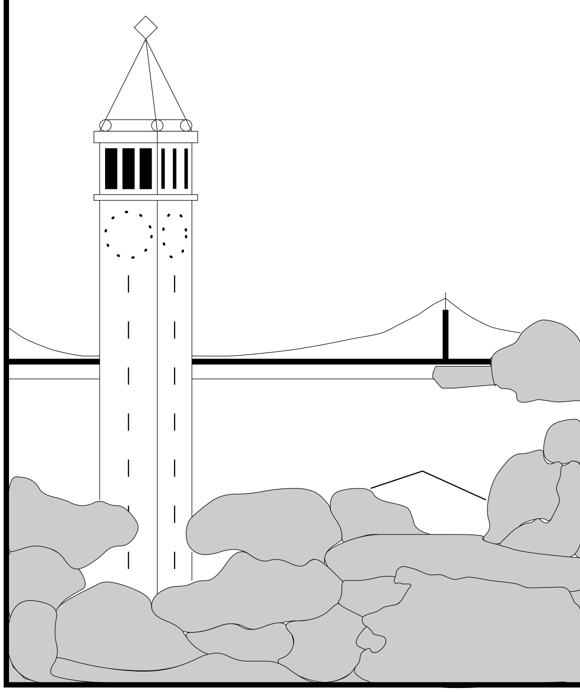
Optimizing the Performance of Sparse Matrix-Vector Multiplication

Eun-Jin Im



Report No. UCB/CSD-00-1104

June 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Optimizing the Performance of Sparse Matrix-Vector Multiplication

by

Eun-Jin Im

Bachelor of Science, Seoul National University, Seoul, 1991
Master of Science, Seoul National University, Seoul, 1993

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 at BERKELEY

Committee in charge:

Professor Katherine A. Yelick, Chair
Professor James W. Demmel
Professor Robert L. Taylor

2000

The dissertation of Eun-Jin Im is approved:

Chair Date

Date

Date

University of California at Berkeley

2000

Optimizing the Performance of Sparse Matrix-Vector Multiplication

Abstract

Optimizing the Performance of Sparse Matrix-Vector Multiplication

by

Eun-Jin Im
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Katherine A. Yelick, Chair

Sparse matrix operations dominate the performance of many scientific and engineering applications. In particular, iterative methods are commonly used in algorithms for linear systems, least squares problems, and eigenvalue problems, which involve a sparse matrix-vector product in the inner loop. The performance of sparse matrix algorithms is often disappointing on modern machines because the algorithms have poor temporal and spatial locality, and are therefore limited by the speed of main memory. Unfortunately, the performance gap between memory and processing is steadily increasing, as processor performance increases by roughly 60% every year, while memory latency drops by only 7%. Performance is also highly dependent on the nonzero structure of the sparse matrix, the organization of the data and its computation, and the exact parameters of the hardware memory system.

This thesis presents a toolkit called Sparsity for the automatic optimization of sparse matrix-vector multiplication. We start with an extensive study of possible memory hierarchy optimizations, in particular, reorganization of the matrix and computation around blocks of the matrix. The research demonstrates that certain kinds of blocking can be effective for both registers and caches, although the nature of that tiling is quite different due to the differences in size between typical register sets and caches. Both types of blocking are shown to be highly effective in some matrices, but ineffective in others, and the choice of block size is also shown to be highly dependent on the matrix and the machine. Thus, to automatically determine when and how the optimizations should be applied, we employ a combination of search over a set of possible optimized versions, along with newly devised

performance models to eliminate or constrain the search to make it practical.

We also consider a common variation of basic sparse matrix-vector multiplication in which a sparse matrix is multiplied by a set of dense vectors. This operation arises, for example, when there are multiple right-hand sides in a linear solver, or when a higher level algorithm has been blocked. The introduction of multiple vectors offers enormous optimization opportunities, effectively changing a matrix-vector operation into a matrix-matrix operation. It is well known that for dense matrices, the latter algorithm has much higher data reuse than the former, and so can achieve much better performance; the same is true in the sparse case.

The SPARSITY system is designed as a web service, so scientists and engineers can easily obtain highly optimized sparse matrix routines without needing to understand the specifics of the optimization techniques or how they are selected.

This thesis also reports on an extensive performance study of over 40 matrices on a variety of machines. The matrices are taken from various scientific and engineering problems, as well as from linear programming and data mining. The machines include the Alpha 21164, UltraSPARC I, MIPS R10000 and PowerPC 604e. These benchmark results are useful for understanding the performance differences across application domains, the effectiveness of the optimizations, and the costs associated with evaluating our performance models in applying the optimizations. The conclusion is that SPARSITY is highly effective, producing routines that are up to 3.1 times faster for a single vector and 6.2 times faster for multiple vectors.

Professor Katherine A. Yelick
Dissertation Committee Chair

To my husband, Yongkyung Kwon,

and my parents,

Sookja Lee and Hyungbin Im

Contents

List of Figures

Acknowledgements

First of all, I would like to deeply thank my advisor Katherine Yelick for her guidance and support through the trials of my graduate study at Berkeley. She has always been available for discussions and provided insightful opinions so that I could pursue my research in the right direction. My thesis research has greatly benefited from her mentorship and her ability to abstract general themes from the specific details of a project.

I am also very grateful to James Demmel for his constant interest in my thesis topic and advice. Research done by Professor Demmel and PHiPAC group provided motivation and insight for this research. I am also grateful to Robert Taylor for reading my thesis at very short notice.

I would like to thank to Xiaoye Li, Mark Adams, Inderjit Dhillon, and Osni Marques for their valuable inputs and for providing sparse matrices for my research. I would also like to thank my fellow graduate students, Ben Liblit, Arvind Krishnamurthy, Rich Vuduc and Randi Thomas.

I deeply thank my dear family; my parents, father-in-law, sisters, brother and brother-in-law. And I truly thank my husband, Yongkyung Kwon, for his love, understanding, patience and constant support throughout my graduate study.

Finally, I acknowledge support from following research grants. This work was supported in part by the Department of Energy contract no. W-7405-ENG-48 through Memorandum Agreement no. B504962 and B244823 with Lawrence Livermore National Laboratory, DOE grant no. DE-FG03-94ER25206, National Science Foundation grant no. CCR-9712410, NSF Infrastructure grant no. CDA-9401156, Army Research Office grant nos. DAAH04-96-1-0079 and DAAG55-98-1-0153, Defense Advanced Research Projects Agency grant no. F30602-95-C-0136 and contract no. F30602-95-C0014, by grants from the IBM Shared University Research (SUR) program and Intel, and by gifts from Sun Microsystems and Hewlett-Packard. This project also utilized the resources of the National Energy Research Scientific Computing Center which is supported by the Office of Energy Research of the U.S. Department of Energy. The information presented here does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be

inferred.

Chapter 1

Introduction

Matrix-vector multiplication is an important computational kernel used in scientific computation, signal and image processing, document retrieval, and many other applications. This operation is often used in iterative solvers for linear systems, in explicit methods for ordinary differential equations, and in eigenvalue computations, just to name a few. Given a sparse matrix A and dense vectors x and y, the problem is to compute $y = A \times x + y$. In many cases, the matrices are sparse, meaning that most of elements are zero. In those cases, only the nonzero elements are stored with extra information regarding the position of the elements in the matrix.

The performance of sparse matrix operations tends to be much lower than their dense matrix counterparts, because the memory access patterns are irregular, and there is more overhead for manipulating the data structure representation. For example, on a 167 MHz UltraSPARC I, the naive implementation of sparse matrix-vector multiplication runs at 25 Mflops for a 1000×1000 dense matrix represented in compressed sparse row format. For comparison, a naive implementation of dense matrix-vector multiplication runs at 38 Mflops on the same machine, and the vendor-optimized routine runs at 57 Mflops. The performance is heavily dependent on the nonzero structure of the matrix, and can be as low as 5 Mflops for matrices with a lower ratio of nonzero elements. The primary reason for this performance difference is poor data locality in access to the source vector x in the sparse case.

Matrix-vector multiplication is a particularly challenging algorithmic kernel. Since each matrix entry is used only once, the ratio of floating point operations to memory operations is low. This is true for dense as well as sparse matrices. Using terminology from the well-known Basic Linear Algebra Subprograms (BLAS) standard [22], matrix-vector operations make up the BLAS-2 class, while matrix-matrix operations make up the BLAS-3 class. Dense BLAS-2 operations perform $O(n^2)$ operations on $O(n^2)$ data ($n \times n$ square matrices), while dense BLAS-3 operations perform $O(n^3)$ operations on $O(n^2)$ data. The potential for $n = n^3/n^2$ fold data reuse in BLAS-3 operations versus little or no reuse in BLAS-2 operations leads to real performance advantages in practice. For example, on the same UltraSPARC machine described above, the vendor-supplied dense matrix-matrix multiplication routine runs at 287 Mflops, 5 times faster than the matrix-vector performance. The reuse of data in sparse matrices is more difficult to analyze, because it depends on the sparsity structure of the matrix, but as we will show, the move from BLAS-2 to BLAS-3 in the sparse setting also has a huge potential performance advantage.

The focus of this thesis is on memory hierarchy optimizations for sparse matrix-vector multiplication. Like dense matrices, these optimizations depend heavily on details of the machine microarchitecture, i.e., the size and speed of register set, caches, and memory. Unlike the dense problems, they also depend on the structure of the matrix, because the distribution of nonzero elements in sparse matrix determines the memory access pattern. As a result, much of the optimization must be done dynamically, using either the matrix of interest or one that reflects the sparsity patterns of the application domain. For example, a structural engineer may use finite-element modeling for many different structures; although the structures may change, the number of degrees of freedom in the simulation may be common across many simulations. The number of degrees of freedom affects the density pattern in the matrix, producing small dense sub-blocks of a particular size, which are important in optimizing the matrix. Similarly, a document retrieval application for the web may construct a matrix of documents and keywords on the web. Our experience with such a matrix indicates a nearly random pattern of nonzeros in the matrix, which is very large and rectangular. Although the specifics of the nonzero pattern and size may change every time the web is searched, the pattern is likely to have similar characteristics and the size will change relatively more slowly over time. Thus, our approach is to take certain characteristics of the matrix into account when performing optimizations, but the resulting code will run well on a class of matrices with similar characteristics.

Two recent trends make the research presented in this thesis particularly timely. First, the gap between processor speed and main memory speed grows every year, with the results that a memory operation on some machines is already equivalent to tens or hundreds of floating point operations. The architectural solution to this problem has been to increase the number of levels of caches and other memory hierarchy structures. The result is machines that are very difficult to understand and model for the purpose of optimization. A second trend is the realization within the scientific computing community of the need for sparse matrix libraries which are highly optimized, just as the dense matrix libraries have been for many years. In particular, a standardization effort called the BLAS Technical Forum is working on a design for a sparse BLAS interface. They have already identified the need for sophisticated runtime optimization for sparse matrix-vector multiplication, and therefore allow users to provide hints at matrix construction time about how the matrix will be used. This approach fits well with our approach, which will automatically optimize matrix-vector multiplication as long as the one-time analysis overhead can be amortized over many operations.

In this thesis we present an optimization framework, called SPARSITY, to generate highly optimized sparse matrix kernels based on information about the matrix structure and target microarchitecture. In particular,SPARSITY generates code for matrix-vector multiplication, where the matrix is sparse and there may be one or more dense vectors. The toolbox takes information about the matrix structure either from the user or from an example matrix that reflects the structure of a set of problems the user intends to run. In general, SPARSITY uses reorganization of the matrix data structure and computational order to improve locality at both the register and cache level. *Register blocking* identifies dense sub-blocks in the matrix and stores these contiguously in memory. To save indexing and branch overhead, a uniform register block size is used across the matrix, which means that zero values may have to be filled in to create some of the blocks. A matrix with two or more natural register block sizes can be written as the sum of two or more matrices, each with a single, natural register block size, and register blocking can be applied to each independently. *Cache blocking* uses a sparse data structure throughout the matrix, but stores the each block contiguously. A generalization of matrix-vector multiplication involves a matrix times a set of vectors. We devise a *multiple-vector optimization* that can be combined with either register or cache blocking. This optimization is useful for applications in which a higher level algorithm has been blocked or there are multiple right-hand sides in solving a system of equations. In some cases, the application admits only a small number of vectors, but in other cases, there are many vectors, and the optimized code groups them into smaller sets.

Both register and cache blocking improve memory system performance for certain matrices and machines if the block sizes are chosen carefully. Similarly, adding multiple vectors can improve performance if the right number of vectors is used. The main challenges in this kind of optimization framework are to determine when the optimizations should be applied and what optimization parameters should be used. We explore a large space of possible techniques, including both searching over a set of parameters on the machine and matrix of interest and performance models to predict which parameter settings will perform well. For setting the register block size, we present a performance model based on some matrix-independent machine characteristics, combined with an analysis of blocking factors that is computed by a statistical sampling of the matrix structure. The model works well in practice and eliminates the need for a large search. For choosing the optimal number of vectors, in applications where multiple vectors are used, we also devise a method for choosing the block size automatically. This heuristic works well for many matrices, but for some, we find that searching over a relatively small number of vectors produces much better results. For cache blocking, we find that search over a limited set of sizes is effective, because a block size that is slightly smaller than the optimal size will still perform well.

For each of these optimizations, we present performance data on a large matrix benchmark suite. The matrices are taken from fluid dynamics, structural modeling, chemistry, economics, circuit simulation, device simulation, linear programming, and document retrieval. In addition, we include one dense matrix in sparse format and one randomly generated sparse matrix, which serve as two extremes on the spectrum of regularity in the memory access patterns.

Our performance study confirms our conjecture that the absolute performance and the effectiveness of each optimization is highly dependent on the matrix and the machine. Roughly speaking, register blocking is most effective on problems that have more structure, whereas cache blocking is most effective on matrices that are very large and have a nearly random structure. For example, register blocking is very effective on finite element problems with multiple degrees of freedom, while cache blocking is effective on a matrix taken from a web search application. Because the two kinds of matrices are so different, we find little benefit from combining the two optimizations. However, if multiple vectors are available in the application, there is an additional payoff to optimizing for them, whether register blocking, cache blocking, or neither is used. For this optimization, we study two additional matrices in detail that come from applications involving multiple vectors.

This thesis makes several contributions in the areas of memory hierarchy optimizations, understanding of sparse matrix computations, and the effectiveness of memory systems designs for irregular access patterns. We summarize these results as follows:

- We introduce optimization techniques for registers, caches, and multiple vectors. Variations of the first two have been described in the literature, but our research includes a more thorough study of various data structure representations and computational organizations to achieve the optimization. We also introduce a third type of optimization for multiple vectors.

- We present a comprehensive study of the performance of a large set of matrices across machines. The machine platforms are the UltraSPARC I, the MIPS R10000, the Alpha 21164, and the IBM PowerPC 604e. This data is useful in understanding the performance of sparse matrix applications on these machines and others.

- Using the same matrices and machines, we demonstrate the effectiveness of each kind of optimization.

- We describe automatic and semi-automatic techniques for choosing optimizations and their parameters. In particular, we define a new performance model for selecting a register block size given some machine characteristics and the sparsity structure of a matrix.

- We describe a new system, SPARSITY, for automatic tuning and code generation of sparse matrix kernels, specifically matrix-vector multiplication. SPARSITY is designed as a hands-on web-service for users who want fast sparse matrix-vector multiplication without the need to understand the details of the optimization techniques.

The remainder of the dissertation is organized as follows. Background information is supplied in chapter 2. This includes a summary of the memory access patterns in matrix-vector multiplication, the characteristics of the 46 matrices in our benchmark suite, and a description of the 4 machines in our performance studies. Chapters 3 and 4 present optimizations for register blocking and cache blocking, along with performance analysis and techniques to determine the block sizes. Chapter 5 extends these two techniques for multiple vectors, and looks at the problems of choosing the right number of vectors. Chapter 6 describes the SPARSITY system for generating optimized sparse matrix-vector multiplication.

Several design choices are also discussed, and there is further discussion of the interface for using the system as a web application. Chapter 7 discusses related work, and chapter 8 draws some conclusions from our results and presents some ideas on future direction of the research.

Chapter 2

Background

In this chapter, we present background information for our research. In section 2.1 we describe the data access patterns that arise in sparse matrix-vector multiplication to provide a foundation for understanding how memory hierarchy optimizations may be done. In section 2.2 we give an overview of the machines that will be used in our performance studies, focusing particularly on the memory hierarchy characteristics such as cache size. Finally, we introduce our benchmark suite of 46 matrices in section 2.3, which are taken from a diverse set of applications and have very different characteristics in terms of the size, percentage of nonzeros, and regularity of nonzero patterns. All of these will be important in understanding the performance studies in the chapters that follow.

2.1 Locality in Matrix-Vector Multiplication

We begin by describing the potential for data reuse in dense matrix-vector multiplication, and then move to the more relevant case of sparse matrix-vector multiplication. We present the dense case because it is simpler, and the contrast is useful in understanding the locality issues in the sparse case. Finally, we will describe how the memory locality issues change when there are multiple vectors.

2.1.1 Dense Matrices

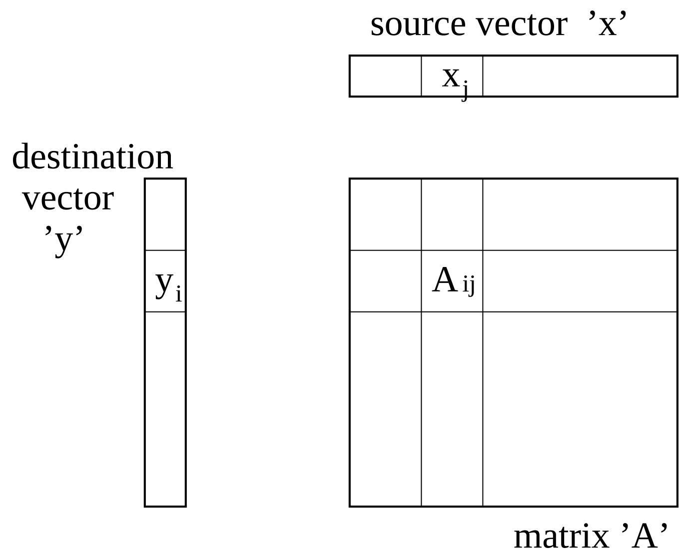
A common optimization for dense matrix computations is to reorganize the computation to improve register or cache reuse by computing on a small block of the array before moving onto the next. This optimization is commonly known as *blocking* or *tiling* and can be performed either by an optimizing compiler or a library [44, 71, 26, 50, 19, 14, 15, 17, 22, 1].

Even for dense matrix-vector multiplication, the potential for reuse is relatively low, because there are only two floating point operations for every element of the matrix. Figure 2.1 shows the basic data structures in computing $y = A \times x$. If A is stored by rows, then a natural way to perform the computation is to compute the dot product of x with each row of A, storing the result in y. In some computations there is an additional scaling constant or a vector to add to the product, but these have relatively minor effects on the memory hierarchy. Therefore, we will limit discussion to the basic matrix-vector product for simplicity. The elements of A are used only once, so there is no opportunity for temporal locality improvements, and spatial locality is already optimal, since the elements are accessed in sequential memory order. Each element of the destination vector is read at the beginning of the dot product and written at the end, which is also optimal. The only potential for locality optimizations are in the source vector. In a dense matrix there is good spatial locality, because source elements are accessed in order, but no reuse unless the rows are short enough that source elements from a previous row's computation are still resident in register or cache when the next dot product is computed. The algorithm can be blocked or tiled to increase reuse in the source vector, although this has an immediate trade-off with increased memory operations on the destination vector.

2.1.2 Sparse Matrices

Many applications generate and use matrices in which most of the elements are zeros. While the dense matrix is simply represented as a two-dimensional array, these sparse matrices are stored only with nonzero elements along with an additional data structure which stores information regarding the location of each nonzero element in the matrix.

The formats of sparse matrices are diverse, and they are discussed in more detail in chapter 7. Among the suggested nine formats in the BLAS Technical Forum standardization [11] we focus on compressed sparse row (CSR) format in our study because it is general and relatively efficient. As shown in the example in figure 2.2, in CSR format, the nonzero elements are stored in rows in the *value* array along with the matching column index in the *col_idx* array. The beginning of each row is pointed to by values in the *row_start* array, in which the last entry points to the end of the last row.



$$y_i = \sum_{j=1}^{n} A_{ij} x_j$$

Figure 2.1: Basic data structures in matrix-vector multiplication

While there is no data reuse in matrix elements in dense matrix-vector multiplication, the memory accesses are sequential for all data structures, providing good spatial locality, which helps in machine structures such as caches. Sparse matrix-vector multiplication is more difficult because the data structure representing the sparse matrix involves indirection, and the source vector is not accessed sequentially. So the performance of sparse matrix-vector multiplication is generally worse than that of dense matrix-vector multiplication. The common factors that affect the performance of dense or sparse matrix- vector multiplication are the size of the matrix and the machine architecture. In addition, the performance of sparse matrix-vector multiplication also depends on the nonzero structure of the sparse matrix, because the way in which the source vector is accessed depends on the location of nonzero elements in the sparse matrix.

2.1.3 Using Multiple Vectors

Some applications of matrix-vector multiplication involve a set of vectors, rather than a single vector, either because the algorithm has been blocked at some higher level or because there are multiple right-hand sides in a system of equations. This turns matrix-

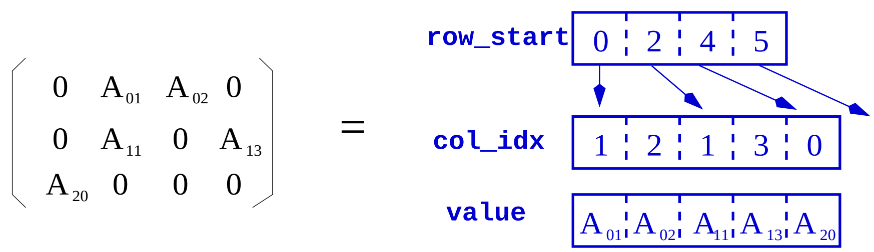


Figure 2.2: Compressed Sparse Row (CSR) representation of a sparse matrix.

```
void smvm (int m, double *value,
           int *col_idx, int *row_start,
           double *x, double *y){
  int i,j;

  for (i=0;i<m;i++) {
    for (j=row_start[i];j<row_start[i+1];j++){
      y[i] += (*value++)*x[*col_idx++];
    }
  }
}
```

Figure 2.3: **Basic sparse matrix-vector multiplication code**

vector multiplication into something closer to matrix-matrix multiplication, where the first matrix is sparse and the second one (which may have only a few columns) is dense. This raises the potential for high performance, because each matrix element will now be reused as many times as there are vectors in the set.

2.2 Processor Architectures

Our experimental data uses the following four processors: UltraSPARC I, MIPS R10000, Alpha 21164, and PowerPC 604e.

UltraSPARC I The UltraSPARC-I microprocessor is a quad-issue superscalar processor implementing the SPARC V9 64-bit RISC architecture. Its floating point unit is composed of five separate pipelined functional units with 32 double-precision floating point registers. The processor has a 16 KB direct-mapped, write-through non-allocating on-chip level 1 (L1) data cache and a 16 KB two-way set-associative instruction cache. The data cache is organized as 512 lines with two 16-byte sub-blocks of data per line. In our experiments, we will use a 167 MHz UltraSPARC I with 512 KB off-chip level 2 (L2) data cache and 512 MB main memory.

MIPS 10000 The MIPS R10000 microprocessor is a quad-issue superscalar processor implementing the 64-bit MIPS IV instruction set architecture. It has 5 separate execution units (2 integer unit, 2 floating point, and one load/store unit) with 32 double-precision floating point registers (64 physical registers that are accessed by register renaming). In addition to two primary floating point units (an adder and multiplier), it has 2 more secondary floating point units which handle long-latency operations such as division and square root. The processor has a 32 KB 2-way set associative on-chip L1 data cache and a 32 KB 2-way set associative on-chip L1 instruction cache. In the experiments, we use a 200 MHz R10000 processor with a 2 MB unified L2 instruction/data cache.

Alpha 21164 The Alpha 21164 is also a quad-issue superscalar processor. It has two integer units and two floating point units (an adder and multiplier), with 32 integer registers and 32 double-precision floating point registers. It has an 8 KB direct-mapped on-chip L1 data cache, and the same for an on-chip instruction cache. Note that the L1 cache is much

Processor	Clock (MHz)	Number of issues/cycle	Number of fp registers	L1 cache size		L2 cache	L3 cache
				Data	Instr.		
UltraSPARC I	167	4	32	16KB	16KB	512KB	
MIPS 10000	200	4	32	32KB	32KB	2MB	
Alpha 21164	533	4	32	8KB	8KB	96KB	4MB
PowerPC 604e	200	4	32	32KB	32KB	512KB	

Figure 2.4: **Summary of processor architectures**

smaller than the other machines, but there is an additional 96KB 3-way set associative on-chip L2 cache which stores both data and instructions. In our experiments, we will use a 533 MHz 21164 with a 4 MB L3 cache that is off chip.

PowerPC 604e The PowerPC 604e microprocessor is a 32-bit implementation of the PowerPC family of RISC microprocessors. The PowerPC 604e is another a quad-issue superscalar processor. It has 7 functional units, consisting of 3 integer units, 1 floating point unit, and 3 load/store units. The architecture has 32 integer registers and 32 double-precision floating point registers. It also has 2 32 KB 4-way set associative on-chip L1 caches, one for instructions and one for data. We used 200MHz PowerPC in the experiment with a 512KB L2 unified cache.

Comparisons The hardware characteristics are summarized in figure 2.4. The Alpha 21164, which has the highest clock rate by more than a factor of two, has the highest peak performance. It also has the most complex memory hierarchy, with three levels of caches, and as it will be shown that this factor makes performance tuning particularly difficult for this machine. Some of our experiments that require very large memory allocations will use only the UltraSPARC and MIPS R10000; for these two machines, the MIPS has a somewhat faster clock rate, but more importantly it has much larger caches for both L1 and L2. In addition, the MIPS has much slower memory access if one misses in both caches: 589 nanoseconds on the MIPS compared to 268 on the UltraSPARC. The MIPS R10000 is likely to get more speedup from the memory hierarchy optimizations because its worst case memory performance is not as good as that of the UltraSPARC.

2.3 Matrix Benchmark Suite

We will use a set of 46 matrices for the performance measurements of the optimizations. The size, number of nonzero elements, and application area of each matrix in the set is summarized in figure 2.5. The matrices are numbered from 1 to 46 in the first column, and the matrices will be referenced by this line number throughout this thesis. Most of the sparse matrices are collected from Tim Davis's matrix collection [20] at the University of Florida. For comparison, we have also included the set of sparse matrices used in Xiaoye Li's thesis [47] in our set. In her thesis, the matrices are used for hand-coded memory hierarchy optimizations for LU factorization. The first matrix is a 1000×1000 dense matrix, which was also included in her set. We also added a synthetic 10000×10000 matrix (matrix 46) with randomly distributed nonzeros and sparsity of 0.15 %.

We have placed the matrices in the table according to our understanding of the application domain from which is was derived. Matrix 1 is a dense matrix. Matrices 2 through 17 are from Finite Element Method (FEM) applications, which in several cases means there are dense sub-locks within much of the matrix. Note however, that the percentage of nonzeros is still very low, so these do not resemble the dense matrix. Matrices 18 through 39 are from structural engineering and device simulation and matrices 40 through 44 are linear programming matrices. Matrix 45 is a matrix built from the existence of keywords within documents on the web (which is processed with an algorithm called Latent Semantic Indexing, or LSI) and matrix 46 is a random matrix with the same density as matrix 45. Most of the matrices are square except matrices 41 through 45, which are linear programming matrices and a document retrieval matrix. All the rectangular matrices has more columns than rows; the number of columns are 5 to 8 times larger than the number of rows for the linear programming matrices and 26 times for the document retrieval matrix.

The matrices are roughly ordered by the regularity of nonzero patterns, with the more regular ones at the top. The distribution of nonzero elements of the matrices in the benchmark suite are shown in appendix A. Note, for example, that the linear programming matrices (40–44) and the document retrieval matrix (45) have much less structure than most of those earlier in the list.

	Name	Application Area	Dimension	Nonzeros	Sparsity
1	dense1000	Dense Matrix	1000x 1000	1000000	100
2	raefsky3	Fluid structure interaction	21200x 21200	1488768	0.33
3	inaccura	Accuracy problem	16146x 16146	1015156	0.39
4	bcsstk35	Stiff matrix automobile frame	30237x 30237	1450163	0.16
5	venkat01	Flow simulation	62424x 62424	1717792	0.04
6	crystk02	FEM Crystal free vibration	13965x 13965	968583	0.50
7	crystk03	FEM Crystal free vibration	24696x 24696	1751178	0.29
8	nasasrb	Shuttle rocket booster	54870x 54870	2677324	0.09
9	3dtube	3-D pressure tube	45330x 45330	3213332	0.16
10	ct20stif	CT20 Engine block	52329x 52329	2698463	0.10
11	bai	Airfoil eigenvalue calculation	23560x 23560	484256	0.09
12	raefsky4	buckling problem	19779x 19779	1328611	0.34
13	ex11	3D steady flow caculation	16614x 16614	1096948	0.40
14	rdist1	Chemical process separation	4134x 4134	94408	0.55
15	vavasis3	2D PDE problem	41092x 41092	1683902	0.10
16	orani678	Economic modeling	2529x 2529	90185	1.41
17	rim	FEM fluid mechanics problem	22560x 22560	1014951	0.20
18	memplus	Circuit Simulation	17758x 17758	126150	0.04
19	gemat11	Power flow	4929x 4929	33185	0.14
20	lhr10	Light hydrocarbon recovery	10672x 10672	232633	0.20
21	goodwin	Fluid mechanics problem	7320x 7320	324784	0.61
22	bayer02	Chemical process simulation	13935x 13935	63679	0.03
23	bayer10	Chemical process simulation	13436x 13436	94926	0.05
24	coater2	Simulation of coating flows	9540x 9540	207308	0.23
25	finan512	Financial portfolio optimization	74752x 74752	596992	0.01
26	onetone2	Harmonic balance method	36057x 36057	227628	0.02
27	pwt	Structural engineering problem	36519x 36519	326107	0.02
28	vibrobox	Structure of vibroacoustic problem	12328x 12328	342828	0.23
29	wang4	Semiconductor device simulation	26068x 26068	177196	0.03
30	lnsp3937	Fluid flow modeling	3937x 3937	25407	0.16
31	lns3937	Fluid flow modeling	3937x 3937	25407	0.16
32	sherman5	Oil reservoir modeling	3312x 3312	20793	0.19
33	sherman3	Oil reservoir modeling	5005x 5005	20033	0.08
34	orsreg1	Oil reservoir simulation	2205x 2205	14133	0.29
35	saylr4	Oil reservoir modeling	3564x 3564	22316	0.18
36	shyy161	Viscous flow calculation	76480x 76480	329762	0.01
37	wang3	Semiconductor device simulation	26064x 26064	177168	0.03
38	mcfe	astrophysics	765x 765	24382	4.17
39	jpwh991	Circuit physics modeling	991x 991	6027	0.61
40	gupta1	Linear programming matrix	31802x 31802	2164210	0.21
41	lpcreb	Linear Programming problem	9648x 77137	260785	0.04
42	lpcred	Linear Programming problem	8926x 73948	246614	0.04
43	lpfit2p	Linear Programming problem	3000x 13525	50284	0.12
44	lpnug20	Linear Programming problem	15240x 72600	304800	0.03
45	lsi	Latent Semantic Indexing	10000x255943	3712489	0.15
46	random	Random Matrix	10000x 10000	150000	0.15

Figure 2.5: **Matrix benchmark suite:** The basic characteristic of each matrix used in our experiments is shown here. The sparsity column is the percentage of nonzeros, which is usually less than 1%.

Chapter 3

Register Blocking Optimization

In this chapter, we present the results of our study on optimization by register blocking. In this optimization, we focus mainly on improving locality at the highest level of the memory, namely register reuse. Register optimizations are different from the other levels of the memory hierarchy in several related aspects. First, the register contents are controlled directly by software, so one may choose an arbitrary set of values to be saved in registers, rather than relying on the mapping of addresses to cache lines, as is typical in cache structures. Although under software control, register allocation is handled by the compiler, as we plan to express our optimizations in C code rather than assembler. Thus, we will not have as much direct control over the way registers are used as we might like in performing the optimizations. Finally, the set of registers is relatively small, which will make certain approaches practical that would not be practical for larger memory structures such as caches.

In section 3.1, we describe the idea and advantage of register blocking and in section 3.2, we present a performance model to determine whether register blocking should be applied for a particular matrix as well as its register block size. We then apply register blocking to a set of test matrices and analyze the performance in section 3.3. There is considerable overhead associated with transforming a matrix for register blocking, meaning that this optimization is recommended when the matrix is to be multiplied sufficiently many times, which is a common case in practice. We quantify that overhead in section 3.4, and a summary of these results is in section 3.5.

3.1 Description of Register Blocking

To optimize for register use, we reorganize the data structure and computation to improve the reuse of values in the source vector. This optimization is commonly known as *blocking* or *tiling* for dense matrix operations [44, 71, 26, 50, 19, 14, 15, 17]. The optimization for dense matrices is relatively simple compared to that of sparse matrices because the block size for a dense matrix is dependent only on the size of the matrix and the machine characteristics, while the block size for a sparse matrix is dependent on the distribution of nonzero elements, in addition to the size of the other factors.

The basic sparse matrix-vector multiplication code for CSR format in figure 2.3 is unlikely to exploit temporal locality of the elements of the source vector x by saving them in registers across rows, in part because the compiler cannot predict the memory accesses statically and also because the set of values needed from x for each row is typically larger than the number of registers. If one identifies small blocks of non-zeros in the matrix, however, and reorganizes the representation to store each of these blocks contiguously, values in the source vector may be reused. The number of registers needed to process an $r \times c$ block of the sparse matrix is at least r values from the destination vector, plus c values from the source vector, plus one register that is used repeatedly for each value in the $r \times c$ block of the sparse matrix.

The code is not only easier to write, but admits more compile-time optimizations such as loop unrolling and software pipelining if the values of r and c are fixed over the entire matrix. Figure 3.1 shows a matrix in compressed sparse block row format, where 2×2 blocks are stored contiguously. The blocked matrix-vector multiplication code becomes a series of dense $r \times c$ matrix-vector multiplications which allow for register reuse; the blocked storage format also saves storage in the col_idx array by a factor of $r \times c$ and turns the loads of indices into simple arithmetic operations, such as integer addition.

We have observed that applications such as Finite Element Methods (FEM) [32] naturally store small dense blocks in sparse matrix when they construct the matrix. However, most matrices do not have uniform block structure throughout, so some zero values are filled in when constructing the blocked representation. In the example in figure 3.1, the original matrix did not have dense 2×2 blocks throughout; in a 2×2 blocked representation, the 0 after a_{00} would be filled in. Such factors mean that we perform more floating point computations for stored zero values in a blocked sparse matrix computation to the CSR

$$A = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & a_{04} & a_{05} \\ a_{10} & a_{11} & 0 & 0 & a_{14} & a_{15} \\ 0 & 0 & a_{22} & 0 & a_{24} & a_{25} \\ 0 & 0 & a_{32} & a_{33} & a_{34} & a_{35} \end{pmatrix}$$

$$row\_start = \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}$$
$$col\_idx = \begin{pmatrix} 0 & 4 & 2 & 4 \end{pmatrix}$$
$$value = \begin{pmatrix} a_{00} & a_{01} & a_{10} & a_{11} & a_{04} & a_{05} & a_{14} & a_{15} & a_{22} & 0 & a_{32} & a_{33} & a_{24} & a_{25} & a_{34} & a_{35} \end{pmatrix}$$

Figure 3.1: **Block compressed sparse row (BSR) storage format:** The elements of each dense 2×2 block are stored contiguously in the *value* array. Only the first column index for each block is stored in *col_idx* array, and *row_start* array points to block row starting positions in the *col_idx* array.

format. Because these operations are not useful, they are not counted in the computation of the performance of sparse matrix-vector multiplication later in this chapter.

The generic code, *bsmvm_generic* in figure 3.2, multiplies a register-blocked sparse matrix with any block size by a vector. The optimized version of this code is shown in figure 3.3 when the block size is 2×2. This optimized code removes some of the branch statements and load stalls by reordering instructions. This improves the performance of register blocked multiplication further. We have written a code generator that produces loop-unrolled codes for given block sizes, which will be described further in chapter 6.

3.2 Determining Register Block Sizes

Register blocking does not always improve performance if the sparse matrix does not have small dense blocks. Even when it has such blocks, the optimizer must pick a good block size for a given matrix and machine. We have developed a performance model that predicts the performance of the multiplication for various block sizes without actually blocking and running the multiplication. The purpose of the performance model is to estimate the performance of many possible block sizes so that a good one may be selected. We will first present the performance model and then describe some practical issues associated with its use in an automatic optimization framework.

```
void bsmvm_generic (int bm, int r, int c,
                    int *row_start, int *col_idx, double *value,
                    double *src, double *dest)
{
  int i, j, ii, jj;

  for (i=0; i<bm; i++,dest+=r){
    for (j=row_start[i]; j<row_start[i+1]; j++,col_idx++,value+=r*c){

      for (ii=0; ii<r; ii++)
        for (jj=0; jj<c; jj+)
          dest[ii] += value[ii*c+jj] * src[(*col_idx)+jj];
    }
  }
}
```

Figure 3.2: **Register-blocked code:** *Row_start*, *col_idx*, and *value* arrays represent the data structure for a register-blocked sparse matrix. *Src* and *dest* represent the multiplier and result vectors, respectively.

3.2.1 A Model for Block Size Selection

There is a trade-off in the choice of block size for sparse matrices. In general, the computation rate will increase with the block size up to some limit at which register spilling becomes necessary. In most sparse matrices, the dense sub-blocks that arise naturally are relatively small: 2×2, 3×3 and 6×6 are typical values. When a matrix is converted to a blocked format, some zero elements are filled in to make a complete $r \times c$ block as shown in figure 3.4. These extra zero values not only consume storage, but tend to increase the number of floating point operations, because they are involved in the sparse matrix computation. (Placing branches in the code to avoid these extra operations proves to provide worse results than computing them.)

We use a simple performance model to determine the block size for a given matrix and machine, based in part on profiling information for that machine. The model has two basic components:

1. An approximation for the Mflop rate of a matrix with a given block size.

2. An approximation for the amount of unnecessary computation that will be performed due to explicitly represented zeros in the blocked matrix.

```
void bsmvm_2x2 (int bm,
                int *row_start, int *col_idx, double *value,
                double *src, double *dest)
{
  int i, j;

  for (i=0; i<bm; i++,dest+=2)
  {
    register double d0, d1;
    d0 = dest[0];
    d1 = dest[1];
    for (j=row_start[i]; j<row_start[i+1]; j++,col_idx++,value+=4)
    {
      d0 += value[0] * src[*col_idx+0];
      d0 += value[1] * src[*col_idx+1];
      d1 += value[2] * src[*col_idx+0];
      d1 += value[3] * src[*col_idx+1];
    }
    dest[0] = d0;
    dest[1] = d1;
  }
}
```

Figure 3.3: **Loop-unrolled, register-blocked code:** *Row_start*, *col_idx*, and *value* arrays represent the data structure for a register-blocked sparse matrix. *Src* and *dest* represent the multiplier and result vectors, respectively.

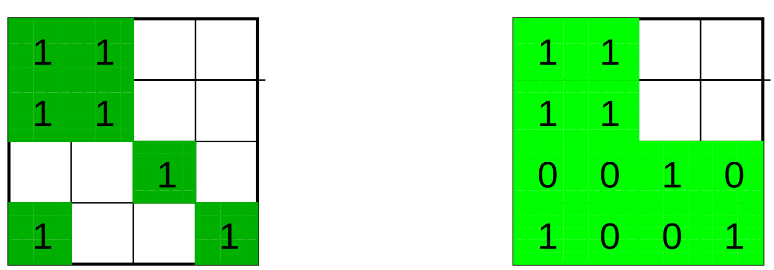


Figure 3.4: **Fill overhead of register blocking:** The BSR format in the right figure stores five extra zero values to make the 2×2 block dense.

The first component cannot be exactly determined without running the resulting blocked matrix (or one with equivalent nonzero structure) on each machine of interest. We therefore use an upper bound for this Mflop rate, which is the performance of a dense matrix stored in the blocked sparse format. The approximation is better for large block sizes than for small ones, as the cost of computing on the dense blocks dominates the other data structure manipulations.

The second component requires the nonzero information about the matrix. However, we prefer to avoid constructing and measuring every blocked variation of the matrix. Even performing this unnecessary computation exactly is very expensive, given that we need to compute this for an entire range of block sizes under consideration. Multiple sweeps over the array or complex data structures must be designed to compute the exact number of zero fills for a set of block sizes. Instead, we develop an approximation that can be done in a single pass over only a subset of the matrix.

Approximating the Performance of a Blocked Sparse Matrix

To approximate the performance of a blocked matrix, we will use a dense matrix in sparse format with the specified number of blocks. While this requires a significant amount of computation to perform for each block size, it will be done once for each machine, not for each matrix.

Figures 3.5 and 3.6 show the performance of sparse matrix vector multiplication for a dense matrix using register-blocked sparse format, (i.e., a dense matrix in sparse format), on different processors: an UltraSPARC I, a MIPS R10000, a Alpha 21164 and a PowerPC 604e. We vary the block size within a range of values for r and c until the performance degrades. Each line shows a particular value of r (number of rows) and each point on the x-axis shows a value for c (number of columns). These graphs are platform dependent. The performance is relatively insensitive to the total matrix size as long as the matrix does not fit in cache but does fit in main memory; the data in the figure uses a 1000×1000 dense matrix.

We make several observations here. First, the performance is fastest on an Alpha, and slowest on a PowerPC. This is because the clock rate of an Alpha is very fast (533 MHz) compared to clock rates of other processors, as shown in figure 2.4. Second, the $1\times$ line, which had no blocking in the vertical direction, is clearly lowest on all machines except

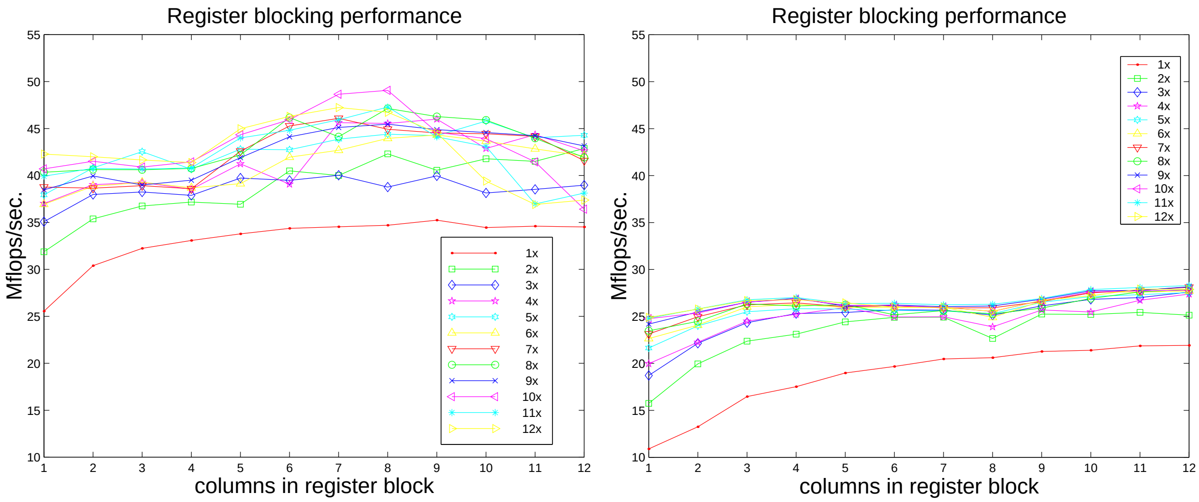


Figure 3.5: **Performance profile of register-blocked code on an UltraSPARC I (left) and a MIPS R10000 (right):** These numbers are taken for a 1000×1000 dense matrix represented in sparse format. Each line is for a fixed number of rows (r), varying the number of columns (c) from 1 to 12.

the Alpha, because there is no reuse in the source vector. Third, on each machine the highest performance is roughly 1.5 to 2 times faster than the lowest performance, which is always the non-blocked performance (block size 1×1). This indicates significant potential for speedup.

We view the absolute performance as an approximate upper bound on the performance of a register-blocked sparse matrix-vector multiplication on each machine, because matrices that are truly sparse are likely to have worse locality at lower levels of the memory hierarchy than a dense matrix in sparse format.

Approximating Fill Overhead

To approximate the unnecessary computation that would result from register blocking, we estimate the number of zeros to be added. This will use a single computation over the matrix of interest, producing an estimate for all block sizes in a particular range.

The number of added zeros in the blocked representation are referred to as *fill*, and the ratio to the original matrix size as *fill overhead*:

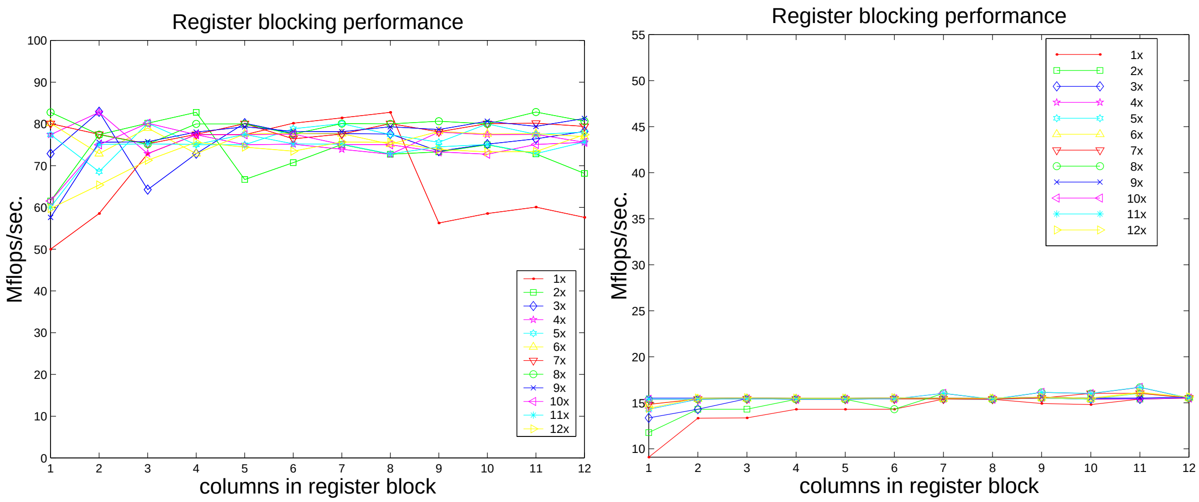


Figure 3.6: **Performance profile of register-blocked code on an Alpha 21164 (left) and a PowerPC 604e (right)** These numbers are taken for a 1000×1000 dense matrix represented in sparse format. Each line is for a fixed number of rows (r), varying the number of columns (c) from 1 to 12.

$$\textit{fill overhead} = \frac{\textit{number of elements stored in register blocked format}}{\textit{number of true nonzeros}}$$

The *fill overhead* is 1 when the matrix has dense blocks of the chosen size spread throughout the matrix; the matrix may still be sparse, but every nonzero is within a block. For sparse matrix-vector multiplication, the number of floating point operations is linear in the number of stored elements (i.e., one multiplication and one addition), so the fill overhead is a good estimate of computation overhead.

We have computed the fill overhead for several benchmark matrices and several block sizes, but the computation time was too high to consider using it as part of an automatic optimization system. To approximate fill overhead, separate computations are made for a column blocking factor and row blocking factor by sampling a fraction of the matrix entries. First, every entry in k rows of the matrix is examined to determine the column blocking factor. (In our implementation, k is chosen to be 100, providing reasonably accurate fill overheads.) A block column index for each element in the matrix was computed by dividing the column index of each nonzero element in the row by the block size in consideration. This gives us an estimate of the total number of blocks for each block size, and thus the fill overhead. For the column block size c, the fill ratio is estimated to be:

$$\textit{estimated fill overhead for column size } c = \frac{\textit{number of blocks} \times c}{\textit{number of nonzero elements in the examined rows}}$$

An estimate of fill overhead for various row sizes is computed independently by an analogous algorithm, namely, examining every 1 out of k columns and computing the number of blocks that would result from a $r \times r$ block for that column.

Instantiating the Model

Given the approximations of performance for a given machine and overhead for a given matrix, we now choose a block size by maximizing the predicted real performance. Since performance was measured in Mflops, real performance was calculated by including only those floating point operations that would have been needed in the unblocked code. Using these estimates, this performance is predicted as:

$$\textit{predicted performance for column size } c = \frac{\textit{performance of a dense matrix in } c \times c \textit{ sparse blocked format}}{\textit{estimated fill overhead for column size } c}$$

$$\textit{predicted performance for row size } r = \frac{\textit{performance of a dense matrix in } r \times r \textit{ sparse blocked format}}{\textit{estimated fill overhead for row size } r}$$

Column blocking factors c in the range from 1 to c_{max} were considered in which c_{max} was the column size with the maximum performance for a dense matrix in sparse-blocked format. Within this range, we chose c by maximizing the predicted performance, and independently did the same to choose the row size r.

It should be noted that the purpose of using the performance model was to find the right block size, rather than to predict the actual performance accurately. The usefulness of the model was validated by running an exhaustive search to find the best performance for a subset of test matrices. Figure 3.7 compares the performance of the block size chosen by the model (in the middle bar) with the block size chosen by an exhaustive search (in the right bar). The ten matrices used in the experiment were matrices 11, 15, 4, 6, 7, 10, 8, 14, 5, and 29 from table 2.5. The block size chosen by the model and that chosen by exhaustive search are same in many cases, and even when they were different (in the second,

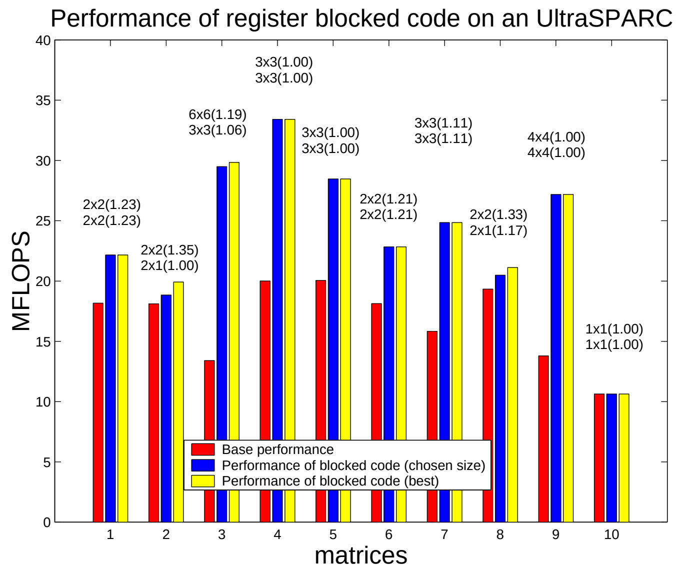


Figure 3.7: **Validation of performance model:** Comparison of the performance of the block size chosen by the performance model (in the middle bar) with the block size chosen by an exhaustive search (in the right bar). on an UltraSPARC I. The left bars indicate unoptimized performance. The upper row of numbers above the bars are the block sizes chosen by the model and the lower row of numbers above the bars are the block sizes chosen by exhaustive search.

third, and eighth matrices), the difference in performance was small (less than 6 % of the performance).

3.2.2 When and How to Apply Register Blocking

We have observed that the performance model described in the previous section generally made a reasonable selection of block size, but the model sometimes selected a block size that was not optimal. At times, register-blocked multiplication with the chosen block size ran slower than unblocked multiplication. So, we made a further decision as to whether to perform register-blocking optimization on a given matrix or not.

The decision process works in the following way. Using the performance model, if any of the predicted performances was better than unblocked performance, the matrix became a candidate for register-blocking optimization. The register block size with the highest predicted performance of multiplication was chosen as the block size, and the matrix was blocked for that size. If after running the multiplication on a blocked matrix and measuring the real performance, the measured performance is shown to be better than non-blocked performance, optimization with register blocking was performed.

It would have been possible to use an exhaustive search by blocking the matrix repeatedly for each block size and measuring the real performance, instead of predicting the performance using a model. However, we developed and used the performance model for the following reasons. First, blocking the matrix in hundreds of possible ways too time-consuming. Second, since the performance graph of multiplication for varying block sizes is quite jagged, it was impossible to speed up a search using heuristics such as hill-climbing to find a maximum.

3.3 Performance of Register Blocking on Benchmark Suite

Figures 3.9 – 3.12 show the effect of register blocking [1] for 46 matrices listed in table 2.5. As it is mentioned in chapter 2, the matrices near the bottom of the list have less structure.

On the Alpha 21164 and the PowerPC 604e, the multiplication performance of some large matrices were measured by fragmenting the whole matrix into smaller matrices due to memory restrictions on these systems. Figures 3.13 – 3.16 give the details of the performance data and show the block size that was selected.

As a reference, optimized dense BLAS operation performance on various processors are shown in table 3.8. The table is drawn from a LAPACK user's guide (3rd. edition) [2]. It shows DGEMV (dense matrix vector multiplication) and DGEMM (dense matrix matrix multiplication) performances for two matrix sizes, 100×100 and 1000×1000. GEMV performance is lower than DGEMM performance since matrix-vector multiplication does not reuse matrix elements. The ratio of DGEMV performance to DGEMM performance is shown in each third column. Even though the matrices are dense, when the ratio is large, we conjecture that we can expect higher speedups for sparse matrix vector multiplication through any kind of memory hierarchy optimization on that processor. If we focus on the larger 1000×1000 matrix in the last column, we can see the MIPS and Power3 (which we have no experimental data on) had fairly high ratios (0.38 and 0.48, respectively). This corresponds to the fact that the best speedups were obtained on the MIPS R10000 (fig-

[1] Each figure shows a Mflop rate comparison (left) and the corresponding speedup (right). The Mflop rate was calculated using only those arithmetic operations required by the original representation, not those induced by fill from blocking. The speedup was calculated by dividing the optimized Mflop rate by the unoptimized rate, or equivalently, the unoptimized runtime by the optimized runtime.

Processor	Clock Speed (MHz)	$n = m = k = 100$ DGEMV Mflops(A)	DGEMM Mflops(B)	(A) /(B)	$n = m = k = 1000$ DGEMV Mflops(C)	DGEMM Mflops(D)	(C) /(D)
UltraSPARC I	200	124	302	0.41	57	287	0.20
UltraSPARC II	300	267	474	0.56	86	527	0.16
MIPS 12000	300	216	563	0.38	210	555	0.38
Alpha 21164	533	66	543	0.12	36	584	0.06
Alpha 21264	500	376	522	0.72	112	500	0.22
PowerPC 604e	190	23	160	0.14	25	212	0.12
Power 3 630	200	304	567	0.54	350	722	0.48

Figure 3.8: BLAS performance on various processors

ure 3.10). The UltraSPARC-I had a fairly high ratio of 0.20, and our speedup for the UltraSPARC (figure 3.9) follows behind the MIPS R10000, while the Alpha 21164 and the PowerPC 604e with low ratios (0.06 and 0.12 respectively) exhibit small speedups in our optimizations (figures 3.11 and 3.12).

As shown in the tables 3.13 – 3.16, the matrices were not register-blocked when the model predicted performance degradation. In the table, the speedups for those matrices are not exactly 1, since we have used slightly different codes for measuring unblocked code and 1×1 blocked code. The latter code is generated by our source code generator and as the table shows, the code performs slightly better for some architectures (R10000 and PowerPC), and slightly worse for others (UltraSPARC). Architecture and compiler specifics contribute to this discrepancy. There is also a small amount of run-time overhead for register blocking in multiplying blocks. In some cases, the small amount of predicted speedup is overwhelmed by this overhead as in matrices 16, 17, and 20 in table 3.13. To prevent those matrices from being register-blocked, SPARSITY's automatic optimization system will add an extra step at the end of the register-blocking optimization, in which the performance of unblocked and register blocked multiplications are measured and compared. If the performance of register-blocked multiplication is worse than that of unblocked multiplication, register blocking will not be applied.

The 43rd matrix, which arises from a linear programming problem, has a nearly dense band starting from the first column to the 25^{th} column, and 1×4 blocks near the diagonal. This kind of matrix confuses our sampling strategy for estimating fill overhead, and the performance was degraded after register blocking because of the incorrectly estimated fill overhead in predicted performance.

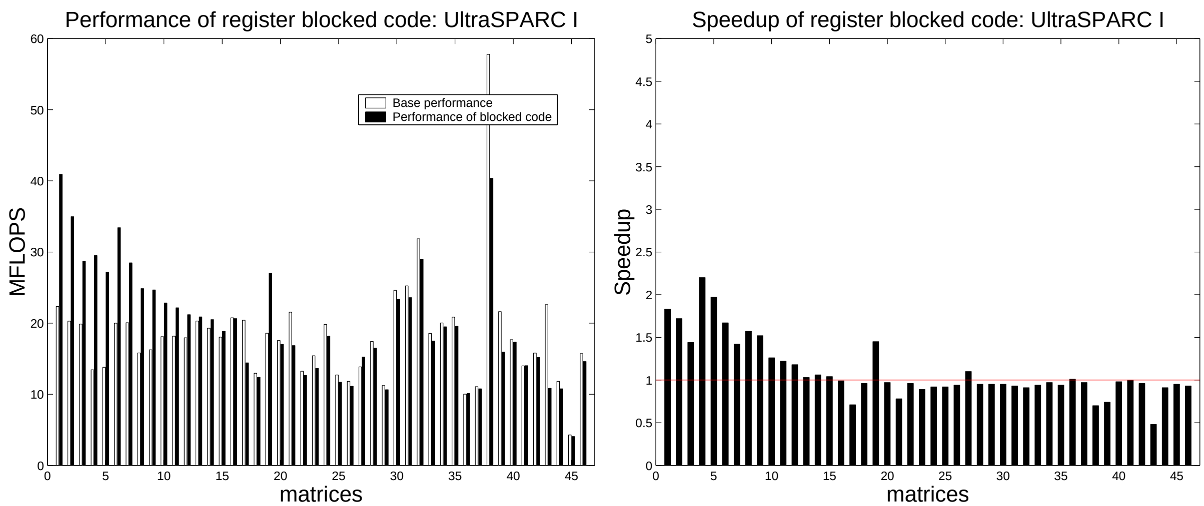


Figure 3.9: **Performance of register-blocked multiplication on an UltraSPARC I**

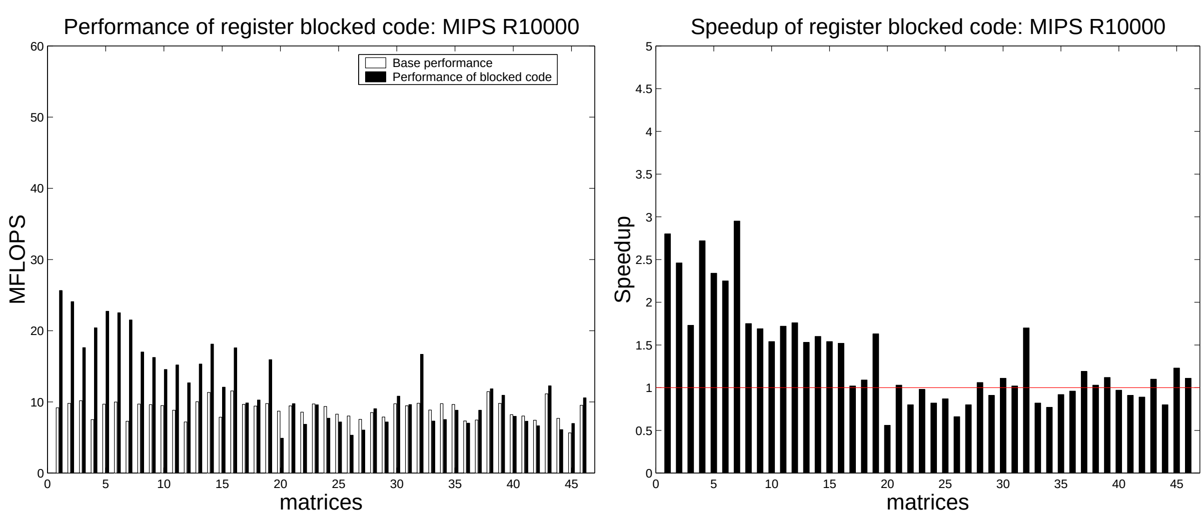


Figure 3.10: **Performance of register-blocked multiplication on a MIPS 10000**

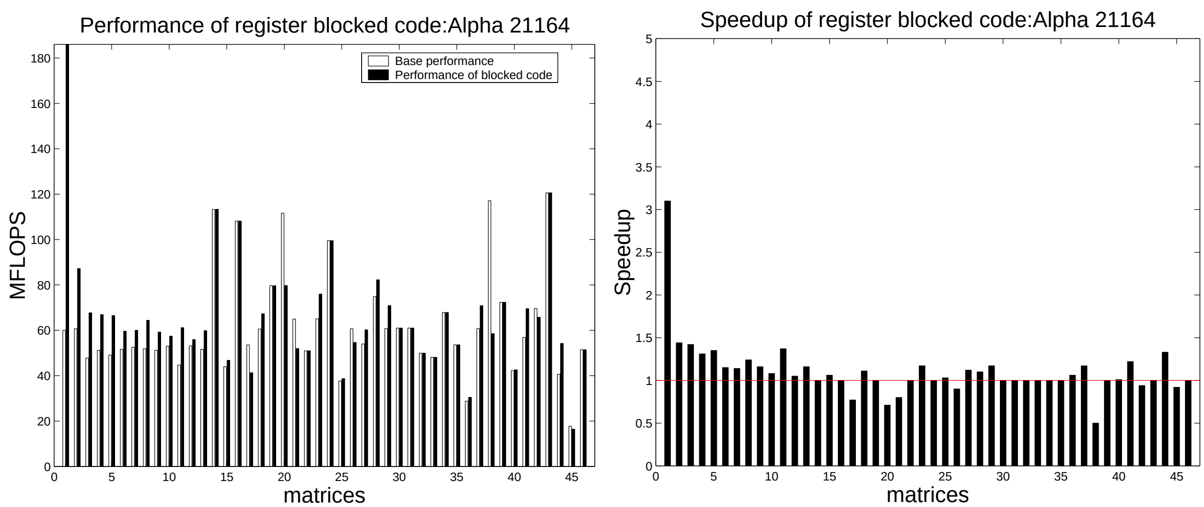


Figure 3.11: **Performance of register-blocked multiplication on an Alpha 21164**

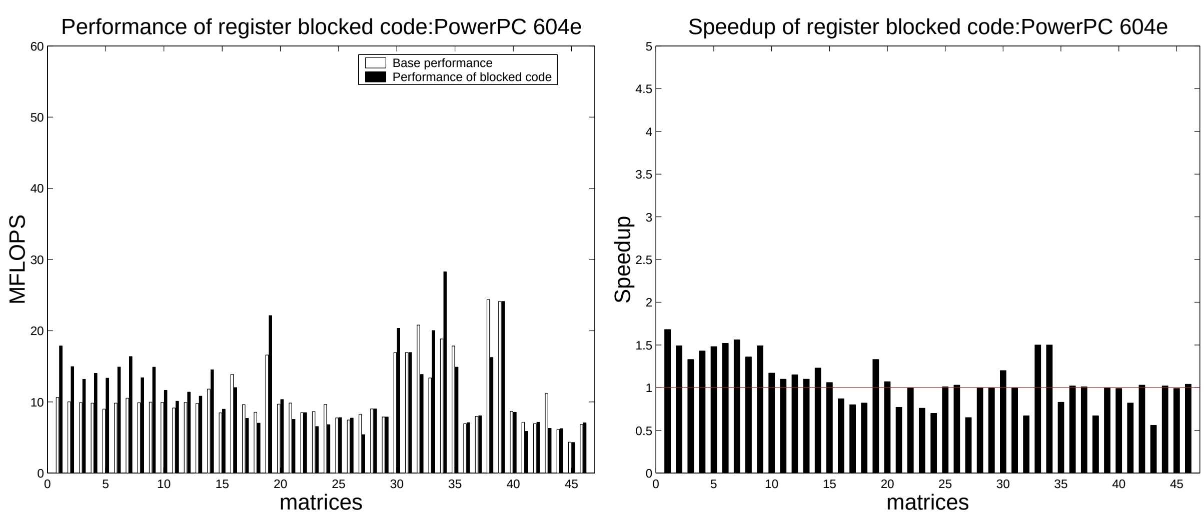


Figure 3.12: **Performance of register-blocked multiplication on a PowerPC 604e**

	Matrix	Block Size	Fill Overhead	Mflops	Speedup
1	45	8x8	1.00	40.91	1.83
2	40	8x8	1.00	34.96	1.72
3	38	6x6	1.12	28.68	1.44
4	4	6x6	1.19	29.49	2.20
5	37	4x4	1.00	27.18	1.97
6	6	3x3	1.00	33.41	1.67
7	7	3x3	1.00	28.47	1.42
8	16	3x3	1.11	24.85	1.57
9	1	3x3	1.13	24.67	1.52
10	8	2x2	1.21	22.84	1.26
11	39	2x2	1.23	22.16	1.22
12	43	2x2	1.24	21.19	1.18
13	41	2x2	1.28	20.87	1.03
14	25	2x2	1.33	20.49	1.06
15	44	2x2	1.35	18.84	1.04
16	26	2x2	1.39	20.63	0.99
17	19	2x2	1.79	14.41	0.71
18	23	2x2	1.79	12.39	0.96
19	24	2x1	1.01	27.03	1.45
20	11	1x2	1.17	17.01	0.97
21	36	1x2	1.36	16.84	0.78
22	2	1x1	1.00	12.66	0.96
23	3	1x1	1.00	13.64	0.89
24	5	1x1	1.00	18.17	0.92
25	9	1x1	1.00	11.69	0.92
26	17	1x1	1.00	11.12	0.94
27	18	1x1	1.00	15.22	1.10
28	20	1x1	1.00	16.47	0.95
29	21	1x1	1.00	10.63	0.95
30	28	1x1	1.00	23.35	0.95
31	29	1x1	1.00	23.59	0.93
32	30	1x1	1.00	28.96	0.91
33	32	1x1	1.00	17.48	0.94
34	33	1x1	1.00	19.48	0.97
35	34	1x1	1.00	19.55	0.94
36	35	1x1	1.00	10.13	1.01
37	42	1x1	1.00	10.76	0.97
38	27	1x1	1.00	40.36	0.70
39	31	1x1	1.00	15.92	0.74
40	10	1x1	1.00	17.33	0.98
41	12	1x1	1.00	14.02	1.00
42	13	1x1	1.00	15.18	0.96
43	14	7x1	3.20	10.84	0.48
44	15	1x1	1.00	10.76	0.91
45	22	1x1	1.00	4.06	0.95
46	46	1x1	1.00	14.60	0.93

Figure 3.13: Summary of register blocking optimization on an UltraSPARC I

	Matrix	Block Size	Fill Overhead	Mflops	Speedup
1	45	11x11	1.00	25.65	2.80
2	40	8x8	1.00	24.09	2.46
3	38	6x6	1.12	17.63	1.73
4	4	6x6	1.19	20.41	2.72
5	37	4x4	1.00	22.73	2.34
6	6	3x3	1.00	22.53	2.25
7	7	3x3	1.00	21.52	2.95
8	16	3x3	1.11	17.02	1.75
9	1	3x3	1.13	16.25	1.69
10	8	3x3	1.57	14.56	1.54
11	39	2x2	1.23	15.19	1.72
12	43	3x3	1.46	12.68	1.76
13	41	3x3	1.52	15.33	1.53
14	25	3x3	1.60	18.12	1.60
15	44	2x2	1.35	12.07	1.54
16	26	3x3	1.69	17.60	1.52
17	19	3x3	2.36	9.85	1.02
18	23	2x2	1.79	10.27	1.09
19	24	2x1	1.01	15.94	1.63
20	11	3x3	2.35	4.90	0.56
21	36	3x3	2.38	9.76	1.03
22	2	2x2	2.71	6.86	0.80
23	3	2x2	2.02	9.58	0.98
24	5	2x2	2.08	7.71	0.82
25	9	2x2	2.45	7.19	0.87
26	17	2x2	2.96	5.32	0.66
27	18	2x2	2.34	6.05	0.80
28	20	1x1	1.00	9.05	1.06
29	21	2x2	1.98	7.19	0.91
30	28	1x1	1.00	10.81	1.11
31	29	2x2	2.31	9.63	1.02
32	30	3x3	1.64	16.69	1.70
33	32	2x2	2.99	7.31	0.82
34	33	2x2	3.09	7.52	0.77
35	34	2x2	2.53	8.84	0.92
36	35	2x2	2.31	7.01	0.96
37	42	2x2	1.98	8.83	1.19
38	27	2x2	1.96	11.83	1.03
39	31	1x1	1.00	10.93	1.12
40	10	2x2	2.33	7.98	0.97
41	12	2x2	2.52	7.28	0.91
42	13	2x2	2.61	6.64	0.89
43	14	4x5	2.68	12.25	1.10
44	15	1x2	1.77	6.11	0.80
45	22	1x1	1.00	6.97	1.23
46	46	1x1	1.00	10.56	1.11

Figure 3.14: Summary of register blocking optimization on a MIPS R10000

	Matrix	Block Size	Fill Overhead	Mflops	Speedup
1	45	8x8	1.00	186.05	3.10
2	40	8x8	1.00	87.25	1.44
3	38	2x2	1.12	67.68	1.42
4	4	2x2	1.07	66.93	1.31
5	37	2x2	1.00	66.49	1.35
6	6	2x2	1.23	59.61	1.15
7	7	2x2	1.22	60.04	1.14
8	16	2x2	1.10	64.38	1.24
9	1	2x2	1.25	59.24	1.16
10	8	2x2	1.21	57.41	1.08
11	39	2x2	1.23	61.17	1.37
12	43	2x2	1.24	55.94	1.05
13	41	2x2	1.28	59.83	1.16
14	25	2x2	1.33	113.30	1.00
15	44	2x2	1.35	46.72	1.06
16	26	2x2	1.39	108.17	1.00
17	19	2x2	1.79	41.29	0.77
18	23	2x2	1.79	67.28	1.11
19	24	2x1	1.01	79.67	1.00
20	11	1x2	1.17	79.76	0.71
21	36	2x2	1.81	51.97	0.80
22	2	1x1	1.00	50.94	1.00
23	3	2x2	2.02	75.94	1.17
24	5	1x1	1.00	99.51	1.00
25	9	1x1	1.00	38.72	1.03
26	17	1x1	1.00	54.63	0.90
27	18	1x1	1.00	60.20	1.12
28	20	1x1	1.00	82.28	1.10
29	21	1x1	1.00	70.88	1.17
30	28	1x1	1.00	61.00	1.00
31	29	1x1	1.00	61.00	1.00
32	30	1x2	1.40	49.92	1.00
33	32	1x1	1.00	48.10	1.00
34	33	1x1	1.00	67.86	1.00
35	34	1x1	1.00	53.58	1.00
36	35	1x1	1.00	30.44	1.06
37	42	1x1	1.00	70.87	1.17
38	27	2x2	1.96	58.54	0.50
39	31	1x1	1.00	72.33	1.00
40	10	1x1	1.00	42.57	1.01
41	12	1x1	1.00	69.54	1.22
42	13	1x1	1.00	65.76	0.94
43	14	2x1	1.52	120.59	1.00
44	15	1x1	1.00	54.19	1.33
45	22	1x1	1.00	16.41	0.92
46	46	1x1	1.00	51.43	1.00

Figure 3.15: Summary of register blocking optimization on an Alpha 21164

	Matrix	Block Size	Fill Overhead	Mflops	Speedup
1	45	11x11	1.00	17.86	1.68
2	40	4x4	1.00	14.96	1.49
3	38	3x3	1.12	13.18	1.33
4	4	3x3	1.06	14.01	1.43
5	37	4x4	1.00	13.34	1.48
6	6	3x3	1.00	14.90	1.52
7	7	3x3	1.00	16.37	1.56
8	16	3x3	1.11	13.39	1.36
9	1	3x3	1.02	14.88	1.49
10	8	2x2	1.21	11.63	1.17
11	39	2x2	1.23	10.09	1.10
12	43	2x2	1.24	11.38	1.15
13	41	2x2	1.28	10.81	1.10
14	25	2x2	1.33	14.52	1.23
15	44	2x2	1.35	8.98	1.06
16	26	2x2	1.39	12.02	0.87
17	19	2x2	1.79	7.70	0.80
18	23	2x2	1.79	7.01	0.82
19	24	2x1	1.01	22.12	1.33
20	11	1x2	1.17	10.34	1.07
21	36	2x2	1.81	7.55	0.77
22	2	1x1	1.00	8.49	1.00
23	3	2x2	2.02	6.55	0.76
24	5	2x2	2.08	6.80	0.70
25	9	1x1	1.00	7.80	1.01
26	17	1x1	1.00	7.72	1.03
27	18	2x2	2.34	5.39	0.65
28	20	1x1	1.00	9.02	1.00
29	21	1x1	1.00	7.88	1.00
30	28	1x1	1.00	20.33	1.20
31	29	1x1	1.00	16.94	1.00
32	30	3x2	2.05	13.86	0.67
33	32	1x1	1.00	20.03	1.50
34	33	1x1	1.00	28.27	1.50
35	34	1x1	1.00	14.88	0.83
36	35	1x1	1.00	7.09	1.02
37	42	1x1	1.00	8.05	1.01
38	27	2x2	1.96	16.25	0.67
39	31	1x1	1.00	24.11	1.00
40	10	1x1	1.00	8.55	0.99
41	12	1x2	1.56	5.86	0.82
42	13	1x1	1.00	7.15	1.03
43	14	3x2	2.39	6.29	0.56
44	15	1x1	1.00	6.25	1.02
45	22	1x1	1.00	4.31	0.99
46	46	1x1	1.00	7.06	1.04

Figure 3.16: Summary of register blocking optimization on a PowerPC 604e

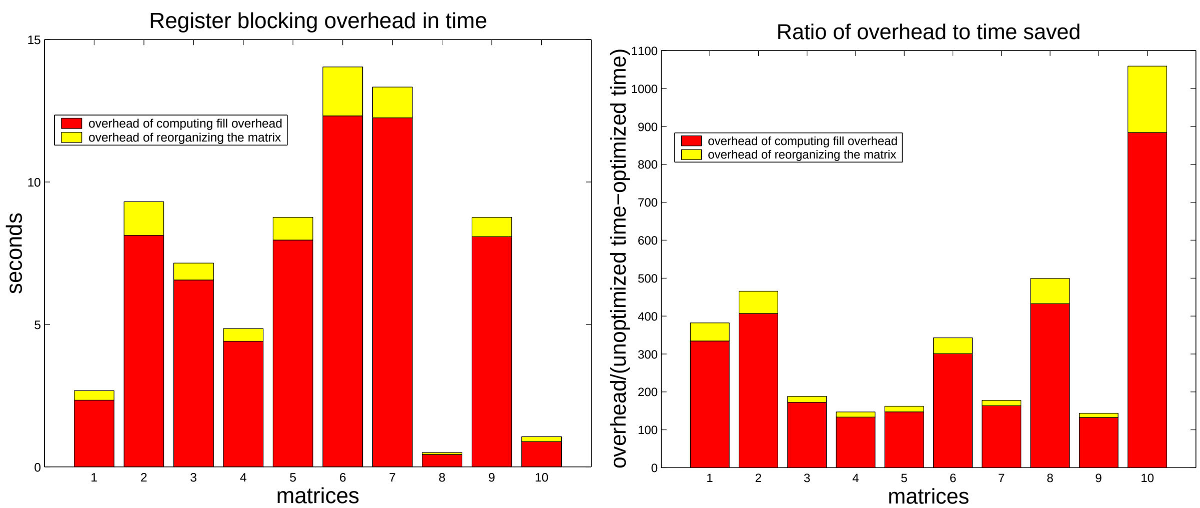


Figure 3.17: **Pre-computation overhead of register-blocked multiplication on 10 sparse matrices taken on a 167 MHz UltraSPARC:** The left figure shows the time used to perform each of the two steps, determining the block size and reorganizing the matrix. The right figure shows the same values divided by the savings of blocked vs. unblocked code; this tells us the number of times a matrix-vector multiplication would have to be performed to amortize the overhead.

3.4 Analysis of Overhead

As we have seen, there are overheads in performing register blocking. The overheads can be distinguished by whether it is a run-time overhead that is paid every time the multiplication is performed, or a preprocessing overhead that is paid only the first time the matrix structure is used. In iterative solvers, for example, a sparse structure may be reused many times with different numeric values. The run-time overhead was already analyzed in the previous section, and we analyze the pre-computation overhead in this section.

There is more than one source of pre-computation overhead seen when applying register blocking. The first of these is the price of determining the block size by using a performance model. As noted earlier, the cost of doing an exact fill overhead computation was very expensive, so we developed the heuristic based on sampling a subset of the rows and columns in a single pass. The second source of overhead is the time used to reorganize the matrix in the blocked format. Both of these overheads are paid only once, while computations like matrix-vector multiplication may be performed many times on the same sparse matrix structure.

Figure 3.17 shows the amount of these two pre-computation overheads, both in absolute time and in the ratio of time spent in overhead to time saved by the optimization. If the block size selection and reorganization are being done at runtime, the figure in the right shows the number of sparse matrix-vector multiplications that must be done before the optimization pays off. While these numbers are typically as high as several hundreds, the optimizations are still likely to be useful in practice, since computations such as iterative solvers repeat the matrix vector multiplication on the same matrix structure many times. If the user is willing to change the matrix representation throughout the application, which is likely if the sparse matrix library is properly encapsulated, the cost of reorganization can be avoided. In addition, there are many application domains in which the block size could be determined for an entire class of applications. For example, dense $k \times k$ sub-blocks will appear in Finite Element problems with k degrees of freedom. Although different finite element problems may produce different sparse matrix structures, the block size chosen for one is likely to work well for others with similar structures (e.g., from the same problem domain) and the same number of degrees of freedom. The sparse BLAS interface standardized by the BLAS Technical Forum accommodates such need to specify the known property of the matrix by a user, and suggests that a user is allowed to provide hints about the matrix when the matrix is constructed.

3.5 Summary

In this chapter, we have studied register blocking optimization. We have learned that finding a right block size with low fill overhead and good performance for the target machine is important for register blocking, and the decision is dependent on the nonzero structure of the sparse matrix and the machine architecture. Realizing that, we have developed a model to determine the block size for a given matrix and a given machine.

As seen from the performance results on a set of test matrices, the matrices with dense blocks (in the top of the list in figure 2.5) have great speedup from register blocking.

As a result of this study, we suggest the following future work. First, currently we use the same multiplication code across machines, with the understanding that a small change in source code can make significant differences on various machines and compilers.

PHiPAC (Portable High Performance ANSI C) [8] and ATLAS (Automatically Tuned Linear Algebra Software) [70] generate different versions of code for dense matrix multiplications and choose the correct version of code for a target machine by compiling the code and measuring the actual performance with search scripts. Similarly, the code generator used in this study may be expanded to generate variants of sparse matrix-vector multiplication code and use a different code for a given target machine to obtain better performance by using search method.

Second, in the analysis of overhead, we note that the run-time overhead of register blocking results from zero fills. In order to decrease the fill overhead, it is possible to mix variable block sizes in one matrix. This incurs a tag for each block to identify block size, and a branch statement for each block to execute the right routine. By storing blocks of the same size together, this overhead can be avoided, but it may reduce the chance of exploiting spatial locality because consecutive elements in the matrix can be stored in separate places if they are in different-sized block. Toledo [68] uses this scheme to store 1x2 and 2x2 blocks together. We have not considered this hybrid method because the number of allowable block sizes is large for our case. However, in a situation where the block size is limited to small range, this variant scheme would be worth trying.

Chapter 4

Cache Blocking Optimization

In this chapter we describe optimization techniques for improving cache utilization. The cost of accessing main memory on modern microprocessors is in the tens to hundreds of cycles, so minimizing cache misses can be critical to high performance. As with register blocking, the basic idea is to reorganize the matrix data structure and associated computation to improve the reuse of data in the source vector, without destroying the locality in the destination vector. Unlike register blocking, the set of values in the cache is not under complete control of the software; hardware controls the selection of data values in each level of cache according to its policies on replacement, associativity, and write strategy [34]. More importantly, the caches can hold thousands of values, while registers only hold tens of values, so it is not practical to fill in a rectangular block of a sparse matrix so that the corresponding source and destination vector values fill the cache. Instead, we rearrange the computation so that a block of values in the matrix are accessed near each other in time, but retain the sparse structure of the matrix, and avoid adding any additional zero elements.

We consider two approaches to a cache blocking algorithm, one that reorganizes the data structure prior to computation, and another that keeps the data structure unmodified but reorganizes the computation to work on logical blocks of the matrix. The approaches are described and compared in section 4.1. Section 4.2 presents the performance results of cache blocking on the set of matrices from chapter 2 as well as a large set of randomly generated matrices. Cache blocking is most effective on matrices that are very large and have a random structure; a specific example of such a matrix comes from a data mining algorithm used for document searching on the web, so the more detailed performance studies, such

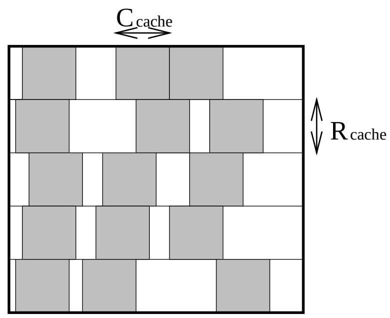


Figure 4.1: **Cache-blocks in a sparse matrix:** The gray areas are sparse matrix blocks that contain nonzero elements in the $r_{cache} \times c_{cache}$ rectangle. The white areas contain no nonzero elements, and are not stored.

as the comparison between blocking methods, focus on that matrix. Section 4.3 evaluates the overhead of performing cache blocking. Roughly speaking, the matrices that benefit most from register blocking are the opposite of those that benefit from cache blocking, but in section 4.4 we look at the performance results of combining the two optimizations. We summarize the cache blocking results in section 4.5.

4.1 Description of Cache Blocking

The idea of cache blocking optimization is to keep c_{cache} elements of the source vector x in the cache along with r_{cache} elements of the destination vector y while an $r_{cache} \times c_{cache}$ block of matrix A is multiplied by this portion of the vector x. The entries of A need not be saved in the cache, but since this decision is under hardware control, interference between elements of the matrix and the two vectors can be a problem.

To simplify the code generation problem and to limit the range of experiments, we start with the assumption that cache blocks within a single matrix should have a fixed size. In other words, r_{cache} and c_{cache} are fixed for a particular matrix and machine. This means that logical block size is fixed, although the amount of data and computation may not be uniform across the blocks, since the number of nonzeros in each block may vary. Figure 4.1 shows a matrix with fixed size cache blocks. Note that the blocks need not begin at the same offsets in each row, unlike those in the implementation of register blocking.

Unlike register blocking, creating dense $r_{cache} \times c_{cache}$ blocks by filling in zeros is impractical, since it would incur excessive storage and computation overhead. Instead, we consider two strategies for cache level blocking. The first optimization, described in section 4.1.1, involves a preprocessing step to reorganize the matrix so that each block is stored contiguously in main memory. This is referred to as *static cache blocking*, because the location of the cache blocks are determined prior to the execution of the matrix vector multiplication. The second optimization, described in section 4.1.2, does not involve any data structure reorganization, but changes the order of computation by retaining a set of pointers into each row of the current logical block. This will be referred to as *dynamic cache blocking*, because the blocks are determined only as the matrix vector multiplication proceeds. Dynamic cache blocking avoids any preprocessing overhead, but as will be seen, it incurs more runtime overhead than static cache blocking.

4.1.1 Static Cache Blocking

In static cache blocking, the sparse matrix is reorganized by changing the order of the column index array and nonzero elements of the sparse matrix, and augmenting another array of indices which points to the beginning of each block. Before reorganization, nonzero elements of each row are stored sequentially in memory. When the matrix is reorganized for cache blocking, the rows of the matrix are broken into groups of r_{cache} rows. Within each group of rows, starting from the column with the nonzero element whose column index is the smallest, any nonzeros that appear in c_{cache} columns are grouped in one rectangular area, which is stored similarly to the compressed sparse row (CSR) format.

The data structures used in a cache-blocked matrix are shown in figure 4.2. The top level array is called row_start and it points to the beginning of each row of blocks. In the figure, there are two rows of blocks, so the row_start matrix has three entries, the last pointing past the end of the *block_ptr* array. The *block_ptr* array points to the beginning of each row within a block, and the *col_idx* and *value* arrays store the column indices and values of each nonzero element. The main difference between this and the CSR format is the extra level of indirection for the blocks.

The code in figure 4.3 multiplies a cache blocked sparse matrix times a dense vector x. In the code, *b_i* is an index into row_start array and *b_j* is an index into *block_ptr* array. The variable *end_r* is used for the last group of rows when the r_{cache} does not divide the number of rows in the matrix. The index *b_j* increases by *end_r* because each sparse block has that many *block_ptr* elements for each row within the block. The nonzero elements are being accessed in the order in which they are stored in memory, which is important for

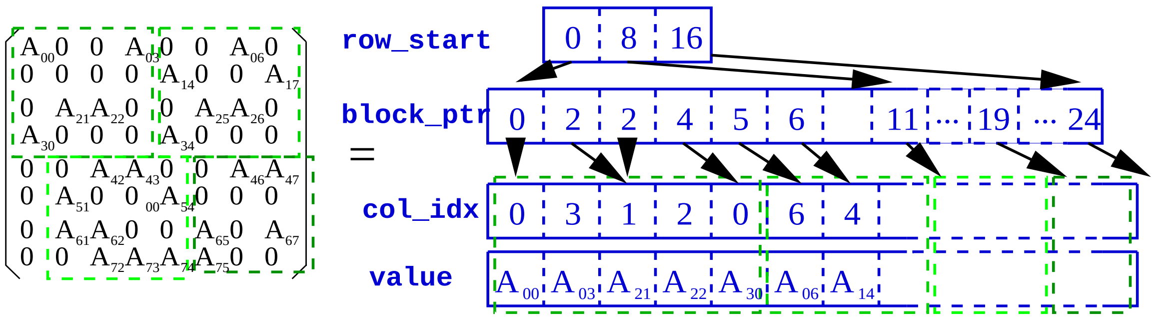


Figure 4.2: **Storage format of a cache-blocked sparse matrix:** In cache blocking, each block is stored in sparse format, similarly to CSR, using data structures *block_ptr*, *col_idx* and *value*. This example matrix has 4, 4×4 blocks. The *row_start* array points to the beginning of each row of blocks, while the *block_ptr* array keeps pointers to the beginnings of individual rows inside those blocks.

preserving spatial locality in the matrix. Referring back to figure 4.1, this means that while processing one gray block, the portions of the x and y vectors that correspond to that block are accessed repeatedly. The sub-arrays of x and y will sit in the cache during processing, as long as they both fit and there is no interference between the two sub-arrays and the matrix entries.

By examining the code, one can see some additional overhead relative to the simpler CSR routine, although the inner loops are roughly the same. We will examine the performance more carefully in section 4.1.3, but even without that quantitative analysis, it is likely that the major disadvantage of static cache blocking is the preprocessing overhead, since reorganization of the matrix is involved.

4.1.2 Dynamic Cache Blocking

In dynamic cache blocking optimization, we change the multiplication code in such a way that the multiplications are performed in the same order as in static cache blocking, but the data structure is unchanged. This has the locality advantages of static cache blocking with respect to the source and destination vectors, but avoids the preprocessing overhead.

The data structures used for dynamic cache blocking are illustrated in figure 4.4. While multiplying a particular block, a set of pointers into each row of the block is maintained. Unfortunately, this adds considerable complexity to the inner loop and does not access matrix elements sequentially, since each block is not stored contiguously. As we

```
void block_smvp_sparse (int r, int m,
                        int *row_start, int *block_ptr,
                        double *value, int *col_idx,
                        double *src, double *dest)
{
  int i, j;
  int b_i, b_j, b_m;
  int end_r;

  b_m = (m+r-1)/r;

  for (b_i=0; b_i<b_m; b_i++){

    end_r = (b_i+1 < b_m) ? r : m - b_i*r ;

    for (b_j=row_start[b_i]; b_j<row_start[b_i+1]; b_j+=end_r){

      for (i=0; i<end_r; i++){
        double t=0;
        for (j=block_ptr[b_j+i]; j<block_ptr[b_j+i+1]; j++)
          t += value[j] * src[col_idx[j]];
        dest[b_i*r+i] += t;
      }
    }
  }
}
```

Figure 4.3: **Code for multiplying a cache-blocked sparse matrix:** In the code, *b_i* is used as an index into the *row_start* array, *b_j* is an index into the *block_ptr* array, and *j* is an index into the *value* and *col_idx* arrays.

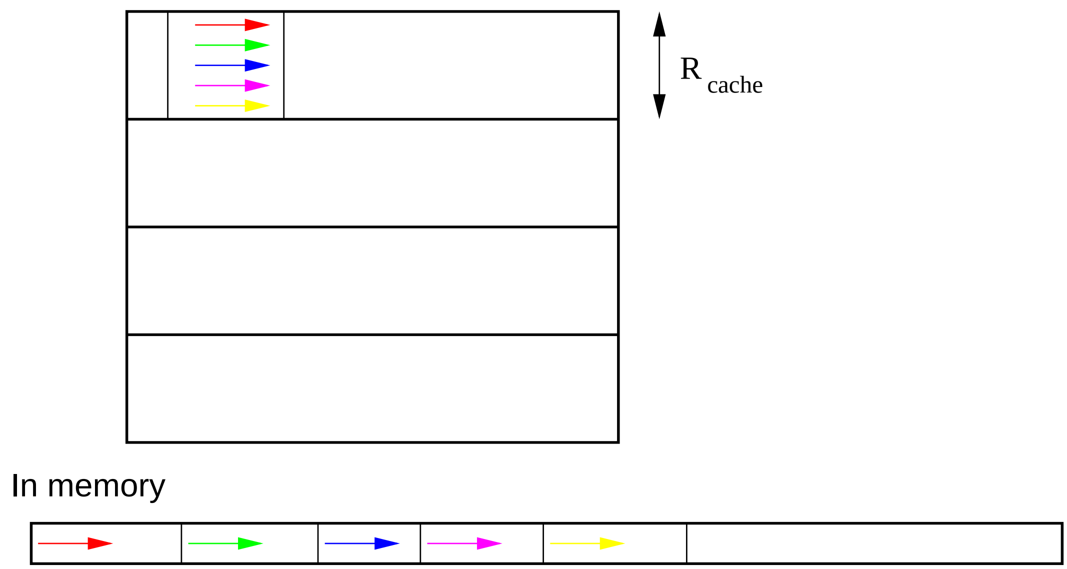


Figure 4.4: **Memory accesses in dynamic cache blocking:** The pointers into the rows in one cache block keep track of the cache block boundaries. Rows in the same cache block are not contiguous in memory.

will see in the next section, this implementation does not speed up very much because the run-time overhead of managing pointers negates the effect of cache blocking.

4.1.3 Performance Comparison of Static and Dynamic Blocking

We now compare the performance of static and dynamic cache blocking, focusing on a matrix used in web search engines with algorithms such as Latent Semantic Indexing (LSI). The matrix is indexed by a set of keywords and a set of documents, with the number of documents being much larger than the set of keywords. The full matrix, obtained from the Inktomi company, is $100K \times 2.6M$ with 380M nonzero elements. These experiments use the first $10K \times 256K$ block. We refer to this matrix as "the LSI matrix," although other algorithms are often used for this type of processing and the use of matrix-vector multiplication in the inner loop is very common. For example, Dhillon and Modha [21] use the same type of processing to cluster documents in large text data. We have chosen to concentrate on the LSI matrix in this evaluation, because cache blocking proves to be particularly effective, so the comparison is more dramatic. We have looked at the relative value of static versus dynamic blocking on other matrices, and the result is the same. Section 4.2 will do a more thorough comparison of cache blocking on other matrices.

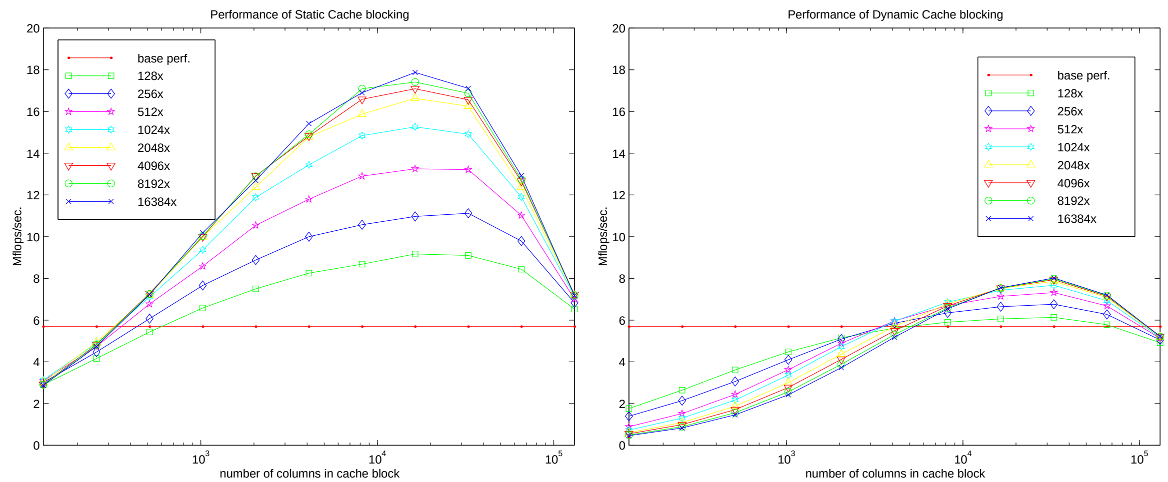
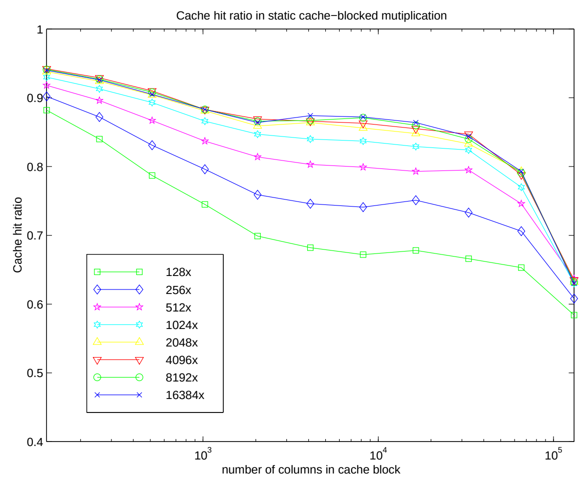


Figure 4.5: **Performance of static and dynamic cache-blocking on the LSI matrix:** This is measured on 167MHz Ultra SPARC I. Each line is for different number of rows in a cache block, r_{cache} and the horizontal axis is the number of columns in a cache block, c_{cache}. The horizontal axis is a log scale.

The graphs in figure 4.5 show the performance of matrix-vector multiplication for static (left) cache blocking and dynamic (right) cache blocking, varying the size of cache blocks. The horizontal line at 5.8 Mflops shows the base performance of the multiplication without any cache blocking. The static cache blocking significantly improves the performance of the LSI matrix as shown in figure 4.5. The best performance is 18.0 Mflops, which is obtained for 16K $\times$ 16K cache blocks. The unblocked performance is only 5.8 Mflops, so the speedup is roughly 3.1. As shown in the figure, the dynamic cache blocking was proven not to be as beneficial as static cache blocking. Because most applications perform several matrix vector multiplications with the same matrix structure, this indicates that it is probably worth paying the preprocessing cost of static blocking. We also note in figure 4.5 that cache blocking for small block sizes degrades the performance, because even in the static case there are runtime costs associated with cache blocking, and the benefits of improved locality are too small to outweigh these costs.

We now study the cache behavior of both implementations to see how cache blocking actually changes the behavior of the cache. Figures 4.6 and 4.7 show cache behavior during multiplication, and help explain the reason that static cache blocking is so much more effective than dynamic cache blocking. The figures at the left show the L2 cache hit ratios and the figures at the right show the total number of L2 cache accesses. They are



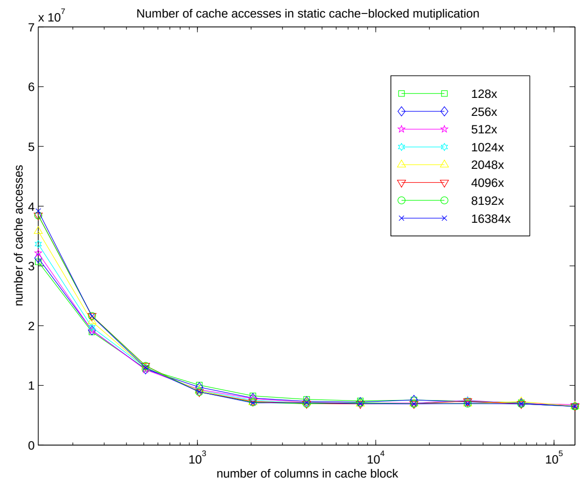


Figure 4.6: **Cache behavior of static cache-blocking on the LSI matrix:** The left figure shows the L2 cache hit ratio and the right figure shows the total number of cache accesses.

measured using an event counter in an UltraSPARC I. A lower number of L2 cache accesses means that more memory accesses were resolved in the L1 cache (L1 cache hit). In static cache blocking, while the cache hit ratio is decreasing as the cache block size gets larger, the total number of cache references rapidly decreases. The peak performance in figure 4.5 (left) occurs when the number of L2 cache accesses is low, but before the cache hit ratio drops dramatically, as it does on the right end of figure 4.6 (left). The cache hit ratio is low in *dynamic cache blocking* because the elements in a block in figure 4.4 are not stored in contiguous locations and the number of extra pointers to be managed increases as the number of rows in the cache block increases. The total number of L2 cache references is larger (almost double) than that of *static cache blocking*, for the same reason.

This examination of cache misses confirms that static cache blocking is more effective than dynamic blocking, backing up the timing results of the previous section. In particular, it shows that the performance differs due to variations in the data cache behavior, rather than being an artifact of our particular implementation of the two blocking strategies, or unrelated compiler or architecture effects.

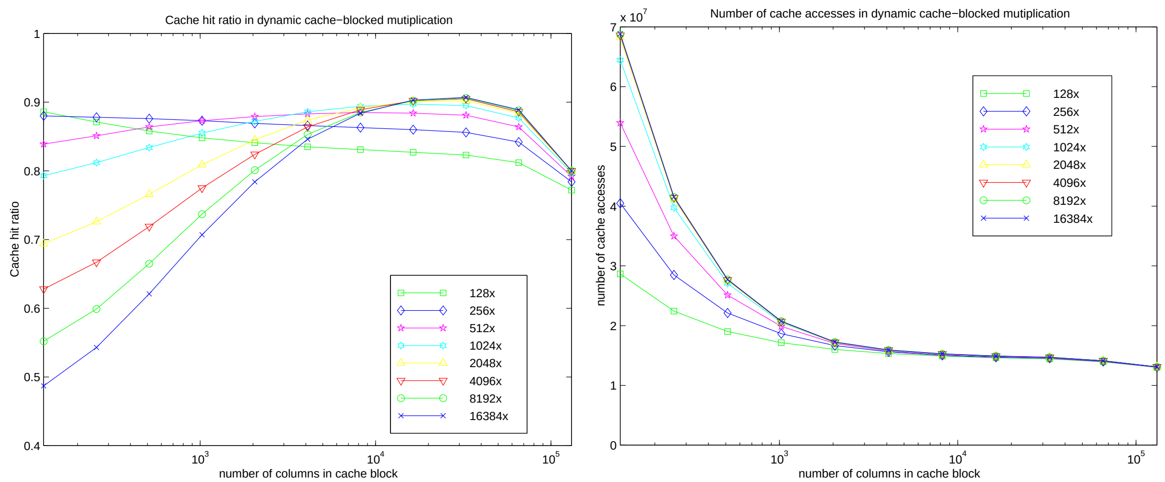


Figure 4.7: **Cache behavior of dynamic cache-blocking on the LSI matrix:** The left figure shows the L2 cache hit ratio and the right figure shows the total number of cache accesses.

4.2 Performance of Cache Blocking

This section provides a more thorough analysis of the benefits of cache blocking, using the idea of static cache blocking. It presents data on the effectiveness of cache blocking on all the matrices in our set, then looks at performance on some randomly generated matrices to help explain the highly variable speedups observed on that set.

4.2.1 Performance on Matrix Benchmark Suite

The results of applying the static cache blocking optimization to a set of the matrices from chapter 2 are shown in figures 4.8 and 4.9 for the UltraSPARC I and MIPS R10000, respectively. The left graph in each figure shows the Mflop rate of both optimized and unoptimized code, and the right side shows the calculated speedups.

The cache block size varies from 64×64 to $64K \times 64K$, including rectangular sizes, and the highest performance was taken for the speedup in the increment of powers of two. Tables 4.10 and 4.11 summarize the block sizes with the highest performance. In general, speedup is better on a MIPS than on an UltraSPARC. There are two reasons for this. First, the performance gap between cache and memory accesses is larger on MIPS R10000. (The L1 and L2 cache miss penalties are 26 and 589 nanoseconds on a MIPS R10000 and 36 and 268 nanoseconds on an UltraSPARC, according to benchmarks.) Second, the MIPS R10000

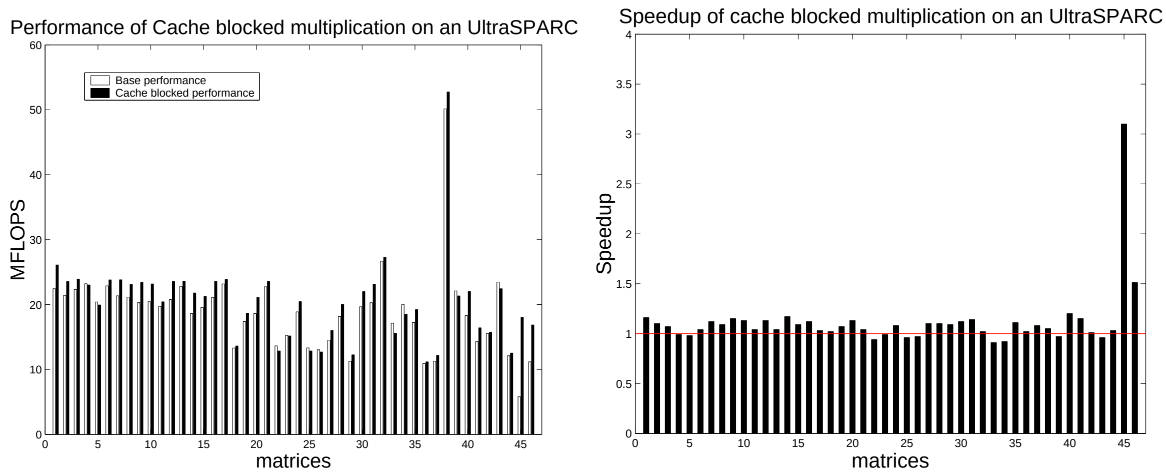


Figure 4.8: **Performance of cache blocked multiplication on an Ultra SPARC I.**

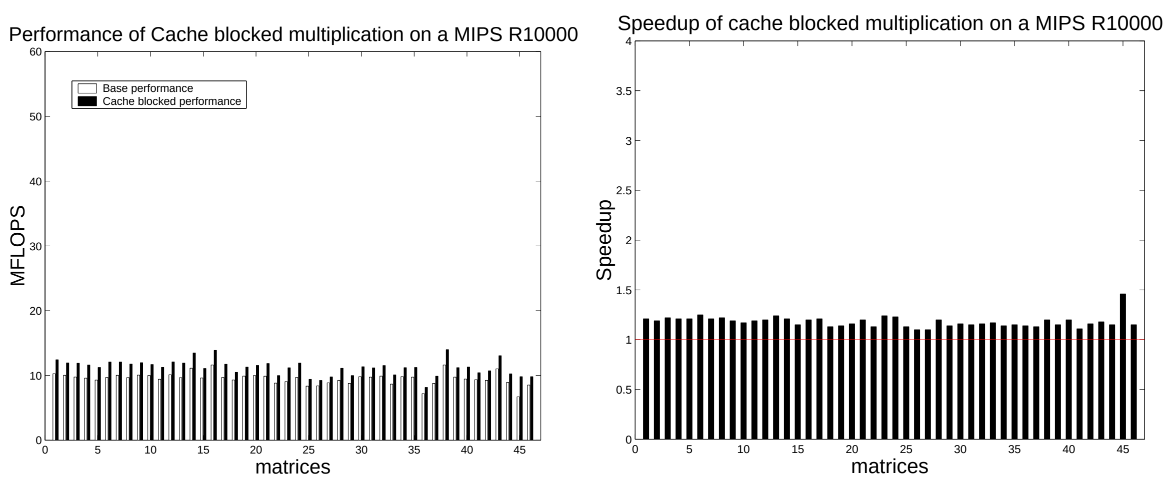


Figure 4.9: **Performance of cache blocked multiplication on a MIPS R10000.**

system used had caches four times larger than those of the UltraSPARC. And it is also observed that the performance of the LSI matrix is noticeably good on an UltraSPARC, where its speedup reaches 3.1. This is caused by particularly slow base performance (5.8 Mflops) of the LSI matrix. The reason behind this is not clearly understood, but it seems to be due to the particular size of the matrix and the way it interferes in the cache.

4.2.2 Performance on Random Matrices

As stated earlier, cache blocking is extremely beneficial to the LSI matrix, while it does not improve the performance of other matrices in the test matrix set. In order to explain why cache blocking is particularly effective on the LSI matrix, we measured the performance of cache blocking on synthetically generated random matrices.

The nonzero pattern of the LSI matrix is unusual compared to most scientific applications, in that it has little discernible structure. Combined with the fact that the size of the matrix is very large, the performance of multiplication on LSI matrix before optimization is very low (5.8 Mflops) relative to the other matrices (10–25 Mflops). So, cache blocking seems to be effective on large matrices with evenly distributed nonzeros, without spots where nonzero elements are clustered, or with only a few such spots.

As further evidence of this previous conjecture, we have generated large synthetic sparse matrices whose nonzero elements were randomly distributed, and measured their performance on these random matrices, varying the density of the nonzero elements. The results are shown in figure 4.12, with the x-axis varying the density of nonzero elements between 0.02% and 0.26%. The size of the random matrix was $64K \times 64K$, and the performance was measured for different cache block sizes. The performance of LSI multiplication for the same cache block sizes are shown in the same figure as separate points above $x = 0.15\%$, the density of the LSI matrix. The performance characteristics of the LSI matrix are visibly similar to those of a random matrix.

4.2.3 Measurement of Randomness of a Sparse Matrix

A performance study using a set of test matrices found that a sparse matrix tends to benefit more when the nonzero structure of the matrix is random. So, it is useful to develop a measurement of how randomly the nonzero elements are spread in the sparse matrix.

Matrix		Matrix Size	NZ	Block Size	Mflops	Speedup
1	45	1000x1000	1000000	512x512	26.10	1.16
2	40	21200x21200	1488768	8192x8192	23.54	1.10
3	38	16146x16146	1015156	65536x65536	23.93	1.07
4	4	30237x30237	1450163	256x256	23.02	0.99
5	37	62424x62424	1717792	65536x65536	19.94	0.98
6	6	13965x13965	968583	1024x1024	23.80	1.04
7	7	24696x24696	1751178	1024x1024	23.82	1.12
8	16	54870x54870	2677324	128x128	23.09	1.09
9	1	45330x45330	3213618	128x128	23.42	1.15
10	8	52329x52329	2698463	256x256	23.17	1.13
11	39	23560x23560	484256	512x512	20.42	1.04
12	43	19779x19779	1328611	512x512	23.56	1.13
13	41	16614x16614	1096948	128x128	23.61	1.04
14	25	4134x4134	94408	16384x16384	21.79	1.17
15	44	41092x41092	1683902	4096x4096	21.26	1.09
16	26	2529x2529	90158	4096x4096	23.56	1.12
17	19	22560x22560	1014951	256x256	23.86	1.03
18	23	17758x17758	126150	2048x2048	13.62	1.02
19	24	4929x4929	33185	2048x2048	18.67	1.07
20	11	10672x10672	232633	8192x8192	21.10	1.13
21	36	7320x7320	324784	256x256	23.56	1.04
22	2	13935x13935	63679	65536x65536	12.85	0.94
23	3	13436x13436	94926	4096x4096	15.16	0.99
24	5	9540x9540	207308	16384x16384	20.43	1.08
25	9	74752x74752	596992	256x256	12.85	0.96
26	17	36057x36057	227628	32768x32768	12.70	0.97
27	18	36519x36519	326107	65536x65536	16.00	1.10
28	20	12328x12328	342828	32768x32768	20.02	1.10
29	21	26068x26068	177196	1024x1024	12.25	1.09
30	28	3937x3937	25407	16384x16384	21.98	1.12
31	29	3937x3937	25407	16384x16384	23.14	1.14
32	30	3312x3312	20793	8192x8192	27.25	1.02
33	32	5005x5005	20033	32768x32768	15.58	0.91
34	33	2205x2205	14133	1024x1024	18.51	0.92
35	34	3564x3564	22316	8192x8192	19.21	1.11
36	35	76480x76480	329762	16384x16384	11.17	1.02
37	42	26064x26064	177168	128x128	12.17	1.08
38	27	765x765	24382	16384x16384	52.75	1.05
39	31	991x991	6027	16384x16384	21.33	0.97
40	10	31802x31802	2164210	512x512	22.01	1.20
41	12	9648x77137	260785	4096x4096	16.41	1.15
42	13	8926x73948	246614	128x128	15.75	1.01
43	14	3000x13525	50284	32768x32768	22.44	0.96
44	15	15240x72600	304800	512x512	12.53	1.03
45	22	10000x255943	3712489	16384x16384	18.03	3.10
46	46	64000x64000	6144000	16384x16384	16.85	1.51

Figure 4.10: Summary of cache blocking optimization on an UltraSPARC I

Matrix		Matrix Size	NZ	Block Size	Mflops	Speedup
1	45	1000x1000	1000000	8192x8192	12.43	1.21
2	40	21200x21200	1488768	512x512	11.94	1.19
3	38	16146x16146	1015156	2048x2048	11.89	1.22
4	4	30237x30237	1450163	32768x32768	11.61	1.21
5	37	62424x62424	1717792	8192x8192	11.23	1.21
6	6	13965x13965	968583	2048x2048	12.08	1.25
7	7	24696x24696	1751178	128x128	12.09	1.21
8	16	54870x54870	2677324	65536x65536	11.76	1.22
9	1	45330x45330	3213618	128x128	11.97	1.19
10	8	52329x52329	2698463	256x256	11.69	1.17
11	39	23560x23560	484256	256x256	11.25	1.19
12	43	19779x19779	1328611	1024x1024	12.10	1.20
13	41	16614x16614	1096948	4096x4096	11.92	1.24
14	25	4134x4134	94408	1024x1024	13.48	1.21
15	44	41092x41092	1683902	512x512	11.07	1.15
16	26	2529x2529	90158	4096x4096	13.88	1.20
17	19	22560x22560	1014951	512x512	11.73	1.21
18	23	17758x17758	126150	8192x8192	10.48	1.13
19	24	4929x4929	33185	32768x32768	11.29	1.14
20	11	10672x10672	232633	32768x32768	11.54	1.16
21	36	7320x7320	324784	128x128	11.84	1.20
22	2	13935x13935	63679	2048x2048	9.98	1.13
23	3	13436x13436	94926	65536x65536	11.19	1.24
24	5	9540x9540	207308	65536x65536	11.91	1.23
25	9	74752x74752	596992	256x256	9.40	1.13
26	17	36057x36057	227628	65536x65536	9.23	1.10
27	18	36519x36519	326107	256x256	9.78	1.10
28	20	12328x12328	342828	256x256	11.10	1.20
29	21	26068x26068	177196	256x256	9.98	1.14
30	28	3937x3937	25407	1024x1024	11.34	1.16
31	29	3937x3937	25407	512x512	11.18	1.15
32	30	3312x3312	20793	32768x32768	11.53	1.16
33	32	5005x5005	20033	1024x1024	10.10	1.17
34	33	2205x2205	14133	512x512	11.20	1.14
35	34	3564x3564	22316	4096x4096	11.23	1.15
36	35	76480x76480	329762	16384x16384	8.15	1.14
37	42	26064x26064	177168	512x512	9.89	1.13
38	27	765x765	24382	16384x16384	13.99	1.20
39	31	991x991	6027	8192x8192	11.20	1.15
40	10	31802x31802	2164210	512x512	11.31	1.20
41	12	9648x77137	260785	2048x2048	10.42	1.11
42	13	8926x73948	246614	256x256	10.72	1.16
43	14	3000x13525	50284	65536x65536	13.05	1.18
44	15	15240x72600	304800	2048x2048	10.26	1.15
45	22	10000x255943	3712489	65536x65536	9.81	1.46
46	46	64000x64000	6144000	32768x32768	9.81	1.15

Figure 4.11: Summary of cache blocking optimization on a MIPS R10000

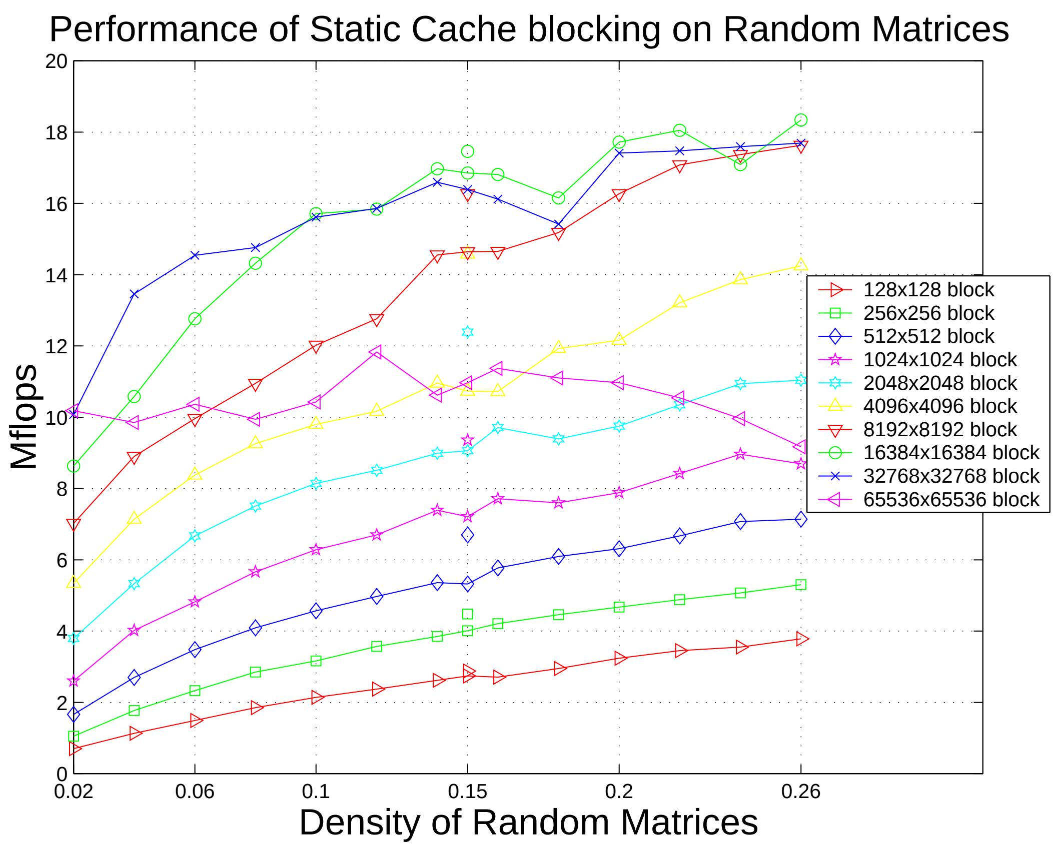


Figure 4.12: **Performance of cache-blocked multiplication on random matrices:** It is measured for $64K \times 64K$ random matrices with varying densities 0.02–0.26% on an UltraSPARC I. Each line represents different cache block sizes, and the separate points at 0.15% show the performance of the LSI matrix, measured in Mflops.

We used a bisection of the matrix to measure this property in our experiment. The ratio of the number of edge-cuts to the number of edges is used as a quantified measurement of this property. If the ratio is large, the matrix is considered to be randomly distributed. But there is a limitation to using a bisection of a matrix: a matrix is interpreted as an undirected graph when it is square and symmetric. Since we include rectangular, non-symmetric matrices in our experiment, we used a hyper-graph representation of the matrix to perform bisection of the matrix, due to its ability to represent a rectangular or non-symmetric matrix. In the hyper-graph representation of a matrix, each row (column) becomes one hyper-edge, and one hyper-edge connects multiple nodes which are nonzero columns (rows) in the row (column). We used the hMETIS hyper-graph partitioner package [40], developed at the University of Minnesota, to bisect a hyper-graph. This package implements a multilevel algorithm [33] for graph partitioning.

In figure 4.13 we show the measurement of the ratio for a subset of test matrices. The matrices are matrices 11, 15, 4, 6, 7, 10, 8, 14, 5, 29, 44, 45 and 46 in figure 2.5. The first ten matrices are also used in section 3.2.2 to validate the performance model for register blocking. Matrix 44 is a linear programming matrix, matrix 45 is the LSI matrix, and matrix 46 is a random matrix. We can verify that the random matrix has the highest value, which is close to 1, and the LSI matrix also has a high value, but not as high as that of the random matrix. Other matrices have very low values.

4.3 Overhead of Cache Blocking

The overhead of reorganizing the LSI matrix for static cache blocking is shown in figure 4.14. For comparison, the time spent for non-optimized multiplication and for cache-blocked multiplication are shown as well. For the optimal size of a cache block (16K $\times$ 16K), the ratio of reorganization time to cache blocking benefit of cache blocking (the difference between non-optimized computation time and cache blocked computation time) is 1.25. Multiplication is expected to be repeated many times in this application, so this one-time overhead is easily amortized.

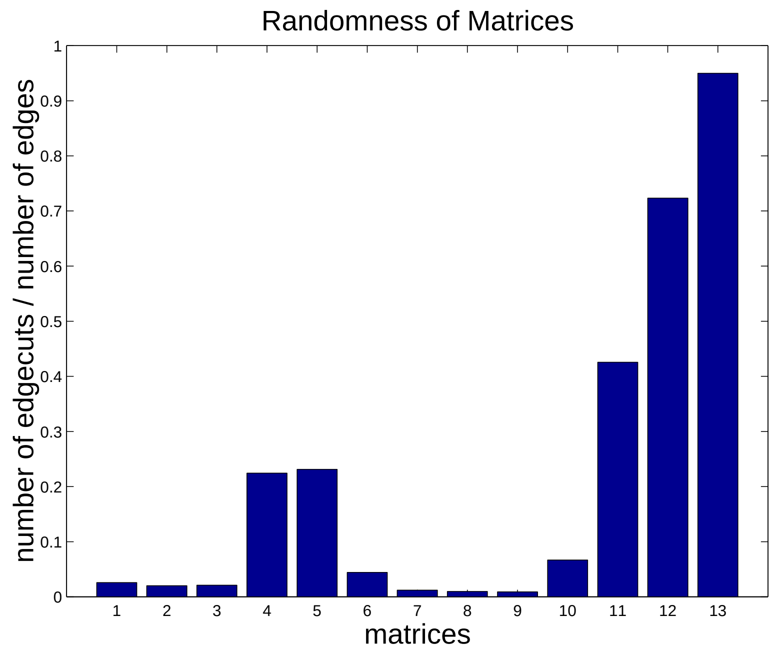


Figure 4.13: **Randomness measure on test matrices**

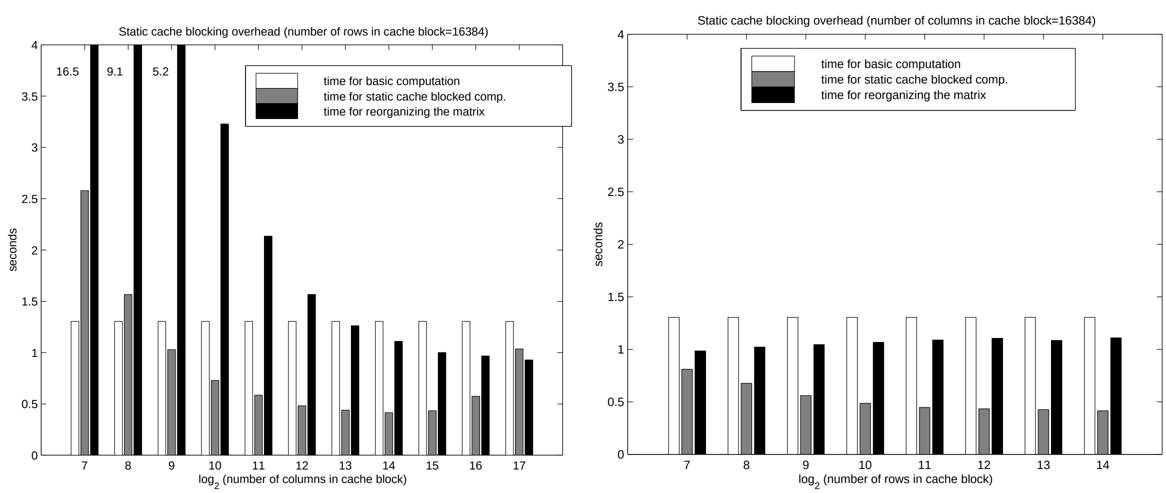


Figure 4.14: **Overhead of reorganizing the matrix for static cache-blocking on the LSI matrix.** The left figure shows the overhead varying the length of cache block columns from 128 to 126K. The right figure shows the overhead varying the length of cache block rows from 128 to 16K. The three bars represent time spent for non-optimized computation (white bar), time spent for static cache-blocked computation (gray bar), and overhead of reorganizing the sparse matrix(black bar), respectively. The three black bars on the left exceed 4 seconds, and the values are written at the top.

4.4 Combining Cache Blocking with Register Blocking

After this study of register blocking and cache blocking optimization, we naturally would like to assess the effectiveness of combining cache blocking and register blocking. We implemented a combination of register and cache blocking and applied it to the set of test matrices. The result is presented in figures 4.15 and 4.16. These figures show the speedup of combined optimization, with comparison to the speedups of register blocking and cache blocking alone on an UltraSPARC and a MIPS R10000.

We observe that the combined optimization generally fails to improve the performance of a single optimization. Although the reason is unclear, we make following conjectures. While register blocking achieves speedup by reducing the number of load/store operations, cache blocking improves performance by reducing the cost of those memory operations. Rather than the benefit from those optimizations coming from separate sources, these optimizations obtain performance benefits from the same source. When they are combined, the effect of one optimization is absorbed by the other, and fails to improve the performance. So, the combination of the register blocking and cache blocking is not recommended.

4.5 Summary

We have introduced a notion of cache blocking for sparse matrix-vector multiplication. We aimed to improve cache performance by organizing the vector products in a way that the elements of vector x can be kept in the cache and reused for the vector products of the next row before they are removed from the cache.

We considered two different implementations of cache blocking, *static* and *dynamic*. In *static* cache blocking the matrix is preprocessed to reorganize itself in units of cache blocks, and in *dynamic* cache blocking, this was done during multiplication. From the experiments, static cache blocking was seen to be much better because there was no overhead due to managing extra pointers during execution.

In the experiments performed with our full suite of matrices provided for this study, cache blocking exhibited a dramatic performance improvement on a particular matrix used in a data mining application, in which nonzero structures are random. Therefore we devised a metric to determine how randomly the nonzero elements were distributed in the sparse

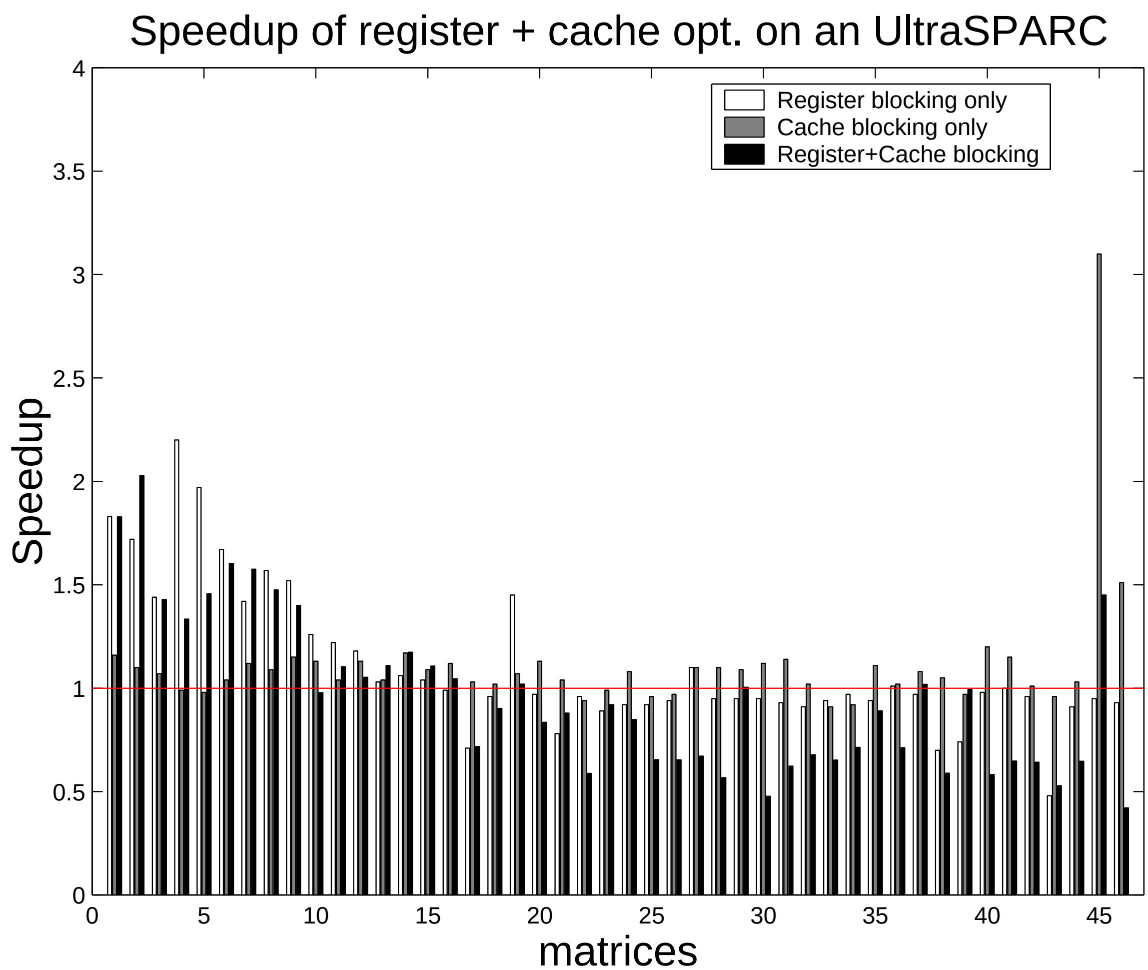


Figure 4.15: **Speedup of register and cache blocked multiplication on an UltraSPARC I**

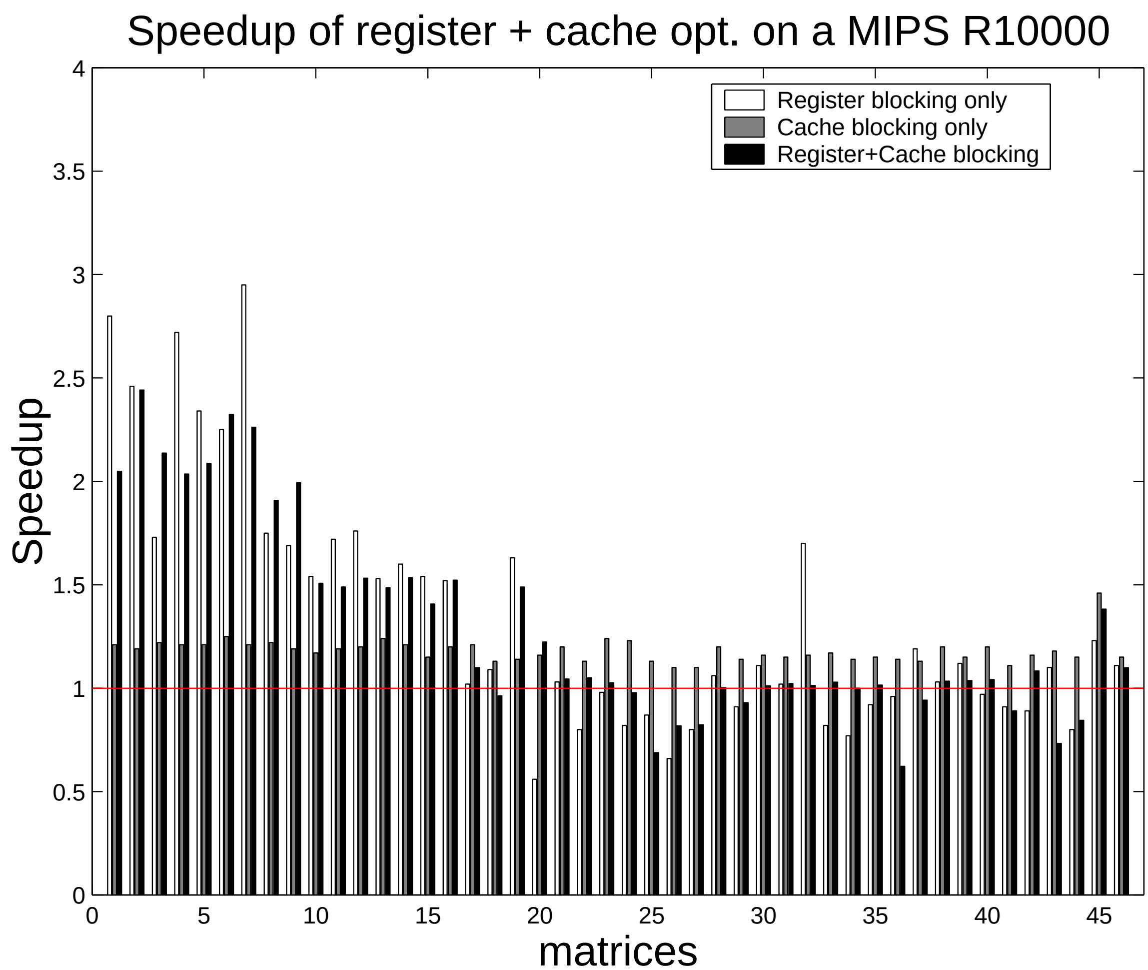


Figure 4.16: **Speedup of register and cache blocked multiplication on a MIPS R10000**

matrix, using the bisection of the matrix.

We also combined register blocking and cache blocking optimizations in such a way that a sparse matrix is blocked for registers and the reorganized matrix is again blocked for cache. The performance result ware disappointing since the combination did not improve the best performance of either optimization.

Chapter 5

Multiplication by Multiple Vectors

In this chapter we consider a generalization of matrix-vector multiplication in which the sparse matrix is multiplied by a set of vectors. This operation, while less common than the single vector case, is important in several applications. In particular, it occurs in practice when there are multiple right-hand sides in an iterative solver, or in blocked eigenvalue algorithms, such as block Lanczos [29, 30, 31, 49, 3] or block Arnoldi [64, 63, 46, 3]. Another application is image segmentation in videos, where a set of vectors is used as the starting guess for a subsequent frame in the video [66].

Multiplying a sparse matrix by a set of vectors has much more potential for memory hierarchy optimizations than the matrix-vector case. Matrix-vector multiplication accesses each matrix element only once, whereas a matrix times a set of k vectors will access each matrix element k times. While there is much more potential for high performance with multiple vectors, the advantage will not be exhibited in straightforward implementations without memory hierarchy optimizations. We therefore extend the optimization techniques for register and cache blocking to handle multiple vectors. In particular, we use the same register and cache blocked matrix formats, but change the code to access elements of the vectors, allowing matrix elements to be reused.

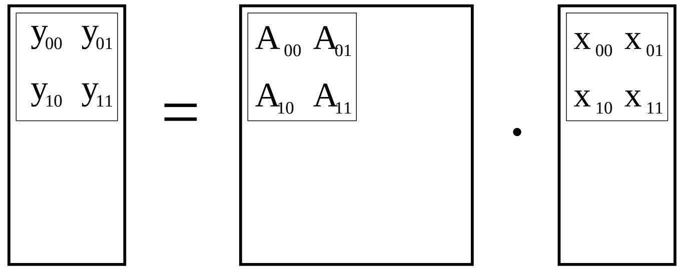
In sections 5.1 and 5.2 we describe the algorithms for register and cache optimizations with multiple vectors. In section 5.3 we examine processes which can automatically determine whether to block and what block sizes should be used. These techniques are largely the same as those used for the single vector case, although the actual decisions for a particular matrix and machine may be different. In addition, as discussed in section 2.1.3, there is a tradeoff between the speed of each multiplication and the convergence rate of algorithms that use multiple vectors. As the number of vectors increases, the speed (Mflop rate) of each iteration in the algorithm increases. Each individual vector multiplication is faster, but the number of iterations to convergence (measured by the number of individual vectors multiplied) may increase. With this tradeoff in mind, we present extensive performance data in section 5.4.

5.1 Register Blocking with Multiple Vectors

The use of multiple vectors essentially turns the problem into matrix-matrix multiplication. The performance numbers from optimized matrix algorithms shown in chapter 3 (figure 3.8) indicate as much as an order of magnitude difference between matrix-matrix and matrix-vector performance for dense matrices. As we have seen, the sparse case typically does not exhibit the same absolute performance as the dense one; however, the same principle of increasing the number of arithmetic operations per matrix entry can improve performance.

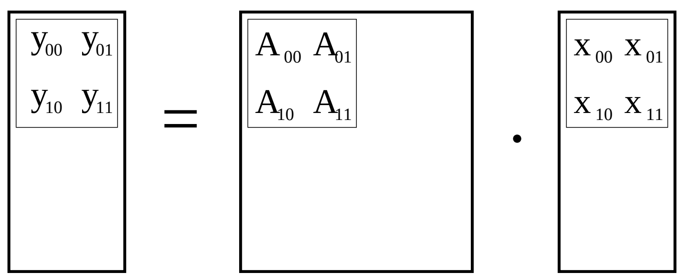
When multiplying a sparse matrix times a set of vectors, the code for multiplication by a single vector can be repeatedly used, but the extra locality advantages are not likely to be exhibited under such conditions. Figure 5.1 illustrates the sequence of steps for the algorithm, showing that two uses of the same matrix element are nz steps apart. Multiplication can be optimized for the memory hierarchy by moving those operations together in time, as shown in figure 5.2.

Our code generator produces code specifically for register-blocked multiplication for a fixed set of vectors. The number of vectors is fixed and all of the loops across the vectors are fully unrolled. For example, rather than producing the 2×2 register-blocked multiplication code in figure 5.3 it generates the code in figure 5.4 for multiplying a 2x2 register-blocked matrix times 2 vectors. This code produces the access order shown in figure 5.2. Because the loop is unrolled for the number of vectors, the code generator produces different versions depending on the number of vectors. The strategy of fully unrolling this loop is used because the code generator is creating the inner kernels of a large computation; if the number of vectors is very large, the loop over the vectors would be strip-mined, with the resulting inner loop becoming one of these unrolled loops.



$$
\begin{aligned}
&(1) \quad y_{00} = A_{00}\,x_{00} + A_{01}\,x_{10} \\
&(2) \quad y_{10} = A_{10}\,x_{00} + A_{11}\,x_{10} \\
&\qquad \cdots \\
&(nz+1) \quad y_{01} = A_{00}\,x_{01} + A_{01}\,x_{11} \\
&(nz+2) \quad y_{11} = A_{10}\,x_{01} + A_{11}\,x_{11}
\end{aligned}
$$

Figure 5.1: **Sequence of steps in single vector code:** In the example, a 4×4 sparse matrix with nz nonzero elements is being multiplied by 2 vectors. The matrix and code are register-blocked using 2×2 blocks.



$$
\begin{aligned}
&(1) \quad y_{00} = A_{00}\,x_{00} + A_{01}\,x_{10} \\
&(2) \quad y_{10} = A_{10}\,x_{00} + A_{11}\,x_{10} \\
&(3) \quad y_{01} = A_{00}\,x_{01} + A_{01}\,x_{11} \\
&(4) \quad y_{11} = A_{10}\,x_{01} + A_{11}\,x_{11}
\end{aligned}
$$

Figure 5.2: **Sequence of multiple vector code:** This example is the same as that in figure 5.1, except that the code has been reorganized to use each element twice (once per vector) before moving to the next element.

```
void bsmvm_2x2 (int m,
               int *row_start, int *col_idx, double *value,
               double *src, double *dest)
{
  int i, j;

  for (i=0; i<m; i++,dest+=2)
  {
    register double d0, d1;
    d0 = dest[0];
    d1 = dest[1];
    for (j=row_start[i]; j<row_start[i+1]; j++,col_idx++,value+=4)
    {
      d0 += value[0] * src[*col_idx+0];
      d0 += value[1] * src[*col_idx+1];
      d1 += value[2] * src[*col_idx+0];
      d1 += value[3] * src[*col_idx+1];
    }
    dest[0] = d0;
    dest[1] = d1;
  }
}
```

Figure 5.3: **Code for multiplying a register-blocked matrix times a single vector:** The variables *row_start*, *col_idx*, and *value* arrays represent the data structure for a register blocked sparse matrix with m block rows, while *src* and *dest* are the multiplier and result vectors, respectively.

```
void bsmvm_m_2x2_2_once (int m, int *row_start, int *col_idx, double *value,
               int src_len, double *src, int dest_len, double *dest)
{
  double *src_p, *dest_p;
  int j, row;

  for (row=0; row<m ; row++, row_start++)
  {
    register double t0_0, t0_1;
    register double t1_0, t1_1;
    t0_0 = 0;
    t0_1 = 0;
    t1_0 = 0;
    t1_1 = 0;

    for (j=*row_start; j<*(row_start+1); j++, col_idx++, value+=4)
    {
      src_p = src + (*col_idx);
      t0_0 += src_p[0] * value[0];
      t0_0 += src_p[1] * value[1];
      t1_0 += src_p[0] * value[2];
      t1_0 += src_p[1] * value[3];
      src_p += src_len;
      t0_1 += src_p[0] * value[0];
      t0_1 += src_p[1] * value[1];
      t1_1 += src_p[0] * value[2];
      t1_1 += src_p[1] * value[3];
    }
    dest_p = dest + 2*row;
    dest_p[0] += t0_0;
    dest_p[1] += t1_0;
    dest_p += dest_len;
    dest_p[0] += t0_1;
    dest_p[1] += t1_1;
  }
}
```

Figure 5.4: **Code for multiplying a register-blocked matrix times 2 vectors:** The variables *row_start*, *col_idx*, and *value* arrays represent the data structure for a register-blocked sparse matrix with m block rows, while *src* and *dest* represent the source and destination vector sets, respectively. The *src_len* and *dest_len* parameters indicate the lengths of the *src* and *dest* vectors. (The *src* and *dest* matrices are stored column-wise, which matches the order in which they are built in most applications.)

5.2 Cache Blocking with Multiple Vectors

Cache blocking with multiple vectors can improve the multiplication performance if it is coded so that a cache-sized block of the matrix is multiplied by multiple vectors before the calculation goes on to the next block. As in register blocking, the matrix elements are reused for each vector, but in this case reuse happens at the level of a cache block rather than at the register level. The code for cache blocking with multiple vectors is similar to that of the single vector case, except there is an inner loop that runs over the set of vectors for each matrix element.

The choice of block size for cache blocking may be different for single vectors than for multiple ones. In particular, block size with multiple vectors may have to be smaller than block size with single vectors to allow sufficient room in the cache to hold the relevant elements of each vector.

5.3 Choosing the Right Number of Vectors

The question of how many vectors to use when multiplying by a set of vectors is partly dependent on the application and partly on the performance of the multiplication operation. For example, there may be a fixed limit to the number of right-hand sides in a system of equations to be solved. In blocked algorithms, there may be a trade-off between the speed of the multiplications and the rate of convergence – using a large number of vectors increases the Mflop rate, but the algorithm converges more slowly. In other algorithms, there may be large number of vectors available, and the only question is one of performance: How does one group the vectors into smaller sets to maximize overall performance?

In an automatic optimization framework, the user will need some control over the number of vectors. When numerical convergence or other higher level algorithmic issues are concerned, it may be better to show users the performance for various numbers of vectors and let them decide how to optimize overall performance. If there are a large number of vectors available, and the only concern is performance, then the system may search over different numbers of vectors, given a representative matrix and machine, to find the optimal number of vectors. Although increasing the number of vectors increases the reuse of matrix elements, it also increases register pressure in the register-blocked case, so the performance does not always increase.

If there are a large number of vectors available at the application level, and the only concern is performance, the optimization space is still quite complex because there are several parameters to consider: the number of rows and columns in register blocks, the number of rows and columns in cache blocks, and the number of vectors. And as with previous optimizations, the machine characteristics and sparsity structures of the matrix may significantly affect the optimal settings of these parameters. Given the large number of dimensions in this space, we will take various cuts in which some of the parameters are fixed and others vary.

In this section, we look at the interaction between the register-blocking factors and the number of vectors. This interaction is particularly important because the register-blocked code for multiple vectors unrolls both the register block and multiple vector loops. How effectively the registers are reused in this inner loop is very dependent on the compiler. We will simplify the discussion by looking at two extremes in the space of matrix structures: a dense $1K \times 1K$ matrix in sparse format, and sparse $10K \times 10K$ randomly generated matrices with $200K$ (.2%) of the entries being nonzero. In both cases, the matrices are blocked for registers, which in the random cases means that the $200K$ nonzero entries will be clustered differently, depending on the block size.

Figures 5.5, 5.6, and 5.7 show the effect of changing the block size and the number of vectors on an UltraSPARC I, MIPS R1000, and Alpha 21164, respectively. Each figure shows the performance of register-blocked code optimized for multiple vectors. The left-hand side shows a matrix with random structure, although each randomly generated "element" is a dense block of the size shown in the legend, since the matrix is register-blocked. The right-hand side shows a dense matrix in sparse format, also blocked according to the block size shown in the legend. The x-axis shows the number of vectors and the y-axis gives the Mflop rate. We have collected this data for all block sizes from 1×1 up to 10×10 on each machine, but present only the data for square block sizes, which is sufficient to illustrate our basic points.

For a dense matrix on an UltraSPARC I, shown on the right-hand side of figure 5.5, performance varies wildly, depending on both the block dimensions and the number of vectors. In general, it appears that smaller blocks with a larger number of vectors performs best. This is borne out by a search over all combinations of blocks sizes and number vectors in the range. The optimal point for the dense matrix is a block size of 1×7 with 11 vectors, and a peak rate of 167 Mflops. The unblocked code (shown as 1×1) does surprisingly well

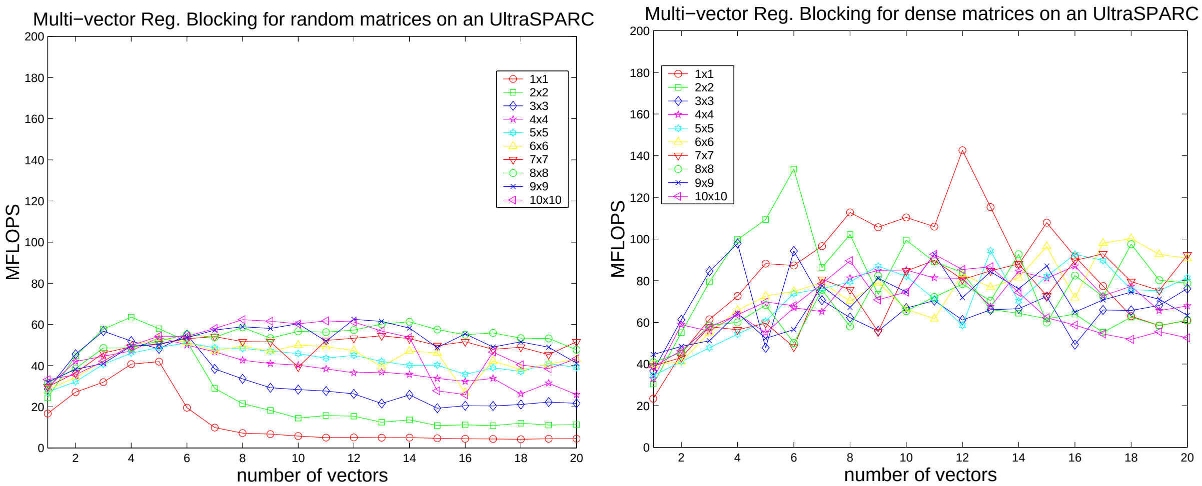


Figure 5.5: **Register-blocked, multiple vector performance on an UltraSPARC I, varying the number of vectors:** The graph on the left uses a random $10K \times 10K$ matrix with $200K$ nonzeros, and the graph on the right uses a $1K \times 1K$ dense matrix in sparse format. Each line indicates a different register blocking. The x-axis shows the number of vectors and the y-axis shows the performance in Mflops.

on the dense matrix, as long as the number of vectors is large (12 is the peak with a rate of 143 Mflops). Although the performance is not as high as 287 Mflops of vendor-optimized BLAS dense matrix-matrix multiplication (DGEMM) for the same matrix on the 200 MHz UltraSPARC I in figure 3.8 from a LAPACK user's guide [2], it is much higher than that of vendor-optimized BLAS dense matrix-vector multiplication (57 Mflops). The performance was comparable to that of vendor-optimized BLAS DGEMM considering that it was not blocked for cache.

The performance of a randomly generated sparse matrix, shown in the left-hand side of figure 5.5, is much lower than the performance of dense case, as expected. The unblocked (1×1) line performs worst of all. More surprising is the clear drop in performance after a peak of approximately 4 or 5 vectors when the block size was small. These observations hold even considering the larger space of non-square block sizes. The performance drop is probably due to the increased size of the inner loop, since a small number of vectors and a small block size will result in a smaller inner loop that is easier for the compiler to analyze and optimize. For larger block sizes, the compiler effects do not vary as much with the vector size, because the loop is already large, and the increasing benefits from the number of vectors (which peak later and are much more gradual) are probably due to the

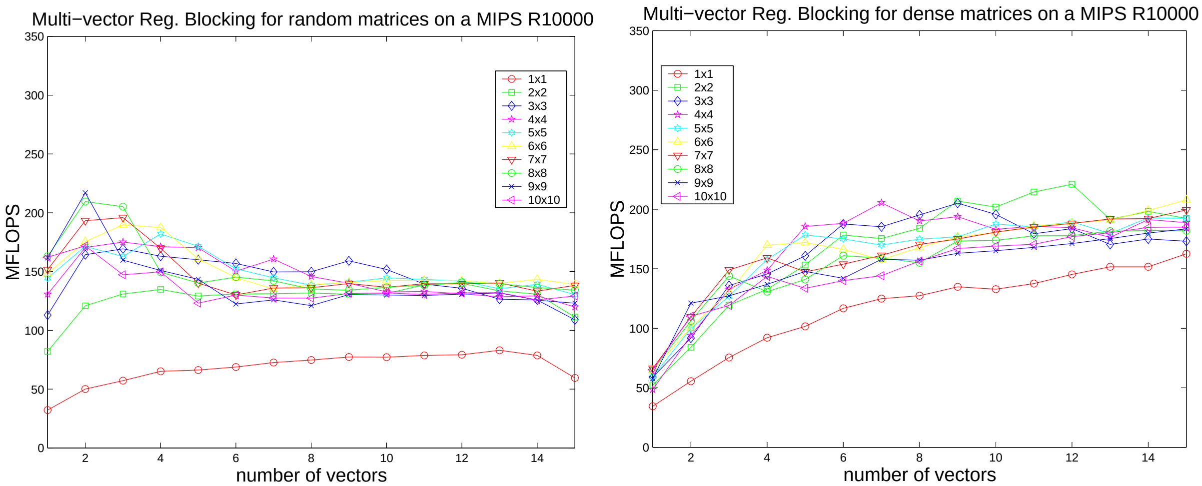


Figure 5.6: **Register-blocked, multiple vector performance on a MIPS R10000, varying the number of vectors:** The graph on the left uses a random $10K \times 10K$ matrix with $200K$ nonzeros, and the graph on the right uses a $1K \times 1K$ dense matrix in sparse format. Each line indicates a different register blocking. The x-axis shows the number of vectors and the y-axis shows the performance in Mflops.

improved reuse of matrix values.

In general, the UltraSPARC numbers indicate that use of multiple vectors significantly improves the performance of matrices on two extremes of regularity and density. Register blocking in combination with multiple vectors is also a good idea, although multiple vectors without blocking produces most of the performance gain (143 rather than 167 Mflops) for the dense matrix. For real sparse matrices, we will also take into the fill overhead for register blocking, which is not seen here – the random matrix was generated to have the desired block size, so there is no fill for blocking.

The graphs for a MIPS R1000K, shown in figure 5.5, show somewhat different factors for that machine. First, as is consistent with our data in other chapters, the MIPS performance is relatively smooth, which makes it a somewhat easier target for optimization. Use of multiple vectors is clearly advantageous, and the best performance is obtained when their use is combined with register blocking. The dense matrix performance increases almost monotonically with the number of vectors, although it appears to be reaching an asymptote under 200 Mflops. As on the UltraSPARC, the dense matrix performs best with relatively small blocks and a large number of vectors: 2×2 with 12 vectors is the best case for square blocks, with a Mflop rate of 221. In considering non-square block sizes as well, the peak

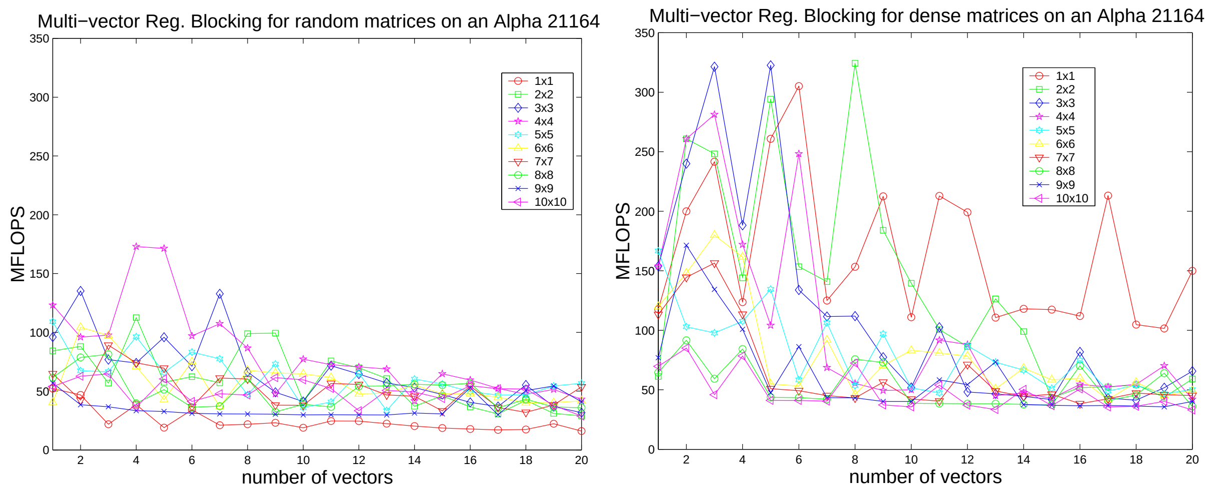


Figure 5.7: **Register-blocked, multiple vector performance on an Alpha 21164, varying the number of vectors:** The graph on the left uses a random $10K \times 10K$ matrix with $200K$ nonzeros, and the graph on the right uses a $1K \times 1K$ dense matrix in sparse format. Each line indicates a different register block size. The x-axis shows the number of vectors and the y-axis shows the performance in Mflops.

performance for the dense matrix is 235 Mflops, obtained with a block size of 2×6 and 14 vectors. Although this is a different block size than was optimal for the UltraSPARC, there are qualitative similarities: both machines use short, fat blocks with a large number of vectors to reach their peak. (For comparison, according to a LAPACK user's guide (figure 3.8), the performance of vendor-optimized BLAS dense matrix-matrix multiplication for the same matrix on 300 MHz MIPS R12000 is 555 Mflops, and that of dense matrix-vector multiplication is 210 Mflops.)

For the random sparse matrices on the MIPS R10000 there is a peak with a small number of vectors and declining performance afterwards, which is even more evident than that on the UltraSPARC. For the R10000, the maximum is at only 2 vectors for larger block sizes, which obtain the best overall performance. It is possible that the increase in inner loop size and complexity causes performance to degrade after the initial benefit from multiple vectors. Why this effect is so much stronger on the random matrix than on the dense one is not entirely clear without more detailed hardware or simulation data from the memory system and compiler. However, the negative impacts of the large loops may be mitigated in the dense matrix because of the overall regularity of memory access.

The performance on the Alpha 21164 is shown in figure 5.7, and, as in the previous chapters, its performance is more sensitive to the optimization parameters than that of the other machines. This is probably due to the extra level of caching, the small size of the L1 cache, and some possible virtual memory (e.g., TLB) interaction. For the dense matrix, relatively small block sizes perform well, e.g., 2×2, 3×3, and even the unblocked 1×1. In considering non-square block sizes, the overall peak performance was 405 Mflops, using a block size of 2×1 and 8 vectors. (The performance of vendor-optimized BLAS for the same matrix on the same processor, reported in a LAPACK user's guide (figure 3.8), is 584 Mflops for dense matrix-matrix multiplication and 36 Mflops for dense matrix-vector multiplication.) For the random sparse matrix, unblocked performance was uniformly bad, although the gap between unblocked and the blocked performance is smaller than that on the other machines. Unlike on the other machines, multiple vectors do not always pay off, as we can see that several of the lines start with a negative slope. However, the effect of a peak at some relatively small number of vectors, in this case four, is also evident for some of the block sizes, including the best one, which is 4×4.

To summarize the results from this section, we note that multiple vectors typically pay off for matrices on both ends of the regularity and density spectrum. For most block sizes and most machines, even changing from one vector to two is a significant improvement. However, with respect to choosing optimization parameters, the dense and random matrices behave very differently. In the next section, we will look at heuristics that use information from one of these synthetic data sets to choose the block size for real matrices. There is also quite a bit of variability across machines in terms of how the parameters should be set. There are two characteristics that appear common across the machines. First, the sparse matrix tends to have a peak with some relatively small number of vectors (2-5), and after that performance declines. Second, the optimal point for the dense matrix is approximately 10 vectors and has a relatively small block size compared to the 8×8 that is best for the single vector code on this matrix.

5.4 Performance Evaluation of Multiple Vectors

In this section we look at the performance benefits of optimizing the code for multiple vectors, first for the register-blocked and then for cache-blocked case. Because the data in chapter 4 shows little evidence that there is any advantage to combining the two kinds of blocking, we do not consider the combination here.

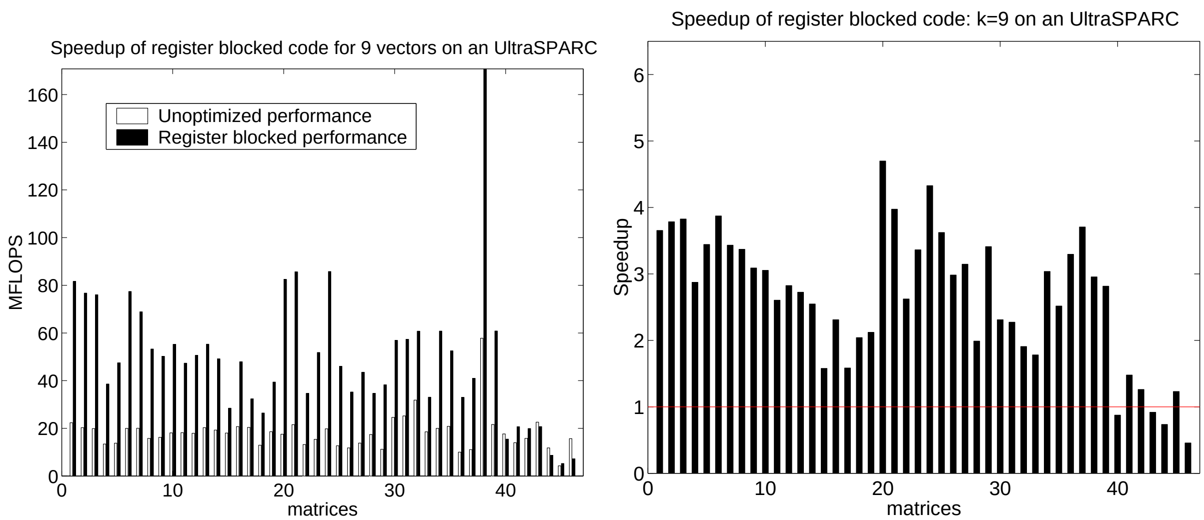


Figure 5.8: **Performance of register-blocked, multiple vector code on an UltraSPARC I:** The left-hand plot shows the performance in Mflops of the unoptimized code and the multiple vector version using 9 vectors. The right-hand plot shows the speedup of multiple vectors relative to unoptimized code.

5.4.1 Register Blocking on a Fixed Number of Vectors

We have generated register blocked codes as shown in figure 5.4 for varying numbers of vectors using a modified code generator, and used those codes for the multiplication of various register-blocked matrices on the UltraSPARC I, MIPS R10000, and Alpha 21164. As shown in the previous section, there is a tight interaction between the parameters for register blocking with multiple vectors, so the question is, how should an automatic optimization system set them. We will start by assuming that the choice of register block size, which is largely dependent on the fill overhead of each possible blocking, should be the same for both multiple and single vectors. All data in this section will use the register-blocking factors that were chosen by our model in chapter 3.

This leaves only the question of how many vectors to use. In practice, this number is often fixed by the higher level application, so from an optimization standpoint, the question is how to subdivide the vectors into smaller groups if there are a very large number. We will start with a fixed number of vectors (9) on all three machines and for all matrices, and show their performance relative to the performance of single vector multiplication. Then we will consider using information from the synthetic matrices to choose the number of vectors.

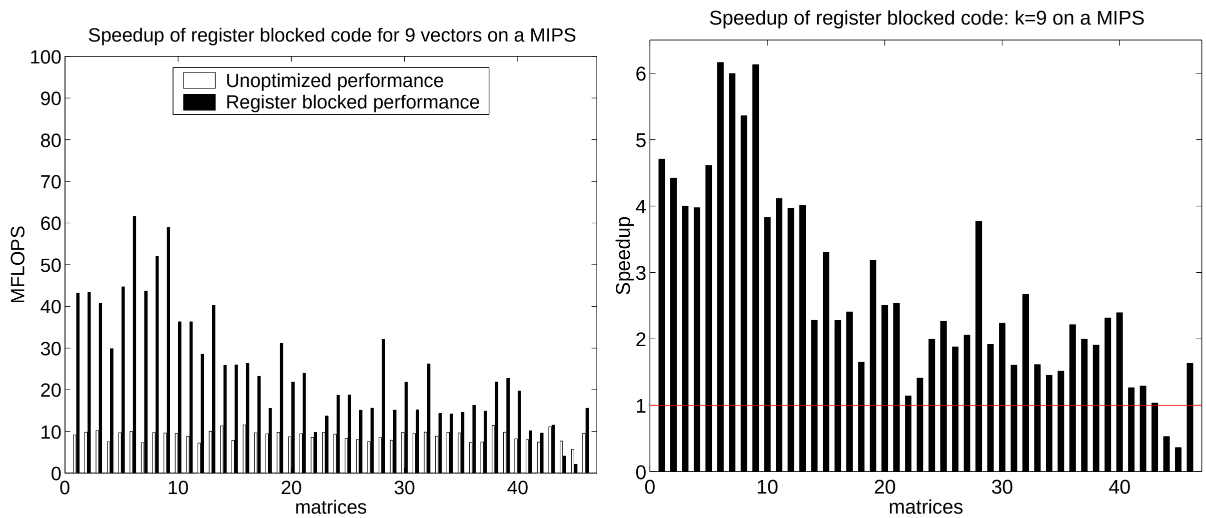


Figure 5.9: **Performance of register-blocked, multiple vector code on a MIPS R10000:** The left-hand plot shows the performance in Mflops of the unoptimized code and the multiple vector version using 9 vectors. The right-hand plot shows the speedup of multiple vectors relative to unoptimized code.

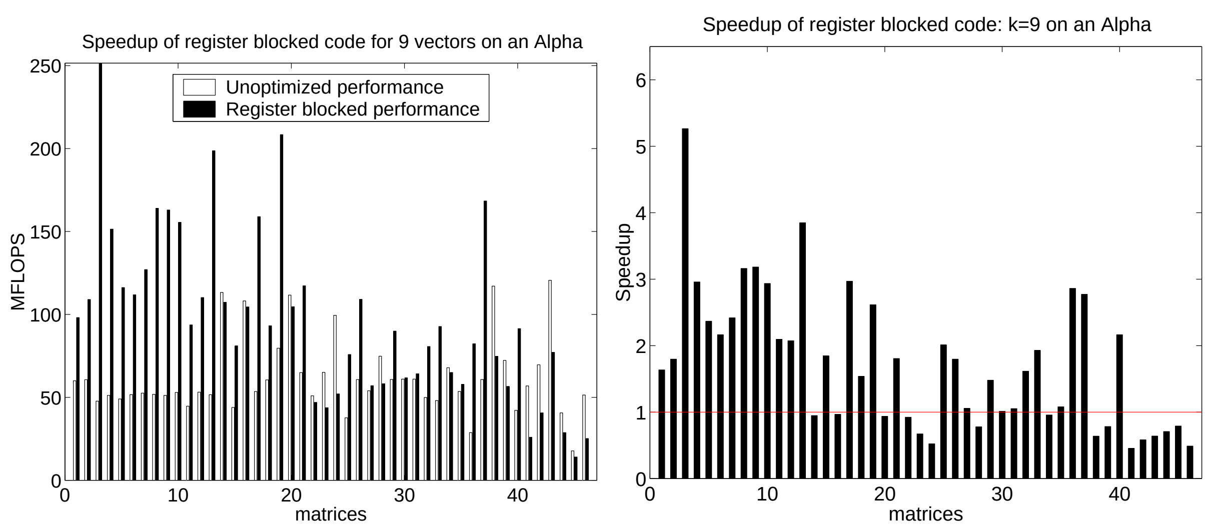


Figure 5.10: **Performance of register-blocked, multiple vector code on an Alpha 21164:** The left-hand plot shows the performance in Mflops of the unoptimized code and the multiple vector version using 9 vectors. The right-hand plot shows the speedup of multiple vectors relative to unoptimized code.

Figures 5.8, 5.9 and 5.10 show the performance and speedup of multiple vectors compared to unoptimized code. We can draw several conclusions from these graphs. First, with 9 vectors, the optimization for multiple vectors typically has a tremendous payoff. On the MIPS R10K, the matrices that benefit most are the lower-numbered ones, which have more regularity and are therefore chosen to have larger register block sizes. On the UltraSPARC, the middle group of matrices sees the highest benefit; these are mostly matrices from scientific simulation problems with some regular patterns, but without the dense sub-blocks that appear naturally in the lower-numbered matrices. The Alpha 21164 continues to be the most challenging machine, both in absolute performance and in speedup from this optimization. On that machine there are several matrices that are degraded by the use of multiple vectors. Given the erratic nature of the detailed performance plots on synthetic matrices, it is possible that good performance is possible with the right number of vectors and the right block size, but our heuristics used in this experiment did not contribute to selection of the right parameters. For the other machines, we are also probably not seeing the best performance possible for multiple vectors, but even the fixed number of vectors and the use of single-vector register block sizes result in very good performance.

5.4.2 Register Blocking Using a Predicted Number of Vectors

The results of section 5.3 were used to develop a strategy for automatic selection of the number of vectors. We continue to use the register block size from the single vector performance model, and for each block size, choose the number of vectors that performed best on one of the synthetic matrices when using the same block size and machine. We consider both the dense and sparse matrix results possible predictors of the number of vectors.

In the tables that follow, there are two possibilities for k, the number of vectors. k_d is the number that performed best for the dense matrix and k_r is the number that performed best for the random sparse matrix. To determine which matrix is a better predictor for the number of vectors, we plot the two Mflop rates for the k_d and k_r. Figures 5.11, 5.12 and 5.13 show these results for each of the three machines. For most of the matrices, the performance of multiplication to k_d vectors are better than the performance of multiplication to k_r vectors. From those figures, it can be seen that k_d is a better choice than k_r for selecting the number of vectors. Although the strategy of using the dense matrix

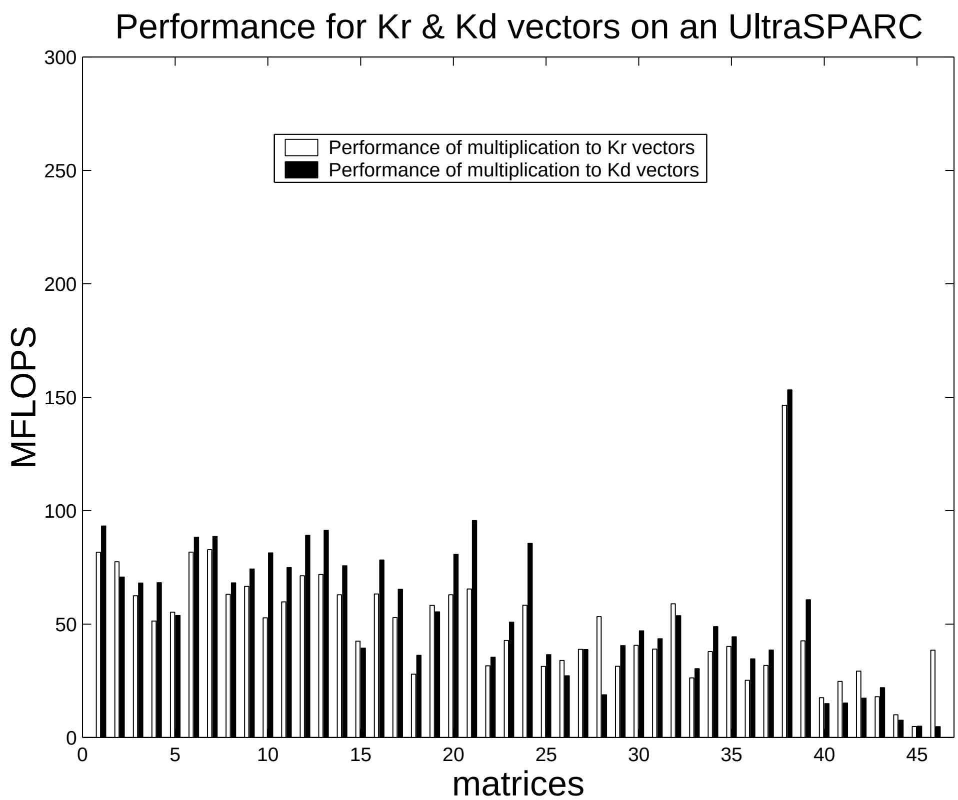


Figure 5.11: **Comparing the dense (k_d) and random sparse (k_r) matrices to choose the number of vectors on an UltraSPARC 1:** The left bar represents the performance when it is multiplied to the peak value for a dense matrix (k_d) and the right bar represents the performance when it is multiplied to the peak value for a random matrix (k_r).

to predict the right number of vectors is good for most matrices in the benchmark suite, it is not good for all of them. For a few matrices, the random matrix was a much better predictor.

5.4.3 Cache Blocking Using a Fixed Number of Vectors

In this section we report on the performance of using multiple vectors in conjunction with cache blocking. As described earlier, the coupling between the number of vectors and the cache size is not as significant as in the register blocking case, because the code is not unrolled, and we are not looking at a restricted resource such as the number of registers. Still we will see that the use of multiple vectors does interact with the choice of cache block size.

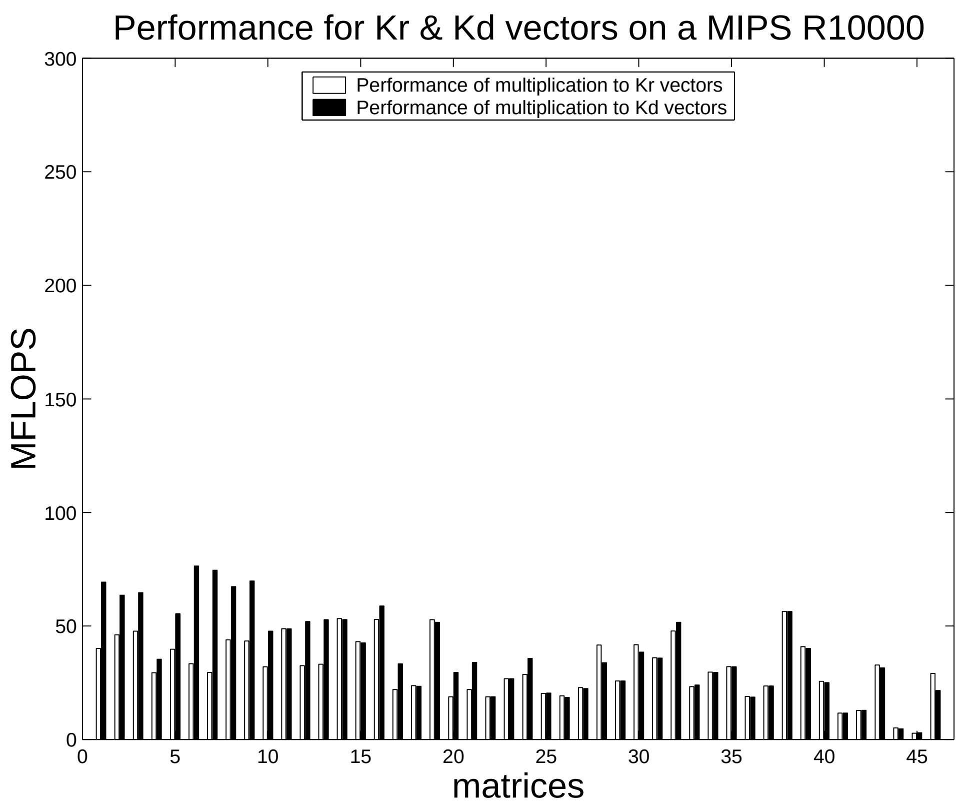


Figure 5.12: **Comparing the dense (k_d) and random sparse (k_r) matrices to choose the number of vectors on a MIPS R10000:** The left bar represents the performance when it is multiplied to the peak value for a dense matrix (k_d) and the right bar represents the performance when it is multiplied to the peak value for a random matrix (k_r).

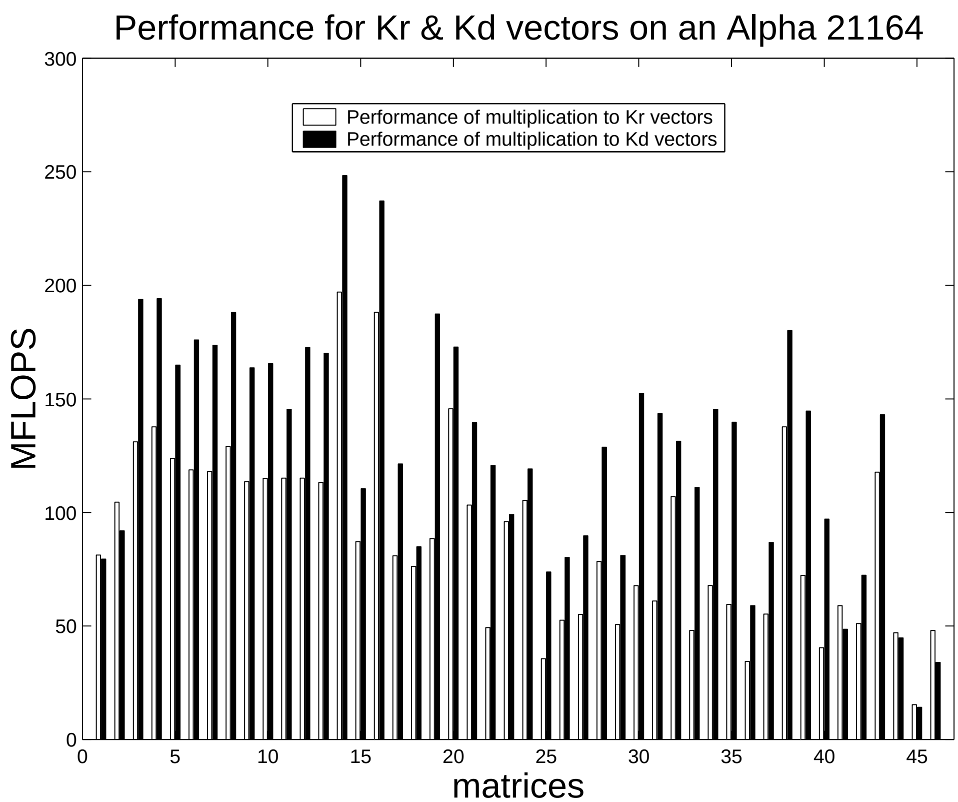


Figure 5.13: **Comparing the dense (k_d) and random sparse (k_r) matrices to choose the number of vectors on an Alpha 21164:** The left bar represents the performance when it is multiplied to the peak value for a dense matrix (k_d) and the right bar represents the performance when it is multiplied to the peak value for a random matrix (k_r).

We use the opposite strategy for exploring this large optimization space by first fixing the number of vectors and then computing a good cache block size. As described above, we expect the number of vectors may be determined at the application level, but here we present data for the case where there are 9 vectors. The cache block size is determined by searching over powers of 2 from 64×64 up to to $64K \times 64K$, including rectangular sizes. We run each of these operations on the matrix and choose the best cache block size. We could use these search strategy because performance is relatively insensitive to small changes in the cache block size.

The optimization was applied only on the UltraSPARC I and MIPS R10000, because of memory limits which were also mentioned in chapter 4. Figures 5.14 and 5.15 show the comparison between unoptimized performance and speedup after cache blocking for multiplication with multiple vectors on Ultra SPARC I and MIPS R10000, respectively. Tables 5.16 and 5.17 show the exact performance numbers and the block sizes that yielded the highest performances.

We make the following observations from the performance results. First, we can see the block sizes that yield the highest performance are much smaller in the multiple-vector case than in the single-vector case in tables 4.10 and 4.11. This is because the matrix elements are being reused in the multi-vector cases, and should therefore be kept in the cache for better performance. In the single vector case, only the source vector needs to be kept in the cache since the matrix values are only used once. In other words, the nonzero elements of every $r_{cache} \times c_{cache}$ block and r_{cache} elements of the source vector should fit in the cache in the multiple vector case, while only r_{cache} elements of the source vector need stay in the cache in the single vector case.

Second, for many matrices, the speedup is close to 200% on an UltraSPARC I, and 140% on a MIPS R10000. For many of these matrices, register blocking with multiple vectors will still be a better option, but for matrices such as LSI (number 45), whose speedup is 300%, cache blocking with multiple vectors is still a better choice, although this is not a further improvement over cache blocking with a single vector. Most block sizes are chosen to be 64×64 on the UltraSPARC, while several larger block sizes are used on the MIPS R10000. This is because the cache on the MIPS R10000 is four times larger than the cache on the UltraSPARC I. While the clock rate of an UltraSPARC and a MIPS R10000 are comparable (167 MHz for the UltraSPARC and 200 MHz for the MIPS R10000), the absolute performance of the UltraSPARC is better than that of the MIPS R10000, because

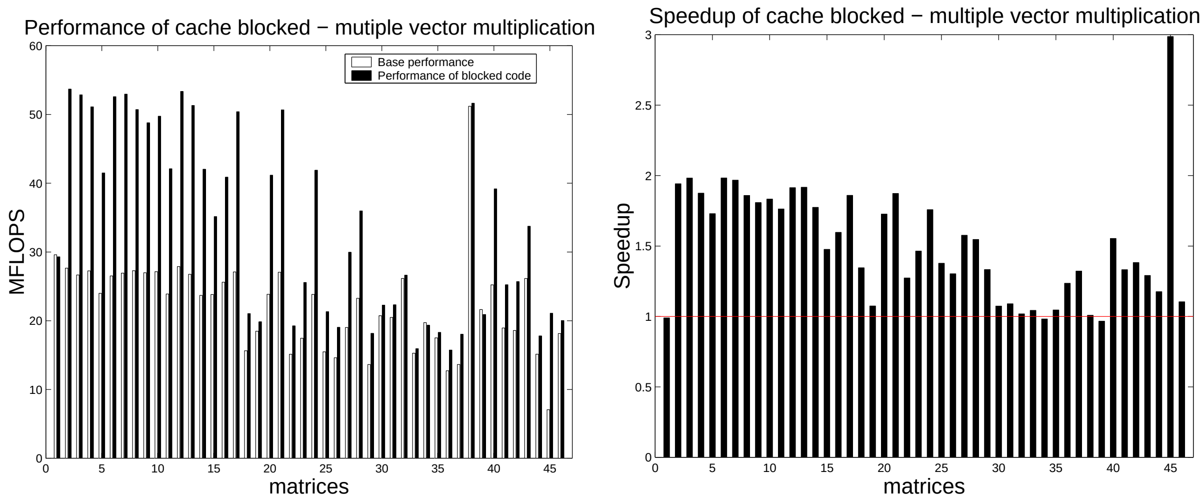


Figure 5.14: **Performance of cache-blocked, multiple-vector multiplication on an Ultra SPARC I:** The left-hand plot shows the absolute performance of the unoptimized and cache-blocked multiple vector code, and the right-hand plot shows the corresponding speedup.

the memory access time is longer on the MIPS R10000 (650 nanoseconds for the MIPS R10000 and 347 nanoseconds for the UltraSPARC).

5.5 Evaluation of Two Multiple-Vector Applications

Many of the benchmarks used in our suite are from other publicly available benchmark suites or other places in which the application context is not entirely known. It is therefore difficult to say which of these applications would benefit from multiple vector optimization. We have therefore collected two additional matrices that originate from applications that use multiple vectors. One is a matrix from global modeling in earth science, whose dimensions are 846968×96300 with 28M nonzeros. The other is a matrix used for text retrieval applications and its dimensions are 13297×5298 with 805K nonzeros. Typically, the first application uses $2 - 5$ vectors and the second application uses $2 - 500$ vectors in each multiplication step. So, for the second application, it was necessary to break the vectors into smaller groups.

We apply the multiple vector optimization to both of these matrices. The performance on an UltraSPARC I, MIPS R10000, and Alpha 21164 for a varying number of vectors are shown in figures 5.18, 5.19, and 5.20. Although the code used is register-blocked,

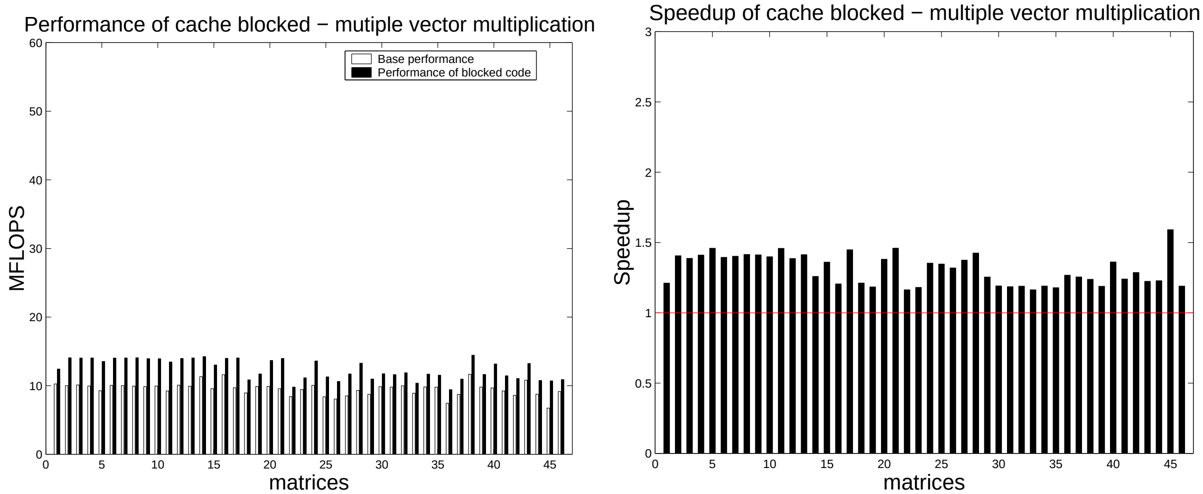


Figure 5.15: **Performance of cache-blocked, multiple-vector multiplication on a MIPS R10000:** The left-hand plot shows the absolute performance of the unoptimized and cache-blocked multiple vector code, and the right-hand plot shows the corresponding speedup.

with the multiple vector loop unrolled, the block size is set to 1×1, because the fill overhead is too large for any other block size.

For the earth science matrix, which would typically have 2-5 vectors available, the performance peaked at three vectors on the UltraSPARC and at four vectors on the MIPS and Alpha. For this matrix, the best strategy from an automatic optimization standpoint is to tell the users the peak for the machine of interest, so they can use the appropriate number of vectors, if possible. Note that the performance for all numbers between 2 and 5 are quite good, so any of the vectors in the typical range would derive significant benefits from multiple vector optimization. However, the numbers predicted by the two synthetic matrices as the appropriate number of vectors do not work well. The random sparse matrix predicts that for a 1×1 blocked matrix, the best performance is obtained using 5 vectors on the UltraSPARC, 13 on the MIPS, and 1 on the Alpha. None of these are very good choices. The dense matrix would have predicted 12 (UltraSPARC), 15 (MIPS) and 6 (Alpha), which are all too high from a performance standpoint and outside the range available in this application. These circumstances emphasize the need for some application programmer control over the number of vectors.

For the text retrieval matrix, the peak performance was at 9 vectors on the Ultra-SPARC, 15 vectors on the MIPS, and 3 on the Alpha. (We could not unroll the loop for

Matrix	Matrix Size	NZ	Block Size	Mflops	Speedup
1	1000x1000	1000000	1000x1000	29.30	1.00
2	21200x21200	1488768	64x64	53.70	1.94
3	16146x16146	1015156	64x64	52.86	1.98
4	30237x30237	1450163	64x64	51.11	1.88
5	62424x62424	1717792	64x64	41.50	1.73
6	13965x13965	968583	64x64	52.58	1.98
7	24696x24696	1751178	64x64	52.96	1.97
8	54870x54870	2677324	64x64	50.71	1.86
9	45330x45330	3213618	64x64	48.80	1.81
10	52329x52329	2698463	64x64	49.75	1.83
11	23560x23560	484256	64x64	42.11	1.76
12	19779x19779	1328611	64x64	53.36	1.91
13	16614x16614	1096948	64x64	51.31	1.92
14	4134x4134	94408	64x64	42.04	1.77
15	41092x41092	1683902	64x64	35.15	1.48
16	2529x2529	90158	64x64	40.89	1.60
17	22560x22560	1014951	64x64	50.40	1.86
18	17758x17758	126150	64x64	21.03	1.35
19	4929x4929	33185	64x64	19.85	1.07
20	10672x10672	232633	64x64	41.17	1.73
21	7320x7320	324784	64x64	50.67	1.87
22	13935x13935	63679	64x64	19.25	1.27
23	13436x13436	94926	64x64	25.56	1.46
24	9540x9540	207308	64x64	41.90	1.76
25	74752x74752	596992	64x64	21.32	1.38
26	36057x36057	227628	64x64	19.04	1.30
27	36519x36519	326107	64x64	29.97	1.58
28	12328x12328	342828	64x64	35.97	1.55
29	26068x26068	177196	64x64	18.17	1.33
30	3937x3937	25407	64x64	22.27	1.07
31	3937x3937	25407	64x64	22.32	1.09
32	3312x3312	20793	64x64	26.63	1.02
33	5005x5005	20033	64x64	15.94	1.04
34	2205x2205	14133	64x64	19.36	0.98
35	3564x3564	22316	64x64	18.31	1.05
36	76480x76480	329762	64x64	15.73	1.24
37	26064x26064	177168	64x64	18.03	1.32
38	765x765	24382	765x765	51.63	1.00
39	991x991	6027	991x991	20.90	1.00
40	31802x31802	2164210	128x128	39.18	1.55
41	9648x77137	260785	64x64	25.23	1.33
42	8926x73948	246614	64x64	25.69	1.38
43	3000x13525	50284	64x64	33.74	1.29
44	15240x72600	304800	256x256	17.80	1.18
45	10000x255943	3712489	2048x2048	21.09	2.99
46	10000x10000	150000	2048x2048	20.03	1.10

Figure 5.16: **Summary of cache-blocked, multiple-vector optimization on an UltraSPARC I**

Matrix	Matrix Size	NZ	Block Size	Mflops	Speedup
1	1000x1000	1000000	8192x8192	12.43	1.21
2	21200x21200	1488768	128x128	14.07	1.41
3	16146x16146	1015156	64x64	14.03	1.39
4	30237x30237	1450163	64x64	14.05	1.41
5	62424x62424	1717792	64x64	13.54	1.46
6	13965x13965	968583	64x64	14.03	1.39
7	24696x24696	1751178	64x64	14.04	1.40
8	54870x54870	2677324	64x64	14.07	1.42
9	45330x45330	3213618	128x128	13.94	1.41
10	52329x52329	2698463	64x64	13.93	1.40
11	23560x23560	484256	64x64	13.46	1.46
12	19779x19779	1328611	64x64	13.97	1.39
13	16614x16614	1096948	64x64	14.04	1.41
14	4134x4134	94408	64x64	14.25	1.26
15	41092x41092	1683902	64x64	13.02	1.36
16	2529x2529	90158	64x64	13.99	1.21
17	22560x22560	1014951	64x64	14.05	1.45
18	17758x17758	126150	128x128	10.85	1.21
19	4929x4929	33185	64x64	11.71	1.18
20	10672x10672	232633	64x64	13.67	1.38
21	7320x7320	324784	64x64	13.97	1.46
22	13935x13935	63679	128x128	9.80	1.16
23	13436x13436	94926	64x64	11.15	1.18
24	9540x9540	207308	64x64	13.60	1.35
25	74752x74752	596992	64x64	11.29	1.35
26	36057x36057	227628	64x64	10.62	1.32
27	36519x36519	326107	64x64	11.71	1.37
28	12328x12328	342828	64x64	13.28	1.42
29	26068x26068	177196	64x64	10.98	1.25
30	3937x3937	25407	64x64	11.75	1.19
31	3937x3937	25407	64x64	11.62	1.19
32	3312x3312	20793	64x64	11.88	1.19
33	5005x5005	20033	128x128	10.37	1.16
34	2205x2205	14133	64x64	11.69	1.19
35	3564x3564	22316	64x64	11.53	1.18
36	76480x76480	329762	64x64	9.43	1.27
37	26064x26064	177168	64x64	10.97	1.26
38	765x765	24382	64x64	14.44	1.24
39	991x991	6027	128x128	11.65	1.19
40	31802x31802	2164210	64x64	13.17	1.36
41	9648x77137	260785	64x64	11.45	1.24
42	8926x73948	246614	64x64	11.05	1.29
43	3000x13525	50284	64x64	13.23	1.22
44	15240x72600	304800	256x256	10.76	1.23
45	10000x255943	3712489	4096x4096	10.71	1.59
46	10000x10000	150000	256x256	10.89	1.19

Figure 5.17: **Summary of cache-blocked, multiple-vector optimization on a MIPS R10000**

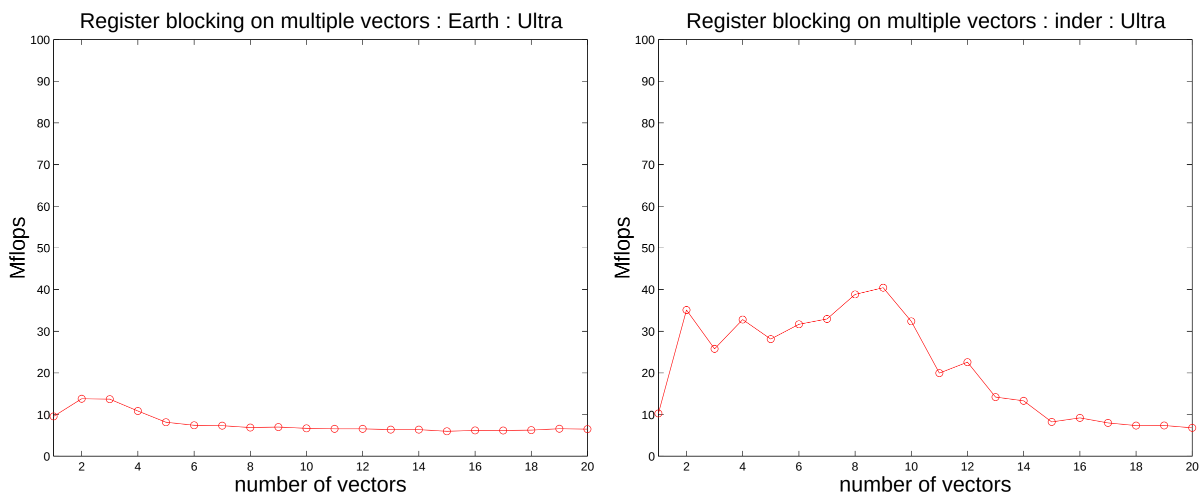


Figure 5.18: **Performance of the multiple-vector optimization on an Earth Science (left) and text retrieval (right) matrix on a UltraSPARC I.**

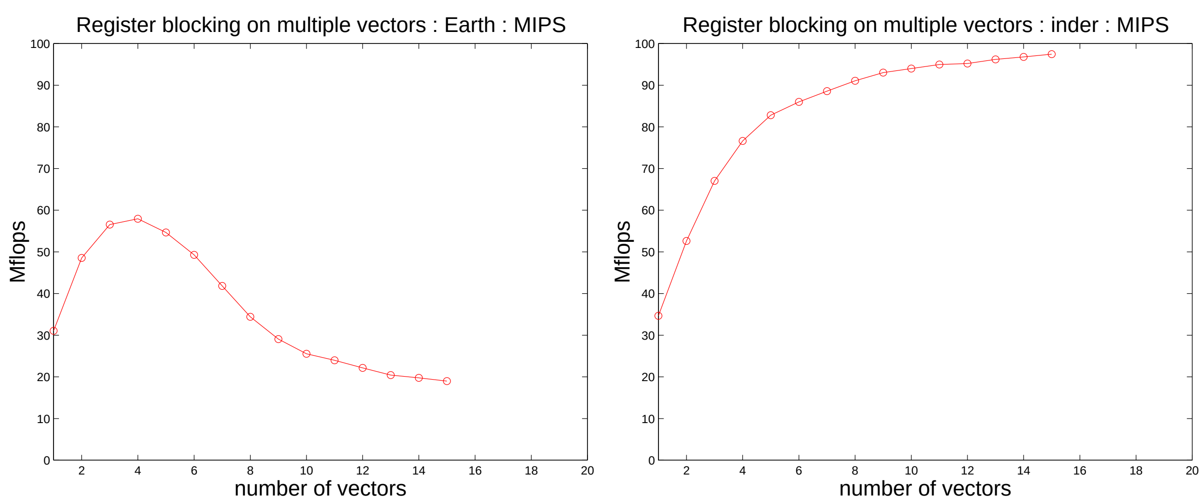


Figure 5.19: **Performance of the multiple-vector optimization on an Earth Science (left) and text retrieval (right) matrix on a MIPS R10000.**

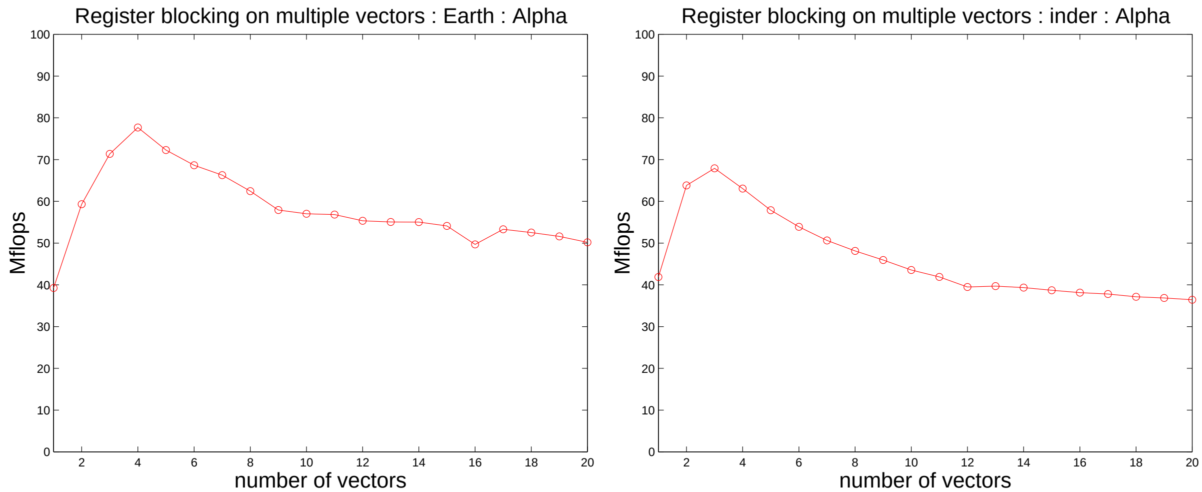


Figure 5.20: **Performance of the multiple-vector optimization on an Earth Science (left) and text retrieval (right) matrix on an Alpha 21164.**

more than 15 vectors on the MIPS, because the compiler failed to compile it.) Because the number of vectors can be anywhere from 2 to 500 in this application, is may be necessary to break the set of matrices into smaller groups of the size indicated from these experiments. The predictions from both the random and dense matrices are therefore possible in this application, although they are still far from optimal. Only the dense matrix prediction of 15 on the MIPS machine is a good one. The conclusion is that if the goals of the system are to provide high performance on each particular matrix, rather than high performance on a large fraction of those available, then the best strategy may be to try many different numbers of vectors, and either automatically choose the best one or provide the user with the performance results.

5.6 Summary

In this chapter we described a generalization of the matrix-vector multiplication problem in which a sparse matrix is multiplied by a set of vectors. We showed that this offers enormous potential for optimizations, just as a matrix-matrix (BLAS-3) operation can run at a significantly higher rate than a matrix-vector (BLAS-2) operation for dense matrices. The optimizations are quite effective when combined with either register or cache blocking. The speedup of register blocking with multiple vectors over unoptimized multiplication was as high as 4.5, and although the speedup of cache blocking with multiple vectors was not as high as that of register blocking, it improved the cache blocking performance for most of the matrices which did not show noticeable speedup with cache blocking with single vector, especially on an UltraSPARC. With most of the matrices in our benchmark suite, which benefited from register blocking but not cache blocking, we show that using multiple vectors adds significant performance to the already optimized code. In addition, cache blocking becomes much more effective when combined with multiple vectors, although the block sizes must be smaller. The reason for this is that with multiple vectors one would like to store the entire block, not just the source vector, in the cache.

We proposed a kind of model to choose the number of vectors, based on the optimal number of vectors for a dense matrix that uses the same blocking factor. Although quite effective on most matrices in our benchmark suite, there are some matrices for which it does not select a reasonable number of vectors. Because of the need for user input which results from numerical concerns related to the algorithm used, we feel that some sort of search over the number of vectors, when combined with input from the user, is the right strategy for an optimization system, even though it would not be fully automatic.

In this chapter, we have chosen the register block size first, independently of the number of vectors and then we have chosen the number of vectors to multiply. Alternatively, we could have used a model in which the number of vectors is used as a parameter, to decide the register block size for the given number of vectors based on the machine-specific profile as shown in figures 5.11, 5.12, and 5.13.

Chapter 6

The Sparsity System

As a result of our study on optimization techniques for sparse matrix-vector multiplication, we have learned that register blocking, cache blocking, and use of multiple vectors can significantly improve performance. We also learned that the right choice of optimizations is crucial to performance improvement, because each optimization technique is beneficial only to a subset of our benchmark matrices and is sometimes detrimental to others. This implies that analysis of the matrix structure and target machine should precede selection of the optimization technique and its parameters. It is unreasonable to expect that the scientists and engineers who are users of sparse matrix operations will also become experts on the optimization techniques described in this thesis. We have therefore built a system, Sparsity, that will choose the optimizations and parameters given little or no input from the user, other than an example matrix and the number of vectors to be multiplied.

Sparsity is an automatic optimization system, and it performs some of the same tasks that an optimizing compiler performs. It does not need to perform the traditional kinds of analyses, because it only compiles one program, sparse matrix-vector multiplication. However, it still performs two other compilation tasks, optimization selection and code generation. Sections 6.1 and 6.2 describe these two components of Sparsity, and section 6.3 provides an overview of the entire system. We then give an overview of the vast domain of sparse matrix formats in section 6.4, which are important in understanding the challenges of building any kind of standardized matrix library or optimization framework. In section 6.5 we describe some of the ways in which Sparsity could be extended in the future.

6.1 Optimization Decisions

In any optimization framework, whether it is a general purpose compiler or a specialized system like SPARSITY, there are various techniques that can be used to make optimization decisions. These include search, general heuristics, and performance models. In our case, the decisions involve choosing both the kinds of optimizations to apply and parameters such as block size. Both the data structure and the code is involved in these transformations.

6.1.1 Search

The simplest solution to selecting transformations on the code is to apply each possible transformation for each possible parameter setting, run the code and measure its performance, and use the minimum setting. In principle, the search may be exhaustive or controlled by some kind of bounded search. For example, one could imagine searching through register block sizes sequentially until the performance starts to decline, or searching over the number of rows and columns using some kind of branch-and-bound technique. Alternatively, one may use a more arbitrary restriction on the search space, such as looking only at block sizes which are powers of two, as was done in cache blocking.

The effectiveness of these search strategies depends on the characteristics of the optimization space. In cache blocking, performance is relatively insensitive to small changes in the cache block size; restriction of the search space may miss the optimal block size, but the resulting performance is probably not much different than for the optimal size. In contrast, the performance can vary wildly given a small change in the register block size, as seen for machines like the Alpha 21164. We therefore believe that exhaustive search over some range of register block size would be necessary under search-based register blocking. Because the overhead of running an exhaustive search for every input matrix is too expensive, in an effort to reduce this overhead, we chose to develop a performance model to complete this phase of selection for the range of register block sizes. In the model, we combine *a priori* knowledge about the machine and information about the matrix.

In SPARSITY, we also use search to determine the optimal number of vectors when the application has many vectors available. This is primarily useful for splitting a large set of vectors (tens or hundreds) into smaller groups. For smaller numbers of vectors the user needs to specify how many are available. Because register blocking with multiple vectors involves two unrolled loops, one over the block and the other over the vectors, making either loop too large can have a serious negative impact on performance.

Search has been effectively used in automatic optimization frameworks for dense matrix kernels [8, 70]. The major disadvantage to search-based optimization is its high cost. While algorithms like simulated annealing are often used for applications like circuit layout, where users are willing to wait for hours or even days for a good solution, such techniques are not employed in the context of general-purpose compilers. Not only is search very expensive, but it requires that the input data be available, which is not the case in static compilation systems.

6.1.2 Heuristics

As an alternative to search, decisions may be based on some kind of heuristic or a performance model. These techniques can also be combined with search to limit the size of the search space.

Heuristics may be based on some knowledge of the machine or algorithm, or on experimental results that indicate it will select good solutions in the search space. For example, we use a somewhat arbitrary cutoff for the maximum block size for register blocking, based on both the observed dense matrix performance and our understanding of the number of registers available on a given machine. Since most of the machines have 32 visible registers, a block size larger than 16×16 is clearly not useful, since we need at least $r + c$ registers to hold the source and destination vectors. We further limit this to 12×12 blocks, because even for the dense matrix benchmark, performance is trailing off at that point, and we have seen no examples of sparse matrices with such large blocks already available. We could probably have limited the space further, and on machines with very small register sets this might be useful for improving the performance of Sparsity.

A second heuristic that we developed the identification of matrices that benefit most from cache blocking. From looking at the nonzero structure in the matrices, we developed a hypothesis that it was most effective on matrices with nearly random structure. We therefore developed a measure of randomness by building a hyper-graph representation of the sparse matrix, bisecting it using a graph partitioning algorithm, and measuring the ratio of the number of edge-cuts to the number of edges. We then chose a threshold for this ratio, which was chosen as 0.4 in our experiments. When combined with some minimum size constraints, this heuristic was able to select those matrices that benefited from cache blocking on the UltraSPARC I, over those that did not. Because most of the matrices gained some smaller benefit on the SGI machine, we did not include this heuristic in the Sparsity system, but instead used the cache block size search described above.

6.1.3 Performance Modeling

A specific class of heuristics are based on performance models, which use some abstraction of the machine performance to predict the performance of the transformed code. There is difficulty in devising a model that is accurate enough to be useful, yet simple enough to evaluate quickly.

The primary example of a performance model within Sparsity is the model of register-blocked performance based on an approximation of the fill overhead, which measures extraneous computation, and dense matrix performance, which is used to approximate the raw performance of the blocked code. Since the estimation of fill overheads for all possible block sizes can be done at one sweep of the sparse matrix, and profiling of the performance of the machine can be done only once for each machine and then reused, model prediction is more efficient that searching for register block sizes by creating each blocked version and measuring the performance of each.

6.2 Code Generation

The second major component of an automatic optimization system is the code generation framework. Because Sparsity is generating code for only one routine, each of the blocked versions could be created by hand, and indeed some of our routines were produced this way. Hand-coding has typically been used for dense matrix kernels, although increasing machine complexity means that an enormous human investment is required to produce each hand-optimized routine. As a result, some vendors have stopped providing routines for their machines, relying instead on their optimizing compilers.

Sparsity takes an intermediate approach to this problem by automating some of the code generation and most of the optimization decisions, but using the special-purpose nature of the system to avoid difficult program analysis problems which are unlikely to work in a sparse matrix context. Specifically, Sparsity uses hand-written codes for some of the drivers and conversion routines as well as the cache-blocked multiplication codes, which are parameterized over the block size. The register blocked multiplication routines, with and without multiple vectors, are generated by a code generation framework, because loops are unrolled for the specific block and vector set size. If this code is parameterized, instead of unrolling the loop, we have found that the performance of multiplication is much lower.

All of the code produced by Sparsity, either by hand or automatically, is C code. (Sparsity itself is written in a combination of Java and C.) The multiplication routines are shown in the previous chapters, in figures 3.3, 5.4 and 4.3. Within the unrolled loops in register blocking, some attempt is made to schedule memory operations by moving certain statements in the code. This code scheduling is not specific to a particular machine or C compiler, although one could imagine more specialized scheduling decisions that search over multiple implementations of the kernels that make up multiplication of a single register block.

If register blocking is selected, then Sparsity produces a hand-written conversion routine and a multiplication routine that is automatically generated. If cache blocking is selected, the code to block the matrix and the multiplication routine are both produced from the hand-written versions. The code generator also produces driver routines, including matrix I/O operations for various file formats, and timing routines, so that users may do their own benchmarking.

6.3 Overview of the Sparsity System

In order to encourage widespread use of our optimized kernels, Sparsity is designed as a web service that provides optimized sparse code for a given matrix and machine. The service aims to choose an appropriate optimization method and parameters and to provide corresponding sparse matrix-vector multiplication code. The general structure of the system is illustrated in figure 6.1. The user may also constrain the optimization system to consider only register blocking, for example, if they believe that it would be much more effective than cache blocking.

Within the Sparsity system, the matrix is tested for several criteria to determine whether register blocking, cache blocking, or both should be applied. As described in chapter 5, the decision to use multiple vectors requires user involvement, and is therefore not fully automatic. If the user does request code for a large number of vectors, an additional optimization step takes place after the other optimization decisions in which the number of

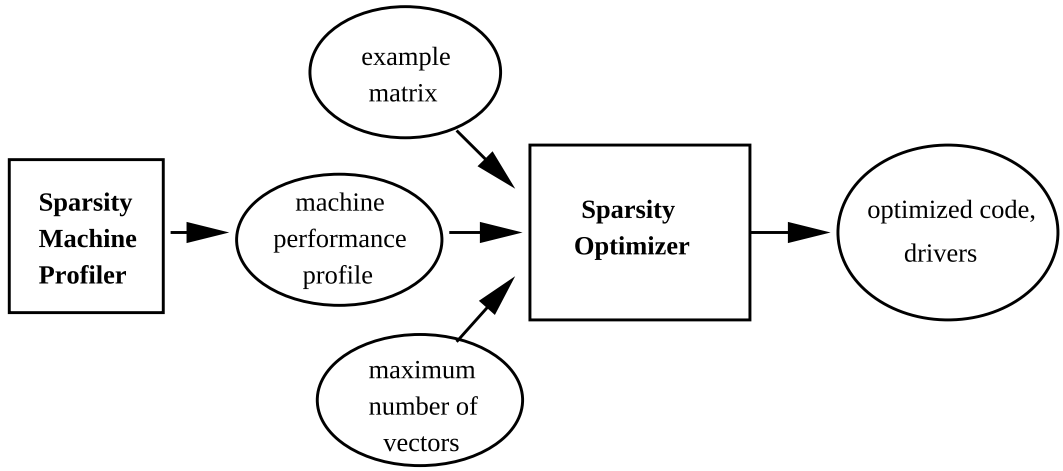


Figure 6.1: **Sparsity system.**

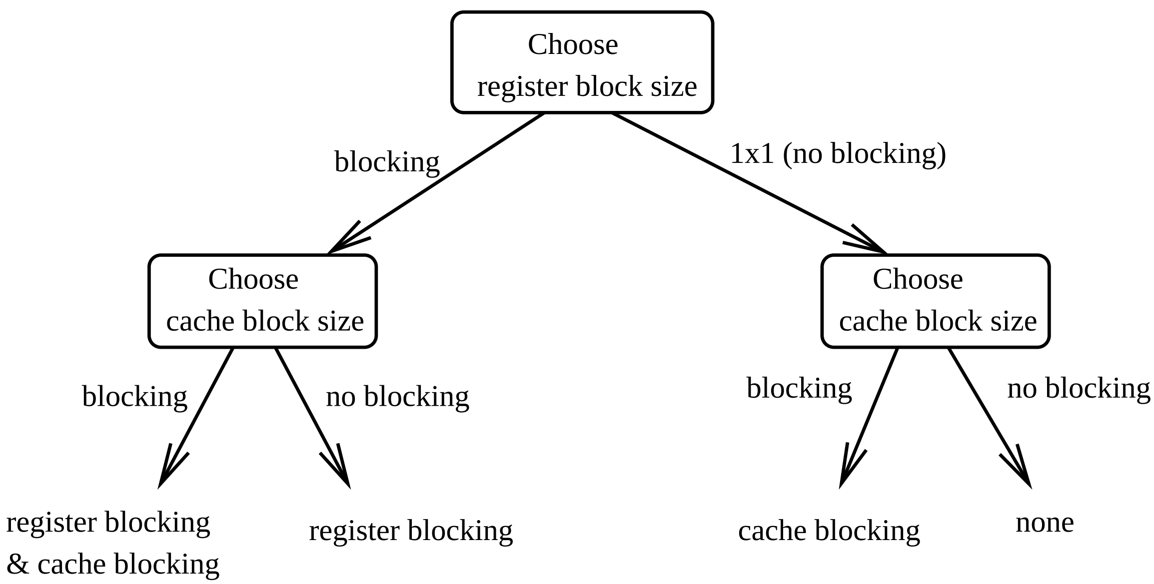


Figure 6.2: **Decision tree for choice of optimization in Sparsity system**

vectors is selected.

For the single vector case, the decision tree in figure 6.2 it used. First, the matrix is tested for register blocking by estimating fill overhead and predicting the blocked multiplication performance using dense performance. Part of the SPARSITY framework includes machine profiling that is done by running each register-blocking size under consideration on a fixed dense matrix, which creates a kind of performance profile for the machine. As discussed above, we limited the possible block sizes up to 12×12, which was more than adequate for all of the machines we considered. After evaluating the performance model for the matrix and the machine profile, the recommended block size was used on the actual matrix and performance compared to the unblocked matrix. This last step is done to ensure that register blocking never degrades overall performance; it can be viewed as a very limited search over two data points, one of which was chosen by our performance model. There are three outputs that result from this test: 1) an answer to the question of whether register blocking is useful; 2) if so, then the selected block size; 3) the code that performs matrix-vector multiplication with the selected block size.

The second test is for cache blocking. As shown in chapter 4, this optimization is unlikely to have a significant payoff on any matrix that was amenable to register blocking. However, we allow for this possibility by applying the cache blocking test to the result of the register blocking test, in other words, either using the register blocked matrix and code as input or, if register blocking did not prove effective, the original matrix and code. The cache blocking test was performed by search over a fixed set of block sizes from 64×64 up to $64K \times 64K$, as well as the unblocked code. For each point in the search space, the matrix is cache-blocked, code is run on the machine of interest, and performance is measured. Although we developed some performance models to aid in decisions related to cache blocking, searching over this limited set of sizes is both practical and more reliable. As with register blocking, the output of this test includes the cache block size and the corresponding code. Again, the output includes the block size and code.

The three possible outcomes of this process are that zero, one or two of the optimizations may be applied. After that, the multiple vector test is performed if requested by the user. Along with the optimized matrix-vector multiplication code, the code generator produces a driver module, benchmarking functions, and matrix I/O routines for commonly used sparse matrix file formats.

SPARSITY is similar to some dynamic or feedback-directed compilation systems in that the code is specific to a particular input. However, the code will work correctly on any matrix, as long as it has been converted to the appropriate block size. Indeed, we expect that a common use of the system will be to produce an optimized matrix-vector multiplication routine for one matrix, to be used for other matrices in the same application domain. Users may choose to use the blocked representation throughout their applications or to convert the matrix before and after iterative solves are performed.

6.4 Sparse Matrix Representations

In sparse matrix-vector multiplication, the nonzero structure of the sparse matrix A directly determines the memory accesses of the source vector x and destination vector y. Any nonzero in a column of A will result in at least one read to the corresponding element of x and any nonzero in a row of A will result in at least one update (read and write) of the corresponding element of y. Thus, multiple nonzeros in a column have the potential for data reuse in the source vector, and multiple nonzeros in a row have the potential for reuse in the destination vector.

There are hundreds of sparse matrix formats used in different application domains, sometimes because the representation is particularly suited to the matrix structures that arise in that domain and sometimes due to arbitrary decisions made by a software designer. There is an ongoing effort called the BLAS Technical Forum to standardize on a set of matrix formats [11]. While the BLAS Technical Forum has been working on expanding the BLAS standard, it considers issues such as the overall functionality, language interfaces, sparse BLAS, extended and mixed precision BLAS, and extensions to the existing BLAS. The forum also considered distributed memory and interval BLAS, but did not come to a consensus for a standard.

In the BLAS Technical Forum standard, nine common sparse matrix formats are supported. They are coordinate (COO), compressed sparse row (CSR), compressed sparse column (CSC) and sparse diagonal (DIA) formats for point entry formats in which individual entries are listed in the storage format. Block coordinate (BCO), block compressed sparse row (BSR), block compressed sparse column (BSC), block sparse diagonal (BDI) and variable block compressed sparse row (VBR) formats are block entry formats where the sparsity structure is represented as a series of small dense blocks. Those storage formats are described in the standard, but the internal representation is left to the implementers, and the sparse matrices are referenced by a handle in the BLAS routines to provide a generic interface. An important aspect of the interface design is that, in the matrix creation routine, the user may provide a hint about the approximate number of times matrix-vector multiplication will be used; this hint may be used by the creation routine to choose a good representation using the kind of analysis that SPARSITY performs.

The coordinate representation stores each nonzero with row and column integer indexes along with a floating point value. The compressed sparse row representation saves some of the storage overhead by storing each row as pairs of column indices and matrix values, so each row index is stored only once. The compressed sparse column representation stores each column as pairs of row indices and matrix values. The sparse diagonal representation stores the sparse matrix with a one-dimensional array of offsets of diagonals and a two-dimensional $m \times n$ array, where m is the number of diagonals and n is the number of rows, for storing nonzero values. Those representations are illustrated in figure 6.3. Block entry formats are block entry versions of corresponding point entry formats where each entry is a small dense block, instead of single value. Variable block compressed sparse row format is a modified version of BSR format that allows each block to be a different size.

In the CSR representation, a matrix-vector multiplication is typically organized around dot products of the matrix rows with the appropriate elements of the source vector. A single value of the destination vector is read and saved in a register during the dot product, and written back to memory once the dot product is complete. We note that a vector-matrix operation for CSR format is organized instead around DAXPY operations (a scalar times a vector plus a vector). A similar representation is seen in compressed sparse column (CSC), which results in opposite design decisions in the two operations.

In SPARSITY, there are two separate matrix formatting issues: the formats supported as input to the optimization system, and the formats that were considered for use as the result of optimizations. We currently support the major point entry formats for our inputs, COO, CSR, and CSC. The diagonal and blocked formats could easily be introduced by adding matrix conversion routines to the system. Internally, SPARSITY uses the compressed sparse row (CSR), which is reasonably efficient across a range of matrices given that SPARSITY has no information about the matrix structure until after processing it.

The more interesting question is what formats are considered as the result of optimization. Among the point entry formats, the coordinate (COO) format is mainly suited for the convenience of generating a sparse matrix in undetermined order, but is not

$$A = \begin{pmatrix} a_{00} & 0 & a_{02} & a_{03} & 0 \\ 0 & 0 & a_{12} & a_{13} & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{31} & 0 & a_{33} & 0 \\ a_{40} & a_{41} & 0 & 0 & a_{44} \end{pmatrix}$$

Coordinate (COO) representation :

$$\text{value} = \begin{pmatrix} a_{00} & a_{02} & a_{03} & a_{12} & a_{13} & a_{20} & a_{21} & a_{22} & a_{23} & a_{31} & a_{33} & a_{40} & a_{41} & a_{44} \end{pmatrix}$$
$$\text{row\_index} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 4 & 4 & 4 \end{pmatrix}$$
$$\text{column\_index} = \begin{pmatrix} 0 & 2 & 3 & 2 & 3 & 0 & 1 & 2 & 3 & 1 & 3 & 0 & 1 & 4 \end{pmatrix}$$

Compressed sparse row (CSR) representation :

$$\text{value} = \begin{pmatrix} a_{00} & a_{02} & a_{03} & a_{12} & a_{13} & a_{20} & a_{21} & a_{22} & a_{23} & a_{31} & a_{33} & a_{40} & a_{41} & a_{44} \end{pmatrix}$$
$$\text{row\_start} = \begin{pmatrix} 0 & 3 & 5 & 9 & 11 & 14 \end{pmatrix}$$
$$\text{column\_index} = \begin{pmatrix} 0 & 2 & 3 & 2 & 3 & 0 & 1 & 2 & 3 & 1 & 3 & 0 & 1 & 4 \end{pmatrix}$$

Compressed sparse column (CSC) representation :

$$\text{value} = \begin{pmatrix} a_{00} & a_{20} & a_{40} & a_{21} & a_{31} & a_{41} & a_{02} & a_{12} & a_{22} & a_{03} & a_{13} & a_{23} & a_{33} & a_{44} \end{pmatrix}$$
$$\text{column\_start} = \begin{pmatrix} 0 & 3 & 6 & 9 & 13 & 14 \end{pmatrix}$$
$$\text{row\_index} = \begin{pmatrix} 0 & 2 & 4 & 2 & 3 & 4 & 0 & 1 & 2 & 0 & 1 & 2 & 3 & 4 \end{pmatrix}$$

Sparse diagonal (DIA) representation :

$$\text{value} = \begin{pmatrix} X & X & X & X & a_{40} \\ X & X & X & 0 & a_{41} \\ X & X & a_{20} & a_{31} & 0 \\ X & 0 & a_{21} & 0 & 0 \\ a_{00} & 0 & a_{22} & a_{33} & a_{44} \\ 0 & a_{12} & a_{23} & 0 & X \\ a_{02} & a_{13} & 0 & X & X \\ a_{03} & 0 & X & X & X \end{pmatrix}$$
$$\text{diag\_offset} = \begin{pmatrix} -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 \end{pmatrix}$$

Figure 6.3: **Example of point-entry sparse matrix representations.** Sparse matrix A is represented in COO, CSR, CSC and DIA formats.

efficient for calculation and is therefore not the result of our optimizations. Register blocking produces BSR, and BSC could be used equally well, with perhaps slightly different results. From examination of the matrix structures in appendix A, few, if any, of the matrices contain large vertical stripes of nonzeros, which would indicate a possible advantage to BSC, since it would increase reuse in the source vector. In some preliminary experiments, we experimented with a variable block compressed sparse row, which allows for variable block sizes, but found the branch overhead and lack of complete loop unrolling to produce worse performance than fixed size blocks, even if the fixed blocks were not an exact fit for the matrix structure. A related idea would be to represent the matrix as the sum of a set of sparser matrices, each with a fixed block size. This would essentially move the branches out of the inner loops and allow for the same kinds of optimizations in our code, although it also means that there would be multiple sweeps over the vectors, and so lower possible reuse.

We did not consider the diagonal format for optimization targets of SPARSITY. These formats are suitable only for sparse matrices where the nonzero elements are distributed around the diagonal, although there are several matrices in our benchmark suite and in general, that have these characteristics. While these formats might perform well on vector supercomputers or other machines with high memory bandwidth, they would not allow for any data reuse in the source or destination vectors. Since each row in the DIA matrix represents a diagonal of the actual matrix, to perform a matrix-vector multiplication, each row would be processed by performing element-wise multiplication with the source vector and element-wise addition with the destination.

The output of our cache-blocked format is not one that has been described in the matrix library literature, probably because it does not arise from any natural structure of the matrix or application, but from a structure in the machine.

6.5 Summary and Further Improvements

SPARSITY provides easy access to highly tuned sparse matrix-vector multiplication routines. The system is accessible through a web interface, and we envision that further improvements in that interface could make the system even more accessible for a larger community of users.

First, a profiling module for measuring machine dependent parameters can be made downloadable to users. The current implementation makes the user download all parts of the program at once, and all the procedures are executed on the target machine, including the code used to decide on the type of optimization and block sizes. The decision process can be moved to a host machine that provides the web service. With this implementation, we could use bisection criteria for cache blocking, since the partitioning package would not have to be installed on the user's machine. Instead, the user would upload the matrix and have the evaluation done on the host machines. This would be more expensive if the user had many and/or large machines.

Secondly, a database of machine dependent parameters can be maintained so that users would not have to run all of the experiments to determine these parameters. This could be done, for example, on a machine in a central computer facility and automatically updated as new machines become available. In this way, the second user of Sparsity at some site would go through a simpler and faster optimization process than the first user. Much of Sparsity is written in Java to simplify downloading and uploading of code and results, and to make the process more transparent to users.

Chapter 7

Related Work

There are several different areas of related work that we discuss in this chapter. We start in section 7.1 with research in optimization of dense matrix libraries which has been ongoing for many years. This includes the work that most closely matches the SPARSITY approach through the use of automatic optimization frameworks for libraries. We then describe prior work on sparse matrix libraries and their optimization in section 7.2.

7.1 Optimization of Dense Matrix Operations

There has been a great deal of research in high performance dense matrix operations. The three basic approaches can be categorized as: general-purpose optimizing compilers, hand-optimized libraries, and automatic optimization frameworks. We include some work on the FFT in this section, even though it is not, strictly-speaking, a dense matrix operation, because it uses regular data structures and therefore has similar optimization issues.

7.1.1 Compilers

The memory hierarchy optimization commonly used for dense matrix operations is known as *blocking* or *tiling*, which can be done for registers or caches. The simplest version of tiling involves reorganization of the computation to improve data locality, while more sophisticated versions change the data representation as well. Tiling is often combined with other loop transformations such as unrolling, loop interchange, and loop skew. These are described in an advanced compiler text such as [51], although we refer to the reader to several other papers on more details and recent work on tiling (and a closely related notion called *shackling*) [18, 50, 16, 12, 44, 71, 41, 42].

This work on compiler-based memory hierarchy optimizations has produced tools that are very sophisticated in performing analysis of the code to determine legal transformations, and apply certain optimizing transformations. Simple linear algebra operations, such as dense matrix-matrix multiplication or matrix-vector multiplication, can easily be analyzed to show that the loops can be legally reordered, unrolled, and tiled while preserving the semantics of the code.

There are some limitations to this pure compiler-based approach. The most important limitation from our perspective is that the analysis techniques cannot handle sparse matrix operations, because the source code contains indirection through an array, which cannot be analyzed at compile time. (Some recent work on compiling sparse matrix code that takes a different approach, is described below.) Even for dense matrices, the approach works only on low level kernels; algorithms such as LU factorization with pivoting or QR decomposition [65, 9] are still open problems. The second problem is that, even for dense matrices, the optimization space is large and complex, making it difficult to find the best implementation. For example, given a set of loops that can be tiled, the problem of choosing a tile size is still quite difficult; existing solutions do not allow for arbitrary numbers of arrays or higher dimensional arrays and they often make unrealistic assumptions about the architecture, such as having a single direct-mapped cache [44, 26, 19]. Our approaches in SPARSITY use experimental models rather than the analytical ones used in this related work.

7.1.2 Hand-Optimized Libraries

A set of basic matrix-vector operations for dense and banded matrices has been standardized in BLAS [45, 24, 23], and its reference implementation is available from Netlib [52]. These basic routines are hand-coded by programmers that work for the machine vendors, who are well-trained in both the details of the architecture and optimization techniques used for high performance. The BLAS approach has successfully been used to produce application-level libraries for dense linear algebra, such as LAPACK [1] (Linear Algebra PACKage) and ScaLAPACK [10] (Scalable Linear Algebra PACKage), which use these BLAS routines as building blocks.

There are two problems with trying this approach for sparse matrices. First, even for the dense case the effort required to produce the optimized kernels has lead some vendors to rely on their compilers to produced the optimized versions. The human effort is increasing due to both the greater complexity of hardware and the demand for a richer set of "primitive" routines. Second, as we has shown, some optimizations are specific to a particular class of matrices, so the burden of writing the routines would likely fall to the applications programmer, who has little understanding of the machine details.

7.1.3 Automatic Generation of Libraries

Due to the large amount of time and effort needed to develop fast numerical codes for different architectures, there have been some recent attempts to automate this process. These approaches share some aspects of the compiler approach, because they often use a code generation facility (although all generate C code rather than assembly language). They avoid the analysis problems that exist in general-purpose compilers, because they are only optimizing one routine or a set of closely related routines. The projects in this area include PHiPAC (Portable High Performance ANSI C) [8] at UC Berkeley and ATLAS (Automatically Tuned Linear Algebra Software) [70] at ORNL. Both of these generate optimized BLAS 3 routines for a given dense matrix size and machine. FFTW (the Fastest Fourier Transform in the West) [28, 27] from MIT generates optimized FFT routines.

PHiPAC The PHiPAC project uses search over a set of possible optimizations to produce a matrix-matrix multiplication routine that is highly tuned to a particular machine. Even excluding algorithmic variants of multiplication such as Strassen's method [4, 36, 67], this routine has a large design space with many parameters such a tile sizes, loop nesting permutations, loop unrolling depths, software pipelining strategies, register allocations, and instruction schedules. Furthermore, these parameters have complicated interactions with increasingly complicated microarchitectures of new microprocessors.

The PHiPAC approach is as follows. First, they developed a generic model of current C compilers and microprocessors that provides guidelines for producing Portable High-Performance ANSI C code (PHiPAC). Second, rather than hand-coding particular routines, they write parameterized generators that produce code according to the guidelines. Third, they write scripts that automatically tune code for a particular system by searching, i.e. varying the generators' parameters, benchmarking the resulting routines, and picking the fastest one. PHiPAC and the vendor BLAS are comparable, close to machine peak for large matrices, and much faster than the naive code that has been fully optimized by the compiler.

PHiPAC, like SPARSITY, relies on the C compiler to do reasonable register allocation, instruction selection, and instruction scheduling. However the group found that compiler could not depend on pointer alias analysis, register and cache tiling, loop unrolling, or software pipelining; even when implemented, they were not done effectively, because there are so many ways to do them. PHiPAC's generated code (1) uses local variables, reordering operations to explicitly remove false dependencies, enabling the compiler to interleave execution and increase parallelism, (2) exploits multiple integer and floating point units, (3) minimizes pointer updates by constant stride, (4) hides multiple instruction FPU latency with independent operations, (5) balances the instruction mix to keep multiple functional units busy, (6) increases locality via tiling to improve cache performance, (7) converts integer multiplications to additions in address calculations, (8) minimizes branches, and avoids magnitude-compares to end loops, (9) explicitly unrolls loops.

The code is generated using these rules and the search scripts then searches over a subset of the combinatorially large space of possible implementations. To limit search time, machine parameters are used to limit tile size choices. Furthermore, they first search for the best "core" register blocked code, then use that to search for the best L1 tiled code, use that for the L2 search, and so on. Searches can take from hours to *weeks*, depending on how thorough they are.

The PHiPAC problem domain is quite different from SPARSITY's, because the data access patterns are independent of input for dense matrices. Although matrix size is not known during optimization, PHiPAC can determine good implementations for various small matrix sizes and then produce a more generic routine for any larger matrix. Thus, the entire optimization problem can be done off-line, with only access to the machine required. We believe that SPARSITY could benefit somewhat from a PHiPAC style search over the code used for small dense blocks, for example, which appear within the register-blocked sparse code. In SPARSITY we currently produce a single machine-independent implementation of this kernel.

ATLAS ATLAS [69] stands for Automatic Tuning of Linear Algebra Software. ATLAS's approach differs from PHiPAC as follows: all machine dependent code (and searching) is done for square matrices assumed to reside in the L1 cache. All other matrices are reduced to this case by copying all or part of the input matrices. This approach has two advantages: (1) Search times are much reduced, from days or weeks down to 1-2 hours, because only one limited kernel is machine dependently searched; and (2) Performance is more uniform than that of PHiPAC result. This is because the copy optimization can make sure that all active submatrices are stored in non-conflicting cache locations. The disadvantages are that copy optimization is inefficient on small matrices, and PHiPAC's more comprehensive search might generate faster code in some cases, although ATLAS works quite well in practice.

Since its original release, ATLAS has been extended to support nearly all the BLAS, not just matrix-multiplication. This is done for other Level 3 BLAS using GEMM-based BLAS as described in [39]. In addition, some LAPACK routines (LU, Cholesky and QR decomposition) are now directly supported by the search procedure. In other words, the search procedure attempts to maximize not the speed of matrix-multiplication per se, but rather the speed of the overall application.

FFTW FFTW [28, 27] automatically produces optimized FFTs. It deals with one-dimensional and multi-dimensional FFTs, strided data, and real or complex input. It also handles parallelism. Its FFTs are comparable with and frequently faster than vendor optimized FFTs. While an FFT is not technically a dense matrix operation, the underlying data structure is a standard array of some dimension, rather than the more complicated sparse data structure that uses index indirection or pointers.

FFTW is structured as follows. As shown by Cooley and Tukey, if $n = n_1 \cdot n_2$, then the n-point FFT can be expressed in terms of n_1 and n_2 point FFTs. Therefore for each factorization of n, there is an FFT algorithm. FFTW does an off-line, machine-independent generation of a number of FFT kernels for many small values of n. Then, at run-time, when the actual n is known, a search is done over a large number of possible factorizations of n, each of which corresponds to an implementation from the prebuilt kernels. Each implementation is timed and the fastest one chosen.

The kernels are machine independent C code, but a number of optimizations are done, based on similar assumptions about what C optimizers can and cannot do as used by PHiPAC and ATLAS. Some optimizations are generic, like common subexpression elimination and constant folding, but some are quite specific to the FFT, such as optimizing both the program DAG and its transposition. The kernel algorithms and the optimizations are all expressed in a single language, a dialect of ML.

An interesting contrast between FFTW on the one hand and ATLAS/PHiPAC on the other is their approach to tiling. For ATLAS and PHiPAC the choice of tile size is perhaps the most important parameter to pick for performance, whereas in FFTW a cache-oblivious algorithm is used, based on a recursive data structure that asymptotically minimizes the number of cache misses, according to a theorem in [35].

The above projects all aimed to optimize the regular operations on arrays. Aside from machine-specific parameters, the only problem-specific parameter is the size of the matrix or vector. However, in optimizing a sparse matrix operation, the nonzero structure of the sparse matrix has enormous effect on the cache performance. Due to the addition of this parameter, the search space for optimal code is much larger for sparse matrices than dense ones, and it is not feasible to perform the optimizations without seeing the actual sparse structures.

7.2 Sparse Matrix Libraries and Optimizations

In this section, we summarize research efforts on sparse matrices, including standardization efforts, optimization studies, and development of sparse matrix packages.

Sparse BLAS: Sparse Matrix Routine Standardization

The BLAS Technical Forum is an effort aimed at expanding the BLAS in a number of ways to reflect developments of modern software, languages, and hardware. This includes a standardization effort to define BLAS functionality on sparse matrices. The draft document [11] defines the functionality and naming conventions of sparse BLAS level 2 (matrix-vector) and level 3 (matrix-matrix) routines. This document also summarizes the diverse storage formats of sparse matrices, which were described in detail in section 6.4.

The NIST Sparse BLAS [59] project has publicly available C and FORTRAN library routines that are sparse counterparts to the dense BLAS level 3 routines, including the solution of triangular systems and matrix-matrix multiplication. They currently support operations on coordinate, compressed sparse row, compressed sparse column, block sparse row, block sparse column, block coordinate, and variable block row formats. Those routines are automatically generated from templates, with some effort put into achieving reasonable performance. However, there is no attempt to choose the formats for the user based on either user's matrix or machine.

Generic Interfaces

A related line of work provides a generic numerical algorithm library, including MV++ (C++ matrix/vector classes) [56], SparseLib++ (sparse matrix computation routines on these classes) [60], and IML++ (Iterative Method Library) [25]. The latest work, TNT (Template Numerical Toolkit) [57], is a successor to all of the others and is still under construction. TNT provides an integrated collection of generic matrix/vector classes based on components of the latest ANSI C++ features along with the Standard Template Library (STL). The algorithms are currently based on the SparseBLAS library for the basic sparse matrix kernel operations, so machine-specific optimizations are not part of their agenda.

Hand Optimization of Matrix-Vector Multiplication

Toledo [68] studied a particular memory optimization of (symmetric) sparse matrix-vector multiplication on uniprocessor. He used mixed 1×2 and 2×2 register blocking, and showed bandwidth reduction by means of reordering rows of the matrix. He considered two ordering strategies and found that both give better performance than a random ordering, sometimes better or worse compared to the ordering given in the original matrix. He also observed that nested-bisection ordering generates more blocks, while reverse Cuthill-McKee [48] ordering decreases the number of blocks found. While Toledo's work is similar to our work on register blocking, it is a much more narrow study of 13 matrices on one machine (IBM RS/6000) and does not attempt to automate the optimization process. Oliker et. al. [54] also performed research on the efficiency of sparse matrix-vector multiplication using various ordering/partitioning algorithms on parallel systems. They compared the following three algorithms against the original natural ordering using a sparse matrix generated from a mesh: Reverse Cuthill-McKee (RCM), self-avoiding walks (SAW), and graph partitioning with the MeTiS package. They conclude that those ordering/partitioning schemes significantly improve the performance on distributed-memory and shared-memory systems. In previous work, we also considered reordering for memory bandwidth reduction, and found some benefits on symmetric multiprocessors [38], but very little effect on uniprocessors.

Sparse Matrix Packages for Multiprocessors

Yousef Saad built SPARSKIT [61] and, with Andrei Malevsky, PSPARSLIB [62]. SPARSKIT is a collection of FORTRAN subroutines for sparse matrix computations. It includes format conversion routines among various sparse matrix formats, matrix-matrix computation routines for the compressed row format, matrix-vector operations for sparse matrices represented in various formats, and some preconditioners. PSPARSLIB [62] is a sparse iterative solver for distributed memory multiprocessors written in FORTRAN. It isolates communication functions so that it can be replaced with any high performance communication method. While communication optimizations are a form of memory hierarchy optimization for parallel machines, the issues are quite different because the messages are entirely under programmer control.

Aztec [37] is a parallel iterative library for solving linear systems, developed at Sandia National Laboratory. This library allows a user to apply standard distributed memory techniques such as locally numbered submatrices and ghost variables, using the notion of a global distributed matrix.

BlockSolve95 [55], developed at Argonne National Laboratory, is a scalable parallel software library primarily intended for the solution of sparse linear systems that arise from physical models, especially problems involving multiple degrees of freedom at each node. BlockSolve95 tries to find a blocked structure by locating cliques and identical nodes and by reordering the matrix columns and rows. By identifying large blocks, it is able to use a higher level dense BLAS routine for better performance. PETSc [5], a toolkit for scientific computations, uses this BlockSolve95 library.

Spark98 at CMU [53] provides various versions of sparse matrix-vector multiplication code for shared memory and message passing systems, along with realistic example matrices. Their code is limited to symmetric matrices.

Compiling Sparse Matrix Code

Two research groups have taken a novel approach to the compilation problem for sparse matrix kernels, which take a dense matrix routine as input, rather than an implementation designed for a sparse matrix format. The input acts as a specification of a particular matrix operation; for many common matrix kernels, this dense matrix code is simple enough that modern compiler analyses can analyze the code quite effectively. The "compiler" is a special-purpose compiler that produces code appropriate for a sparse matrix.

A. J. C. Bik's thesis [6, 7] describes a *sparse compiler* that takes a dense matrix routine as input, along with a sparse matrix, and generates a sparse version of the code along with an automatically selected sparse matrix data structure. Within the compiler, the sparse matrix is analyzed to classify the input matrix, and then the dense matrix loop is transformed using dependence analysis of data accesses. Bik's work is similar to ours in that it analyzes the nonzero structure of a sparse matrix for the purpose of optimization. However, its nonzero structure analyzer is quite different in that it identifies the distribution of nonzeros locally, i.e., where they are clustered, while we analyze the global structure based on the assumption that it is identical throughout the matrix. One can envision a hybrid analyzer that combines both techniques such that after each cluster of nonzeros in the matrix is identified, each cluster is analyzed for blocks individually. Another difference between the two approaches is that Sparsity uses machine information that can affect the data storage selection while the sparse compiler does not.

Kotlyar et al. from Cornell, in a related effort, produced work on the Bernoulli compiler [43]. Bernoulli also uses dense matrix format for the input code. It also takes some identification of the desired sparse matrix storage format as input, but not a particular matrix. Bernoulli uses relational algebra to generate parallel sparse code from dense code input. The goal of this project is to simplify programming of sparse matrix algorithms, rather than optimizations that are specific to a particular matrix or machine.

On-demand Code Generation

Several of the projects described above provide a web service that generates some sparse matrix routines. NIST SparseBLAS [58] and A. J. C. Bik's *sparse compiler*[13] provide on-demand code generation service for BLAS level 2 and 3 routines. The NIST SparseBLAS system only provides a reference implementation, not one that is tuned to any particular matrix or machine. *Sparse compiler*'s service is closest to ours, since it performs matrix-specific optimization and matrix format selection. The output of the sparse compiler is FORTRAN, whereas Sparsity produces C, and as described above, the analysis of the matrices is quite different.

Chapter 8

Conclusions

In this thesis, we have described new optimization techniques for sparse matrix-vector multiplication and described an optimization framework, SPARSITY, for automatically selecting and applying these optimizations. The optimization framework uses limited search over a set of possible optimized versions, along with performance models and other heuristics to restrict the search space. We also presented a thorough study of memory hierarchy optimizations, demonstrating that the choice of optimizations is highly dependent on the nonzero structure of the matrix and the target machine. Our benchmark matrices were taken from a wide range of applications, including fluid dynamics, structural modeling, chemistry, economics, circuit simulation, device simulation, linear programming, and document retrieval.

Our optimization techniques address the increasingly deep and complex layering of memory systems in modern machines, which has come about due to the widening gap between processor and DRAM memory performance. At the top of the memory hierarchy is a fixed set of registers, which are normally under control of the compiler. To optimize for registers, we demonstrated that an effective strategy is to identify fixed-size dense blocks within a sparse matrix, filling in zeros as necessary. We introduced a performance model to help select the appropriate block size for a machine, using a kind of machine performance profile combined with an analysis of the sparse matrix structure. Even on matrices where the blocks were not evident at the application level, small blocks proved useful on some machines.

The next two or three levels in most processor memory hierarchies are caches, which differ across machines in their size, speed, and replacement policies. To optimize for cache reuse, we devised a kind of two-level sparse block structure for matrices, which is particularly effective for very large matrices with a nearly random sparsity pattern. We introduced heuristics to help identify this class of matrices, which work quite well in practice, although we found that search over a relatively limited set of possible block sizes is also practical and more reliable.

For a class of sparse matrix algorithms, the problem can be reduced to a matrix times a set of vectors, rather than a single vector. No other known work on optimization directly addresses this problem. We extended our optimization framework to take advantage of multiple vectors, which can be used to increase the reuse of data within registers or caches. We also extended our benchmark suite for this work, adding two matrices that are known to come from applications that use multiple matrices. The benefits of multiple vectors turn out to be very high, often a factor of two over the previously optimized code.

The benefits of our approach are summarized in the speedup graphs shown in figures 8.1 and 8.2. Figure 8.1 shows the UltraSPARC I results, and Figure 8.2 shows the MIPS R10000 results. The plots show the speedup from code generated by Sparsity using whichever optimizations it determines to be best. The left graph in each figure shows the single vector speedup and the right plot shows the multiple vector speedup using nine vectors; both are relative to the unoptimized single vector performance. The matrices are roughly categorized by the application domain from which they were derived. The figures show that our system is highly effective, with speedups as high as three-fold for single vector multiplication and six-fold for multiple vector multiplication. The finite element matrices show the highest benefit from our optimizations, typically due to register blocking; text retrieval matrices benefit from cache blocking, while linear programming matrices have relatively small speedups. Multiple vectors roughly double the performance of the already optimized single vector performance, and also work well on matrices that do not derive any benefit from register or cache blocking.

The only major drawback to automatic optimization of sparse matrix operations is the relatively high overhead for analyzing and reorganizing the matrix. (The overhead of single-vector multiplication for a subset of benchmark matrices on an UltraSPARC is summarized in figure 8.3.) In particular, register blocking analysis it quite expensive, because it is searching over the matrix structure to look for not one or two possible block sizes, but a large set of sizes. Search over block sizes, as is used by dense matrix optimizations systems like PHiPAC and Atlas, are probably not practical for the same reasons. The reorganization

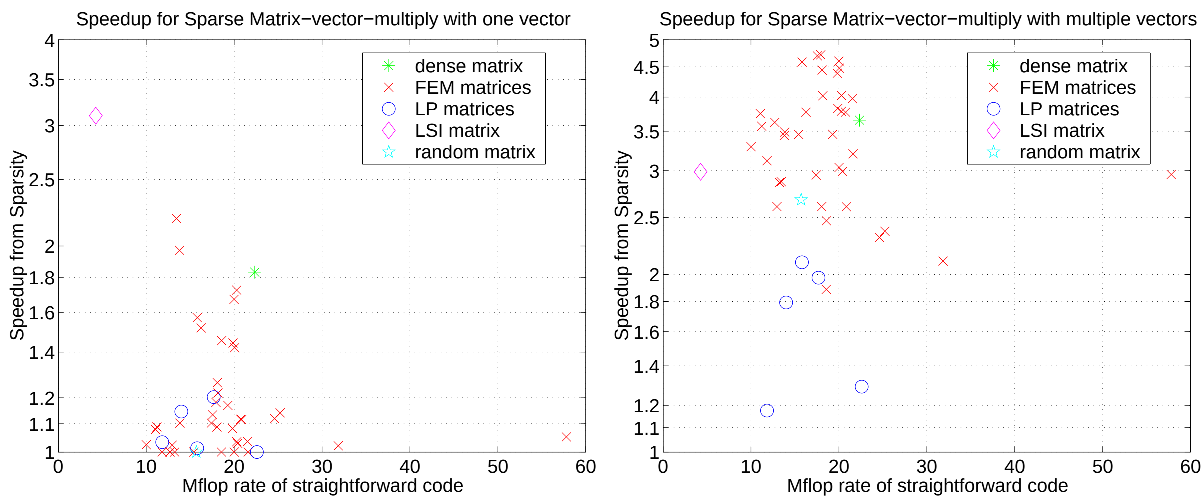


Figure 8.1: **Summary of optimized speedup on an UltraSPARC**

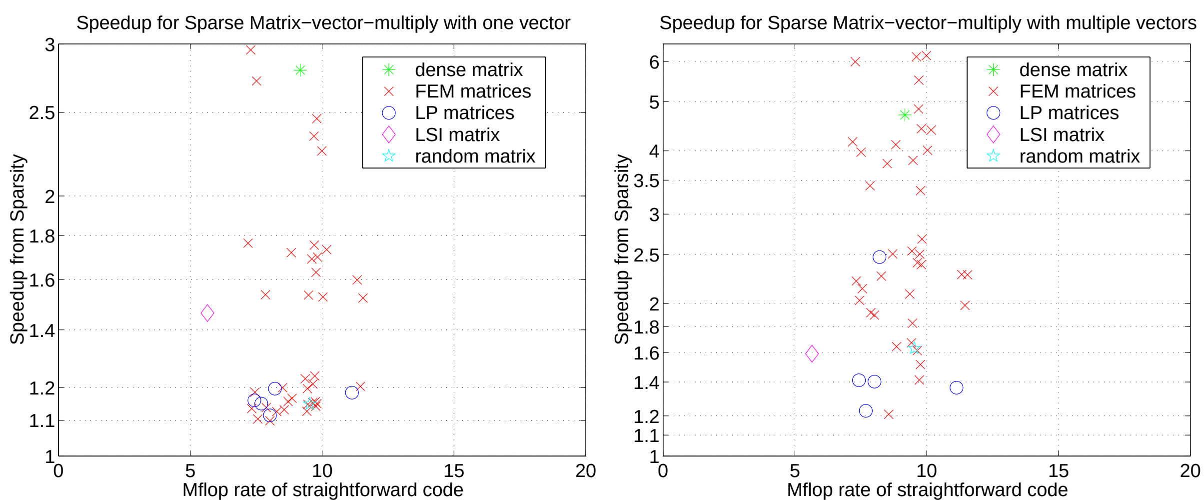


Figure 8.2: **Summary of optimized speedup on a MIPS R10000**

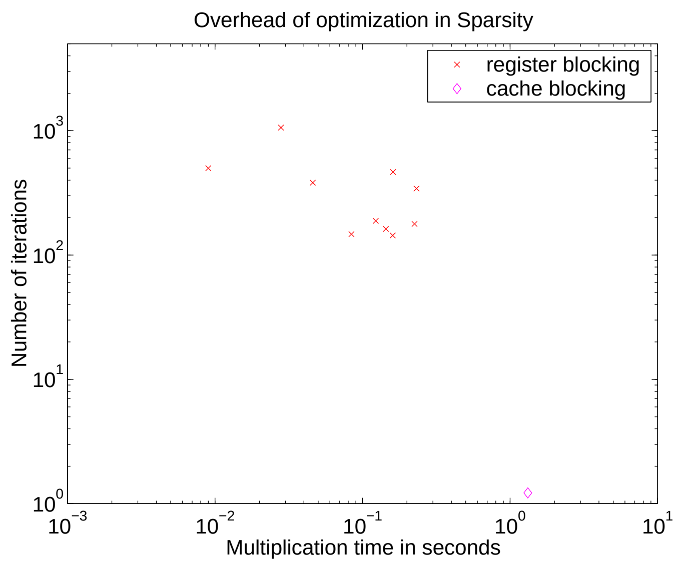


Figure 8.3: **Summary of overhead in Sparsity:** The horizontal axis shows the unoptimized multiplication time in log scale and the vertical axis shows the overhead time divided by the difference between unoptimized multiplication time and optimized multiplication time in log scale. The amount is equivalent to the number of iterations that the optimization pays off. A graph is shown for the subset of benchmark matrices.

of a matrix for a particular register block size is very expensive, since memory must be rearranged at a precise level within the matrix. By comparison, cache-blocking reorganization can be done more efficiently using block copy operations, since relatively large portions of the data structure are retained.

The overhead of analysis can be avoided at runtime if the appropriate data structure is used throughout the application run. The analysis and transformation cost can at least be amortized over many matrix-vector multiplications if the matrix can be translated before calling an iterative solver, for example. If multiple solves are being done, as might happen in a time-stepped simulation, then one may pay the translation cost with each solve but avoid the analysis by reusing the results of analysis from the first solve, assuming that the matrix structure is similar enough across times steps that the same block sizes will be appropriate. The BLAS Technical Forum has already identified the need for runtime optimization of sparse matrix routines, since they include a parameter in the matrix creation routine to indicate how frequently matrix-vector multiplication will be performed. In some cases, it would be beneficial if the user could also indicate some other property of the matrix, such as the size of dense blocks within the matrix, which could either come from the application programmer's background knowledge of the problem or from previous feedback from the optimization system.

The SPARSITY system is set up as a web service to encourage users from science and engineering disciplines outside of Computer Science. We believe that the SPARSITY framework is very general, and could easily admit additional optimizations, such as those based on a diagonal representation or mixed block size format, or machine-specific instruction ordering for the dense matrix operations within the register-blocked code. SPARSITY should also prove useful to others doing research on sparse matrix optimization. The set of matrices and machine benchmarks provided in this thesis will serve as a starting point for a database of sparse matrix performance understanding, which may help machine architects, algorithm designers, library and compiler writers, and application scientists to better understand the performance of sparse matrix applications.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Feb. 1995.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide (third edition)*. SIAM, Philadelphia, 1999.

[3] Z. Bai, T.-Z. Chen, D. Day, J. Dongarra, A. Edelman, T. Ericsson, R. Freund, M. Gu, B. Kagstrom, A. Knyazev, T. Kowalski, R. Lehoucq, R.-C. Li, R. Lippert, K. Maschoff, K. Meerbergen, R. Morgan, A. Ruhe, Y. Saad, G. Sleijpen, D. Sorensen, and H. Van der Vorst. Templates for the solution of algebraic eigenvalue problems: A practical guide. in preparation, 2000.

[4] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:97–371, 1991.

[5] S. Balay, W. Gropp, L. C. McInns, and B. Smith. PETSc 2.0 user's manual. Technical Report ANL-95/11, Argonne National Laboratory, 1996.

[6] A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, 1996.

[7] A. J. C. Bik and H. A. G. Wjishoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, 1996.

[8] J. A. Bilmes, K. Asanovic, J. Demmel, C. Chin, and D. Lam. Optimizing matrix

multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, July 1997.

[9] C. Bischof. Adaptive blocking in the QR factorization. *J. Supercomputing*, 3(3):193–208, 1989.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.

[11] BLAST Forum. *Documentation for the Basic Linear Algebra Subprograms (BLAS)*, Oct. 1999. http://www.netlib.org/blast/blast-forum.

[12] P. Boulet, J. Dongarra, Y. Robert, and F. A. A. Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25(5):547–568, mai 1999.

[13] P. Brinkhaus, A. J. Bik, and H. A. Wijshoff. Subroutine on demand-service : Sparse BLAS 2 & 3. http://hp137a.wi.leidenuniv.nl:8080/blas-service/blas.html.

[14] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, July 1994.

[15] S. Carr. Combining optimization for cache and instruction-level parallelism. In *Proceedings of the 1996 International Conference on Parallel Architectures and Compiler Techniques (PACT 96)*, Oct. 1996.

[16] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *TOPLAS*, 16(6):1768–1810, 1994.

[17] S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Transactions on Mathematical Software*, 23(3), Sept. 1997.

[18] S. Carr and R. B. Lehoucq. Compiler blockability of dense matrix factorizations. *TOMS*, 23(3):336–361, 1997.

[19] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[20] T. Davis. University of Florida Sparse Matrix Collection, 1997. http://www.cise.ufl.edu/ davis/sparse/.

[21] I. S. Dhillon and D. S. Modha. Concept decompositions for large sparse text data using clustering. Technical Report RJ 10147, IBM, July 1999. to appear in Machine Learning.

[22] J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14:1–17, Mar. 1988.

[23] J. Dongarra, J. Du 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.

[24] J. Dongarra, J. Du Croz, 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.

[25] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference*, pages 214–218, 1992.

[26] K. Essenghir. Improving data locality for caches. Master's thesis, Rice University, Sept. 1993.

[27] M. Frigo. A fast fourier transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia*, May 1999.

[28] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *International Conference on Acoustics, Speech and Signal Processing*, 1998.

[29] G. H. Golub and R. Underwood. The Block Lanczos Method for Computing Eigenvalues. In J. R. Rice, editor, *Mathematical Sotware III*, pages 361–377. Academic Press, Inc., 1977.

[30] R. G. Grimes, J. G. Lewis, and H. D. Simon. A Shifted Block Lanczos Algorithm for Solving Sparse Symmetric Eigenvalue Problems. *SIAM J. Matrix Anal. Appl.*, 15:228–272, 1994.

[31] K. K. Gupta and C. L. Lawson. Development of a Block Lanczos Algorithm for Free Vibration Analysis of Spinning Structures. *Int. J. for Numer. Meth. in Eng.*, 26:1029–1037, 1988.

[32] M. T. Heath. *Scientific Computing: An Introductory Survey.* McGraw-Hill, 1997.

[33] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[34] J. L. Hennesy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, second edition, 1996.

[35] X. Hong and H. T. Kung. I/O complexity: the red blue pebble game. In *Proceedings of the 13th Symposium on the Theory of Computing*, pages 326–334. ACM, 1981.

[36] S. Huss-Lederman, E. Jacobson, J. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Supercomputing 96.* IEEE, 1996.

[37] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro. Aztec user's guide: Version 1.1. Technical Report SAND95-1559, Sandia National Laboratories, 1995.

[38] E.-J. Im and K. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, Mar. 1999.

[39] B. Kågström, P. Ling, and C. Van Loan. Portable High Performance GEMM-based Level 3 BLAS. In R. F. S. et al., editor, *Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993. SIAM. software available at `www.netlib.org/blas/gemm_based`.

[40] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. *hMETIS A Hypergraph Partitioning Package.* University of Minnesota, Jan. 1998.

[41] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[42] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Interional Conference on Supercomputing*, 1999.

[43] V. Kotlyar, K. Pingali, and P. Stodghill. Compiling parallel code for sparse matrix applications. In *Supercomputing*, 1997.

[44] 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*, Apr. 1991.

[45] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[46] R. Lehoucq and K. Maschhoff. Implementation of an implicitly restarted block Arnoldi method. Preprint MCS-P649-0297, Argonne National Lab, 1997.

[47] X. S. Li. *Sparse Gaussian Elimination on High Performance Computers.* PhD thesis, University of California, Berkeley, 1996.

[48] W.-H. Liu and A. H. Sherman. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. In *SIAM J. Numerical Analysis*, pages 198–213, 1976.

[49] O. A. Marques. BLZPACK: Decsription and User's guide. Technical Report TR/PA/95/30, CERFACS, 1995.

[50] 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.

[51] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[52] Netlib. Netlib repository at UTK and ORNL. http://www.netlib.org.

[53] D. R. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, 1997.

[54] L. Oliker, X. Li, G. Heber, and R. Biswas. Ordering unstructured meshes for sparse matrix computations on leading parallel systems. In J. R. et al., editor, *Parallel and*

Distributed Processing, 15 IPDPS 2000 Workshops, pages 497–503, Springer-Verlag, Berlin, 2000. Lecture Notes in Computer Science 1800.

[55] P. Plassmann and M. T. Jones. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, 1995.

[56] R. Pozo. MV++, 1995. http://math.nist.gov/mv++.

[57] R. Pozo. Template numerical toolkit (TNT), 1997. http://math.nist.gov/tnt.

[58] R. Pozo and K. Remington. NIST Sparse BLAS sourceservice code request form. http://math.nist.gov/spblas/sourceservice.html.

[59] R. Pozo and K. Remington. NIST Sparse BLAS, 1997. http://math.nist.gov/spblas.

[60] R. Pozo, K. Remington, and A. Lumsdaine. SparseLib++, 1996. http://math.nist.gov/sparselib++.

[61] Y. Saad. *SPARSKIT: A basic tool-kit for sparse matrix computations*, June 1994.

[62] Y. Saad and A. V. Malevsky. *P-SPARSLIB: A Portable Library of Distributed Memory Sparse Iterative Solvers*, 1995.

[63] M. Sadkane. Block-Arnoldi and Davidson methods for unsymmetric large eigenvalue problems. *Numer. Math.*, 64:195–211, 1993.

[64] M. Sadkane. A block Arnoldi-Chebyshev method for computing the leading eigenpairs of large sparse unsymmetric matrices. *Numer. Math.*, 64:181–193, 1993.

[65] R. Schreiber and C. Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.*, 10:53–57, 1989.

[66] J. Shi and J. Malik. Motion segmentation and tracking using normalized cuts. In *International Conference on Computer Vision*, Jan. 1998.

[67] M. Throttethodl, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of Supercomputing '98*, Orlando, FL, November 1998.

[68] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, Mar. 1997.

[69] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. Computer Science Department CS-97-366, University of Tennessee, Knoxville, TN, December 1997. (LAPACK Working Note #131; see `http://www.netlib.org/utk/projects/atlas/index.html`).

[70] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software (ATLAS). http://www.netlib.org/atlas.

[71] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Computer Systems Laboratory, Stanford University, Aug. 1992.

Appendix A

Nonzero Structure of Test Matrices

Figures A.1 – A.12 illustrate the distribution of nonzero elements of each matrix. The darker point means denser nonzeros elements.

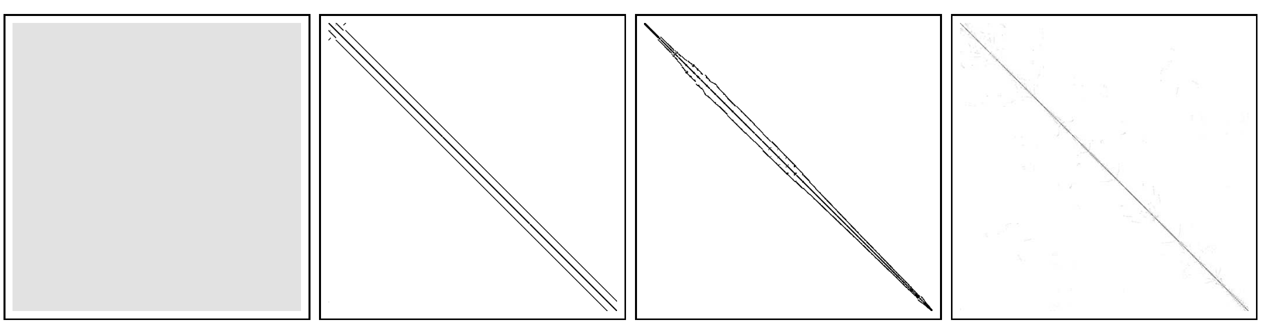


Figure A.1: (1)dense1000 (2)raefsky3 (3)inaccura (4)bcsstk35

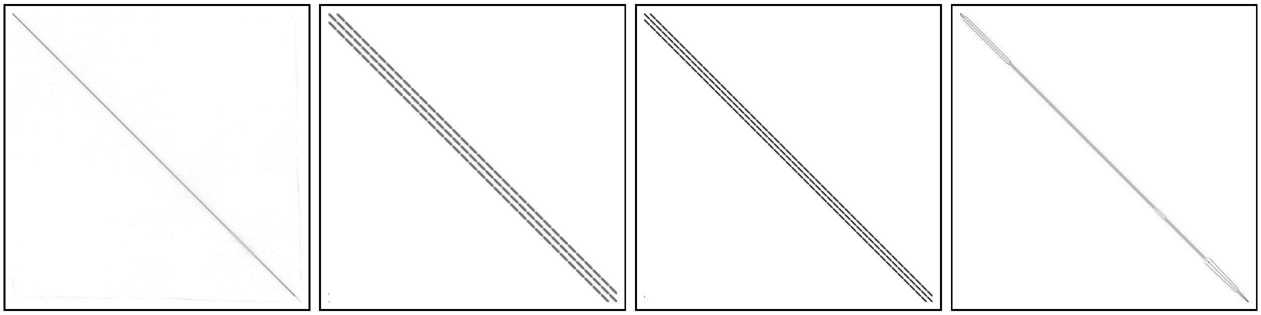


Figure A.2: (5)venkat01 (6)crystk02 (7)crystk03 (8)nasasrb

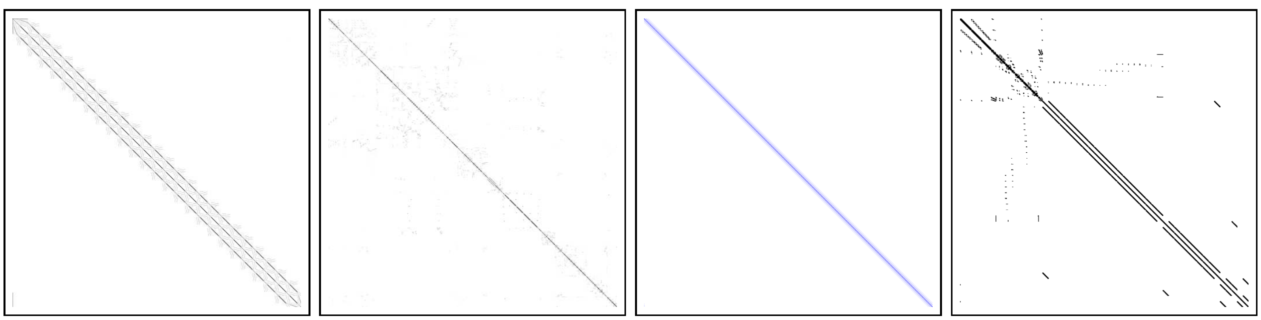


Figure A.3: (9)3dtube (10)ct20stif (11)bai (12)raefsky4

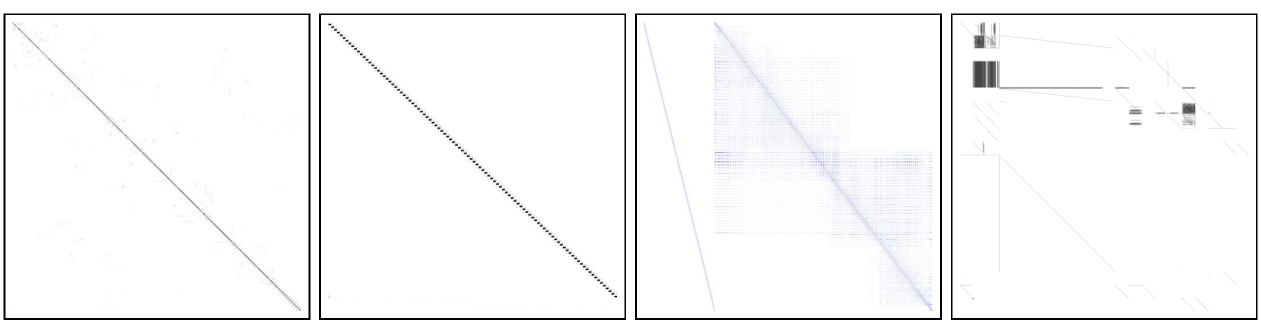


Figure A.4: (13)ex11 (14)rdist1 (15)vavasis3 (16)orani678

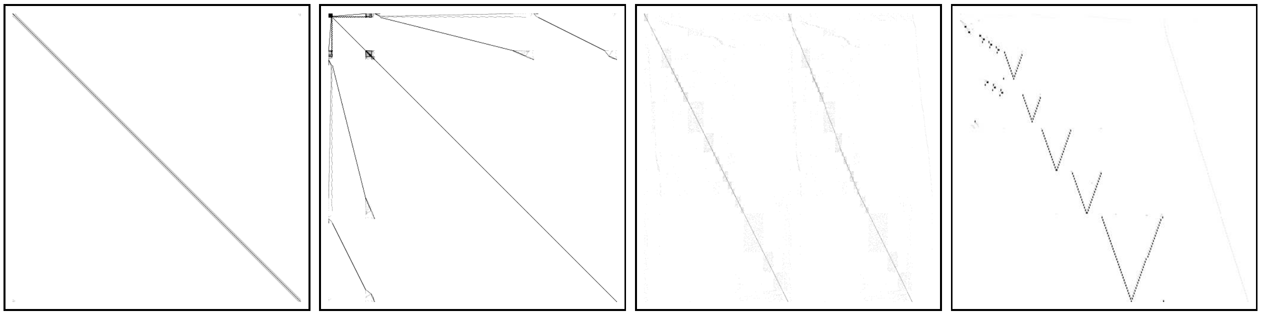


Figure A.5: (17)rim (18)memplus (19)gemat11 (20)lhr10

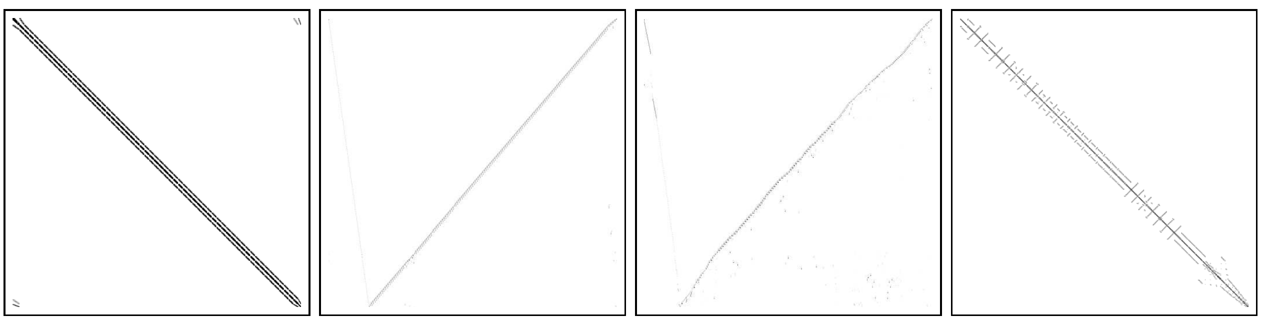


Figure A.6: (21)goodwin (22)bayer02 (23)bayer03 (24)coater2

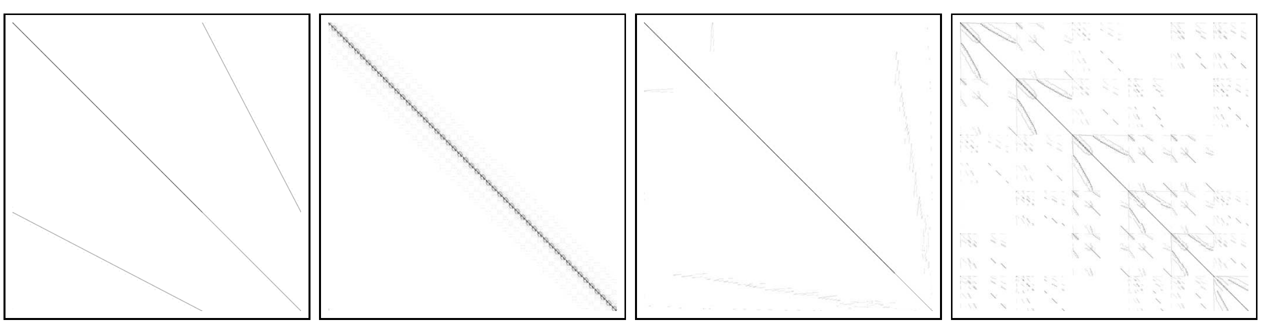


Figure A.7: (25)finan512 (26)onetone2 (27)pwt (28)vibrobox

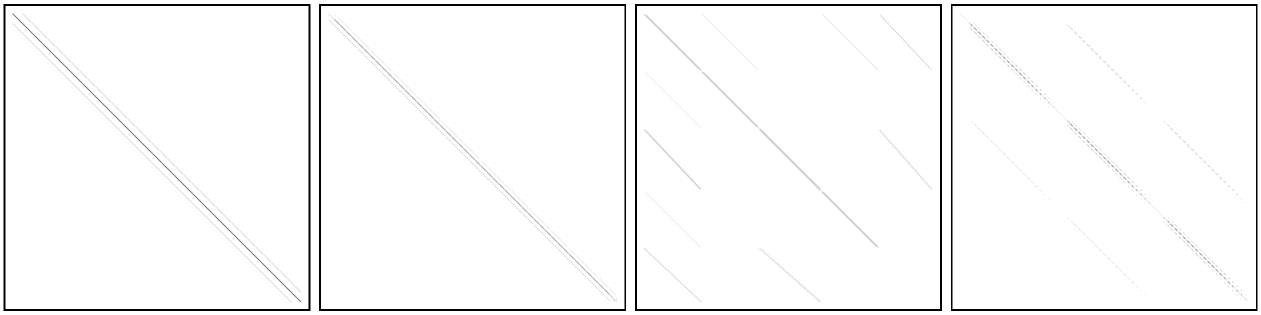


Figure A.8: (29)wang4 (30)lnsp3937 (31)lns3937 (32)sherman5

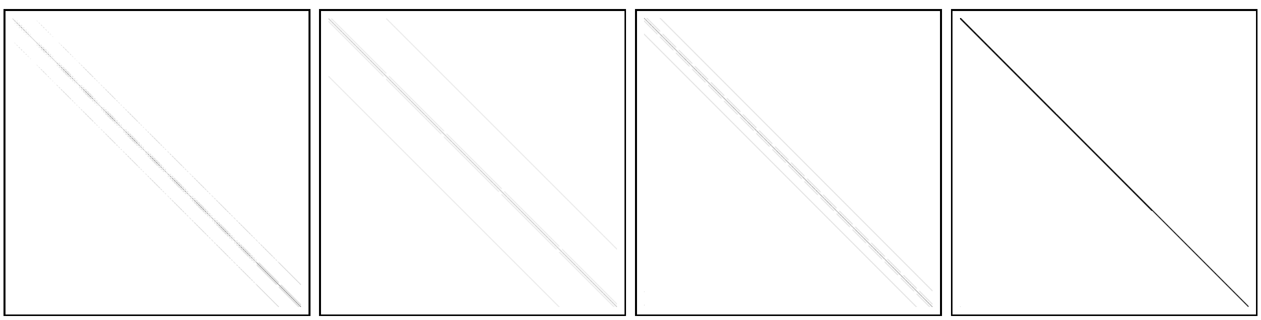


Figure A.9: (33)sherman3 (34)orsreg1 (35)saylr4 (36)shyy161

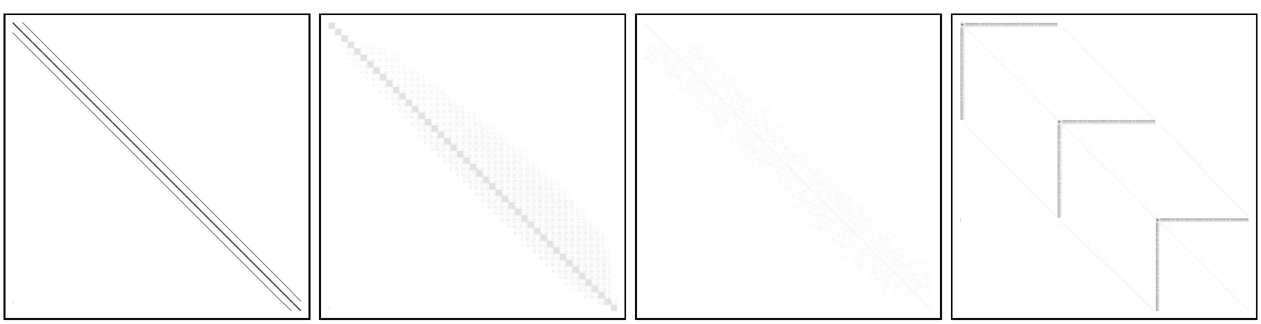


Figure A.10: (37)wang3 (38)mcfe (39)jpwh991 (40)gupta1

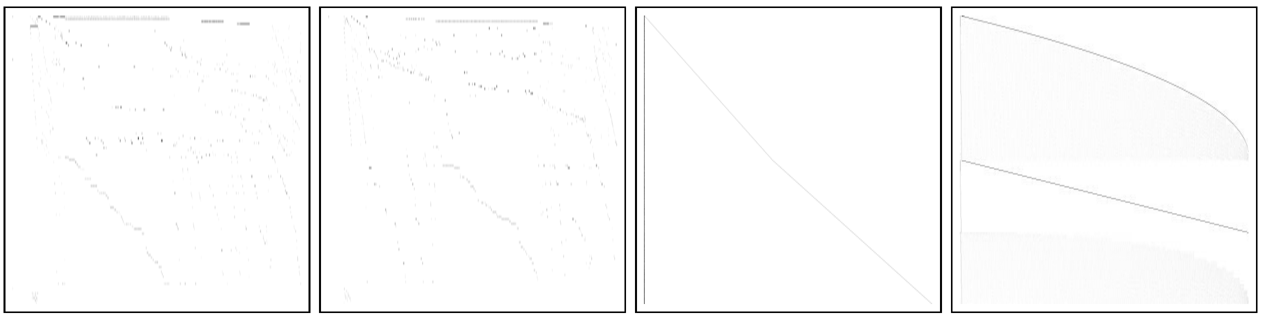


Figure A.11: (41)lpcred (42)lpfit2p (43)lpnug20 (44)nasasrb

Figure A.12: (45)lsi (46)random