ACM Press.

[330] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[331] J. Worley, B. Worley, T. Christian, and C. Worley. AES finalists on PA-RISC and IA-64: implementations and performance. In *Proceedings of the 3rd AES Candidate Conference*, New York, NY, USA, April 2000.

[332] C. F. J. Wu and M. Hamada. *Experiments: Planning, Analysis, and Parameter Design Optimization*. John Wiley and Sons, Inc., 2000.

[333] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[334] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 169–181, Vancouver, BC Canada, June 2000.

[335] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2003.

[336] D. M. Young, T. C. Oppe, D. R. Kincaid, and L. J. Hayes. On the use of vector computers for solving large sparse linear systems. Technical Report CNA-199, Center for Numerical Analysis, University of Texas at Austin, Austin, TX, USA, 1995.

Appendix A

Sparse Matrix-Vector Multiply Historical Data

Strictly speaking, the individual data points shown in Figure 1.1 are not directly comparable because different reports use different test matrices. Nevertheless, all data points except NAS CG data points [23] correspond to performance reported for matrices from physical simulation applications based on finite element method (FEM) or finite difference methods. Thus, we would expect the data to be roughly comparable.

We show the data in tabulated form in Tables A.1–A.2, and note the sources. "Year" indicates year of processor production at the specified clock rate. The year is prefixed by the month (*e.g.*, "11/1992") if known; otherwise, we take the month to be June ("6") when plotting and performing fits on the data points.

Performance on the NAS CG data points [23] have been increased by a factor of 4/3 for two reasons: (1) CG performs more than just sparse matrix-vector multiply (SpMV), so the 4/3 factor assumes the SpMV component runs faster than what is reported by NAS CG, and (2) the CG matrices are randomly structured unblocked matrices.

We argue that the fitted regression lines in Figure 1.1 are reasonable:

- The R^2 values (fraction of variance in performance accounted for by the independent variable, "year") for the untuned microprocessor fit is .62, tuned microprocessor data is .67, and tuned vector data is .50. Thus, for the microprocessor data, more than 60% of the variability in performance is explained by the independent variable, "year."
- We plot the residuals for the microprocessor tuned and untuned data in Figure A.1

(*top*). Specifically, the residual is $\log_2(P_i) - \log_2(p(t_i))$, where each data point is (t_i, P_i), and $p(t)$ is the fitted performance of the form $p_0 2^{\frac{t}{\tau}}$. Qualitatively speaking, the scatter of the data points about 0 indicates that there is probably no reason to suspect a systematic fitting error.

- We plot the tuned microprocessor performance divided by untuned performance (or "speedups") over time in Figure A.1 (*bottom*). We show the NAS CG data points distinctly, and omit a few data points for which untuned performance was not available (see Table A.1). The solid line shows the tuned microprocessor fitted line divided by the untuned line. The trend of this ratio roughly capture the general trend toward increasing speedups over time.

Together, these observations suggest that the regression fits are reasonable, though not all of the variability can be accounted for only by "year."

	Processor	Year	MHz	Peak Mflop/s	Ref. Mflop/s	Tuned Mflop/s	Source
1	Intel i860	1987	5	23	5	8	[266]
2	Motorola 88100	1989	20	20	—	1	[23]
3	Kendall Sq. KSR1	1991	20	40	2	3	[23]
4	Matsushita Adenart	1991	20	20	—	1	[23]
5	DEC alpha21064	1992	150	150	20	25	[88]
6	Kendall Sq. KSR2	1992	40	80	5	6	[23]
7	HP PA-7100	11/1992	100	100	7	9	[23]
8	nCUBE 2s	1993	25	192	—	51	[23]
9	DEC 21164	1994	500	1000	43	90	[167]
10	Meiko CS-2	1994	50	40	—	13	[23]
11	MIPS R8000	6/1994	76	300	—	39	[23]
12	HP PA-7200	6/1994	120	240	13	22	[326]
13	DEC Alpha 21164	1995	500	1000	43	58	[129]
14	Sun Ultra 1	5/1995	143	286	17	22	[301]
15	IBM PowerPC 604e	9/1995	190	190	20	25	[167]
16	IBM Power2	1996	66	266	40	100	[301]
17	MIPS R10000	1996	250	500	45	90	[293]
18	Hitachi SR-2201	1996	150	300	25	40	[197]
19	HP PA-8000	3/1996	180	360	48	80	[129]
20	HP PA-8200	9/1996	240	480	6	8	[197]
21	IBM Power2	1997	160	640	56	140	[129]
22	IBM Power3	1997	375	1500	164	240	[316]
23	Intel Pentium II	9/1997	266	266	11	11	[52]
24	Sun Ultra 2i	3/1998	333	666	36	73	[316]
25	MIPS R12000	11/1998	300	600	94	109	[221]
26	DEC Alpha 21264a	1999	667	1334	160	254	[221]
27	Intel Pentium III-m	9/2000	800	800	59	122	[316]
28	Intel Pentium 4	11/2000	1500	3000	327	425	[316]
29	Hitachi SR-8000	2001	250	1000	45	145	[223]
30	IBM Power4	2001	1300	5200	595	805	[316]
31	Intel Itanium	5/2001	800	3200	120	345	[316]
32	Sun Ultra 3	2002	900	1800	53	108	[316]
33	Intel Itanium 2	3/2002	900	3600	295	1200	[316]

Table A.1: **Historical SpMV data: microprocessors**. Data points taken from NAS CG results are cited accordingly [23]. "Year" indicates year of processor production at the specified clock rate. The year is prefixed by the month (*e.g.*, "11/1992") if known; otherwise, we take the month to be June ("6") when plotting the data points. For a few data points, reference performance was not available (*e.g.*, Platform 2 based on the Motorola 88100).

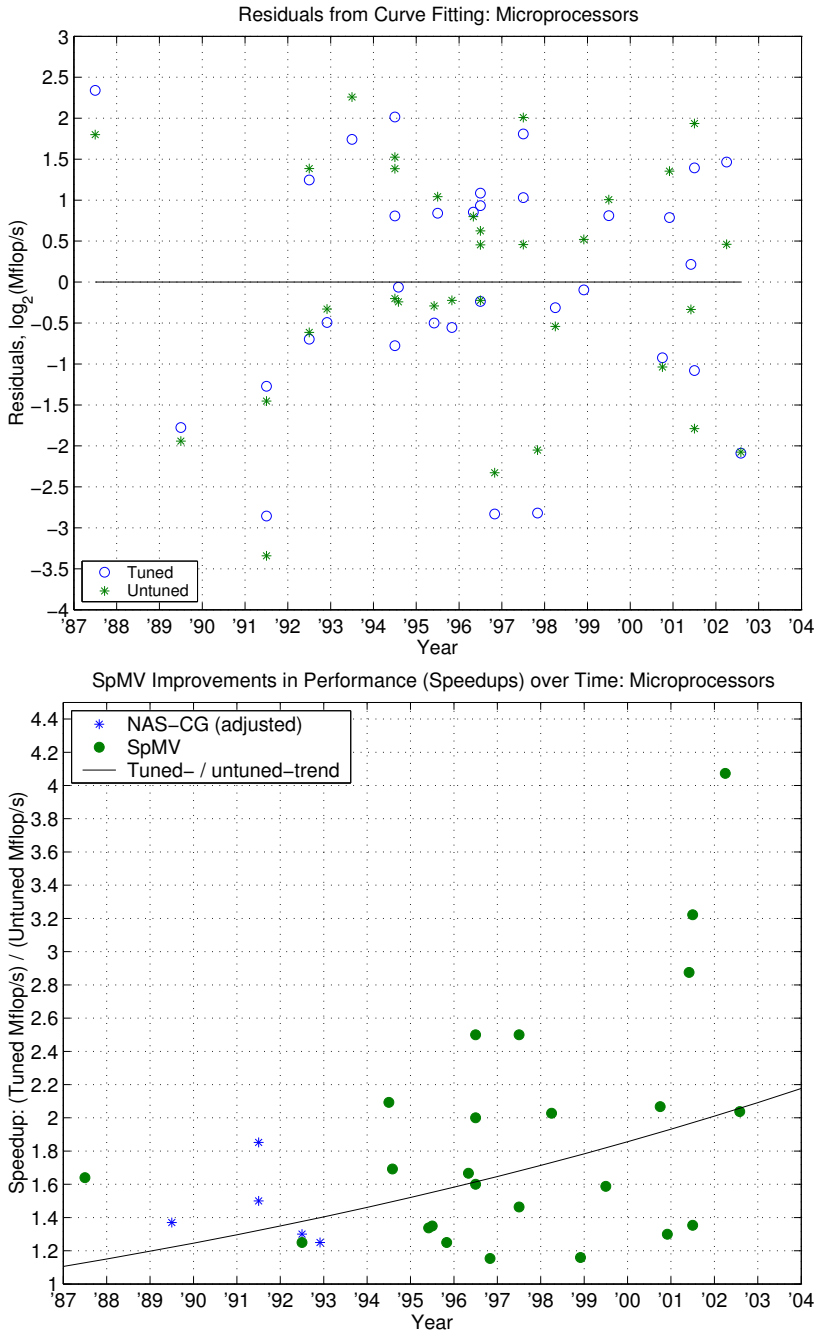

Figure A.1: **Partial, qualitative justification for fitted trends**. (*Top*) The scatter of the residuals between the fitted and true data points about 0 suggest that there is no systematic error in the fit. (*Bottom*) The ratio of the fitted lines (black solid line) roughly captures the true trend in "speedup" (true tuned performance divided by untuned performance) over time.

	Processor	Year	MHz	Peak Mflop/s	Ref. Mflop/s	Tuned Mflop/s	Source
1	Cray Y-MP	6/1988		333	127	127	[23]
2	Cray EL	6/1989		133	33	33	[23]
3	Cray C90	1992	240	960	120	234	[51]
4	Fujitsu VPP500	1994	100	1600	267	267	[23]
5	Cray SV1	1996	500	2000	125	125	[197]
6	NEC SX-4	1996	125	2000	600	675	[197]
7	Fujitsu VPP5000	1999	303	9600	1405	1881	[223]
8	NEC SX-5	2000	250	8000	1100	1200	[197]
9	Hitachi SR-8000	2001	250	1000	68	174	[223]
10	Japan Earth Simulator	2002	500	8000	1750	2375	[223]
11	NEC SX-6	2002	500	8000	620	620	[239]
12	UCB VIRAM	2003	200	1600	91	511	[239]

Table A.2: **Historical SpMV data: vector processors**. Data points taken from NAS CG results are cited accordingly [23]. "Year" indicates year of processor production at the specified clock rate. The year is prefixed by the month (*e.g.*, "11/1992") if known; otherwise, we take the month to be June ("6") when plotting the data points.

Appendix B

Experimental Setup

B.1 Machines, compilers, libraries, and tools

All experimental evaluations are conducted on machines based on the microprocessors shown in Tables B.1–B.2. This table summarizes each platform's hardware and compiler configurations, and performance results on key dense matrix kernels.

The dense kernels shown are double-precision dense matrix-matrix multiply (DGEMM), double-precision dense matrix-vector multiply (DGEMV), and dense band matrix-vector multiply (DGBMV). The matrix dimension is chosen to be the smallest $n = k \cdot 1000$ such that $n^2 > C_\kappa$, where k is an integer and C_κ is the size of the largest cache (in doubles).

Latency estimates are obtained as discussed in Section 4.2.1 using the memory system microbenchmarks due to Saavedra-Barrera [269] and Snavely [282].

We also indicate whether PAPI v2.3.4 was available for each platform at the time these experiments were performed.

Throughout this dissertation, we assume IEEE double-precision (64-bit) floating point values and 32-bit integers.

B.2 Matrix benchmark suite

Most of the experiments in this dissertation were conducted using the test matrix benchmark suite used by Im [164]. Tables B.3–B.5 summarizes the size of each matrix and the application area in which each matrix arises. Matrices are available from either of the collections at NIST (MatrixMarket [53]) and the University of Florida [90].

	Sun Ultra 2i	Sun Ultra 3	Intel Pentium III	Intel Pentium III-M
MHz	333	900	500	800
OS	Solaris v8	Solaris v8	Linux	Linux
Compiler	Sun cc v6	Sun cc v6	Intel C v6.0	Intel C v7.0
PAPI v2.3.4?	yes	no	yes	no
Peak Mflop/s	667	1800	500	800
DGEMM Mflop/s	425 ATLAS	1600 Sun PerfLib	330 ITXGEMM v1.1	640 Goto
DGEMV Mflop/s	58 ATLAS	322 Sun PerfLib	96 Intel MKL v5.2	147 Intel MKL v5.2
DGBMV Mflop/s	48 ATLAS	48 ATLAS	23 Intel MKL v5.2	102 Ref
Peak MB/s	664	4800	680	915
STREAM Triad MB/s	215	504	350	570
No. FP regs (double)	16	32	8	8
L1 size Line size Associativity Latency	16 KB 16 B direct 2 cy	64 KB 32 B 4-way 1–2 cy	16 KB 32 B 4-way 2 cy	16 KB 32 B 4-way 1–2 cy
L2 size Line size Associativity Latency	2 MB 64 B 2-way 6–7 cy	8 MB 64 B 2-way 5–11 cy	512 KB 32 B 4-way 18 cy	256 KB 32 B 4-way 5–18 cy
TLB entries Page size	64 8 KB	512 8 KB	64 4 KB	64 4 KB
Memory size Latency	256 MB 38–66 cy	4 GB 28–200 cy	128 MB 25–60 cy	256 MB 40–60 cy
β_s MB/s	333	1477	516	595

Table B.1: **Hardware platforms (1/2)**. machine configurations, compilers, and compiler optimization flags used

The matrices in Tables B.3–B.5 are arranged in roughly four groups. Matrix 1 is a dense matrix stored in sparse format; Matrices 2–17 arise in finite element method (FEM) applications; 18–39 come from assorted applications (including chemical process engineering, oil reservoir modeling, circuits, and finance); 40–44 are linear programming examples.

The largest cache on some machines (notably, the L_3 cache on the Power4) is large enough to contain some of the matrices. To avoid inflated findings, for each platform we report performance results only on the subset of out-of-cache matrices. Matrices that

	IBM Power3	IBM Power4	Intel Itanium 1	Intel Itanium 2
MHz	375	1300	800	900
OS	AIX	AIX	Linux	Linux
Compiler	IBM xlc v5	IBM xlc v6	Intel C v6.0	Intel C v7.0
PAPI v2.3.4?	yes	no	yes	yes
Peak Mflop/s	1500	5200	3200	3600
DGEMM Mflop/s	1300 ESSL	3500 ESSL	2200 ATLAS	3500 Goto
DGEMV Mflop/s	260 ESSL	900 ESSL	310 Intel MKL v5.2	1330 Goto
DGBMV Mflop/s	230 ESSL	606 Ref	167 ATLAS	463 Ref
Peak MB/s	1600	11000	2100	6400
STREAM Triad MB/s	748	2286	1103	4028
No. FP regs (double)	32	32	128	128
L1 size Line size Associativity Latency	64 KB 128 B 128-way 0.5–2 cy	32 KB 128 B 2-way 0.7–1.4 cy	16 KB 32 B 4-way 0.5–2 cy	32 KB 64 B 4-way 0.34–1 cy
L2 size Line size Associativity Latency	8 MB 128 B direct 9 cy	1.5 MB 128 B 8-way 4.4–91 cy	96 KB 64 B 6-way 0.75–9 cy	256 KB 128 B 8-way 0.5–4 cy
L3 size Line size Associativity Latency	—	16 MB 512 B 8-way 21.5–1243 cy	2 MB 64 B 2-way 21–24 cy	1.5 MB 128 B 8-way 3–20 cy
TLB entries Page size	256 4 KB	1024 4 KB	96 16 KB	128 16 KB
Memory size Latency	1 GB 35–139 cy	4 GB 60–10000 cy	1 GB 36–85 cy	2 GB 11–60 cy
β_s MB/s	1129	3998	1288	6227

Table B.2: **Hardware platforms (2/2)**. Machine configurations, compilers, and compiler optimization flags used in this dissertation. Additional material may be found in various processor manuals and papers for the Power3 [15], Itanium 2 [74].

fit within the largest cache on all platforms shown in Tables B.1–B.2 have been omitted. Figures always use the numbering scheme shown in Tables B.3–B.5 when referring to these matrices.

Chapter 5 uses of a number of supplemental matrices, listed in Table B.6.

	Name and Application area	Dimension	Non-zeros	Nnz per row	Max. active elems.
1	`dense2000` Dense matrix	2000	4000000	2000	2000
2	`raefsky3` Fluid structure interaction	21200	1488768	70.2	1448
3	`olafu` Accuracy problem	16146	1015156	62.9	1116
4	`bcsstk35` Stiff matrix automobile frame	30237	1450163	48.0	2359
5	`venkat01` Flow simulation	62424	1717792	27.5	8340
6	`crystk02` FEM crystal free vibration	13965	968583	69.4	921
7	`crystk03` FEM crystal free vibration	24696	1751178	70.9	1143
8	`nasasrb` Shuttle rocket booster	54870	2677324	48.8	1734
9	`3dtube` 3-D pressure tube	45330	3213332	70.9	4749

Table B.3: **Sparsity matrix benchmark suite: Matrices 1–9 (finite element matrices)**. For an explanation of the last column, see Section B.2.1. For additional characterizations of the non-zero structure of these matrices, see Chapter 5 and Appendix F.

B.2.1 Active elements

This dissertation does not specifically explore the technique of *cache blocking*, though we include a summary of this technique in Section 5.3 [164, 165, 235]. Indeed, none of the matrices considered in this dissertation benefit from cache blocking [235]. To see roughly why, we show the *minimum number of active source vector elements* for each matrix in the last column in each of Tables B.3–B.6, where we define this quantity as follows.

Suppose the matrix A is stored in compressed sparse row (CSR) format. We define $\text{active}(A, i)$ to be the *number of active source vector elements* at row $i > 0$ of A to be the number of elements of the source vector x that are loaded in a row $i' < i$ and also loaded in some row $l' \geq i$. We define the *maximum* number of active elements to be $\max_i \text{active}(A, i)$, and show this quantity in the last column of Tables B.3–B.6.

The maximum number of active elements is an intuitive measure of source vector locality that is matrix-dependent but machine-independent. We can interpret this quantity

	Name and Application area	Dimension	Non-zeros	Nnz per row	Max. active elems.
10	`ct20stif` CT20 Engine block	52329	2698463	51.6	14570
11	`bai` Airfoil eigenvalue calculation	23560	484256	20.6	608
12	`raefsky4` Buckling problem	19779	1328611	67.2	2446
13	`ex11` 3D steady flow caculation	16614	1096948	66.0	1031
15	`vavasis3` 2D PDE problem	41092	1683902	41.0	3731
17	`rim` FEM fluid mechanics problem	22560	1014951	45.0	448

Table B.4: **Sparsity matrix benchmark suite: Matrices 10–17 (finite element matrices)**. For an explanation of the last column, see Section B.2.1. For additional characterizations of the non-zero structure of these matrices, see Chapter 5 and Appendix F.

as being the minimum size (in words) of a fully associative cache needed to guarantee that we incur only compulsory misses in performing a row-oriented traversal of A. Inspecting Tables B.3–B.6, only in the case of Matrix `2anova2` is this quantity equivalent to more than 1 MB of storage. Therefore, we might expect that only on this matrix will cache-level blocking lead to performance improvements over a CSR implementation of sparse matrix-vector multiply (SpMV). However, examining the non-zero structure of Matrix `2anova2` reveals that the number of active elements is relatively high because the first row of A is full. Excluding this row, the maximum number of active elements drops to 1023. Thus, for none of these matrices would we expect large performance increases due to cache blocking based on the maximum number of active elements.

B.3 Measurement methodology

We use the PAPI v2.3.4 library for access to hardware counters on all platforms [60]; we use the cycle counters as timers. Counter values reported are the median of 25 consecutive trials. The standard deviation of these trials is typically less than 1% of the median.

If PAPI is not available, we use the highest resolution timer available. We use the IPM/RPRF timing package, which detects this timer automatically on many platforms

	Name and Application area	Dimension	Non-zeros	Nnz per row	Max. active elems.
18	`memplus` Circuit simulation	17758	126150	7.1	17706
19	`gemat11` Power flow	4929	33185	6.7	1123
20	`lhr10` Light hydrocarbon recovery	10672	232633	21.8	489
21	`goodwin` Fluid mechanics problem	7320	324784	44.4	394
22	`bayer02` Chemical process simulation	13935	63679	4.6	286
23	`bayer10` Chemical process simulation	13436	94926	7.1	1170
24	`coater2` Simulation of coating flows	9540	207308	21.7	508
25	`finan512` Financial portfolio optimization	74752	596992	8.0	45625
26	`onetone2` Harmonic balance method	36057	227628	6.3	662
27	`pwt` Structural engineering problem	36519	326107	8.9	14242
28	`vibrobox` Structure of vibroacoustic problem	12328	342828	27.8	10124
29	`wang4` Semiconductor device simulation	26068	177196	6.8	1800
36	`shyy161` Viscous flow calculation	76480	329762	4.3	320
37	`wang3` Semiconductor device simulation	26064	177168	6.8	1800
40	`gupta1` Linear programming matrix	31802	2164210	68.1	20369
41	`lpcreb` Linear Programming problem	9648×77137	260785	27.0	68354
42	`lpcred` Linear Programming problem	8926×73948	246614	27.6	66307
43	`lpfit2p` Linear Programming problem	3000×13525	50284	16.8	25
44	`lpnug20` Linear Programming problem	15240×72600	304800	20.0	72600

Table B.5: **Sparsity matrix benchmark suite: Matrices 18–44 (matrices from assorted applications and linear programming problems)**. For an explanation of the last column, see Section B.2.1. For additional characterizations of the non-zero structure of these matrices, see Chapter 5 and Appendix F.

	Name and Application Area	Dimension	Non-zeros	Nnz per row	Max. active elems.
A	`bmw7st_1` Car body analysis [1]	141347	7339667	51.9	34999
B	`cop20km` Accelerator cavity design [129]	121192	4826864	39.8	121192
C	`pwtk` Pressurized wind tunnel [90]	217918	11634424	53.4	16439
D	`rma10` Charleston Harbor [90]	46835	2374001	50.7	19269
E	`s3dkq4m2` Cylindrical shell [53]	90449	4820891	53.3	1224
F	`2anova2` Statistical analysis [327]	254284	1261516	5.0	254282
G	`3optprice` Option pricing (finance) [133]	59319	1081899	18.2	3120
H	`marca_tcomm` Telephone exchange [248]	547824	2733595	5.0	452
I	`mc2depi` Ridler-Rowe epidemic [248]	525825	2100225	4.0	770
S1	`dsq_S_625` 2D 5-pt stencil	388129	1938153	5.0	1246
S2	`sten_r2d9` 2D 9-pt stencil	250000	2244004	9.0	1002
S3	`sten_r3d27` 3D 27-pt stencil	74088	1906624	25.7	3614

Table B.6: **Supplemental matrices**. Summary of supplemental matrices used in Chapter 5. For an explanation of the last column, see Section B.2.1. For additional characterizations of the non-zero structure of these matrices, see Chapter 5 and Appendix F.

[46, 20, 21].

For SpMV, reported performance in Mflop/s always uses "ideal" flops. That is, if a transformation of the matrix requires filling in explicit zeros (as with register blocking, described in Section 3.1), arithmetic with these extra zeros are *not* counted as flops when determining performance.

Appendix C

Baseline Sparse Format Data

Tables C.1–C.8 show the raw measured performance for the matrices in Tables B.3–B.5. The following formats are compared:

- compressed sparse row (CSR) format
- compressed sparse column (CSC) format
- modified sparse row (MSR) format
- diagonal (DIAG) format
- jagged diagonal (JAD) format
- ELLPACK/ITPACK (ELL) format

We show the best performance of either Fortran or hand-translated C implementations of the sparse matrix-vector multiply (SpMV) routines available in the SPARSKIT library [267].

Matrix	Performance (Mflop/s)					
No.	CSR	CSC	MSR	DIAG	ELL	JAD
1	34	22	33	26	21	22
2	34	26	33	25	16	19
3	34	25	33	30	13	17
4	33	24	32	31	3.9	18
5	31	25	29	30	9.5	14
6	34	24	33	28	16	18
7	34	26	34	29	16	18
8	34	26	32	33	2.9	16
9	34	26	34	31	—	—
10	34	24	33	31	4.2	16
11	29	23	27	19	17	16
12	33	24	33	31	6.6	17
13	34	24	33	31	12	17
15	31	23	29	31	—	—
17	32	25	32	33	7.0	17
21	29	22	28	29	6.2	16
25	21	21	20	20	2.0	9.9
27	20	19	19	19	8.8	13
28	27	21	24	25	3.5	15
36	18	18	16	18	11	10
40	32	24	31	31	—	—
44	23	18	22	—	—	15

Table C.1: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Ultra 2i**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Matrix No.	Performance (Mflop/s)					
	CSR	CSC	MSR	DIAG	ELL	JAD
1	68	58	68	*83	52	66
2	53	48	52	53	33	42
4	53	46	52	49	7.6	41
5	49	44	48	45	21	36
7	52	46	49	50	31	39
8	49	44	48	48	6.1	34
9	49	44	48	47	1.0	36
10	49	44	46	46	8.4	36
12	55	49	53	55	13	42
13	60	54	59	59	27	42
15	47	44	46	47	0.6	41
40	46	41	46	46	0.3	40

Table C.2: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Ultra 3**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Matrix	Performance (Mflop/s)					
No.	CSR	CSC	MSR	DIAG	ELL	JAD
1	41	38	42	42	40	39
2	40	36	38	21	14	14
3	40	36	38	36	11	15
4	40	36	38	34	3.4	11
5	38	34	36	35	9.8	8.5
6	40	36	39	31	14	17
7	40	37	39	29	14	13
8	40	36	38	38	2.8	8.2
9	40	36	40	35	—	—
10	40	36	38	37	4.0	9.9
11	37	33	36	24	15	11
12	40	36	39	37	6.1	14
13	40	36	38	36	12	15
15	39	35	36	—	—	—
17	39	35	37	—	5.9	11
18	28	25	23	25	0.2	11
20	35	33	31	—	6.7	24
21	38	35	36	—	14	28
23	28	26	22	—	4.5	12
24	36	31	31	—	7.5	22
25	30	27	26	27	2.3	3.8
26	28	26	21	—	3.1	4.6
27	31	24	27	25	8.5	5.8
28	37	31	35	35	4.4	20
29	28	25	24	15	14	6.6
36	26	23	20	21	11	4.8
37	28	26	23	16	14	6.6
40	39	33	38	38	—	—
41	31	35	30	—	—	16
42	31	35	30	—	—	16
44	29	31	27	—	—	13

Table C.3: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Pentium III**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Matrix	Performance (Mflop/s)					
No.	CSR	CSC	MSR	DIAG	ELL	JAD
1	75	61	77	73	53	56
2	67	58	67	60	27	31
3	68	60	68	59	21	32
4	61	54	61	49	5.5	26
5	61	53	58	39	17	23
6	65	58	67	61	27	32
7	64	57	66	58	24	26
8	60	55	62	48	4.4	21
9	62	53	63	54	—	—
10	61	54	62	50	6.3	24
11	60	52	54	40	27	25
12	68	60	69	59	11	30
13	69	60	69	60	22	31
15	64	54	62	46	—	—
17	69	58	66	53	11	31
18	42	36	38	18	0.4	26
20	61	52	55	38	12	45
21	71	58	65	59	19	44
23	48	41	37	20	8.2	30
24	65	52	53	40	14	41
25	37	34	34	15	3.3	12
26	42	38	33	14	5.1	15
27	39	35	37	17	13	15
28	56	47	53	40	7.4	36
29	44	39	37	22	26	18
36	38	36	31	18	19	13
37	44	39	38	22	26	18
40	62	54	65	56	—	—
41	48	36	46	—	—	35
42	47	36	46	—	—	35
44	42	39	39	—	—	24

Table C.4: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Pentium III-M**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Matrix	Performance (Mflop/s)					
No.	CSR	CSC	MSR	DIAG	ELL	JAD
1	153	139	151	149	120	141
2	147	135	148	147	104	113
4	141	134	143	142	25	96
5	130	125	128	131	68	81
7	141	132	140	141	99	111
8	139	133	139	135	20	78
9	143	136	143	141	—	—
10	140	134	140	139	27	88
12	147	139	148	148	43	107
13	151	138	151	152	85	94
15	126	126	125	127	—	—
40	132	122	132	133	—	—

Table C.5: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Power3**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Matrix	Performance (Mflop/s)					
No.	CSR	CSC	MSR	DIAG	ELL	JAD
1	607	503	604	598	459	498
8	436	390	419	408	31	197
9	500	445	486	473	6.1	286
10	434	392	421	444	54	216
40	453	402	431	445	—	—

Table C.6: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Power4**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Matrix No.	Performance (Mflop/s)					
	CSR	CSC	MSR	DIAG	ELL	JAD
1	142	54	120	65	*196	194
2	133	54	113	70	157	150
3	132	54	112	120	118	137
4	129	53	108	114	36	130
5	121	52	102	116	77	87
6	133	54	112	96	154	150
7	134	54	113	97	155	146
8	130	53	109	126	24	104
9	133	54	112	114	4.1	135
10	130	53	110	120	35	114
11	117	51	97	78	*172	105
12	133	54	112	124	67	145
13	133	54	112	120	131	141
15	127	53	104	127	3.0	130
17	129	53	108	129	69	120
21	129	53	108	128	69	148
25	90	44	74	81	16	44
27	96	46	76	88	92	64
28	122	52	102	118	40	118
36	67	38	53	56	*85	37
40	131	54	109	128	0.0	134
44	112	38	90	—	125	81

Table C.7: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Itanium 1**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Matrix No.	Performance (Mflop/s)					
	CSR	CSC	MSR	DIAG	ELL	JAD
1	296	155	291	132	279	265
2	275	151	272	134	242	247
3	275	151	273	240	203	254
4	248	145	258	221	59	230
5	251	145	246	225	142	174
6	262	149	268	186	243	252
7	260	149	268	186	247	250
8	247	143	253	239	44	207
9	261	148	267	225	7.6	242
10	250	145	258	233	62	215
11	241	142	236	142	274	196
12	276	151	272	251	107	254
13	277	151	270	239	211	258
15	260	148	249	249	5.4	216
17	269	149	262	258	113	236
21	273	149	264	258	114	244
25	130	105	145	116	27	84
27	141	108	158	130	132	116
28	222	136	236	216	61	209
36	128	109	120	99	*157	83
40	250	146	260	251	2.1	231
44	230	109	210	—	—	175

Table C.8: **Comparison of sparse matrix-vector multiply performance using the baseline formats: Itanium 2**. Missing data indicates that there was not sufficient memory to convert the matrix to the corresponding format. Performance values more than 1.2× faster than CSR are shwon in red and marked by an asterisk.

Appendix D

Data on the Sparsity Heuristic

	Matrix 9 $r_{\text{opt}}\times c_{\text{opt}}$= 3×3 (54 Mflop/s) (1×1 → 35 Mflop/s)			Matrix 10 $r_{\text{opt}}\times c_{\text{opt}}$= 2×1 (40 Mflop/s) (1×1 → 34 Mflop/s)			Matrix 40 $r_{\text{opt}}\times c_{\text{opt}}$= 1×1 (33 Mflop/s) (1×1 → 33 Mflop/s)		
σ	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs
0.0006	3×3	54	0.1	4×2	37	0.2	1×1	33	0.0
0.0007	3×3	54	0.2	1×2	37	0.2	1×1	33	0.1
0.0008	3×3	54	0.2	8×2	30	0.2	1×1	33	0.2
0.0009	3×3	54	0.2	6×2	33	0.2	1×1	33	1.2
0.001	3×3	54	0.2	6×2	33	0.2	1×1	33	0.2
0.002	3×3	54	0.5	4×2	37	0.5	1×1	33	0.4
0.0025	3×3	54	0.6	2×2	40	0.6	1×1	33	0.6
0.003	3×3	54	0.7	2×2	40	0.7	1×1	33	0.4
0.004	3×3	54	0.9	2×2	40	0.9	1×1	33	0.7
0.005	3×3	54	1.2	8×2	30	1.2	1×1	33	0.7
0.006	3×3	54	1.4	2×2	40	1.4	1×1	33	0.9
0.007	3×3	54	1.7	2×2	40	1.7	1×1	33	0.6
0.0075	3×3	54	1.8	2×2	40	1.7	1×1	33	2.7
0.008	3×3	54	1.9	2×2	40	1.9	1×1	33	0.7
0.009	3×3	54	2.3	2×2	40	2.2	1×1	33	3.5
0.01	3×3	54	2.3	2×2	40	2.3	1×1	33	2.7
0.02	3×3	54	4.8	2×2	40	4.6	1×1	33	3.3
0.025	3×3	54	5.8	2×2	40	5.8	1×1	33	3.8
0.03	3×3	54	7.0	2×2	40	6.9	1×1	33	6.9
0.04	3×3	54	9.4	2×2	40	9.4	1×1	33	10.0
0.05	3×3	54	11.9	2×2	40	11.8	1×1	33	12.2
0.06	3×3	54	14.0	2×2	40	14.0	1×1	33	13.6
0.07	3×3	54	16.4	2×2	40	16.9	1×1	33	19.6
0.08	3×3	54	18.8	2×2	40	18.7	1×1	33	21.0
0.09	3×3	54	21.0	2×2	40	21.0	1×1	33	22.6
0.1	3×3	54	23.8	2×2	40	23.2	1×1	33	21.0
0.2	3×3	54	47.0	2×2	40	46.6	1×1	33	41.2
0.3	3×3	54	70.3	2×2	40	69.9	1×1	33	66.2
0.4	3×3	54	93.6	2×2	40	93.1	1×1	33	85.3
0.5	3×3	54	118	2×2	40	117	1×1	33	113
0.6	3×3	54	141	2×2	40	140	1×1	33	128
0.7	3×3	54	165	2×2	40	166	1×1	33	154
0.8	3×3	54	188	2×2	40	187	1×1	33	177
0.9	3×3	54	211	2×2	40	210	1×1	33	198
1	3×3	54	235	2×2	40	233	1×1	33	220

Table D.1: **Heuristic accuracy as the matrix sampling fraction (σ) varies: Matrices 9, 10, and 40 on Ultra 2i**. We show the block size $r_{\text{h}}\times c_{\text{h}}$ chosen by the heuristic, the resulting performance P_h (in Mflop/s), and the time to execute the heuristic in units of the time to execute one unblocked SpMV.

	Matrix 9 $r_{\text{opt}}\times c_{\text{opt}}$= 3×3 (102 Mflop/s) (1×1 → 62 Mflop/s)			Matrix 10 $r_{\text{opt}}\times c_{\text{opt}}$= 2×2 (77 Mflop/s) (1×1 → 61 Mflop/s)			Matrix 40 $r_{\text{opt}}\times c_{\text{opt}}$= 1×1 (62 Mflop/s) (1×1 → 62 Mflop/s)		
σ	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs
0.0006	3×3	102	0.2	2×2	77	0.2	2×1	45	0.1
0.0007	3×3	102	0.2	2×2	77	0.2	2×1	45	0.1
0.0008	3×3	102	0.3	2×2	77	0.2	2×1	45	0.4
0.0009	3×3	102	0.3	2×2	77	0.3	2×1	45	0.2
0.001	3×3	102	0.3	2×2	77	0.3	1×2	54	0.3
0.002	3×3	102	0.6	2×2	77	0.6	1×1	62	0.4
0.0025	3×3	102	0.7	2×2	77	0.7	2×1	45	0.4
0.003	3×3	102	0.9	2×2	77	0.9	1×1	62	0.9
0.004	3×3	102	1.2	2×2	77	1.2	1×1	62	1.4
0.005	3×3	102	1.5	2×2	77	1.5	1×1	62	1.4
0.006	3×3	102	1.8	2×2	77	1.9	2×1	45	1.6
0.007	3×3	102	2.1	2×2	77	2.1	1×1	62	2.4
0.0075	3×3	102	2.2	2×2	77	2.2	1×1	62	1.8
0.008	3×3	102	2.4	2×2	77	2.4	1×1	62	3.3
0.009	3×3	102	2.7	2×2	77	2.7	1×1	62	2.6
0.01	3×3	102	3.3	2×2	77	3.0	1×1	62	4.4
0.02	3×3	102	6.1	2×2	77	5.9	1×1	62	7.3
0.025	3×3	102	7.9	2×2	77	7.4	1×1	62	8.5
0.03	3×3	102	9.0	2×2	77	8.9	1×1	62	9.1
0.04	3×3	102	11.8	2×2	77	11.8	1×1	62	14.2
0.05	3×3	102	15.2	2×2	77	14.8	1×1	62	18.0
0.06	3×3	102	18.9	2×2	77	17.8	1×1	62	20.1
0.07	3×3	102	20.9	2×2	77	20.9	1×1	62	23.2
0.08	3×3	102	24.3	2×2	77	23.6	1×1	62	28.3
0.09	3×3	102	27.3	2×2	77	26.8	1×1	62	30.3
0.1	3×3	102	30.1	2×2	77	29.8	1×1	62	35.5
0.2	3×3	102	59.5	2×2	77	59.5	1×1	62	65.3
0.25	3×3	102	74.7	2×2	77	75.2	1×1	62	90.5
0.3	3×3	102	89.5	2×2	77	88.6	1×1	62	106
0.4	3×3	102	120	2×2	77	119	1×1	62	147
0.5	3×3	102	149	2×2	77	150	1×1	62	178
0.6	3×3	102	178	2×2	77	179	1×1	62	209
0.7	3×3	102	209	2×2	77	209	1×1	62	276
0.75	3×3	102	224	2×2	77	224	1×1	62	285
0.8	3×3	102	239	2×2	77	236	1×1	62	294
0.9	3×3	102	268	2×2	77	268	1×1	62	313
1	3×3	102	298	2×2	77	298	1×1	62	328

Table D.2: **Heuristic accuracy as the matrix sampling fraction (σ) varies: Matrices 9, 10, and 40 on Pentium III-M**. We show the block size $r_{\text{h}}\times c_{\text{h}}$ chosen by the heuristic, the resulting performance P_h (in Mflop/s), and the time to execute the heuristic in units of the time to execute one unblocked SpMV.

	Matrix 9 $r_{\text{opt}}\times c_{\text{opt}}$= 3×3 (705 Mflop/s) (1×1 → 500 Mflop/s)			Matrix 10 $r_{\text{opt}}\times c_{\text{opt}}$= 2×1 (549 Mflop/s) (1×1 → 434 Mflop/s)			Matrix 40 $r_{\text{opt}}\times c_{\text{opt}}$= 1×1 (453 Mflop/s) (1×1 → 453 Mflop/s)		
σ	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs
0.0006	3×3	705	0.5	4×1	526	0.5	1×1	453	0.2
0.0007	3×1	693	0.6	1×1	434	0.5	1×1	453	0.3
0.0008	3×3	705	0.6	3×1	540	0.6	1×1	453	0.6
0.0009	3×3	705	0.7	2×1	549	0.6	1×1	453	3.8
0.001	3×3	705	0.8	1×1	434	0.6	1×1	453	0.6
0.002	3×3	705	1.5	4×1	526	1.4	1×1	453	1.4
0.0025	3×3	705	1.9	1×1	434	1.8	1×1	453	1.9
0.003	3×3	705	2.2	2×1	549	2.0	1×1	453	1.3
0.004	3×3	705	3.0	1×1	434	2.6	1×1	453	2.3
0.005	3×3	705	3.8	2×1	549	3.4	1×1	453	2.4
0.006	3×3	705	4.4	2×1	549	3.9	1×1	453	3.0
0.007	3×3	705	5.3	2×1	549	4.7	1×1	453	2.1
0.0075	3×3	705	5.6	2×1	549	4.9	1×1	453	8.7
0.008	3×3	705	5.9	2×1	549	5.4	1×1	453	2.4
0.009	3×3	705	7.1	2×1	549	6.1	1×1	453	11.3
0.01	3×3	705	7.4	2×1	549	6.5	1×1	453	8.6
0.02	3×3	705	15.1	2×1	549	13.0	1×1	453	10.8
0.025	3×3	705	18.3	2×1	549	16.4	1×1	453	12.6
0.03	3×3	705	22.1	2×1	549	19.3	1×1	453	22.5
0.04	3×3	705	29.6	2×1	549	26.2	1×1	453	32.5
0.05	3×3	705	37.1	2×1	549	32.6	1×1	453	39.3
0.06	3×3	705	44.2	2×1	549	39.3	1×1	453	43.9
0.07	3×3	705	51.9	2×1	549	45.5	1×1	453	63.7
0.08	3×3	705	59.4	2×1	549	52.5	1×1	453	68.3
0.09	3×3	705	66.2	2×1	549	59.0	1×1	453	74.0
0.1	3×3	705	75.1	2×1	549	64.9	1×1	453	68.6
0.2	3×3	705	148	2×1	549	131	1×1	453	135
0.25	3×3	705	185	2×1	549	163	1×1	453	165
0.3	3×3	705	223	2×1	549	196	1×1	453	217
0.4	3×3	705	297	2×1	549	261	1×1	453	283
0.5	3×3	705	371	2×1	549	327	1×1	453	356
0.6	3×3	705	444	2×1	549	393	1×1	453	434
0.7	3×3	705	519	2×1	549	457	1×1	453	496
0.75	3×3	705	555	2×1	549	490	1×1	453	541
0.8	3×3	705	592	2×1	549	524	1×1	453	560
0.9	3×3	705	666	2×1	549	588	1×1	453	652
1	3×3	705	739	2×1	549	653	1×1	453	718

Table D.3: **Heuristic accuracy as the matrix sampling fraction (σ) varies: Matrices 9, 10, and 40 on Power4**. We show the block size $r_{\text{h}}\times c_{\text{h}}$ chosen by the heuristic, the resulting performance P_h (in Mflop/s), and the time to execute the heuristic in units of the time to execute one unblocked SpMV.

	Matrix 9 $r_{\text{opt}}\times c_{\text{opt}}$= 6×1 (720 Mflop/s) (1×1 → 261 Mflop/s)			Matrix 10 $r_{\text{opt}}\times c_{\text{opt}}$= 4×2 (698 Mflop/s) (1×1 → 250 Mflop/s)			Matrix 40 $r_{\text{opt}}\times c_{\text{opt}}$= 4×1 (327 Mflop/s) (1×1 → 250 Mflop/s)		
σ	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs	$r_{\text{h}}\times c_{\text{h}}$	P_h	Time 1×1 SpMVs
0.0006	6×1	720	0.1	4×2	698	0.1	1×1	249	0.3
0.0007	3×2	702	1.1	8×1	578	0.1	2×2	290	0.0
0.0008	3×2	702	0.1	4×2	698	0.1	2×2	290	0.0
0.0009	3×2	702	0.2	4×2	698	0.1	8×1	250	0.5
0.001	3×2	702	0.2	4×2	698	0.2	3×2	299	0.3
0.002	3×2	702	0.3	4×2	698	0.3	1×1	249	0.1
0.0025	3×2	702	0.4	4×2	698	0.4	1×1	249	0.3
0.003	3×2	702	0.8	4×2	698	2.5	1×1	249	0.2
0.004	3×2	702	0.7	4×2	698	0.6	6×1	287	3.4
0.005	3×2	702	1.9	4×2	698	1.6	6×1	287	0.9
0.006	3×2	702	1.0	4×2	698	2.6	6×1	287	1.2
0.007	3×2	702	1.1	4×2	698	1.1	1×1	249	0.6
0.0075	3×2	702	1.2	4×2	698	1.2	4×2	307	1.2
0.008	3×2	702	2.8	4×2	698	2.3	2×2	290	2.0
0.009	3×2	702	2.1	4×2	698	1.9	4×1	327	0.8
0.01	3×2	702	2.7	4×2	698	2.8	2×1	296	3.8
0.02	3×2	702	3.2	4×2	698	6.0	5×1	280	3.6
0.025	3×2	702	7.1	4×2	698	7.1	2×2	290	5.4
0.03	3×2	702	6.6	4×2	698	6.2	4×1	327	9.5
0.04	3×2	702	11.8	4×2	698	10.8	2×2	290	9.7
0.05	3×2	702	12.1	4×2	698	12.9	7×1	274	17.0
0.06	3×2	702	13.9	4×2	698	15.7	8×1	250	14.6
0.07	3×2	702	16.3	4×2	698	17.8	4×1	327	15.1
0.08	3×2	702	18.4	4×2	698	21.3	4×1	327	18.9
0.09	3×2	702	23.0	4×2	698	24.1	4×1	327	27.4
0.1	3×2	702	25.6	4×2	698	26.3	4×1	327	30.2
0.2	3×2	702	47.6	4×2	698	52.6	4×1	327	50.1
0.3	3×2	702	74.0	4×2	698	73.6	4×1	327	76.9
0.4	3×2	702	98.9	4×2	698	106	4×1	327	95.5
0.5	3×2	702	125	4×2	698	128	4×1	327	131
0.6	3×2	702	147	4×2	698	145	4×1	327	135
0.7	3×2	702	176	4×2	698	182	4×1	327	177
0.8	3×2	702	202	4×2	698	195	4×1	327	200
0.9	3×2	702	230	4×2	698	221	4×1	327	218
1	3×2	702	255	4×2	698	251	4×1	327	254

Table D.4: **Heuristic accuracy as the matrix sampling fraction (σ) varies: Matrices 9, 10, and 40 on Itanium 2**. We show the block size $r_{\text{h}}\times c_{\text{h}}$ chosen by the heuristic, the resulting performance P_h (in Mflop/s), and the time to execute the heuristic in units of the time to execute one unblocked SpMV.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s
1	4×8	1.00	72	4×8	1.00	72	8×8	1.00	69
2	8×8	1.00	63	8×2	1.00	58	8×8	1.00	63
3	6×6	1.12	55	6×2	1.12	49	6×6	1.12	55
4	6×2	1.13	54	6×2	1.13	54	3×3	1.06	52
5	4×4	1.00	51	4×4	1.00	51	4×4	1.00	51
6	3×3	1.00	56	3×3	1.00	56	3×3	1.00	56
7	3×3	1.00	56	3×3	1.00	56	3×3	1.00	56
8	6×6	1.15	52	6×2	1.13	49	6×3	1.13	49
9	3×3	1.02	54	3×3	1.02	54	3×3	1.02	54
10	2×1	1.10	40	2×2	1.21	40	2×2	1.21	40
11	2×2	1.23	36	2×2	1.23	36	2×2	1.23	36
12	2×2	1.24	40	2×2	1.24	40	3×3	1.46	39
13	2×1	1.14	39	1×2	1.14	37	3×3	1.52	37
				[2×2	1.28	38]			
15	2×1	1.00	41	2×1	1.00	41	2×2	1.35	34
17	1×1	1.00	34	1×1	1.00	34	1×1	1.00	34
21	1×1	1.00	35	1×1	1.00	35	1×1	1.00	35
25	1×1	1.00	22	1×1	1.00	22	1×1	1.00	22
27	2×1	1.53	24	1×1	1.00	22	1×1	1.00	22
28	1×1	1.00	31	1×1	1.00	31	1×1	1.00	31
36	1×1	1.00	19	1×1	1.00	19	1×1	1.00	19
40	1×1	1.00	33	1×1	1.00	33	1×1	1.00	33
44	1×1	1.00	28	1×1	1.00	28	1×1	1.00	28

Table D.5: **Comparison of register blocking heuristics: Ultra 2i**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s
1	12×12	1.00	90	12×12	1.00	90	12×12	1.00	90
2	8×8	1.00	109	8×8	1.00	109	8×8	1.00	109
4	3×3	1.06	83	6×6	1.19	77	3×6	1.12	80
				[3×3	1.06	83]			
5	4×4	1.00	76	4×4	1.00	76	4×4	1.00	76
7	3×3	1.00	82	3×3	1.00	82	3×3	1.00	82
8	6×6	1.15	68	6×6	1.15	68	6×6	1.15	68
9	3×3	1.02	69	3×3	1.02	69	3×3	1.02	69
10	2×1	1.10	53	2×2	1.21	52	2×2	1.21	52
12	2×1	1.13	61	2×2	1.24	58	2×2	1.24	58
13	2×1	1.14	66	2×2	1.28	64	2×2	1.28	64
15	2×1	1.00	61	2×1	1.00	61	1×1	1.00	48
40	1×1	1.00	47	1×1	1.00	47	1×1	1.00	47

Table D.6: **Comparison of register blocking heuristics: Ultra 3**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s
1	5×2	1.00	101	5×2	1.00	101	6×6	1.00	98
2	4×8	1.00	95	4×8	1.00	95	4×4	1.00	85
3	6×2	1.12	86	6×2	1.12	86	3×3	1.12	83
4	3×3	1.06	86	3×3	1.06	86	3×3	1.06	86
5	4×2	1.00	85	4×2	1.00	85	4×4	1.00	77
6	3×3	1.00	92	3×3	1.00	92	3×3	1.00	92
7	3×3	1.00	92	3×3	1.00	92	3×3	1.00	92
8	6×2	1.13	85	3×3	1.11	82	3×3	1.11	82
				[6×2	1.13	85]			
9	3×3	1.02	89	3×3	1.02	89	3×3	1.02	89
10	4×2	1.45	64	2×2	1.21	61	3×2	1.38	61
				[4×2	1.45	64]			
11	2×2	1.23	56	2×2	1.23	56	2×2	1.23	56
12	3×3	1.46	65	4×2	1.48	64	3×3	1.46	65
				[3×3	1.46	65]			
13	3×3	1.52	62	5×2	1.66	59	3×3	1.52	62
				[3×3	1.52	62]			
15	2×1	1.00	57	2×1	1.00	57	2×2	1.35	51
17	4×1	1.75	48	4×1	1.75	48	2×3	2.06	41
18	2×1	1.36	34	2×2	1.79	32	2×2	1.79	32
				[1×1	1.00	31]			
20	1×2	1.17	46	1×2	1.17	46	1×3	1.35	43
21	3×1	1.59	47	3×1	1.59	47	3×2	2.07	42
				[4×1	1.77	47]			
23	2×1	1.46	33	1×1	1.00	32	1×1	1.00	32
24	2×1	1.52	39	1×1	1.00	38	1×1	1.00	38
25	1×1	1.00	31	1×1	1.00	31	1×1	1.00	31
26	1×1	1.00	29	1×1	1.00	29	1×1	1.00	29
27	2×1	1.53	34	1×1	1.00	33	1×1	1.00	33
28	1×1	1.00	38	1×1	1.00	38	1×1	1.00	38
29	2×2	1.98	30	1×1	1.00	30	2×2	1.98	30
36	1×1	1.00	27	1×1	1.00	27	1×1	1.00	27
37	2×2	1.98	31	1×1	1.00	30	2×2	1.98	31
40	1×1	1.00	39	1×1	1.00	39	1×1	1.00	39
41	1×1	1.00	33	1×1	1.00	33	1×1	1.00	33
42	1×1	1.00	33	1×1	1.00	33	1×1	1.00	33
44	1×1	1.00	30	1×1	1.00	30	1×1	1.00	30

Table D.7: **Comparison of register blocking heuristics: Pentium III**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s
1	3×11	1.00	143	3×11	1.00	143	10×10	1.00	122
2	2×8	1.00	120	8×8	1.00	114	8×8	1.00	114
3	6×1	1.10	113	6×6	1.12	104	6×6	1.12	104
4	3×3	1.06	106	3×3	1.06	106	3×3	1.06	106
5	4×2	1.00	105	4×4	1.00	101	4×4	1.00	101
6	3×3	1.00	107	3×3	1.00	107	3×3	1.00	107
7	3×3	1.00	106	3×3	1.00	106	3×3	1.00	106
8	6×6	1.15	96	3×6	1.13	92	6×6	1.15	96
9	3×3	1.02	102	3×3	1.02	102	3×3	1.02	102
10	2×2	1.21	77	2×2	1.21	77	2×2	1.21	77
11	2×2	1.23	76	2×2	1.23	76	2×2	1.23	76
12	2×2	1.24	83	2×2	1.24	83	2×2	1.24	83
13	3×2	1.40	84	2×2	1.28	82	3×2	1.40	84
15	2×1	1.00	79	2×1	1.00	79	2×2	1.35	71
17	1×1	1.00	69	1×1	1.00	69	1×1	1.00	69
18	2×1	1.36	45	2×2	1.79	42	2×2	1.79	42
				[1×1	1.00	42]			
19	2×1	1.01	55	2×1	1.01	55	2×1	1.01	55
20	1×1	1.00	61	1×2	1.17	61	1×2	1.17	61
21	1×1	1.00	71	1×1	1.00	71	1×1	1.00	71
22	1×1	1.00	40	1×1	1.00	40	1×1	1.00	40
23	1×1	1.00	48	1×1	1.00	48	1×1	1.00	48
				[2×1	1.46	48]			
24	1×1	1.00	65	1×1	1.00	65	1×1	1.00	65
25	1×1	1.00	37	1×1	1.00	37	1×1	1.00	37
26	1×1	1.00	42	1×1	1.00	42	1×1	1.00	42
27	2×1	1.53	39	1×1	1.00	39	1×1	1.00	39
28	1×1	1.00	56	1×1	1.00	56	1×1	1.00	56
29	1×1	1.00	44	1×1	1.00	44	1×1	1.00	44
36	1×1	1.00	38	1×1	1.00	38	1×1	1.00	38
37	1×1	1.00	44	1×1	1.00	44	1×1	1.00	44
40	1×1	1.00	62	1×1	1.00	62	1×1	1.00	62
41	1×1	1.00	48	1×1	1.00	48	1×1	1.00	48
42	1×1	1.00	47	1×1	1.00	47	1×1	1.00	47
43	1×1	1.00	50	2×1	1.52	50	2×5	2.14	50
				[1×2	1.52	50]			
44	1×1	1.00	42	1×1	1.00	42	1×1	1.00	42

Table D.8: **Comparison of register blocking heuristics: Pentium III-M**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s
1	4×4	1.00	256	4×4	1.00	256	4×4	1.00	256
2	4×4	1.00	197	4×4	1.00	197	4×4	1.00	197
4	3×3	1.06	164	3×3	1.06	164	3×3	1.06	164
5	4×2	1.00	161	4×4	1.00	153	4×4	1.00	153
7	3×3	1.00	170	3×3	1.00	170	3×3	1.00	170
8	3×3	1.11	145	3×1	1.06	140	3×3	1.11	145
				[6×2	1.13	140]			
9	3×3	1.02	159	3×3	1.02	159	3×3	1.02	159
10	1×1	1.00	137	1×1	1.00	137	1×1	1.00	137
12	1×1	1.00	143	1×1	1.00	143	1×1	1.00	143
13	1×1	1.00	148	1×1	1.00	148	1×1	1.00	148
15	2×1	1.00	145	2×1	1.00	145	2×1	1.00	145
40	1×1	1.00	132	1×1	1.00	132	1×1	1.00	132

Table D.9: **Comparison of register blocking heuristics: Power3**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s
1	8×1	1.00	766	8×1	1.00	766	12×12	1.00	789
8	6×2	1.13	581	3×1	1.06	547	3×3	1.11	545
				[6×3	1.13	542]			
9	3×3	1.02	705	3×3	1.02	705	3×3	1.02	705
10	2×1	1.10	549	2×1	1.10	549	2×2	1.21	497
40	1×1	1.00	453	1×1	1.00	453	1×1	1.00	453

Table D.10: **Comparison of register blocking heuristics: Power4**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s
1	4×1	1.00	250	4×1	1.00	250	2×2	1.00	239
2	4×1	1.00	229	4×1	1.00	229	2×2	1.00	215
3	3×1	1.06	208	3×1	1.06	208	2×2	1.12	184
4	3×1	1.04	204	3×1	1.04	204	2×2	1.07	177
5	4×1	1.00	188	4×1	1.00	188	2×2	1.00	176
6	3×1	1.00	220	3×1	1.00	220	2×2	1.23	165
7	3×1	1.00	221	3×1	1.00	221	2×2	1.22	167
8	3×1	1.06	201	3×1	1.06	201	2×2	1.10	167
9	3×1	1.01	217	3×1	1.01	217	2×2	1.25	165
10	3×1	1.27	169	2×1	1.10	162	2×2	1.21	158
11	4×1	1.70	119	2×2	1.23	118	2×2	1.23	118
12	3×1	1.24	182	2×1	1.13	176	2×2	1.24	164
13	3×1	1.26	179	2×1	1.14	174	2×2	1.28	160
15	2×1	1.00	164	2×1	1.00	164	2×2	1.35	138
17	3×1	1.59	141	2×1	1.36	139	1×1	1.00	136
21	3×1	1.59	142	3×1 [1×1	1.59 1.00	142 138]	1×1	1.00	138
25	1×1	1.00	64	1×1	1.00	64	1×1	1.00	64
27	3×1	1.94	78	1×1	1.00	67	1×1	1.00	67
28	1×1	1.00	121	1×1	1.00	121	1×1	1.00	121
36	3×1	2.31	54	1×1	1.00	50	1×1	1.00	50
40	1×1	1.00	128	1×1	1.00	128	1×1	1.00	128
44	1×1	1.00	75	1×1	1.00	75	1×1	1.00	75

Table D.11: **Comparison of register blocking heuristics: Itanium 1**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Matrix	Exhaustive best			Version 2 heuristic			Version 1 heuristic		
No.	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s	$r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s
1	4×2	1.00	1220	4×2	1.00	1220	2×2	1.00	748
2	4×2	1.00	1122	4×2	1.00	1122	2×2	1.00	693
3	6×1	1.10	946	6×1	1.10	946	2×2	1.12	598
4	4×2	1.23	807	4×2	1.23	807	2×2	1.07	566
				[6×1	1.10	790]			
5	4×2	1.00	1012	4×2	1.00	1012	2×2	1.00	595
6	4×2	1.46	740	3×2	1.12	719	2×2	1.23	536
7	4×2	1.45	734	3×2	1.11	711	2×2	1.22	535
8	6×1	1.12	778	6×1	1.12	778	2×2	1.10	542
9	6×1	1.34	720	3×2	1.12	702	2×2	1.25	506
10	4×2	1.45	698	4×2	1.45	698	2×2	1.21	503
11	4×2	1.70	620	4×2	1.70	620	2×2	1.23	496
12	4×2	1.48	774	4×2	1.48	774	2×2	1.24	558
13	4×2	1.54	749	4×2	1.54	749	2×2	1.28	535
15	4×1	1.78	514	2×1	1.00	490	2×2	1.35	490
17	4×1	1.75	536	7×1	2.09	531	2×2	1.79	388
				[6×1	1.98	536]			
20	4×2	2.33	477	4×2	2.33	477	2×2	2.06	314
21	6×1	2.01	537	6×1	2.01	537	2×2	1.81	387
				[7×1	2.10	537]			
24	4×1	2.28	393	4×1	2.28	393	2×2	2.08	312
25	4×1	3.13	226	2×1	1.71	210	2×2	2.45	202
26	4×1	2.91	271	4×1	2.91	271	2×2	2.96	191
27	4×1	2.39	273	4×1	2.39	273	2×2	2.34	216
28	4×1	2.83	314	4×1	2.83	314	2×2	2.54	253
36	4×2	3.53	252	4×2	3.53	252	2×2	2.31	206
40	4×1	2.39	327	2×1	1.56	296	2×2	2.33	290
				[4×1	2.39	327]			
41	2×1	1.78	252	2×1	1.78	252	2×2	2.52	252
42	2×1	1.80	248	1×1	1.00	243	2×2	2.61	248
				[2×1	1.80	248]			
44	1×1	1.00	230	1×1	1.00	230	1×1	1.00	230

Table D.12: **Comparison of register blocking heuristics: Itanium 2**. For each matrix (column 1), we show the best block size, fill, and performance using exhaustive search (columns 2–4), Version 2 heuristic (columns 5–7), and Version 1 heuristic (columns 8–10). The sampling fraction $\sigma = .01$. If the block size when $\sigma = 1$ differs from that when $\sigma = .01$, we show the results of using Version 2 heuristic with $\sigma = 1$ in square brackets.

Appendix E

Performance Bounds Data

Matrix No.	$r \times c$	Fill	Lower bound			Actual		
			Loads per non-zero	L_1 Misses per non-zero	L_2 Misses per non-zero	Loads per non-zero	L_1 Misses per non-zero	L_2 Misses per non-zero
1	6×8	1.00	1.19	0.51	0.13	1.19	0.60	0.13
2	8×8	1.00	1.16	0.52	0.13	1.16	0.55	0.14
3	6×6	1.12	1.34	0.58	0.15	1.36	0.63	0.15
4	6×2	1.13	1.42	0.61	0.15	1.45	0.66	0.15
5	4×4	1.00	1.36	0.55	0.14	1.37	0.66	0.14
6	3×3	1.00	1.46	0.54	0.14	1.47	0.62	0.14
7	3×3	1.00	1.46	0.54	0.14	1.47	0.62	0.14
8	6×6	1.15	1.37	0.60	0.15	1.40	0.66	0.15
9	3×3	1.02	1.49	0.55	0.14	1.50	0.64	0.14
10	2×1	1.10	2.18	0.71	0.18	2.25	0.79	0.18
11	2×2	1.23	2.12	0.75	0.19	2.26	0.87	0.19
12	2×2	1.24	2.08	0.72	0.18	2.21	0.79	0.18
13	2×1	1.14	2.24	0.73	0.18	2.32	0.83	0.18
15	2×1	1.00	2.04	0.65	0.16	2.05	0.90	0.17
17	1×1	1.00	3.04	0.78	0.19	3.07	0.86	0.20
21	1×1	1.00	3.05	0.78	0.19	3.07	0.87	0.19
25	1×1	1.00	3.25	0.91	0.23	3.38	1.04	0.24
27	2×1	1.53	2.96	1.08	0.27	3.29	1.18	0.28
28	1×1	1.00	3.07	0.79	0.20	3.11	1.02	0.20
36	1×1	1.00	3.46	1.04	0.26	3.70	1.16	0.26
40	1×1	1.00	3.03	0.77	0.19	3.05	1.18	0.20
44	1×1	1.00	3.10	0.91	0.23	3.26	1.63	0.24

Table E.1: **Comparison of analytic and measured load counts: Ultra 2i**.

Matrix No.	$r \times c$	Fill	Lower bound Loads per non-zero	Lower bound L_1 Misses per non-zero	Lower bound L_2 Misses per non-zero	Actual Loads per non-zero	Actual L_1 Misses per non-zero	Actual L_2 Misses per non-zero
1	2×10	1.00	1.55	0.26	0.26	1.56	0.38	0.26
2	4×2	1.00	1.39	0.27	0.27	1.44	0.28	0.28
3	6×2	1.12	1.40	0.30	0.30	1.46	0.31	0.30
4	3×3	1.06	1.54	0.29	0.29	1.64	0.30	0.29
5	4×2	1.00	1.42	0.28	0.28	1.54	0.33	0.29
6	3×3	1.00	1.46	0.27	0.27	1.52	0.30	0.27
7	3×3	1.00	1.46	0.27	0.27	1.52	0.30	0.27
8	6×2	1.13	1.41	0.30	0.30	1.49	0.32	0.31
9	3×3	1.02	1.49	0.28	0.28	1.55	0.30	0.29
10	4×2	1.45	1.91	0.40	0.40	2.09	0.42	0.40
11	2×2	1.23	2.12	0.37	0.37	2.52	0.42	0.38
12	3×3	1.46	1.98	0.39	0.39	2.19	0.41	0.40
13	3×3	1.52	2.04	0.41	0.41	2.28	0.43	0.41
14	3×2	1.47	2.11	0.42	0.42	2.46	0.44	0.43
15	2×1	1.00	2.04	0.33	0.33	2.18	0.48	0.33
16	4×1	1.43	2.08	0.42	0.42	2.29	0.46	0.42
17	4×1	1.75	2.46	0.50	0.50	2.72	0.52	0.50
18	2×1	1.36	2.75	0.50	0.50	3.64	0.63	0.55
20	1×2	1.17	2.84	0.39	0.39	3.50	0.40	0.40
21	4×1	1.77	2.49	0.51	0.51	2.76	0.52	0.51
23	2×1	1.46	2.91	0.54	0.54	3.97	0.55	0.54
24	2×1	1.52	2.84	0.50	0.50	3.37	0.52	0.50
25	1×1	1.00	3.25	0.45	0.45	4.62	0.50	0.48
26	1×1	1.00	3.32	0.47	0.47	5.10	0.57	0.47
27	2×1	1.53	2.96	0.54	0.54	3.89	0.58	0.57
28	1×1	1.00	3.07	0.40	0.40	3.46	0.50	0.41
29	2×2	1.98	3.20	0.64	0.64	4.54	0.71	0.64
36	1×1	1.00	3.46	0.52	0.52	5.67	0.53	0.52
37	2×2	1.98	3.20	0.64	0.64	4.55	0.71	0.64
40	1×1	1.00	3.03	0.38	0.38	3.19	0.62	0.40
41	1×1	1.00	3.07	0.46	0.46	3.42	0.65	0.59
42	1×1	1.00	3.07	0.46	0.46	3.41	0.64	0.59
44	1×1	1.00	3.10	0.45	0.45	3.91	0.79	0.61

Table E.2: **Comparison of analytic and measured load counts: Pentium III**.

Matrix No.	$r \times c$	Fill	Lower bound Loads per non-zero	Lower bound L_1 Misses per non-zero	Lower bound L_2 Misses per non-zero	Actual Loads per non-zero	Actual L_1 Misses per non-zero	Actual L_2 Misses per non-zero
1	4×4	1.00	1.31	0.06	0.06	1.31	0.07	0.07
2	4×4	1.00	1.33	0.07	0.07	1.35	0.10	0.07
4	3×3	1.06	1.54	0.07	0.07	1.61	0.11	0.08
5	4×2	1.00	1.42	0.07	0.07	1.47	0.08	0.07
7	3×3	1.00	1.46	0.07	0.07	1.50	0.10	0.07
8	3×3	1.11	1.60	0.08	0.08	1.68	0.08	0.08
9	3×3	1.02	1.49	0.07	0.07	1.53	0.08	0.07
10	1×1	1.00	3.04	0.10	0.10	3.14	0.10	0.10
12	1×1	1.00	3.03	0.10	0.10	3.10	0.14	0.10
13	1×1	1.00	3.03	0.10	0.10	3.10	0.14	0.10
15	2×1	1.00	2.04	0.08	0.08	2.10	0.14	0.08
40	1×1	1.00	3.03	0.10	0.10	3.10	0.18	0.10

Table E.3: **Comparison of analytic and measured load counts: Power3**.

Matrix No.	$r \times c$	Fill	Lower bound			Actual		
			Loads per non-zero	L_2 Misses per non-zero	L_3 Misses per non-zero	Loads per non-zero	L_2 Misses per non-zero	L_3 Misses per non-zero
1	8×1	1.00	1.25	0.13	0.13	1.38	0.13	0.13
2	4×1	1.00	1.52	0.14	0.14	1.65	0.15	0.14
3	3×1	1.06	1.77	0.16	0.16	1.97	0.16	0.16
4	3×1	1.04	1.75	0.16	0.16	1.94	0.16	0.16
5	4×1	1.00	1.55	0.15	0.15	1.68	0.16	0.15
6	3×1	1.00	1.69	0.15	0.15	1.86	0.15	0.15
7	3×1	1.00	1.69	0.15	0.15	1.86	0.15	0.15
8	3×1	1.06	1.77	0.16	0.16	1.97	0.16	0.16
9	3×1	1.01	1.70	0.15	0.15	1.87	0.16	0.15
10	3×1	1.27	2.05	0.19	0.19	2.35	0.20	0.19
11	2×2	1.23	2.12	0.19	0.19	2.40	0.19	0.19
12	3×1	1.24	2.00	0.18	0.18	2.29	0.19	0.18
13	3×1	1.26	2.04	0.19	0.19	2.34	0.19	0.19
15	2×1	1.00	2.04	0.16	0.16	2.17	0.21	0.16
17	3×1	1.59	2.48	0.24	0.24	2.94	0.24	0.24
21	1×1	1.00	3.05	0.19	0.19	3.21	0.20	0.19
25	1×1	1.00	3.25	0.23	0.23	3.55	0.24	0.24
27	3×1	1.94	3.07	0.31	0.31	3.73	0.33	0.32
28	1×1	1.00	3.07	0.20	0.20	3.25	0.23	0.20
36	3×1	2.31	3.72	0.40	0.40	4.63	0.40	0.40
40	1×1	1.00	3.03	0.19	0.19	3.18	0.27	0.19
44	1×1	1.00	3.10	0.23	0.23	3.31	0.32	0.25

Table E.4: **Comparison of analytic and measured load counts: Itanium 1**.

Matrix No.	$r \times c$	Fill	Lower bound: Loads per non-zero	Lower bound: L_2 Misses per non-zero	Lower bound: L_3 Misses per non-zero	Actual: Loads per non-zero	Actual: L_2 Misses per non-zero	Actual: L_3 Misses per non-zero
1	4×2	1.00	1.38	0.07	0.07	1.45	0.07	0.07
2	4×2	1.00	1.39	0.07	0.07	1.48	0.07	0.07
3	6×1	1.10	1.47	0.08	0.08	1.58	0.08	0.08
4	4×2	1.23	1.66	0.08	0.08	1.83	0.09	0.08
5	4×2	1.00	1.42	0.07	0.07	1.52	0.07	0.07
6	4×2	1.46	1.92	0.10	0.10	2.15	0.10	0.10
7	4×2	1.45	1.90	0.10	0.10	2.13	0.10	0.10
8	6×1	1.12	1.50	0.08	0.08	1.62	0.08	0.08
9	6×1	1.34	1.75	0.09	0.09	1.91	0.10	0.09
10	4×2	1.45	1.91	0.10	0.10	2.14	0.10	0.10
11	4×2	1.70	2.23	0.12	0.12	2.57	0.12	0.12
12	4×2	1.48	1.94	0.10	0.10	2.18	0.10	0.10
13	4×2	1.54	2.00	0.10	0.10	2.26	0.11	0.10
15	4×1	1.78	2.51	0.13	0.13	2.84	0.15	0.13
17	4×1	1.75	2.46	0.13	0.13	2.80	0.13	0.13
20	4×2	2.33	2.93	0.16	0.16	3.47	0.16	0.16
21	6×1	2.01	2.54	0.14	0.14	2.87	0.14	0.14
24	4×1	2.28	3.16	0.17	0.17	3.70	0.17	0.17
25	4×1	3.13	4.32	0.24	0.24	5.20	0.25	0.24
26	4×1	2.91	4.08	0.23	0.23	4.94	0.23	0.23
27	4×1	2.39	3.37	0.18	0.18	4.01	0.19	0.19
28	4×1	2.83	3.84	0.20	0.20	4.54	0.22	0.21
36	4×2	3.53	4.51	0.27	0.27	5.58	0.27	0.27
40	4×1	2.39	3.26	0.17	0.17	3.76	0.20	0.17
41	2×1	1.78	3.22	0.16	0.16	3.72	0.20	0.17
42	2×1	1.80	3.26	0.16	0.16	3.80	0.20	0.17
44	1×1	1.00	3.10	0.11	0.11	3.31	0.16	0.12

Table E.5: **Comparison of analytic and measured load counts: Itanium 2**.

Mat.	Best			1×1	*Upper bounds* Analytic			PAPI
No.	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Mflop/s	$r_{\text{up}} \times c_{\text{up}}$	Fill	Mflop/s	Mflop/s
1	6×8	1.00	73	34	12×12	1.00	81	78
2	8×8	1.00	57	34	8×8	1.00	78	75
3	6×6	1.12	49	34	6×6	1.12	69	66
4	6×2	1.13	50	33	6×3	1.12	67	65
5	4×4	1.00	48	31	4×4	1.00	71	66
6	3×3	1.00	50	34	3×3	1.00	70	67
7	3×3	1.00	53	34	3×3	1.00	71	67
8	6×6	1.15	50	34	6×6	1.15	67	64
9	3×3	1.02	52	34	3×3	1.02	69	65
10	2×1	1.10	39	34	2×2	1.21	53	50
11	2×2	1.23	32	29	2×2	1.23	50	47
12	2×2	1.24	38	33	2×2	1.24	52	50
13	2×1	1.14	37	34	2×2	1.28	51	48
15	2×1	1.00	40	31	2×1	1.00	56	51
17	1×1	1.00	32	32	1×1	1.00	43	42
21	1×1	1.00	29	29	1×1	1.00	43	42
25	1×1	1.00	21	21	1×1	1.00	38	35
27	2×1	1.53	22	20	1×1	1.00	39	36
28	1×1	1.00	27	27	1×1	1.00	42	39
36	1×1	1.00	18	18	1×1	1.00	34	32
40	1×1	1.00	32	32	1×1	1.00	44	39
44	1×1	1.00	23	23	1×1	1.00	39	32

Table E.6: **Comparison of register blocked SpMV performance to the upper bound model: Ultra 2i**.

Mat.	Best			1×1	Analytic upper bound		
No.	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Mflop/s	$r_{\text{up}} \times c_{\text{up}}$	Fill	Mflop/s
1	12×12	1.00	90	50	12×12	1.00	360
2	8×8	1.00	109	54	8×8	1.00	348
4	3×3	1.06	83	53	6×3	1.12	299
5	4×4	1.00	76	49	4×4	1.00	318
7	3×3	1.00	82	51	3×3	1.00	317
8	6×6	1.15	68	49	6×3	1.13	297
9	3×3	1.02	69	49	3×3	1.02	312
10	2×1	1.10	53	49	2×2	1.21	240
12	2×1	1.13	61	56	2×2	1.24	236
13	2×1	1.14	66	62	2×2	1.28	230
15	2×1	1.00	61	48	2×1	1.00	254
40	1×1	1.00	47	47	1×1	1.00	200

Table E.7: **Comparison of register blocked SpMV performance to the upper bound model: Ultra 3**.

Mat.	Best			1×1	*Upper bounds* Analytic			PAPI
No.	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	Mflop/s	$r_{\text{up}}\times c_{\text{up}}$	Fill	Mflop/s	Mflop/s
1	2×10	1.00	107	41	12×12	1.00	125	108
2	4×2	1.00	90	40	8×8	1.00	121	108
3	6×2	1.12	82	40	6×6	1.12	106	100
4	3×3	1.06	83	40	6×3	1.12	104	98
5	4×2	1.00	82	38	4×4	1.00	110	97
6	3×3	1.00	88	40	3×3	1.00	109	103
7	3×3	1.00	90	40	3×3	1.00	109	103
8	6×2	1.13	83	40	6×6	1.15	103	98
9	3×3	1.02	88	40	3×3	1.02	107	101
10	4×2	1.45	63	40	2×2	1.21	82	77
11	2×2	1.23	53	37	2×2	1.23	78	69
12	3×3	1.46	63	40	2×2	1.24	81	77
13	3×3	1.52	60	40	2×2	1.28	79	75
14	3×2	1.47	42	33	2×2	1.33	74	69
15	2×1	1.00	56	39	2×1	1.00	86	69
16	4×1	1.43	42	35	2×1	1.17	75	68
17	4×1	1.75	47	39	1×1	1.00	67	64
18	2×1	1.36	31	28	2×1	1.36	58	47
20	1×2	1.17	42	35	1×2	1.17	68	62
21	4×1	1.77	44	38	1×1	1.00	66	64
23	2×1	1.46	29	28	1×1	1.00	58	49
24	2×1	1.52	36	36	1×1	1.00	65	60
25	1×1	1.00	30	30	1×1	1.00	59	49
26	1×1	1.00	28	28	1×1	1.00	57	44
27	2×1	1.53	32	31	1×1	1.00	60	50
28	1×1	1.00	37	37	1×1	1.00	65	56
29	2×2	1.98	28	28	1×1	1.00	58	46
36	1×1	1.00	26	26	1×1	1.00	53	43
37	2×2	1.98	28	28	1×1	1.00	58	46
40	1×1	1.00	39	39	1×1	1.00	67	52
41	1×1	1.00	31	31	1×1	1.00	60	47
42	1×1	1.00	31	31	1×1	1.00	60	47
44	1×1	1.00	29	29	1×1	1.00	60	40

Table E.8: **Comparison of register blocked SpMV performance to the upper bound model: Pentium III**.

Mat.	Best			1×1	Analytic upper bound		
No.	$r_{opt}\times c_{opt}$	Fill	Mflop/s	Mflop/s	$r_{up}\times c_{up}$	Fill	Mflop/s
1	8×12	1.01	122	75	12×12	1.00	146
2	2×8	1.00	120	67	8×8	1.00	142
3	6×1	1.10	113	68	6×6	1.12	126
4	3×3	1.06	106	61	3×3	1.06	124
5	4×2	1.00	105	61	4×4	1.00	132
6	3×3	1.00	107	65	3×3	1.00	133
7	3×3	1.00	106	64	3×3	1.00	133
8	6×6	1.15	96	60	6×3	1.13	122
9	3×3	1.02	102	62	3×3	1.02	130
10	2×2	1.21	77	61	2×2	1.21	101
11	2×2	1.23	76	60	2×2	1.23	96
12	2×2	1.24	83	68	2×2	1.24	100
13	3×2	1.40	84	69	2×2	1.28	97
14	2×2	1.33	79	67	2×1	1.17	91
15	2×1	1.00	79	64	2×1	1.00	108
16	3×3	1.69	78	75	2×1	1.17	93
17	1×1	1.00	69	69	1×1	1.00	88
18	2×1	1.36	45	42	1×1	1.00	75
19	2×1	1.01	55	55	2×1	1.01	90
20	1×1	1.00	61	61	1×2	1.17	88
21	1×1	1.00	71	71	1×1	1.00	88
22	1×1	1.00	40	40	1×1	1.00	68
23	1×1	1.00	48	48	1×1	1.00	75
24	1×1	1.00	65	65	1×1	1.00	85
25	1×1	1.00	37	37	1×1	1.00	76
26	1×1	1.00	42	42	1×1	1.00	73
27	2×1	1.53	39	39	1×1	1.00	78
28	1×1	1.00	56	56	1×1	1.00	86
29	1×1	1.00	44	44	1×1	1.00	74
36	1×1	1.00	38	38	1×1	1.00	67
37	1×1	1.00	44	44	1×1	1.00	74
40	1×1	1.00	62	62	1×1	1.00	89
41	1×1	1.00	48	48	1×1	1.00	76
42	1×1	1.00	47	47	1×1	1.00	76
43	1×1	1.00	50	50	1×1	1.00	75
44	1×1	1.00	42	42	1×1	1.00	77

Table E.9: **Comparison of register blocked SpMV performance to the upper bound model: Pentium III-M**.

Mat.	Best			1×1	*Upper bounds* Analytic			PAPI
No.	$r_{opt}\times c_{opt}$	Fill	Mflop/s	Mflop/s	$r_{up}\times c_{up}$	Fill	Mflop/s	Mflop/s
1	4×4	1.00	256	142	12×12	1.00	276	268
2	4×4	1.00	168	135	8×8	1.00	266	241
4	3×3	1.06	145	130	6×3	1.12	229	206
5	4×2	1.00	148	125	4×4	1.00	244	230
7	3×3	1.00	155	133	3×3	1.00	244	218
8	3×3	1.11	141	133	6×3	1.13	228	219
9	3×3	1.02	155	135	3×3	1.02	240	228
10	1×1	1.00	133	133	2×2	1.21	185	174
12	1×1	1.00	130	130	2×2	1.24	182	162
13	1×1	1.00	130	130	2×2	1.28	177	157
15	2×1	1.00	136	119	2×1	1.00	196	169
40	1×1	1.00	127	127	1×1	1.00	155	133

Table E.10: **Comparison of register blocked SpMV performance to the upper bound model: Power3**.

Mat.	Best			1×1	Analytic upper bound		
No.	$r_{opt}\times c_{opt}$	Fill	Mflop/s	Mflop/s	$r_{up}\times c_{up}$	Fill	Mflop/s
1	10×8	1.00	819	607	12×12	1.00	970
8	6×2	1.13	581	436	6×6	1.15	798
9	3×3	1.02	705	500	3×3	1.02	826
10	2×1	1.10	549	434	2×2	1.21	633
40	1×1	1.00	453	453	1×1	1.00	516

Table E.11: **Comparison of register blocked SpMV performance to the upper bound model: Power4**.

Mat.	Best			1×1	*Upper bounds* Analytic			PAPI
No.	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Mflop/s	$r_{\text{up}} \times c_{\text{up}}$	Fill	Mflop/s	Mflop/s
1	4×1	1.00	250	156	12×12	1.00	304	306
2	4×1	1.00	214	142	8×8	1.00	294	286
3	3×1	1.06	192	140	6×6	1.12	260	253
4	3×1	1.04	192	135	3×3	1.06	255	246
5	4×1	1.00	180	118	4×4	1.00	272	257
6	3×1	1.00	201	139	3×3	1.00	273	260
7	3×1	1.00	210	141	3×3	1.00	273	261
8	3×1	1.06	195	136	6×3	1.13	253	244
9	3×1	1.01	211	140	3×3	1.02	268	249
10	3×1	1.27	164	134	2×2	1.21	209	194
11	2×2	1.23	109	107	2×2	1.23	197	183
12	3×1	1.24	172	140	2×2	1.24	205	192
13	3×1	1.26	167	140	2×1	1.14	200	186
15	2×1	1.00	159	126	2×1	1.00	223	187
17	3×1	1.59	133	133	1×1	1.00	179	163
21	1×1	1.00	127	127	1×1	1.00	179	161
25	1×1	1.00	63	63	1×1	1.00	157	133
27	3×1	1.94	70	63	1×1	1.00	160	136
28	1×1	1.00	111	111	1×1	1.00	176	148
36	3×1	2.31	51	49	1×1	1.00	139	121
40	1×1	1.00	127	127	1×1	1.00	181	142
44	1×1	1.00	62	62	1×1	1.00	159	119

Table E.12: **Comparison of register blocked SpMV performance to the upper bound model: Itanium 1**.

Mat.	Best			1×1	*Upper bounds* Analytic			PAPI
No.	$r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	Mflop/s	$r_{\text{up}}\times c_{\text{up}}$	Fill	Mflop/s	Mflop/s
1	4×2	1.00	1200	296	12×12	1.00	1486	1452
2	4×2	1.00	1122	275	8×8	1.00	1433	1387
3	6×1	1.10	946	275	6×6	1.12	1258	1229
4	4×2	1.23	807	248	6×3	1.12	1230	1151
5	4×2	1.00	1012	251	4×4	1.00	1290	1176
6	4×2	1.46	740	262	3×3	1.00	1261	1139
7	4×2	1.45	734	260	3×3	1.00	1261	1142
8	6×1	1.12	778	247	6×3	1.13	1222	1191
9	6×1	1.34	720	261	3×3	1.02	1240	1115
10	4×2	1.45	698	250	2×2	1.21	948	797
11	4×2	1.70	620	241	2×2	1.23	904	743
12	4×2	1.48	774	276	3×1	1.24	944	801
13	4×2	1.54	749	277	3×1	1.26	925	778
15	4×1	1.78	514	260	2×1	1.00	1001	778
17	4×1	1.75	536	269	2×1	1.36	770	594
20	4×2	2.33	477	243	1×2	1.17	763	567
21	6×1	2.01	537	273	2×1	1.38	761	590
24	4×1	2.28	393	253	1×1	1.00	735	499
25	4×1	3.13	226	130	1×1	1.00	680	467
26	4×1	2.91	271	158	1×1	1.00	660	439
27	4×1	2.39	273	141	1×1	1.00	689	478
28	4×1	2.83	314	222	1×1	1.00	743	517
36	4×2	3.53	252	128	1×1	1.00	618	412
40	4×1	2.39	327	250	1×1	1.00	759	539
41	2×1	1.78	252	239	1×1	1.00	693	479
42	2×1	1.80	248	243	1×1	1.00	693	480
44	1×1	1.00	230	230	1×1	1.00	697	483

Table E.13: **Comparison of register blocked SpMV performance to the upper bound model: Itanium 2**.

Appendix F

Block Size and Alignment Distributions

Figures F.1–F.15 show the block size distributions for our test matrices arising in finite element method (FEM) applications. Each matrix was first partitioned using the greedy algorithm described in Section 5.1.2, and then converted to variable block row (VBR) format using the compressed sparse row (CSR) format-to-VBR conversion routine provided by SPARSKIT [267]. We show the partitioning of each matrix when $\theta = 1$ and $\theta = \theta_{\min}$, as described in Section 5.1.4.

The top plot in each of Figures F.1–F.15 shows the fraction of total non-zeros contained within $r \times c$ blocks. Each square is an $r \times c$ block size shaded by the fraction of total non-zeros contained in blocks of that size, and labeled by the same fraction rounded to two decimal digits. Thus, a '0' entry at a particular $r \times c$ indicates that there is at least 1 block of size $r \times c$, but that fewer than .5% of all non-zeros were contained within $r \times c$ blocks. No numerical label on a given square indicates that no blocks of the corresponding size occurs.

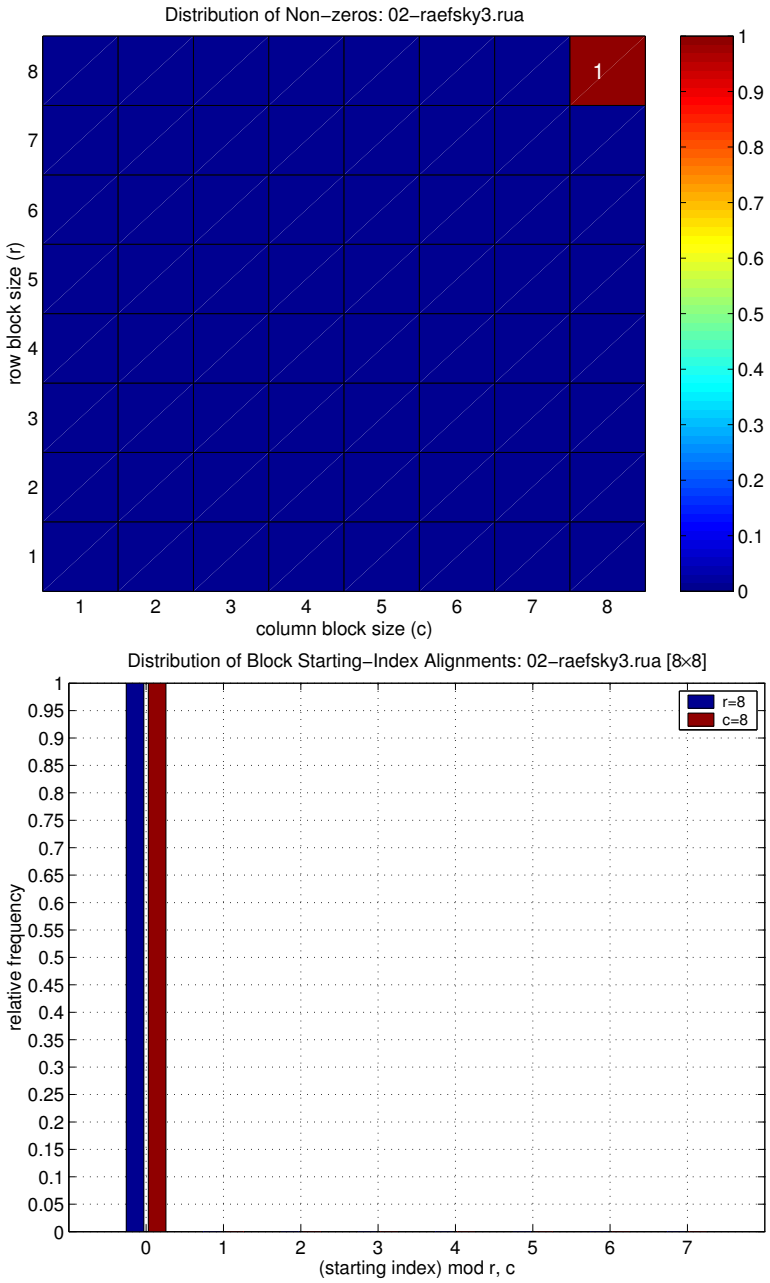

Figure F.1: **Distribution and alignment of block sizes: Matrix `raefsky3`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 8×8 blocks. Specifically, we plot the fraction of 8×8 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

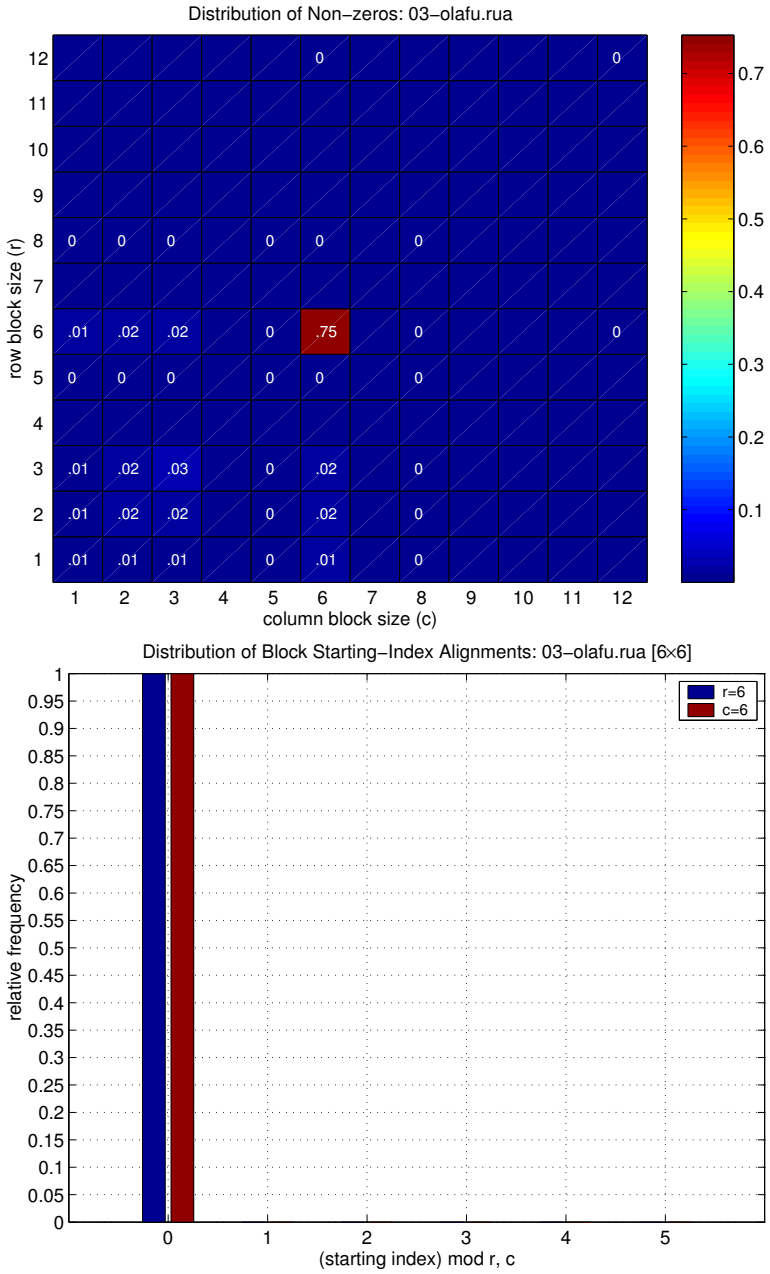

Figure F.2: **Distribution and alignment of block sizes: Matrix `olafu`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 6×6 blocks. Specifically, we plot the fraction of 6×6 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

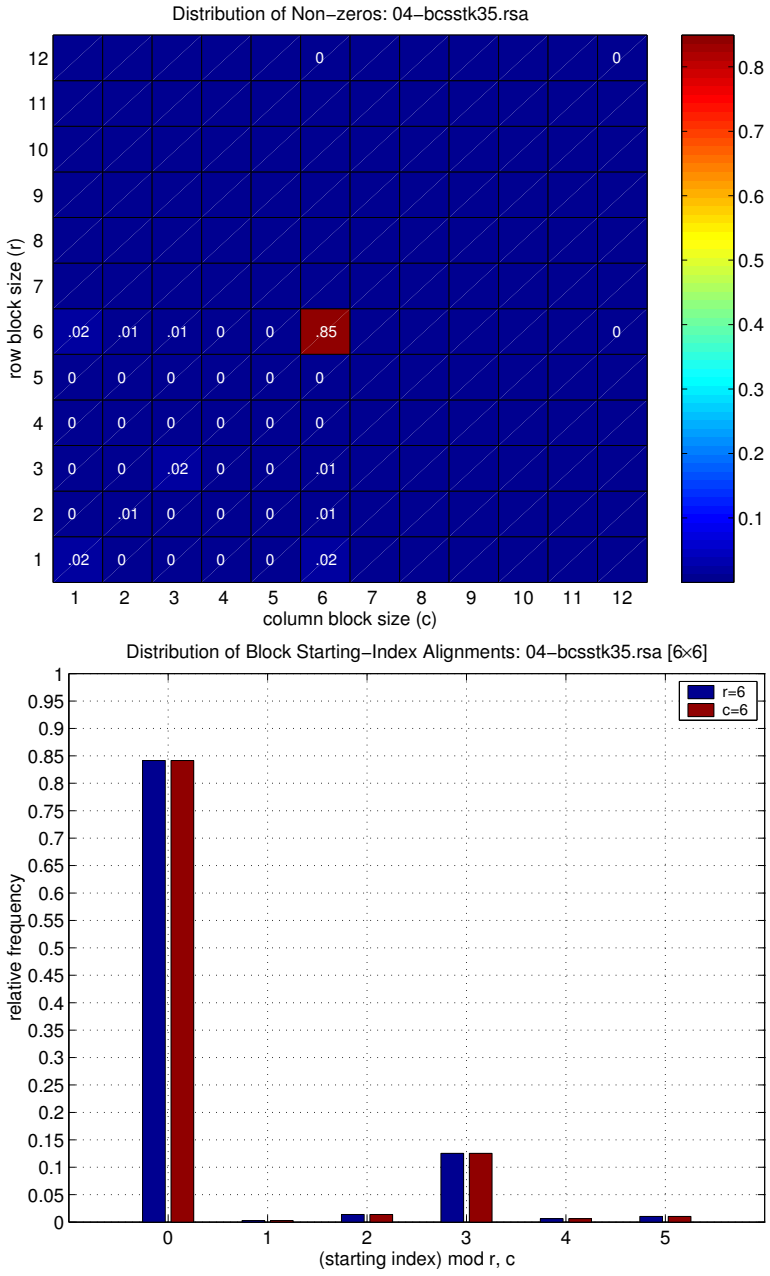

Figure F.3: **Distribution and alignment of block sizes: Matrix** `bcsstk35`. (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 4×4 blocks. Specifically, we plot the fraction of 4×4 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

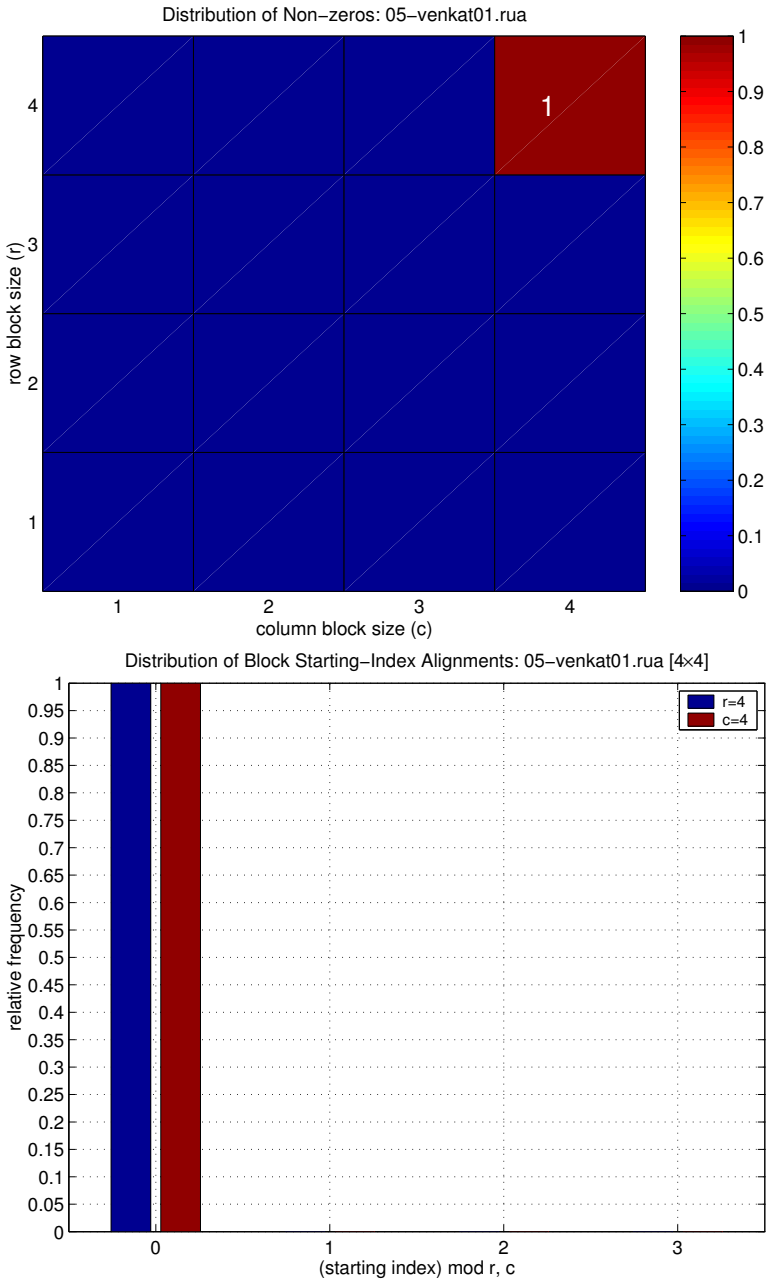

Figure F.4: **Distribution and alignment of block sizes: Matrix `venkat01`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

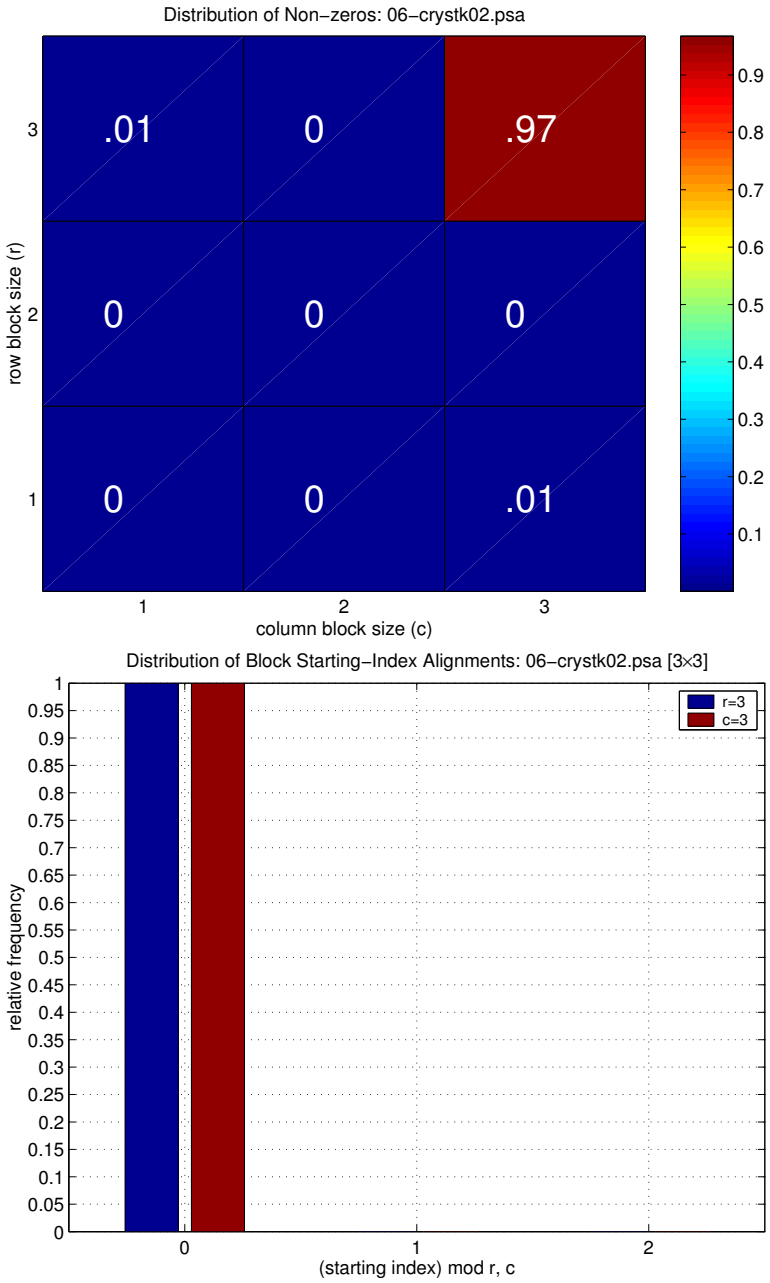

Figure F.5: **Distribution and alignment of block sizes: Matrix `crystk02`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

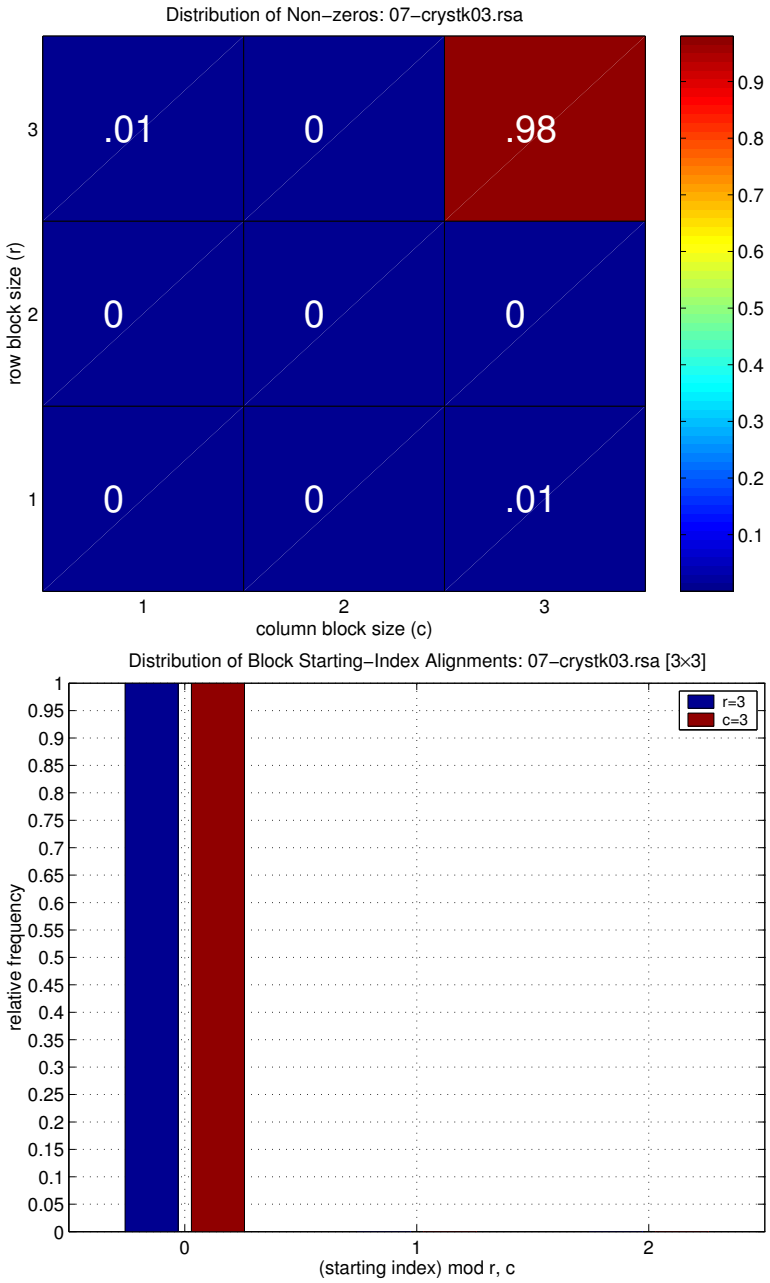

Figure F.6: **Distribution and alignment of block sizes: Matrix `crystk03`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 6×6 blocks. Specifically, we plot the fraction of 6×6 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

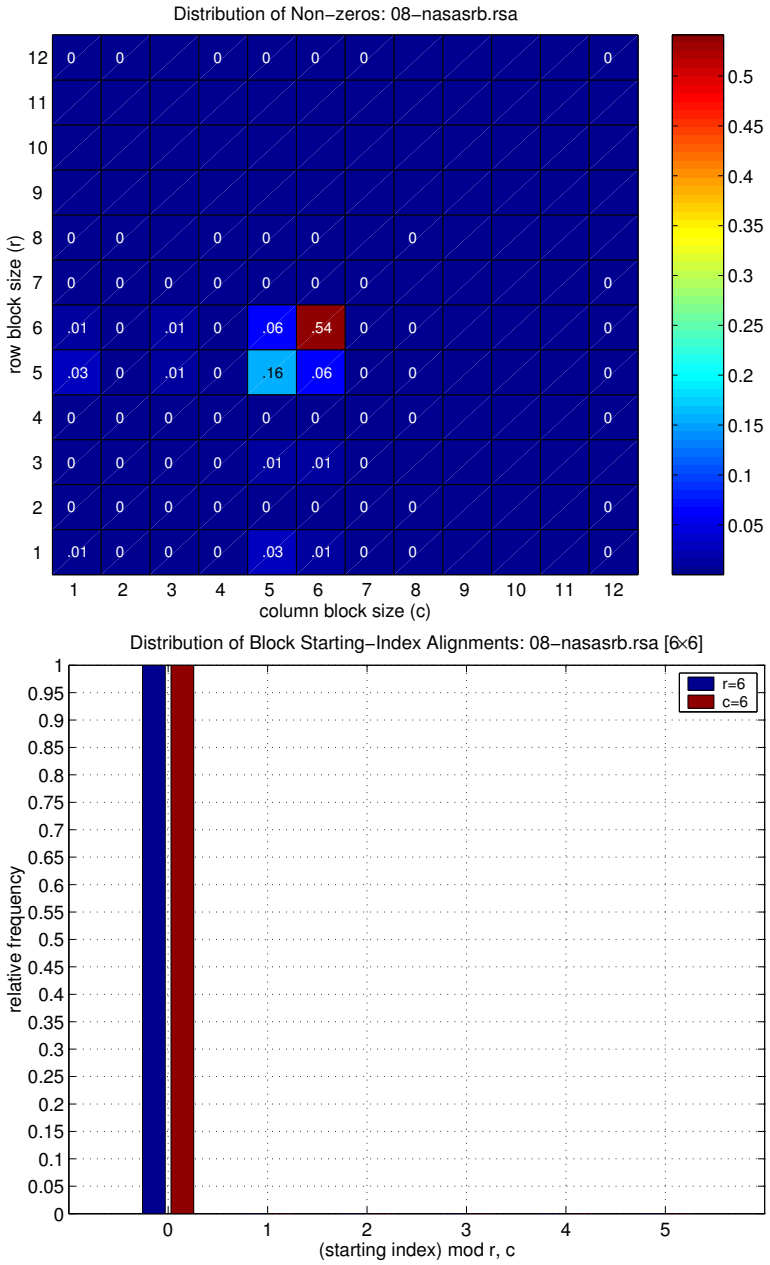

Figure F.7: **Distribution and alignment of block sizes: Matrix `nasasrb`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

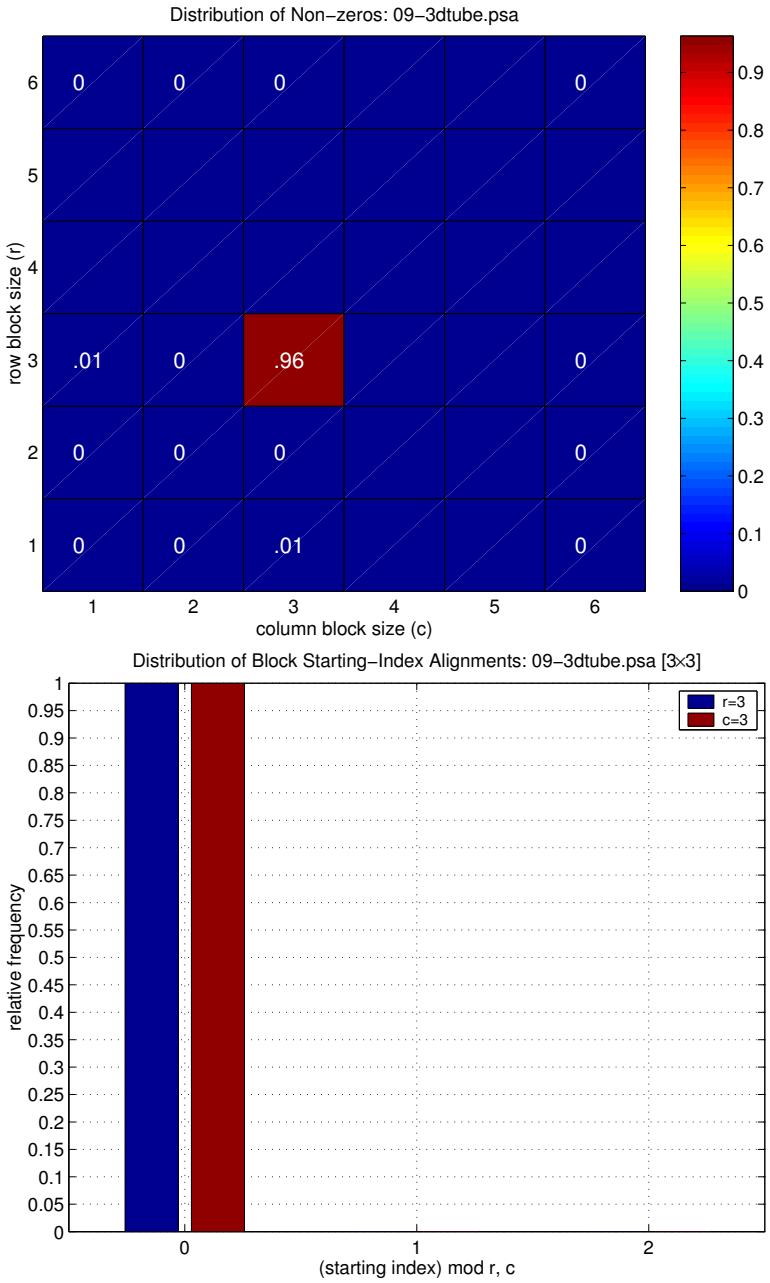

Figure F.8: **Distribution and alignment of block sizes: Matrix `3dtube`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 6×6 blocks. Specifically, we plot the fraction of 6×6 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

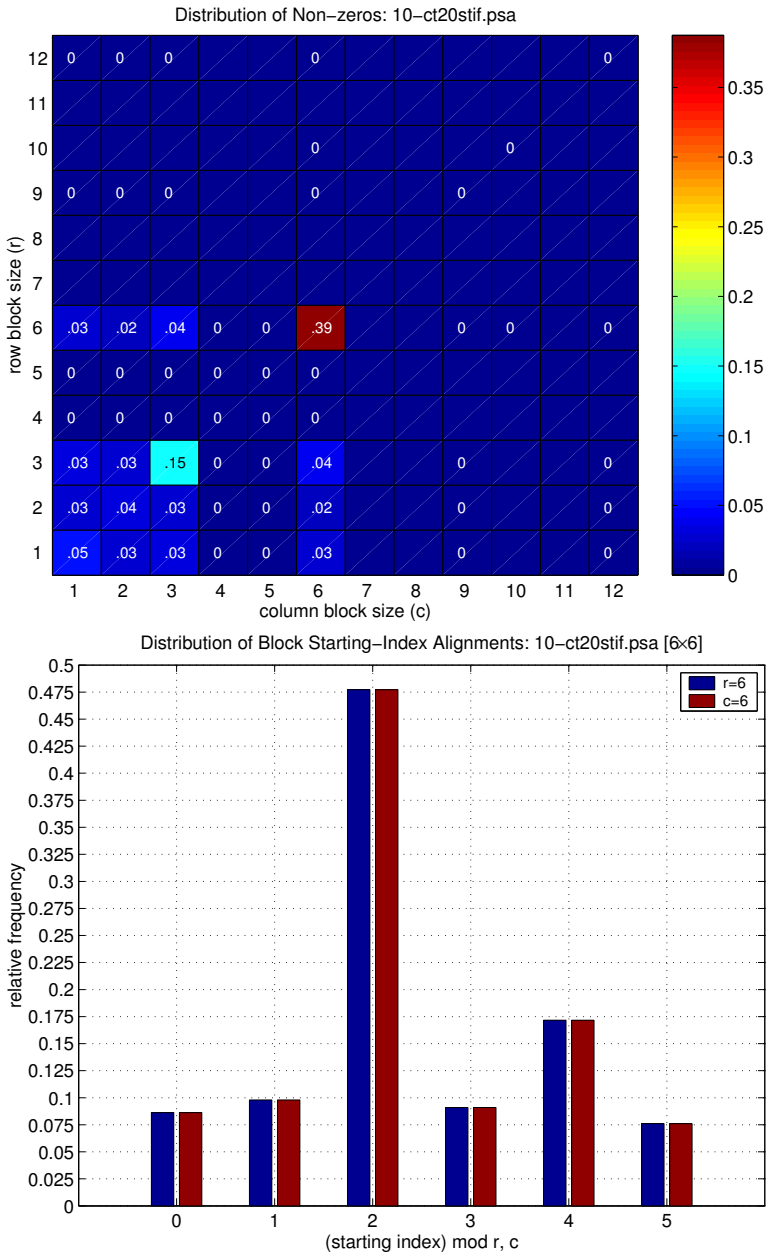

Figure F.9: **Distribution and alignment of block sizes: Matrix `ct20stif`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

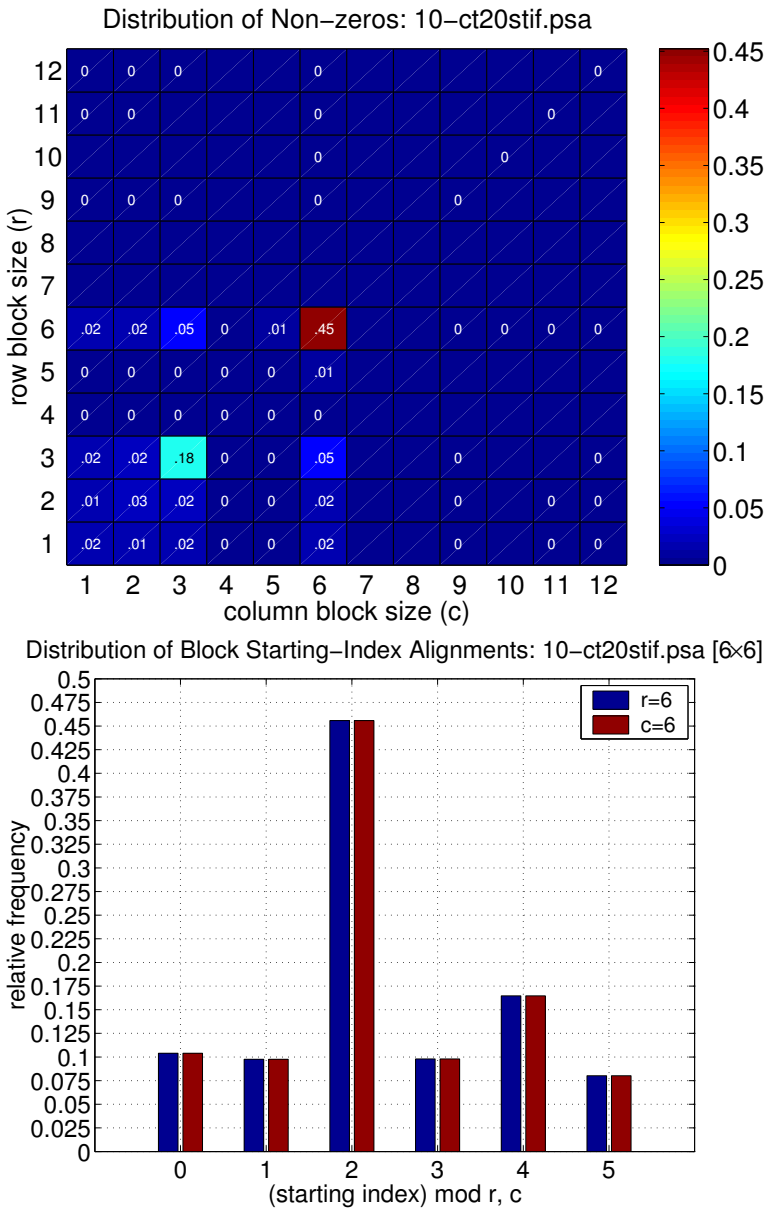

Figure F.10: **Distribution and alignment of block sizes ($\theta = .9$): Matrix `ct20stif`.** Compare to Figure F.9. (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format *with fill*, where the partitioning threshold is set to $\theta = .9$. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks.

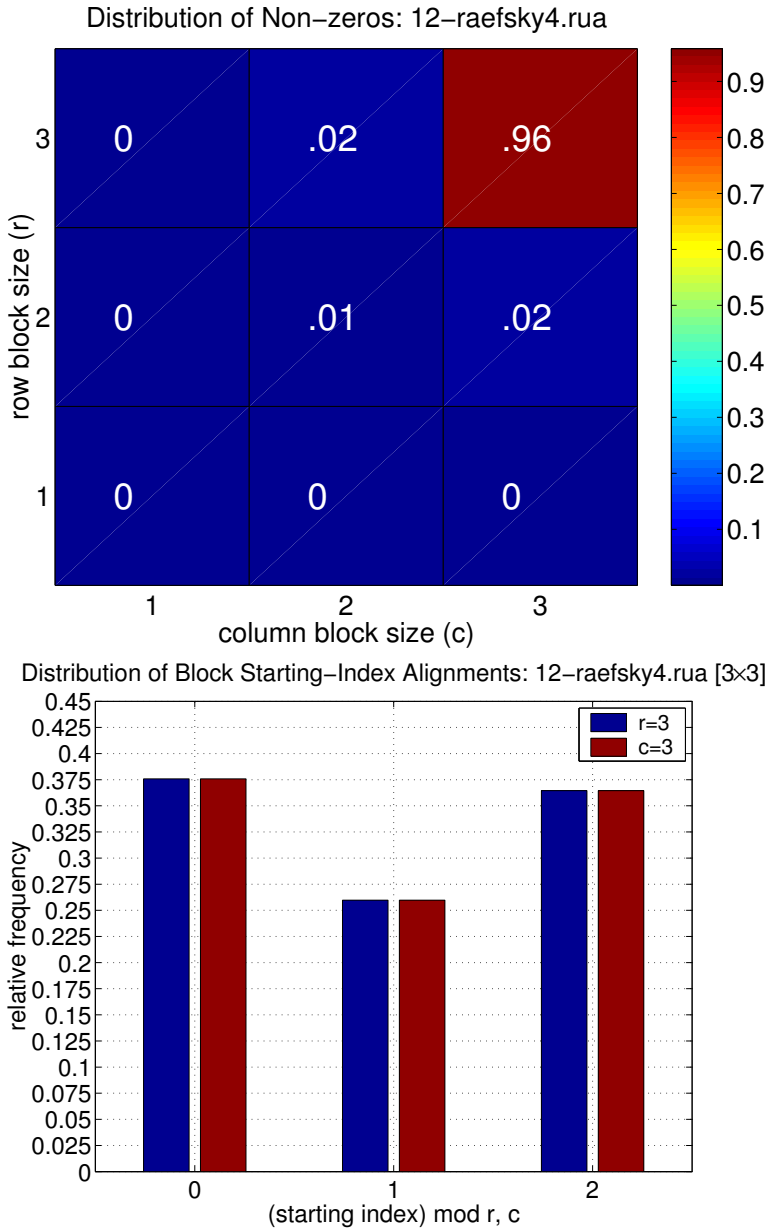

Figure F.11: **Distribution and alignment of block sizes: Matrix `raefsky4`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

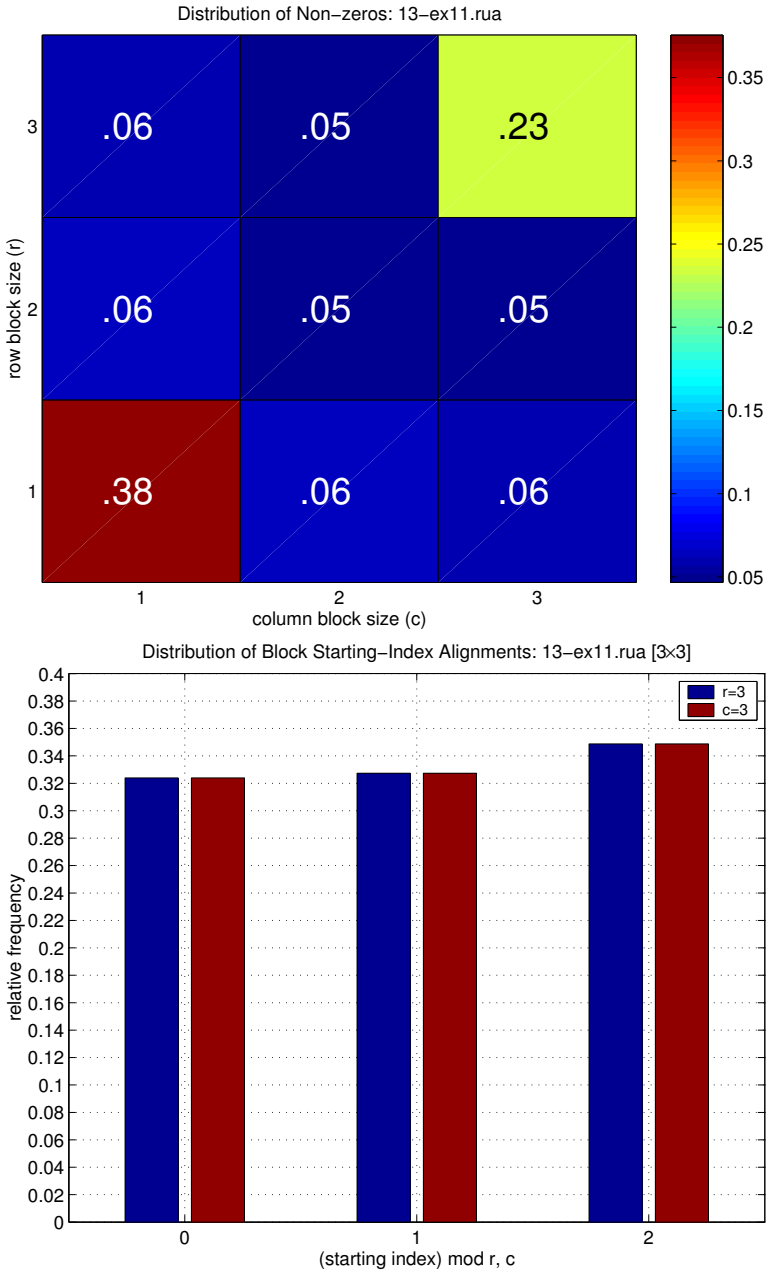

Figure F.12: **Distribution and alignment of block sizes: Matrix `ex11`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

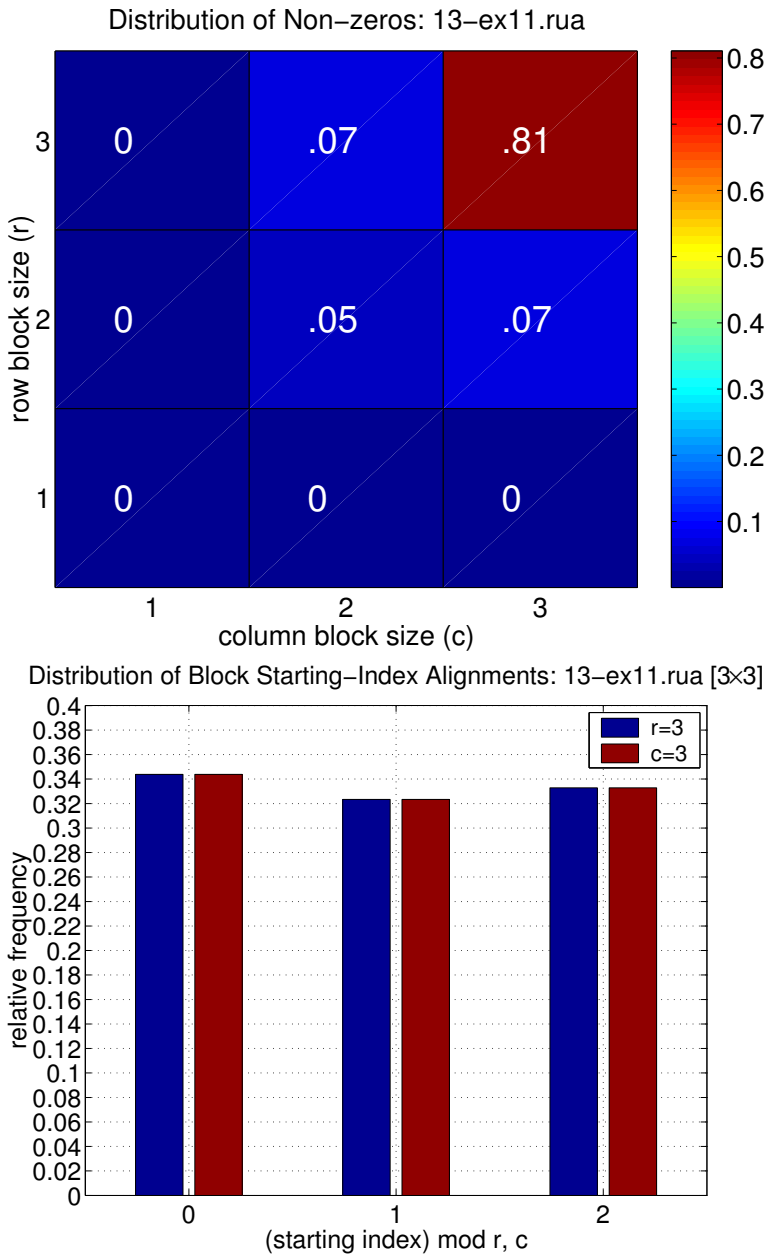

Figure F.13: **Distribution and alignment of block sizes ($\theta = .7$): Matrix `ex11`.** Compare to Figure F.12. (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format *with fill*, where the partitioning threshold is set to $\theta = .7$. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks.

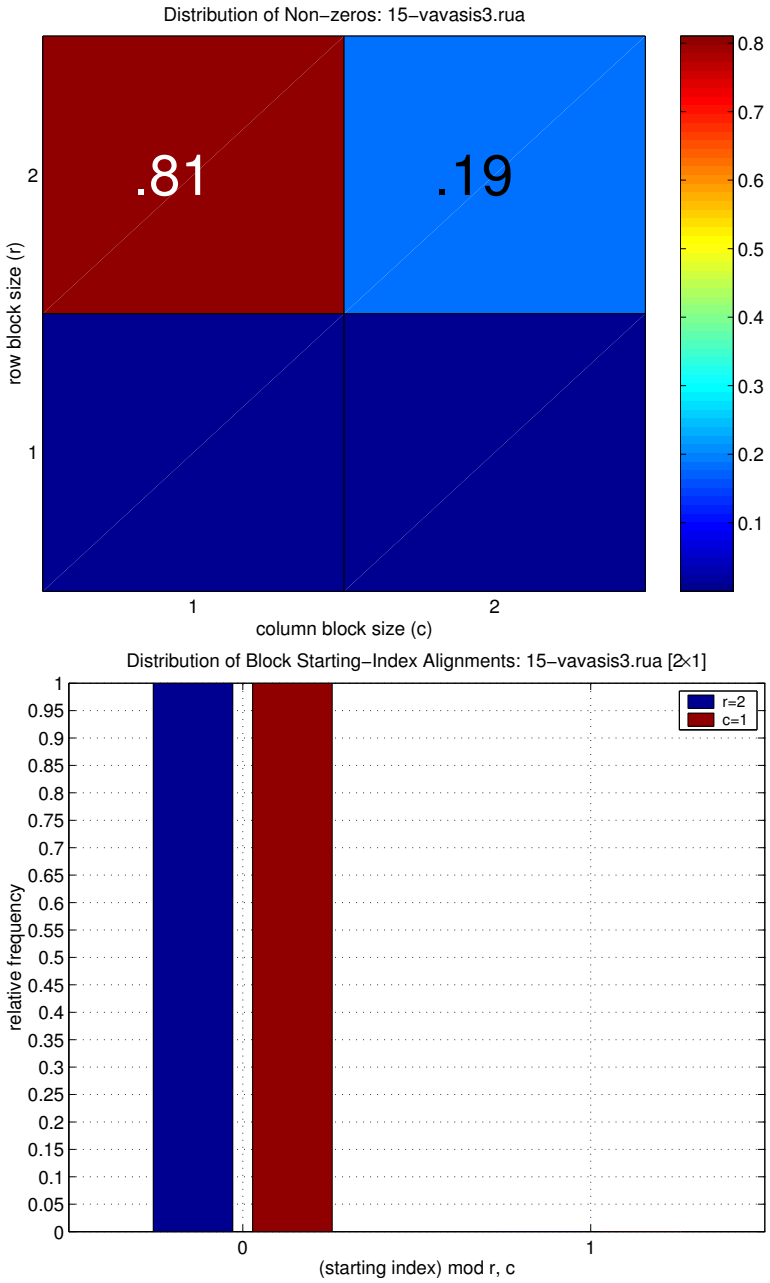

Figure F.14: **Distribution and alignment of block sizes: Matrix `vavasis3`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 2×1 blocks. Specifically, we plot the fraction of 2×1 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

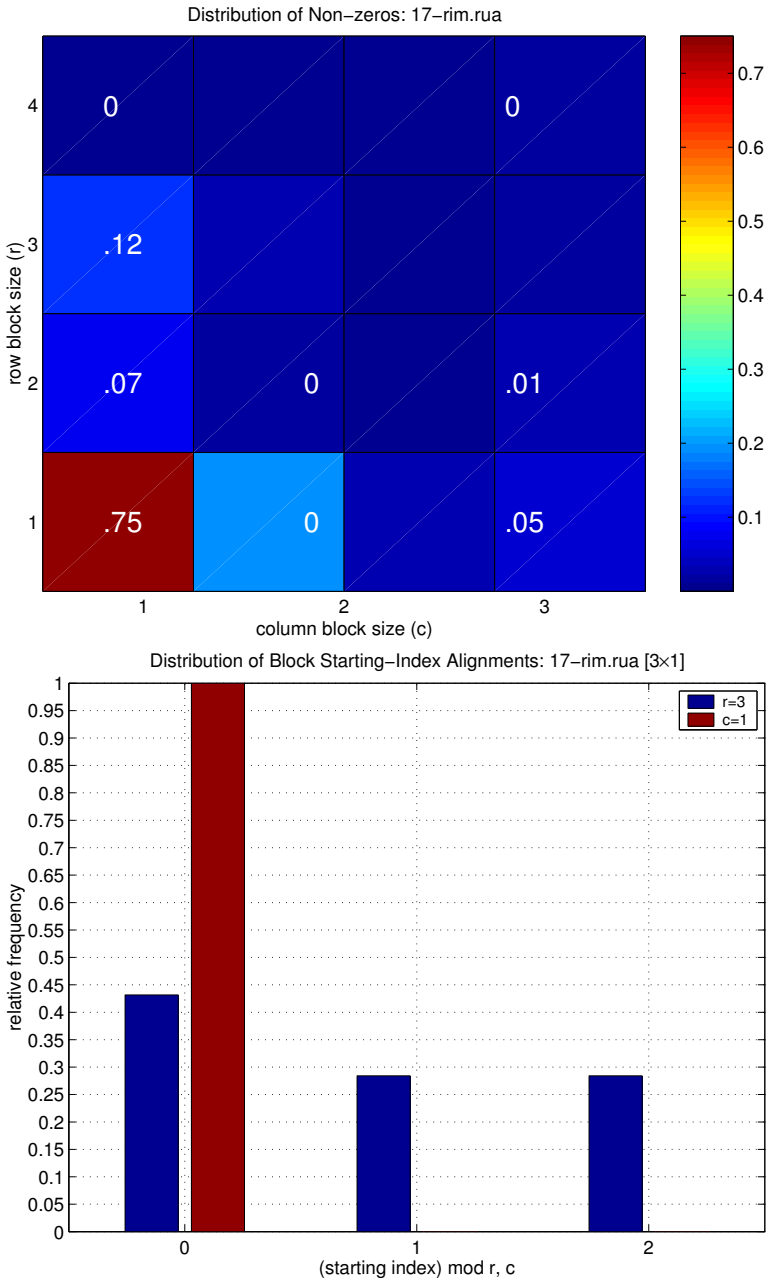

Figure F.15: **Distribution and alignment of block sizes: Matrix `rim`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×1 blocks. Specifically, we plot the fraction of 3×1 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

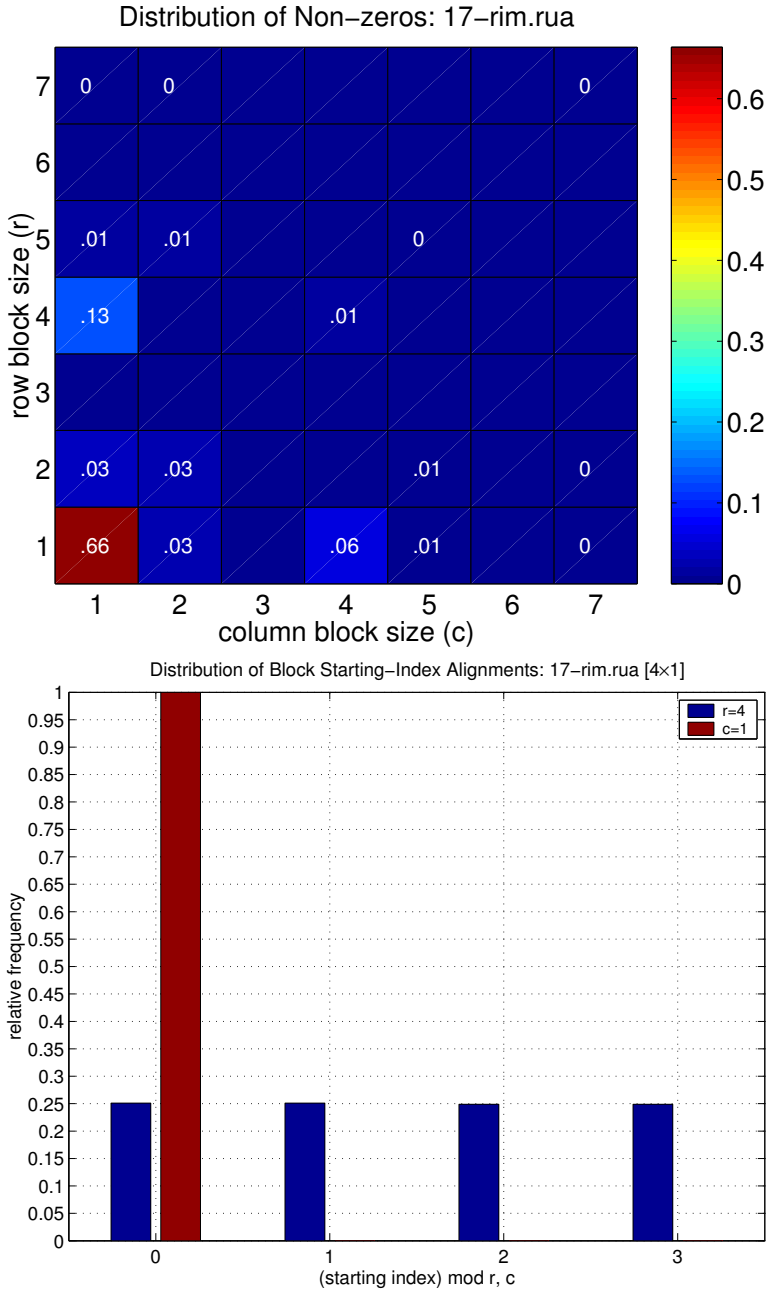

Figure F.16: **Distribution and alignment of block sizes ($\theta = .8$): Matrix `rim`.** Compare to Figure F.15. (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format *with fill*, where the partitioning threshold is set to $\theta = .8$. (*Bottom*) Distribution of row and column alignments for the 3×1 blocks.

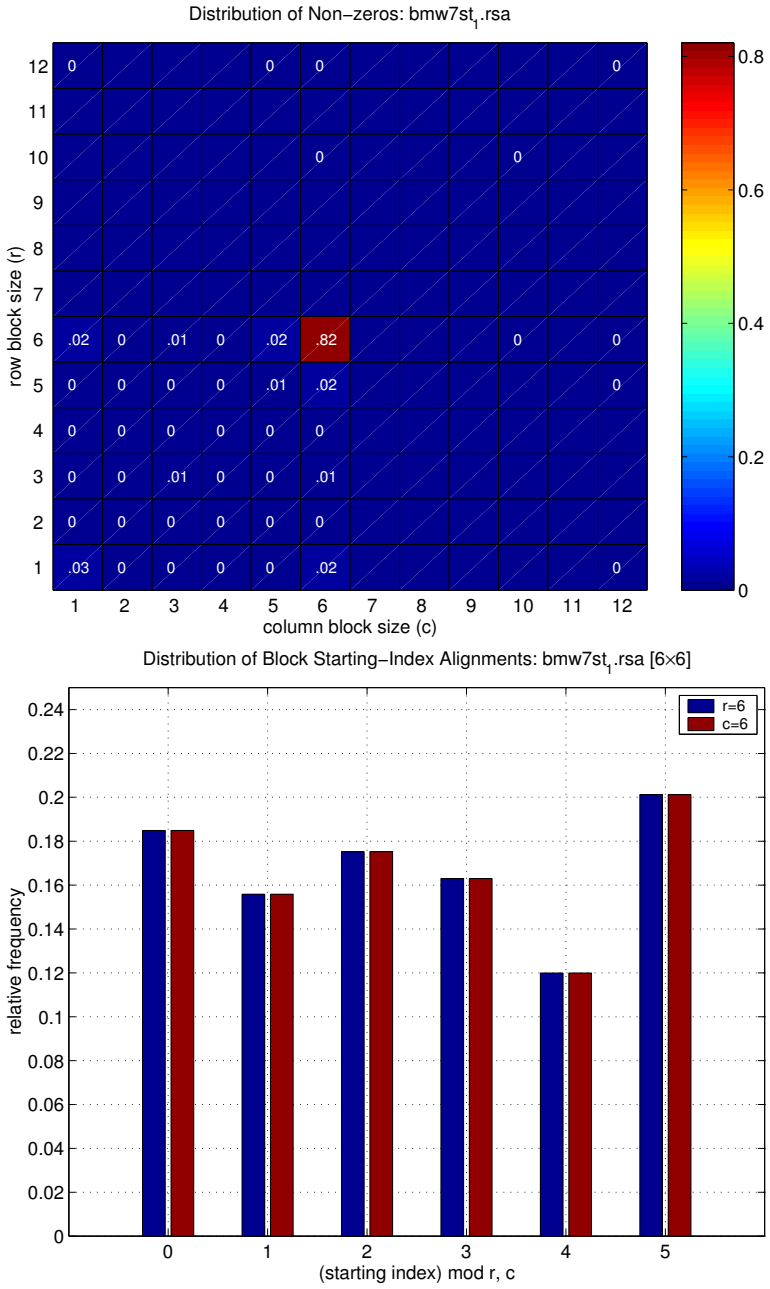

Figure F.17: **Distribution and alignment of block sizes: Matrix `bmw7st_1`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 6×6 blocks. Specifically, we plot the fraction of 6×6 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

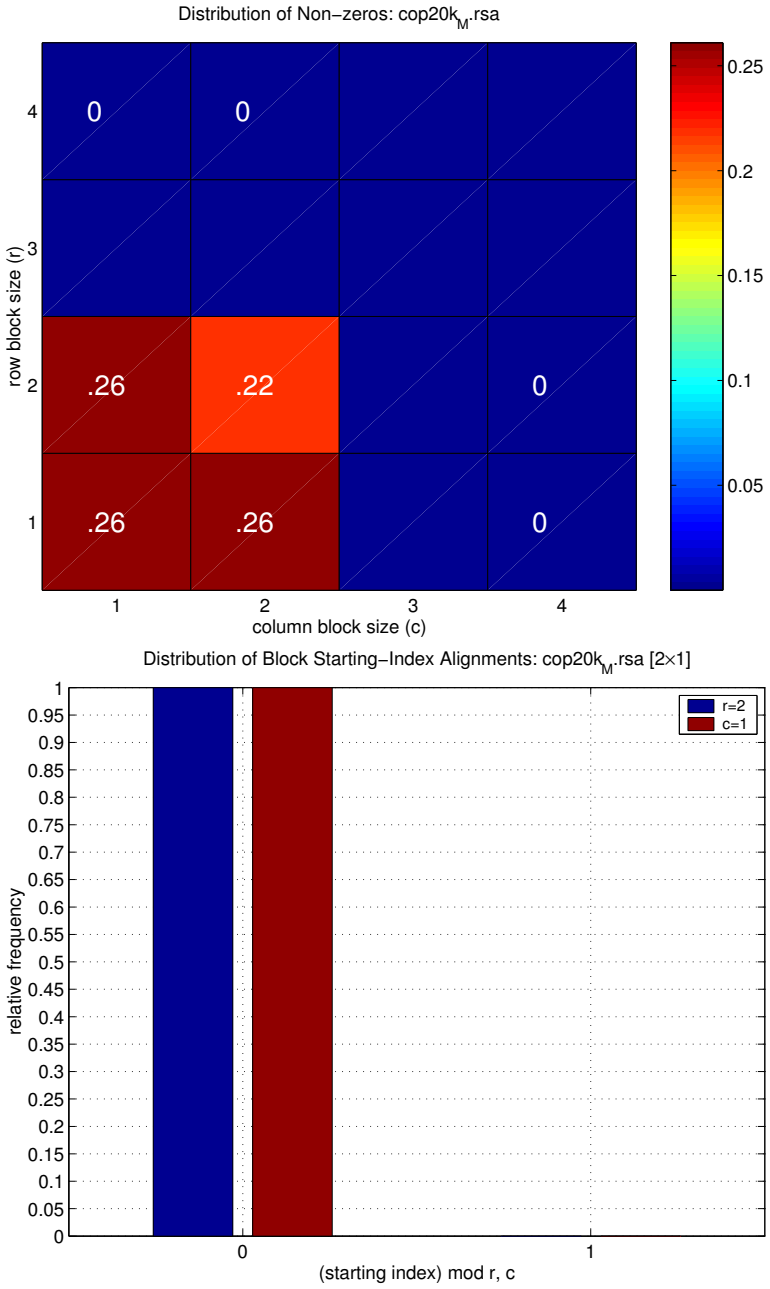

Figure F.18: **Distribution and alignment of block sizes: Matrix** `cop20k_M`. (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 2×1 blocks. Specifically, we plot the fraction of 2×1 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

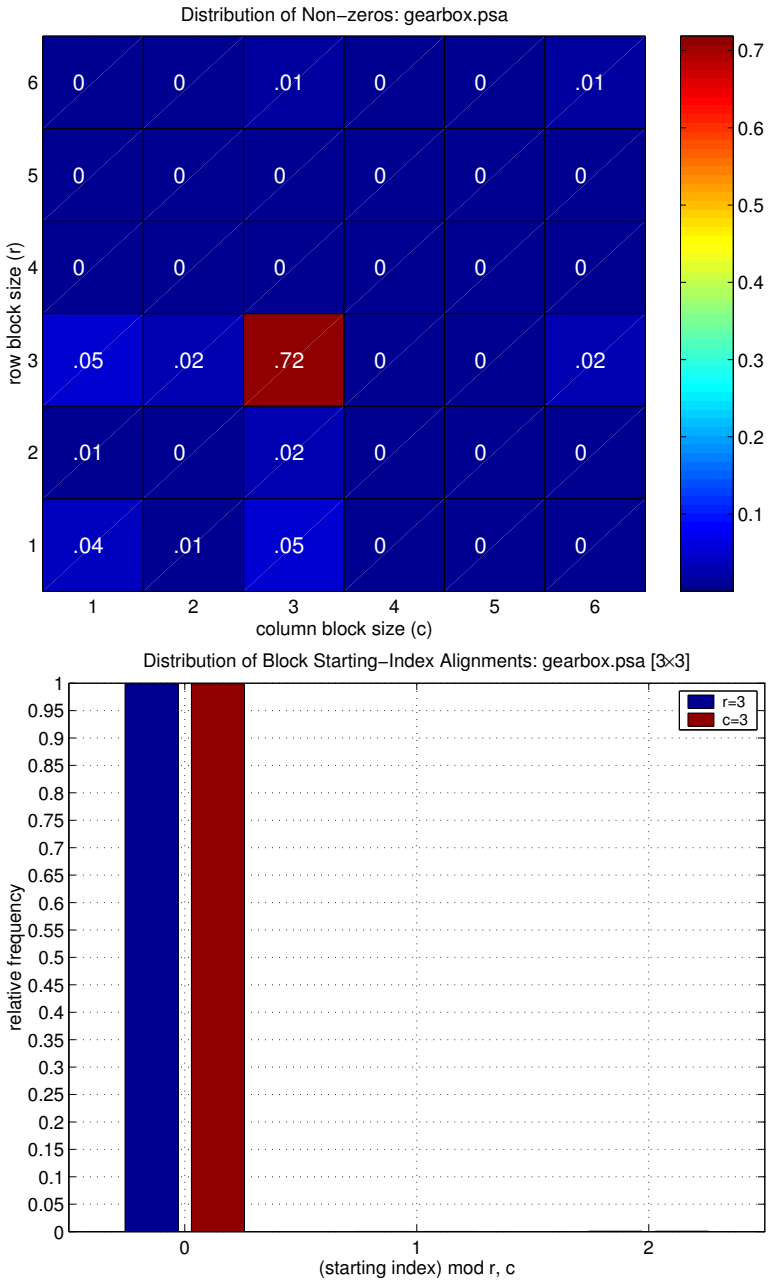

Figure F.19: **Distribution and alignment of block sizes: Matrix `gearbox`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

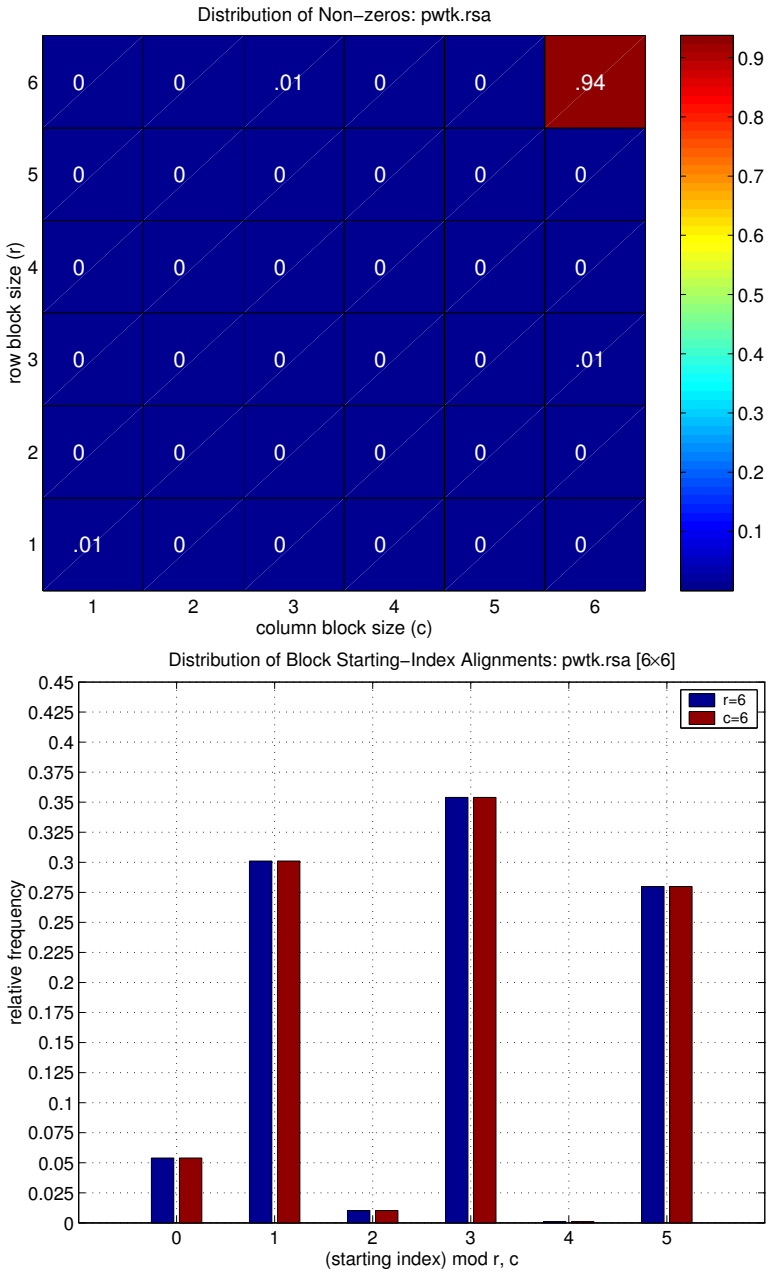

Figure F.20: **Distribution and alignment of block sizes: Matrix `pwtk`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 6×6 blocks. Specifically, we plot the fraction of 6×6 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

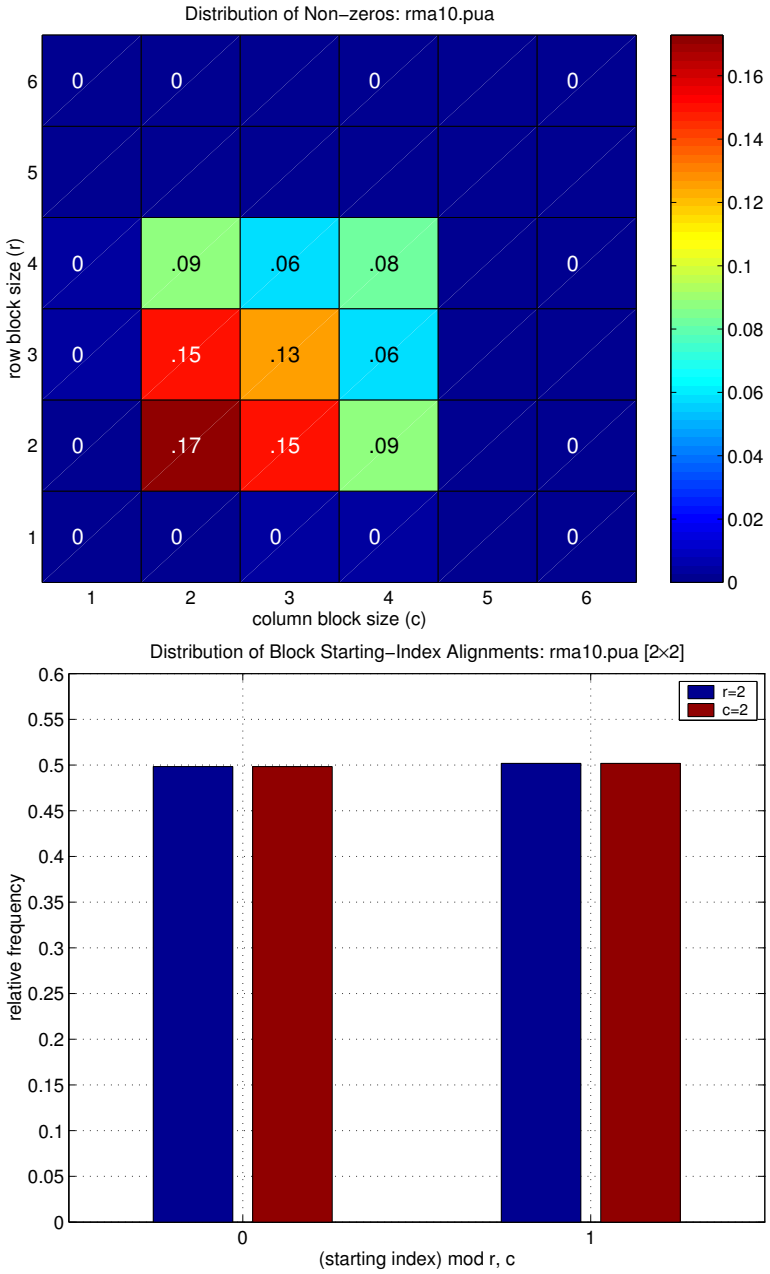

Figure F.21: **Distribution and alignment of block sizes: Matrix `rma10`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 2×2 blocks. Specifically, we plot the fraction of 2×2 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

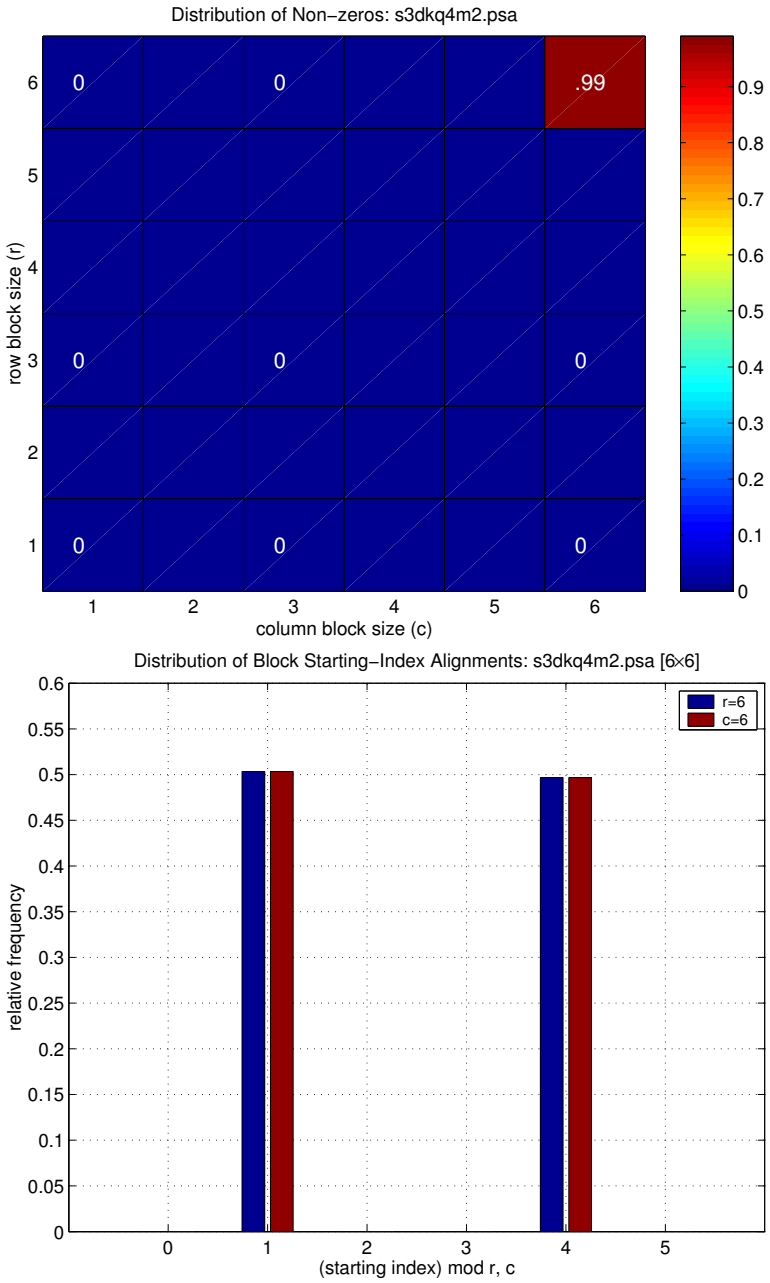

Figure F.22: **Distribution and alignment of block sizes: Matrix** `s3dkq4m2`. (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 6×6 blocks. Specifically, we plot the fraction of 6×6 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

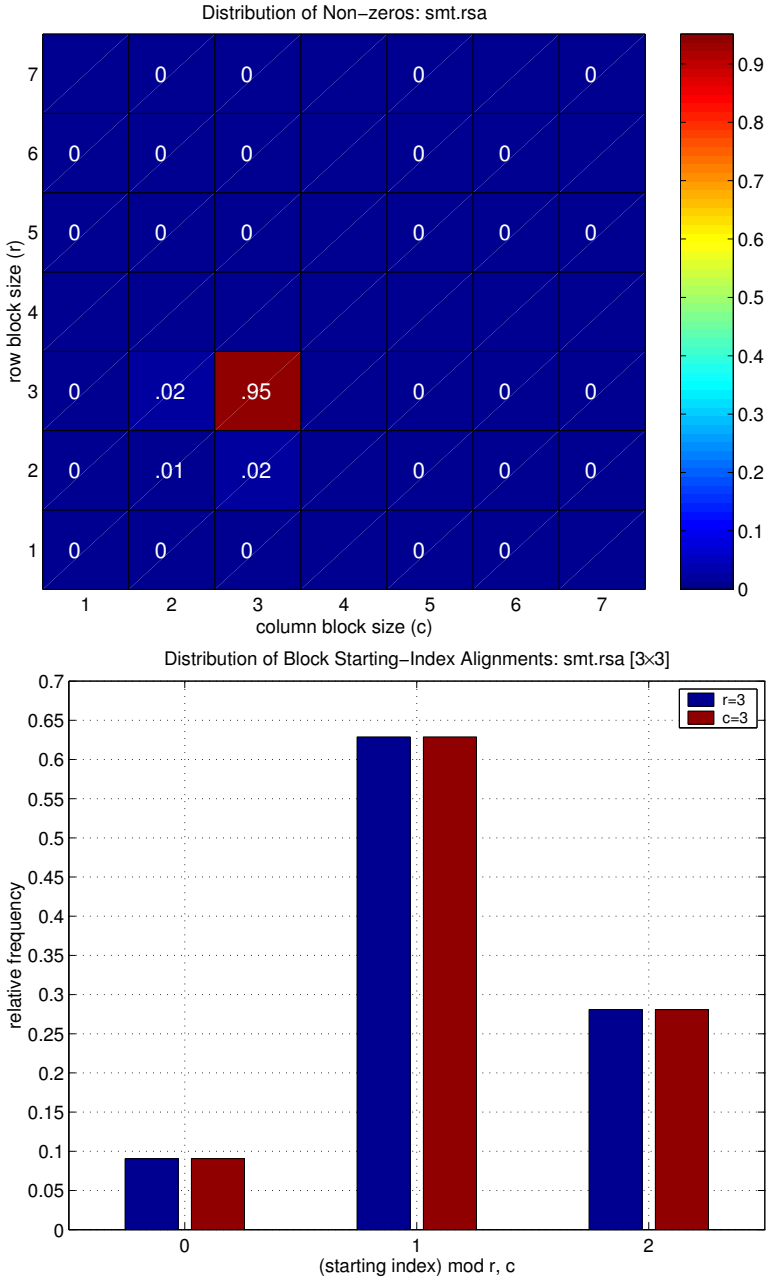

Figure F.23: **Distribution and alignment of block sizes: Matrix `smt`.** (*Top*) Distribution of non-zeros by block size when the matrix is supplied in VBR format with no fill. A numerical label, even if 0, indicates that at least 1 block had the corresponding block size. A lack of a label indicates exactly 0 blocks of the given block size. (*Bottom*) Distribution of row and column alignments for the 3×3 blocks. Specifically, we plot the fraction of 3×3 blocks whose starting row index i satisfies $i \mod r = 0$, and whose starting column index j satisfies $j \mod c = 0$.

Appendix G

Variable Block Splitting Data

Tables G.1–G.4 show the splittings used in Figures 5.6–5.9. For each matrix (column 1), we show the following.

- Columns 2–4: The best register blocking performance and corresponding block size, fill ratio.
- Columns 5–9: The best performance with splitting, using unaligned block compressed sparse row (UBCSR) format. The matrix is initially converted to VBR using a fill threshold of θ (column 6). We show the block size $r_k \times c_k$ used for each component of the splitting (column 7). We also show the corresponding number of non-zeros (divided by ideal non-zeros) for the k-th component (column 8), and estimated performance of just the k-th component (column 9) using the non-zero count in column 8.

  If the best splitting performance occurs for $\theta < 1$, we also show the data corresponding to the best performance when $\theta = 1$.
- Column 10: Reference performance, using CSR.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	θ	$r_k \times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	39	2×1	1.10	43	0.9	3×3	0.77	52	34
						1×1	0.24	28	
				41	1	3×3	0.63	52	
						1×1	0.37	30	
12	38	2×2	1.24	51	1	3×3	0.96	54	33
						1×1	0.04	31	
13	37	2×1	1.14	37	1	3×1	0.34	52	34
						1×1	0.66	34	
15	40	2×1	1.00	39	1	2×1	1.00	40	31
						1×1	0.00	0	
17	32	1×1	1.00	34	0.8	2×1	0.24	61	32
						1×1	0.77	33	
				33	1	3×1	0.12	139	
						1×1	0.88	33	
A	39	2×2	1.22	47	1	6×6	0.82	57	32
						1×1	0.18	27	
B	25	1×1	1.00	27	1	2×1	0.48	32	25
						1×1	0.52	24	
C	44	3×3	1.22	53	1	6×6	0.94	58	34
						1×1	0.06	23	
D	38	2×1	1.14	39	1	2×2	0.77	46	34
						1×1	0.23	25	
E	38	8×2	1.45	57	1	6×2	0.99	60	34
						1×1	0.01	8	

Table G.1: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Ultra 2i**. Splitting data for Figure 5.6.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	θ	$r_k \times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	77	2×2	1.21	93	0.9	3×3	0.77	120	61
						1×1	0.25	57	
				90	1	3×3	0.63	118	
						3×1	0.11	78	
						1×1	0.26	59	
12	83	2×2	1.24	115	1	3×3	0.96	111	68
						1×1	0.04	40	
13	84	3×2	1.40	105	0.7	3×3	0.82	112	69
						3×2	0.07	93	
						1×1	0.12	55	
				76	1	3×3	0.23	121	
						2×1	0.23	78	
						1×1	0.53	65	
15	79	2×1	1.00	79	1	2×1	1.00	79	64
						1×1	0.00	0	
17	69	1×1	1.00	70	0.8	4×1	0.17	100	69
						1×1	0.84	67	
				68	1	3×1	0.12	84	
						1×1	0.88	67	
A	83	2×2	1.22	102	1	3×3	0.88	108	65
						1×1	0.12	50	
B	46	1×1	1.00	52	1	2×1	0.48	59	46
						1×2	0.26	51	
						1×1	0.26	43	
C	91	3×3	1.22	105	1	3×6	0.94	115	65
						1×1	0.06	42	
D	79	2×2	1.29	80	1	3×2	0.45	102	67
						2×2	0.36	91	
						1×1	0.19	47	
E	90	2×2	1.11	115	1	6×6	0.99	114	65
						1×1	0.01	16	

Table G.2: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Pentium III-M**. Splitting data for Figure 5.7.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	θ	$r_k \times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	549	2×1	1.10	643	0.9	6×1	0.56	758	434
						3×1	0.30	681	
						2×1	0.09	623	
						1×1	0.07	323	
				579	1	3×2	0.61	703	
						3×1	0.13	590	
						1×1	0.26	414	
A	442	2×1	1.10	596	1	6×3	0.84	728	334
						1×1	0.16	355	
B	199	1×1	1.00	363	1	2×2	0.22	355	199
						2×1	0.26	461	
						1×2	0.26	354	
						1×1	0.26	336	
C	332	3×1	1.11	453	1	6×3	0.94	463	224
						1×1	0.06	254	
D	477	1×1	1.00	524	1	3×2	0.45	701	477
						2×2	0.36	550	
						1×1	0.19	280	
E	530	4×1	1.17	657	1	6×3	0.99	731	427
						1×1	0.01	71	

Table G.3: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Power4**. Splitting data for Figure 5.8.

	Register Blocking			Splitting					1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	θ	$r_k \times c_k$	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
10	698	4×2	1.45	537	1	6×1	0.49	945	250
						3×1	0.25	607	
						1×1	0.26	287	
12	774	4×2	1.48	643	1	3×1	0.97	710	276
						1×1	0.03	138	
13	749	4×2	1.54	622	0.7	3×1	0.89	720	277
						2×1	0.12	534	
						1×1	0.00	25	
				421	1	3×1	0.34	681	
						2×1	0.16	517	
						1×1	0.50	329	
15	514	4×1	1.78	510	1	2×1	1.00	559	260
						1×1	0.00	0	
17	536	4×1	1.75	338	0.8	4×1	0.17	723	269
						1×1	0.84	326	
				330	1	2×1	0.16	442	
						1×1	0.84	325	
A	772	4×2	1.43	689	1	6×1	0.87	957	333
						1×1	0.13	240	
B	342	2×2	1.82	315	1	2×1	0.48	389	254
						1×2	0.26	302	
						1×1	0.26	255	
C	826	4×2	1.34	795	1	6×1	0.95	982	337
						1×1	0.05	166	
D	718	4×2	1.55	433	1	4×2	0.21	804	331
						2×2	0.56	582	
						1×1	0.23	226	
E	895	4×2	1.23	842	1	6×1	0.99	984	337
						1×1	0.01	36	

Table G.4: **Best unaligned block compressed sparse row splittings on variable block matrices, compared to register blocking: Itanium 2**. Splitting data for Figure 5.9.

Appendix H

Row Segmented Diagonal Data

H.1 Sparse matrix-vector multiply

We show the sparse matrix-vector multiply (SpMV) wrapper for $u = 2$ that calls the subroutine shown in Figure 5.14. Row segments are not required to be an exact multiple of u (*i.e.*, see line 5), allowing row segments to be chosen independently of the unrolling factor. (The layout of data in the value array `val` does, however, depend on u.)

H.2 Implementation configuration and performance

Tables H.1–H.4 show the splittings used in Figures 5.15–5.18. For each matrix (column 1), we show the following.

- Columns 2–4: The best register blocking performance and corresponding block size, fill ratio.
- Columns 5–9: The best performance with splitting, using row segmented diagonal (RSDIAG) format for the diagonal substructure and block compressed sparse row (BCSR) format format for any block structure (column 5). For the RSDIAG component, we show the value of the unrolling depth tuning parameter u (column 6). For the register blocked component, we show the block size $r_k \times c_k$ splitting (column 6). We also show the corresponding number of non-zeros (divided by ideal non-zeros) for each component (column 7), and estimated performance of each component (column 8) using the non-zero count in column 7.

```
    void sparse_mvm_rsegdiag_2( int n_segs,
        const double* val, const int* src_ind,
        const int* num_diags, const int* seg_starts,
        const double* x, double* y )
    {
        int S;
1       for( S = 0; S < n_segs; S++ ) // loop over segments
        {
2           int n_diags_seg = num_diags[S];
3           int num_rows = seg_starts[S+1] - seg_starts[S];

4           sparse_mvm_onerseg_2( num_rows / 2, n_diags_seg,
               val, src_ind, x, y );

5           if( num_rows % 2 ) // leftover rows
            {
               /* ... call cleanup routine ... */
            }

            /* advance pointers to the next segment */
6           val += num_rows * n_diags_seg;
7           src_ind += n_diags_seg;
8           y += num_rows;
        }
    }
```

Figure H.1: **Row segmented diagonal sparse matrix-vector multiply routine**. An example of the complete sparse matrix-vector multiply routine for $u = 2$.

- Column 9: Reference performance, using CSR.

	Register Blocking			RSDIAG-based splitting				1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Tuning Param	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
11	35	2×2	1.23	45	$u = 8$	0.24	45	30
					4×2	0.77	52	
S1	19	1×1	1.00	36	$u = 9$	1.00	36	18
S2	22	1×1	1.00	42	$u = 9$	1.00	42	22
S3	31	1×1	1.00	48	$u = 9$	1.00	48	31
F	17	1×1	1.00	25	$u = 11$	0.60	24	17
					8×8	0.40	29	
G	27	1×1	1.00	47	$u = 8$	1.00	47	27
H	19	1×1	1.00	35	$u = 9$	1.00	35	19
I	18	1×1	1.00	30	$u = 15$	1.00	30	18

Table H.1: **Best row segmented diagonal + register blocking performance, compared to register blocking only: Ultra 2i**. Row segmented diagonal data for Figure 5.15.

	Register Blocking			RSDIAG-based splitting				1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Tuning Param	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
11	74	2×2	1.23	90	$u = 17$	0.24	69	59
					4×2	0.77	102	
S1	42	1×1	1.00	70	$u = 6$	1.00	70	41
S2	47	1×2	1.33	83	$u = 7$	1.00	83	47
S3	58	1×2	1.32	93	$u = 7$	1.00	93	58
F	46	1×1	1.00	54	$u = 19$	0.60	54	44
					4×8	0.40	52	
G	56	2×2	1.59	89	$u = 7$	1.00	89	55
H	46	1×1	1.00	74	$u = 7$	1.00	74	46
I	41	1×1	1.00	67	$u = 7$	1.00	67	41

Table H.2: **Best row segmented diagonal + register blocking performance, compared to register blocking only: Pentium III-M**. Row segmented diagonal data for Figure 5.16.

	Register Blocking			RSDIAG-based splitting				1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Tuning Param	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
S1	242	1×1	1.00	573	$u = 13$	1.00	522	242
S2	304	1×1	1.00	668	$u = 6$	1.00	668	304
S3	344	1×1	1.00	531	$u = 1$	1.00	432	344
I	198	1×1	1.00	489	$u = 7$	1.00	489	198

Table H.3: **Best row segmented diagonal + register blocking performance, compared to register blocking only: Power4**. Row segmented diagonal data for Figure 5.17.

	Register Blocking			RSDIAG-based splitting				1×1
	Mflop/s	$r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Tuning Param	(stored nz) / (ideal nz)	Mflop/s	Mflop/s
11	620	4×2	1.70	549	$u = 13$	0.24	510	272
					4×2	0.77	624	
S1	187	4×1	2.80	402	$u = 16$	1.00	402	150
S2	277	4×1	2.00	500	$u = 16$	1.00	500	204
S3	326	3×2	1.94	644	$u = 4$	1.00	644	289
F	130	1×1	1.00	154	$u = 16$	0.60	154	130
					2×4	0.40	184	
G	318	2×2	1.59	572	$u = 4$	1.00	572	258
H	156	2×2	2.00	262	$u = 16$	1.00	257	117
I	139	2×1	1.75	223	$u = 15$	1.00	223	100

Table H.4: **Best row segmented diagonal + register blocking performance, compared to register blocking only: Itanium 2**. Row segmented diagonal data for Figure 5.18.

Appendix I

Supplemental Data on $A^T A \cdot x$

I.1 Deriving Cache Miss Lower Bounds

Below, we derive each of the 2 cases given in Section 7.2.2. We assume the same notation. To simplify the discussion, let $l_i = 1$; the case of $l_i > 1$ reduces each of the miss counts below by a factor of $\frac{1}{l_i}$, as shown in Equation (7.9).

1. $\hat{W} + \hat{V} \leq C_i$: *The total working set fits in cache.*
   In this case, there is sufficient cache capacity to hold both the matrix and vector working sets in cache. Therefore, we incur only compulsory misses: 1 miss for each of the $\frac{m}{r}\hat{W}$ matrix data words, and 1 miss for each of the $2n$ vector elements (x and y).

2. $\hat{W} + \hat{V} > C_i$: *The total working set exceeds the cache size.*
   To obtain a lower bound in this case, suppose (1) the cache is fully associative, and (2) we have complete control over how data is placed in the cache. Suppose we choose to devote a fraction α of the cache to the matrix elements, and a fraction $1 - \alpha$ of the cache to the vector elements. The following condition ensures that α lies in a valid subset of the interval $[0, 1]$:

   $$\max\left\{0, 1 - \frac{\hat{V}}{C_i}\right\} \leq \alpha \leq \min\left\{\frac{\hat{W}}{C_i}, 1\right\} \quad .$$

   (The case of α at the lower bound means that we devote the maximum possible number of elements to the vector working set, and as few as possible to the matrix working set. When α meets the upper bound, we devote as many cache lines as possible to the matrix and as few as possible to the vector.)

First consider misses on the matrix. In addition to the $\frac{m}{r}\hat{W}$ compulsory misses, we incur capacity misses: for each block row of A, each of the $\hat{W} - \alpha C_i$ words exceeding the alotted capacity for the matrix will miss. Summing the capacity misses over all $\frac{m}{r}$ block rows and adding the compulsory misses, we find $\frac{m}{r}\hat{W} + \frac{m}{r}\left(\hat{W} - \alpha C_i\right)$ misses to the matrix elements.

A similar argument applies to the x and y vectors. There are $2n$ compulsory misses and, for each block row, $\hat{V} - (1-\alpha)C_i$ capacity misses, or $2n + \frac{m}{r}\left[\hat{V} - (1-\alpha)C_i\right]$ misses in all.

Thus, a lower bound on cache misses in this case is

$$\begin{aligned} M_i &\geq \frac{m}{r}\hat{W} + \frac{m}{r}(\hat{W} - \alpha C_i) + 2n + \frac{m}{r}\left[\hat{V} - (1-\alpha)C_i\right] \\ &= \frac{m}{r}\hat{W} + 2n + \frac{m}{r}(\hat{W} + \hat{V} - C_i) \end{aligned}$$

which is independent of how cache capacity is allocated among the matrix and vector data, *i.e.*, independent of α.

We can further refine the bounds by considering each block row individually, *i.e.*, taking $\hat{W}$ and $\hat{V}$ to be functions of the non-zero structure of the actual block row. Such refinement would have produced tighter upper bounds.

I.2 Tabulated Performance Data

Tables I.1–I.4 list the block sizes and corresponding performance values and measurements for Figures 7.6–7.9. In particular, each table shows the following data:

- **Best cache optimized, register blocked** block size ($r_{\text{opt}} \times c_{\text{opt}}$) and performance: Best block size and corresponding performance based on an exhaustive search over block sizes.
- **Heuristic cache optimized, register blocked** block size ($r_{\text{h}} \times c_{\text{h}}$) and performance: Block size chosen by the heuristic and its corresponding performance. Items in this column marked with a * show when this choice of block size yields performance that is more than 10% worse than the optimal block size, $r_{\text{opt}} \times c_{\text{opt}}$.
- **Register blocking only** block size ($r_{\text{reg}} \times c_{\text{reg}}$) and performance.

No.	Best cache-opt. + reg. blocking $r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s	Reg. blocking only $r_{\text{reg}}\times c_{\text{reg}}$	Fill	Mflop/s
1	7×7	1.00	91	7×7	1.00	91	8×5	1.00	59
2	8×8	1.00	97	4×8	1.00	91	8×2	1.00	58
3	6×6	1.12	82	6×6	1.12	82	6×6	1.12	50
4	6×3	1.12	80	6×6	1.19	78	6×2	1.13	49
5	4×4	1.00	77	4×4	1.00	77	4×4	1.00	50
6	3×3	1.00	79	3×3	1.00	79	3×3	1.00	50
7	3×3	1.00	82	3×3	1.00	82	3×3	1.00	51
8	6×2	1.13	87	6×6	1.15	83	6×6	1.15	50
9	3×3	1.02	79	3×3	1.02	79	3×3	1.02	50
10	2×2	1.21	60	5×2	1.58	56	2×2	1.21	35
11	2×2	1.23	51	2×2	1.23	51	2×2	1.23	31
12	2×2	1.24	59	3×2	1.36	58	3×2	1.36	36
13	3×2	1.40	57	3×2	1.40	57	3×2	1.40	35
15	2×1	1.00	50	2×1	1.00	50	2×1	1.00	33
17	2×1	1.36	45	2×1	1.36	45	1×1	1.00	28
21	2×1	1.38	41	2×1	1.38	41	1×1	1.00	28
25	2×1	1.71	28	1×1	1.00	28	1×1	1.00	21
27	2×1	1.53	33	1×1	1.00	31	1×1	1.00	21
28	1×1	1.00	35	1×1	1.00	35	1×1	1.00	26
36	1×1	1.00	27	1×1	1.00	27	1×1	1.00	18
40	1×1	1.00	34	1×1	1.00	34	1×1	1.00	27
44	1×1	1.00	28	1×1	1.00	28	1×1	1.00	21

Table I.1: **Block size summary data for the Sun Ultra 2i platform**. An asterisk (∗) by a heuristic performance value indicates that this performance was less than 90% of the best performance.

- **Cache optimized, register blocked** implementation using the same block size, $r_{\text{reg}}\times c_{\text{reg}}$, as in the register blocking only case. Items in this column marked with a ∗ show when this choice of block size yields performance that is more than 10% worse than the optimal block size, $r_{\text{opt}}\times c_{\text{opt}}$. That is, marked items show when the sparse $A^TA\cdot x$ (SpA^TA)-specific heuristic makes a better choice than using the optimal block size based only on SpMV performance.

No.	Best cache-opt. + reg. blocking $r_{opt}\times c_{opt}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_h\times c_h$	Fill	Mflop/s	Reg. blocking only $r_{reg}\times c_{reg}$	Fill	Mflop/s
1	8×4	1.00	121	8×4	1.00	121	6×2	1.00	89
2	4×2	1.00	138	8×4	1.00	121*	4×8	1.00	84
3	3×6	1.12	121	3×3	1.12	120	6×2	1.12	79
4	3×3	1.06	124	3×3	1.06	124	6×2	1.13	78
5	4×2	1.00	116	4×2	1.00	116	4×2	1.00	76
6	3×3	1.00	132	3×3	1.00	132	3×3	1.00	79
7	3×3	1.00	134	3×3	1.00	134	3×3	1.00	80
8	3×3	1.11	119	3×3	1.11	119	6×2	1.13	79
9	3×3	1.02	124	3×3	1.02	124	3×3	1.02	78
10	4×2	1.45	93	4×2	1.45	93	4×2	1.45	56
11	2×2	1.23	79	2×2	1.23	79	2×2	1.23	50
12	4×2	1.48	94	3×2	1.36	94	3×2	1.36	57
13	3×2	1.40	91	3×2	1.40	91	3×2	1.40	55
14	3×2	1.47	70	3×2	1.47	70	3×2	1.47	44
15	2×1	1.00	68	2×1	1.00	68	2×1	1.00	52
16	4×2	1.66	65	4×1	1.43	63	4×1	1.43	45
17	3×1	1.59	66	4×1	1.75	66	6×1	1.98	44
18	2×1	1.36	41	2×1	1.36	41	2×1	1.36	31
20	1×2	1.17	64	1×2	1.17	64	1×2	1.17	42
21	3×1	1.59	64	4×1	1.77	64	5×1	1.88	42
23	2×1	1.46	45	1×1	1.00	44	2×1	1.46	31
24	1×1	1.00	55	1×1	1.00	55	2×1	1.52	36
25	1×1	1.00	42	1×1	1.00	42	1×1	1.00	30
26	1×1	1.00	41	1×1	1.00	41	1×1	1.00	28
27	1×1	1.00	45	1×1	1.00	45	2×1	1.53	32
28	1×1	1.00	52	1×1	1.00	52	1×1	1.00	35
29	2×2	1.98	42	1×1	1.00	42	1×1	1.00	28
36	1×1	1.00	37	1×1	1.00	37	1×1	1.00	27
37	2×2	1.98	43	1×1	1.00	43	1×1	1.00	28
40	1×1	1.00	48	1×1	1.00	48	1×1	1.00	35
41	1×1	1.00	34	1×1	1.00	34	1×1	1.00	28
42	1×1	1.00	34	1×1	1.00	34	1×1	1.00	28
44	1×1	1.00	29	1×1	1.00	29	1×1	1.00	26

Table I.2: **Block size summary data for the Intel Pentium III platform**. An asterisk (*) by a heuristic performance value indicates that this performance was less than 90% of the best performance.

No.	Best cache-opt. + reg. blocking $r_{\text{opt}} \times c_{\text{opt}}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_{\text{h}} \times c_{\text{h}}$	Fill	Mflop/s	Reg. blocking only $r_{\text{reg}} \times c_{\text{reg}}$	Fill	Mflop/s
1	4×4	1.00	260	4×4	1.00	260	2×4	1.00	173
2	4×4	1.00	248	4×4	1.00	248	4×2	1.00	166
4	3×6	1.12	220	3×3	1.06	217	3×2	1.07	151
5	4×4	1.00	223	4×4	1.00	223	4×2	1.00	156
7	3×3	1.00	229	3×3	1.00	229	3×3	1.00	156
8	3×6	1.13	221	2×6	1.13	215	2×2	1.10	143
9	3×3	1.02	231	3×3	1.02	231	3×3	1.02	151
10	2×1	1.10	186	2×1	1.10	186	2×1	1.10	136
12	2×2	1.24	184	2×1	1.13	183	2×1	1.13	136
13	2×1	1.14	179	2×1	1.14	179	2×1	1.14	135
15	2×1	1.00	177	2×1	1.00	177	2×1	1.00	141
40	1×1	1.00	137	1×1	1.00	137	1×1	1.00	124

Table I.3: **Block size summary data for the IBM Power3 platform**. An asterisk (∗) by a heuristic performance value indicates that this performance was less than 90% of the best performance.

I.3 Speedup Plots

Figures I.1–I.4 compare the observed speedup when register blocking and the cache optimization are combined with the product (register blocking only speedup) × (cache optimization only speedup). When the former exceeds the latter, we say there is a synergistic effect from combining the two optimizations. This effect occurs on all the platforms but the Pentium III, where the observed speedup and the product of individual speedups are nearly equal.

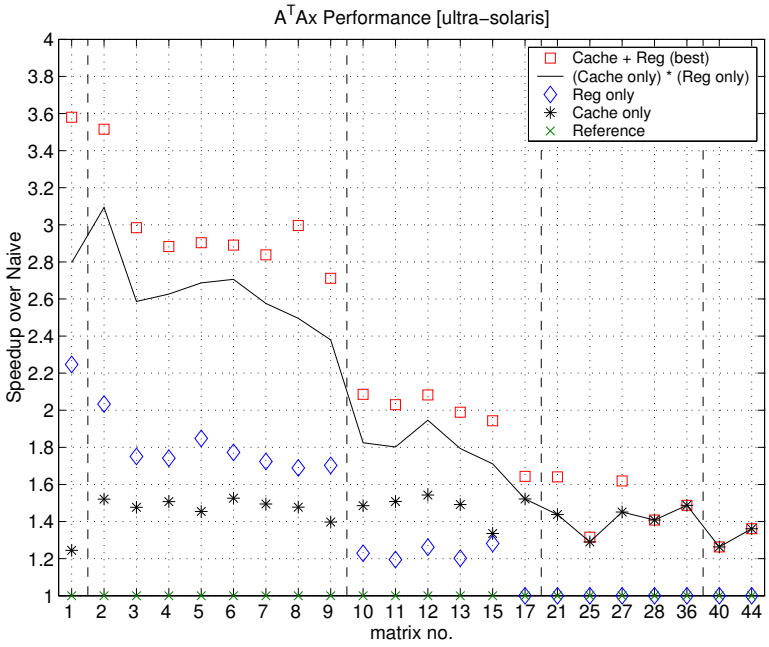

Figure I.1: **Combined effect of register blocking and the cache optimization on the Sun Ultra 2i platform**.

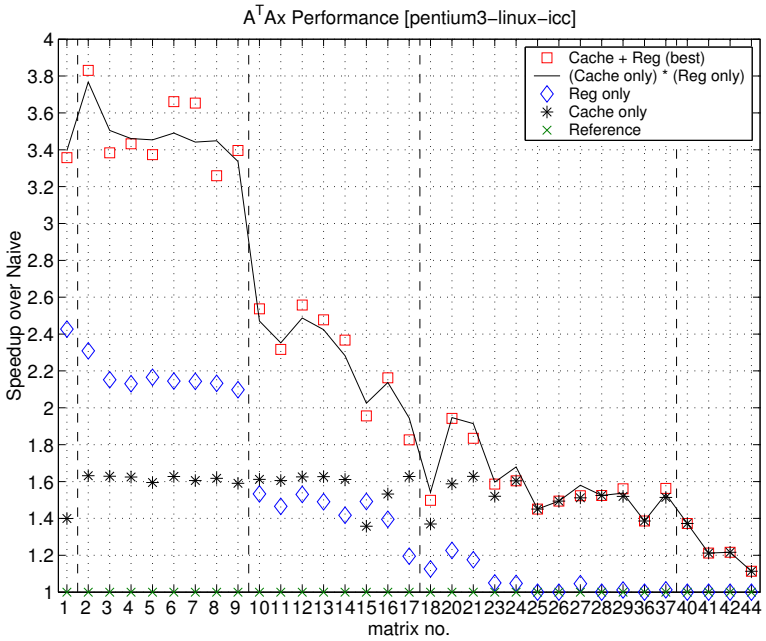

Figure I.2: **Combined effect of register blocking and the cache optimization on the Intel Pentium III platform**. The observed speedup of combining register and cache optimizations equals the product of (cache optimization only speedup) and (register blocking only speedup), shown as a solid line.

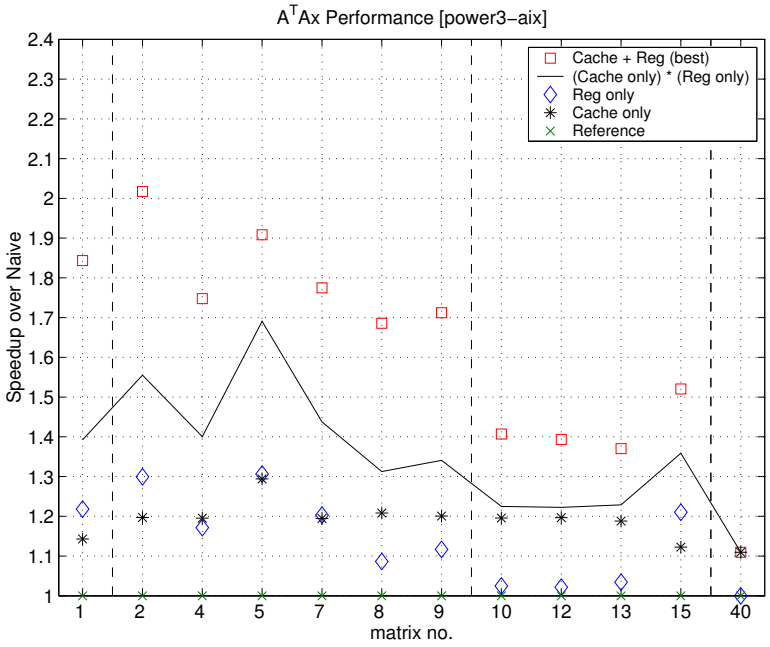

Figure I.3: **Combined effect of register blocking and the cache optimization on the IBM Power3 platform**.

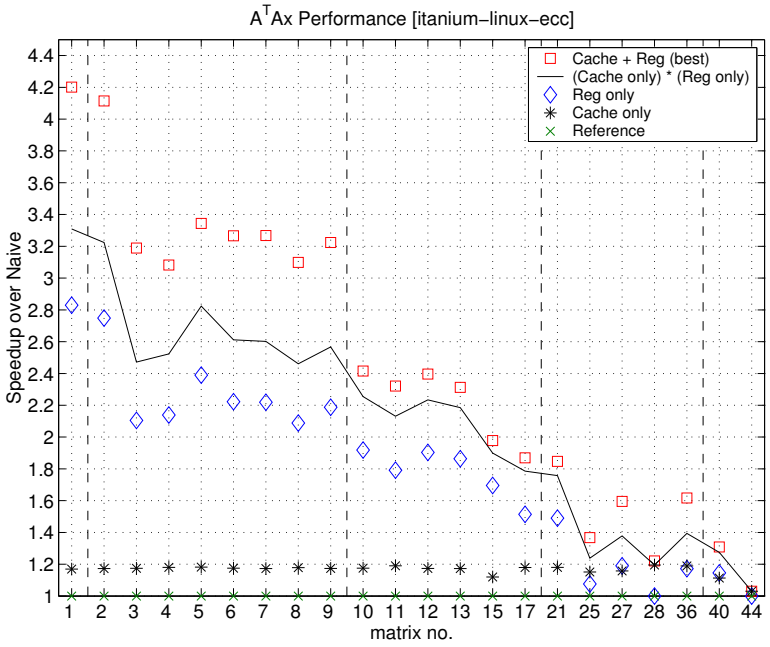

Figure I.4: **Combined effect of register blocking and the cache optimization on the Intel Itanium platform**.

No.	Best cache-opt. + reg. blocking $r_{\text{opt}}\times c_{\text{opt}}$	Fill	Mflop/s	Heuristic cache-opt. + reg. blocking $r_{\text{h}}\times c_{\text{h}}$	Fill	Mflop/s	Reg. blocking only $r_{\text{reg}}\times c_{\text{reg}}$	Fill	Mflop/s
1	8×2	1.00	321	8×2	1.00	321	8×1	1.00	225
2	8×8	1.00	309	8×2	1.00	307	8×1	1.00	211
3	6×6	1.12	237	6×6	1.12	237	3×1	1.06	159
4	3×3	1.06	229	3×3	1.06	229	2×2	1.07	159
5	4×2	1.00	234	4×2	1.00	234	4×2	1.00	168
6	3×3	1.00	245	3×3	1.00	245	3×1	1.00	168
7	3×3	1.00	250	3×3	1.00	250	3×1	1.00	171
8	6×6	1.15	233	6×6	1.15	233	3×1	1.06	158
9	3×3	1.02	249	3×3	1.02	249	3×1	1.01	169
10	4×2	1.45	182	4×2	1.45	182	4×1	1.33	144
11	2×2	1.23	155	2×2	1.23	155	2×2	1.23	117
12	4×2	1.48	184	4×2	1.48	184	4×1	1.37	145
13	4×2	1.54	177	4×2	1.54	177	4×1	1.40	142
15	2×2	1.35	144	2×2	1.35	144	2×1	1.00	123
17	4×1	1.75	138	4×1	1.75	138	4×1	1.75	112
21	4×1	1.77	129	4×1	1.77	129	4×1	1.77	105
25	3×1	2.37	67	1×1	1.00	56∗	2×1	1.71	52
27	3×1	1.94	82	3×1	1.94	82	3×1	1.94	62
28	2×2	2.54	82	1×1	1.00	81	1×1	1.00	69
36	3×1	2.31	64	2×2	2.31	61	3×1	2.31	46
40	3×1	1.99	96	3×1	1.99	96	3×1	1.99	84
44	1×1	1.00	45	1×1	1.00	45	1×1	1.00	46

Table I.4: **Block size summary data for the Intel Itanium platform**. An asterisk (∗) by a heuristic performance value indicates that this performance was less than 90% of the best performance.

Appendix J

Supplemental Data on $A^{\rho} \cdot x$

Matrix	Reference Mflop/s	$r \times c$	$A \cdot x$ Mflop/s	Sparse Tiled $A^2 \cdot x$ Mflop/s	Sparse Tiled $A^3 \cdot x$ Mflop/s	Sparse Tiled $A^4 \cdot x$ Mflop/s
1	34	6×8	73	73	73	73
2	34	8×8	57	89	102	110
3	34	6×6	49	77	90	98
4	33	6×2	50	72	82	88
5	31	4×4	48	72	83	87
6	34	3×3	50	78	89	95
7	34	3×3	53	75	88	93
8	34	6×6	50	72	83	89
9	34	3×3	52	68	56	53
10	34	2×1	39	50	52	51
11	29	2×2	32	49	55	59
12	33	2×2	38	54	60	61
13	34	2×1	37	55	60	60
15	31	2×1	40	40	40	40
17	32	1×1	32	44	50	53
21	29	1×1	29	45	50	53
25	21	1×1	21	24	25	26
27	20	2×1	22	33	37	39
28	27	1×1	27	34	32	31
36	18	1×1	18	20	20	21
40	32	1×1	32	32	32	32

Table J.1: **Tabulated performance data under serial sparse tiling: Ultra 2i**. The block size is selected and fixed based on the best performance of register blocked SpMV. Columns 2–4 also appear in Appendix D.

Matrix	Reference Mflop/s	$r \times c$	$A \cdot x$ Mflop/s	Sparse Tiled $A^2 \cdot x$ Mflop/s	Sparse Tiled $A^3 \cdot x$ Mflop/s	Sparse Tiled $A^4 \cdot x$ Mflop/s
1	41	2×10	107	107	107	107
2	40	4×2	90	91	90	90
3	40	6×2	82	86	84	82
4	40	3×3	83	113	118	118
5	38	4×2	82	104	111	111
6	40	3×3	88	100	95	91
7	40	3×3	90	95	91	90
8	40	6×2	83	86	83	83
9	40	3×3	88	88	88	88
10	40	4×2	63	69	68	66
11	37	2×2	53	74	80	77
12	40	3×3	63	70	69	69
13	40	3×3	60	67	63	62
15	39	2×1	56	56	56	56
17	39	4×1	47	66	67	62
18	28	2×1	31	33	33	33
20	35	1×2	42	47	46	46
21	38	4×1	44	66	68	64
23	28	2×1	29	36	39	38
24	36	2×1	36	57	61	60
25	30	1×1	30	37	41	43
26	28	1×1	28	34	36	35
27	31	2×1	32	46	52	53
28	37	1×1	37	43	42	41
29	28	2×2	28	36	36	34
36	26	1×1	26	28	31	34
37	28	2×2	28	36	36	34
40	39	1×1	39	45	44	44

Table J.2: **Tabulated performance data under serial sparse tiling: Pentium III**. The block size is selected and fixed based on the best performance of register blocked SpMV. Columns 2–4 also appear in Appendix D.