SPARSITY: Optimization Framework for Sparse Matrix Kernels [*]

Eun-Jin Im
School of Computer Science, Kookmin University, Seoul, Korea

Katherine Yelick
Richard Vuduc
Computer Science Division, University of California, Berkeley

Abstract

Sparse matrix-vector multiplication is an important computational kernel that performs poorly on most modern processors due to a low compute-to-memory ratio and irregular memory access patterns. Optimization is difficult because of the complexity of cache-based memory systems and because performance is highly dependent on the nonzero structure of the matrix. The Sparsity system is designed to address these problems by allowing users to automatically build sparse matrix kernels that are tuned to their matrices and machines. Sparsity combines traditional techniques such as loop transformations with data structure transformations and optimization heuristics that are specific to sparse matrices. It provides a novel framework for selecting optimization parameters, such as block size, using a combination of performance models and search.

In this paper we discuss the optimization of two operations: a sparse matrix times a dense vector and a sparse matrix times a set of dense vectors. Our experience indicates that register level optimizations are effective for matrices arising in certain scientific simulations, in particular finite-element problems. Cache level optimizations are important when the vector used in multiplication is larger than the cache size, especially for matrices in which the nonzero structure is random. For applications involving multiple vectors, reorganizing the computation to perform the entire set of multiplications as a single operation produces significant speedups. We describe the different optimizations and parameter selection

[*]This research was supported in part by U.S. Army Research Office under grants DAAH04-96-1-0079 and DAAG55-93-1-0153, by the Department of Energy through Memorandum Agreement No. B504962 with LLNL, and under Grant No. DE-FC02-01ER25478, by the National Science Foundation under grants ACI-9619020, ACI-0090127, and ACI-9813362, by Grant No. R04-2002-000-20066-0 from Korea Science and Engineering Foundation, by the research program 2003 of Kookmin University in Korea, and a gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

techniques and evaluate them on several machines using over 40 matrices taken from broad set of application domains. Our results demonstrate speedups of up to 4× for the single vector case and up to 10× for the multiple vector case.

1 Introduction

Matrix-vector multiplication is used in scientific computation, signal and image processing, document retrieval, and other applications. In many cases the matrices are sparse so only the nonzero elements and their indices are stored. The performance of sparse matrix operations tends to be much lower than their dense matrix counterparts due to: (1) the overhead of accessing the index information in the matrix structure, and (2) the irregularity of many of the memory accesses. For example, on an 167 MHz UltraSPARC I, there is a 2x slowdown from the index overhead (measured by comparing a dense matrix in both dense and sparse format) and an additional 5x slowdown for matrices that have a nearly random nonzero structure [14]. This performance gap is due entirely to the memory system performance, and is likely to increase as the gap between processor speed and memory speed increases.

The Sparsity system is designed to help users obtain highly tuned sparse matrix kernels without having to know the details of their machine's memory hierarchy or how their particular matrix structure will be mapped onto that hierarchy. Sparsity performs register level and cache level optimizations [14], which are quite different than those performed by compilers for dense code. In particular, the data structure is changed and in some cases explicit zeros are added to the matrix to improve memory system behavior.

In this paper we describe optimization techniques used in Sparsity for tuning sparse matrix-vector multiplication. Section 3 presents the register blocking optimization, which is most useful on matrices from Finite Element Methods, because they tend to have naturally occuring dense sub-blocks. The challenge is to select the register block size, which involves a trade-off between the memory system overhead that comes from poor locality and the additional computation required to perform multiplications with explicit zeros. Section 4 describes cache blocking, which is used for problems in which the source vector is too large to fit in the cache. Section 5 considers a variation on matrix-vector multiplication in which the matrix is multiplied by a set of vectors. Sparsity reorganizes the computation to take advantage of multiple vectors, improving locality and performance as a result. An overview of the Sparsity system architecture is presented in section 6.

2 Benchmark Matrices and Machines

To evaluate Sparsity's optimization techniques, we present performance data on four machines based on the following micrprocessors: a 333 MHz Sun UltraSPARC IIi, an 800 MHz Intel Mobile Pentium III, a 900 MHz Intel Itanium

Processor and compiler	Clock (MHz)	Data cache sizes	DGEMV (MFLOPS)	DGEMM (MFLOPS)
Sun UltraSPARC IIi Sun C v6.0	333	L1: 16 KB L2: 2 MB	58	425
Intel Pentium III Mobile (Coppermine) Intel C v6.0	800	L1: 16 KB L2: 256 KB	147	590
IBM Power4 IBM xlc v6	1300	L1: 64 KB L2: 1.5 MB L3: 32 MB	915	3500
Intel Itanium 2 Intel C v7.0	900	L1: 16 KB L2: 256 KB L3: 3 MB	1330	3500

Table 1: **Summary of Machines**

2, and a 1.3 GHz IBM Power4. The machines are summarized in Table 1, where we show each processors' clock speed and cache configuration, along with performance of optimized dense Basic Linear Algebra Subroutines (BLAS) operations for comparison. We include the performance of the BLAS routines as an approximate upper bound on the sparse performance: dense matrix-vector multiplication (DGEMV) is an upper bound for sparse matrix-vector multiplication, while dense matrix-matrix multiplication (DGEMM) is an upper bound for the multiple vector case. The BLAS numbers are are for double-precision floating point numbers on a 2000×2000 matrix. For each platform, we measured the performance of the vendor-supplied, hand-optimized BLAS library, Goto's assembly-coded BLAS libraries [9], and automatically generated BLAS libraries using ATLAS; we report the performance of the best of these implementations in Table 1.

Since the optimizations also depend strongly on matrix size and structure, we use a large set of matrices taken from a range of application domains for our experiments. Table 2 summarizes the matrices. We have placed the matrices in the table according to our understanding of the application domain from which it was derived. Matrix 1 is a dense matrix, which is included in our suite for comparison to the DGEMV performance. Matrices 2 through 17 are from Finite Element Method (FEM) applications, which in several cases means there are dense sub-locks within many parts 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 a variety of engineering and science applications, including circuit simulation, computational fluid dynamics, chemistry, and structural dynamics. Matrix 40 comes from a text retrieval (latent semantic indexing) application [4]. Matrices 41 through 43 come from linear programming problems. Finally, matrix 44 comes from a statistical experimental design problem. All of the first 39 matrices are square, and although some are symmetric, we do not try to take advantage of symmetry in this paper. The matrices are roughly ordered by the regularity of nonzero patterns, with the more regular ones at the top.

3 Register Optimizations for Sparse Matrices

3.1 Description of the Register Optimizations

The performance of sparse matrix operations are typically limited by the memory system, because the ratio of the number of memory reference instructions to the number of floating point operation is high, due to the indirect data structure representing sparse matrix. Hence our first optimization technique is designed to eliminate loads and stores by reusing values that are in registers. For matrix-vector multiplication, there are few opportunities for register reuse, because each element of the matrix is used only once. To make this discussion concrete, we assume that we are starting with a fairly general representation of a sparse matrix called Compressed Sparse Row (CSR) format. In CSR, all row indices are stored (by row) in one vector, all matrix values are stored in another, and a separate vector of indices indicates where each row starts within these two vectors. In the calculation of $y = A \times x$, where A is a sparse matrix and x and y are dense vectors, the computation may be organized as a series of dot-products on the rows. In computing $y = A \times x$, the elements of A are accessed sequentially but not reused. The elements of y are also accessed sequentially, and they are reused for each nonzero in the row of A. The access to x is irregular, as it depends on the column indices of nonzero elements in matrix A.

Register reuse of y and A cannot be improved, but access to x may be optimized if there are elements in A that are in the same column and nearby one another, so that an element of x may be saved in a register. To improve locality, SPARSITY stores a matrix as a sequence of small dense blocks, and organizes the computation to compute each block before moving on to the next. This blocked format also has the advantage of reducing the amount of memory required to store indices for the matrices, since a single index is stored per block. To take full advantage of the improved locality for register allocation, we fix the block sizes at compile time. SPARSITY therefore generates code for matrices containing only full dense blocks of some fixed size $r \times c$, where each block starts on a row that is a multiple of r and a column that is a multiple of c. The code for each block is unrolled, with instruction scheduling and other optimizations applied by the C compiler. The assumption is that all nonzeros must be part of some $r \times c$ block, so SPARSITY will transform the data structure by adding explicit zeros where necessary. We show an example of the unblocked, reference code in Figure 1, and 2×2 blocked code in Figure 2.

	Name	Application Area	Dimension	Nonzeros	%
1	dense1000	Dense Matrix	1000x1000	1000000	100
2	raefsky3	Fluid/structure	21200x21200	1488768	0.33
3	inaccura	Accuracy problem	16146x16146	1015156	0.39
4	bcsstk35	Automobile frame	30237x30237	1450163	0.16
5	venkat01	Flow simulation	62424x62424	1717792	0.04
6	crystk02	FEM Crystal	13965x13965	968583	0.50
7	crystk03	FEM Crystal	24696x24696	1751178	0.29
8	nasasrb	Shuttle rocket booster	54870x54870	2677324	0.09
9	3dtube	3-D pressure tube	45330x45330	3213332	0.16
10	ct20stif	CT20 Engine block	52329x52329	2698463	0.10
11	bai	Airfoil eigenvalue	23560x23560	484256	0.09
12	raefsky4	Buckling problem	19779x19779	1328611	0.34
13	ex11	3D steady flow	16614x16614	1096948	0.40
14	rdist1	Chemical processes	4134x4134	94408	0.55
15	vavasis3	2D PDE problem	41092x41092	1683902	0.10
16	orani678	Economic modeling	2529x2529	90185	1.41
17	rim	FEM fluid mechanics	22560x22560	1014951	0.20
18	memplus	Circuit Simulation	17758x17758	126150	0.04
19	gemat11	Power flow	4929x4929	33185	0.14
20	lhr10	Light hydrocarbon	10672x10672	232633	0.20
21	goodwin	Fluid mechanics	7320x7320	324784	0.61
22	bayer02	Chemical process	13935x13935	63679	0.03
23	bayer10	Chemical process	13436x13436	94926	0.05
24	coater2	Coating flows	9540x9540	207308	0.23
25	finan512	Financial optimizaion	74752x74752	596992	0.01
26	onetone2	Harmonic balance	36057x36057	227628	0.02
27	pwt	Structural engineering	36519x36519	326107	0.02
28	vibrobox	Vibroacoustics	12328x12328	342828	0.23
29	wang4	Semiconductor devices	26068x26068	177196	0.03
30	lnsp3937	Fluid flow	3937x3937	25407	0.16
31	lns3937	Fluid flow	3937x3937	25407	0.16
32	sherman5	Oil reservoir	3312x3312	20793	0.19
33	sherman3	Oil reservoir	5005x5005	20033	0.08
34	orsreg1	Oil reservoir	2205x2205	14133	0.29
35	saylr4	Oil reservoir	3564x3564	22316	0.18
36	shyy161	Viscous flow	76480x76480	329762	0.01
37	wang3	Semiconductor devices	26064x26064	177168	0.03
38	mcfe	Astrophysics	765x765	24382	4.17
39	jpwh991	Circuit physics	991x991	6027	0.61
40	webdoc	Document Clustering	10000x255943	37124897	0.15
41	nug30	Linear programming	52260x379350	1567800	0.0079
42	osa60	Linear programming	10280x243246	1408073	0.056
43	rail4284	Railway scheduling	4284x1092610	11279748	0.24
44	bibd_22_8	Experimental design	231x319770	8953560	12.0

Table 2: **Matrix benchmark suite:** The basic characteristic of each matrix used in our experiments is shown. The sparsity column is the percentage of nonzeros.

```c
void smvm_1x1( int m, const double* value,
           const int* col_idx, const int* row_start,
           const double* x, double* y )
{
    int i, jj;

    /* loop over rows */
    for( i = 0; i < m; i++ ) {
        double y_i = y[i];

        /* loop over non-zero elements in row i */
        for( jj = row_start[i]; jj < row_start[i+1];
             jj++, col_idx++, value++ ) {
            y_i += value[0] * x[col_idx[0]];
        }
        y[i] = y_i;
    }
}
```

Figure 1: **Reference implementation.** A standard C implementation of SMVM for $y = y + Ax$, assuming CSR storage and C-style 0-based indexing. A is an $m \times n$ matrix. This is a modification of the corresponding NIST routine.

```c
void smvm_2x2( int bm, const int *b_row_start,
               const int *b_col_idx, const double *b_value,
               const double *x, double *y )
{
    int i, jj;

    /* loop over block rows */
    for( i = 0; i < bm; i++, y += 2 ) {
        register double d0 = y[0];
        register double d1 = y[1];
        for( jj = b_row_start[i]; jj < b_row_start[i+1];
               jj++, b_col_idx++, b_value += 2*2 ) {
            d0 += b_value[0] * x[b_col_idx[0]+0];
            d1 += b_value[2] * x[b_col_idx[0]+0];
            d0 += b_value[1] * x[b_col_idx[0]+1];
            d1 += b_value[3] * x[b_col_idx[0]+1];
        }
        y[0] = d0;
        y[1] = d1;
    }
}
```

Figure 2: **Example: 2×2 register blocked code.** Here, `bm` is the number of block rows, i.e., the number of rows in the matrix is `2*bm`. The dense sub-blocks are stored in row-major order.

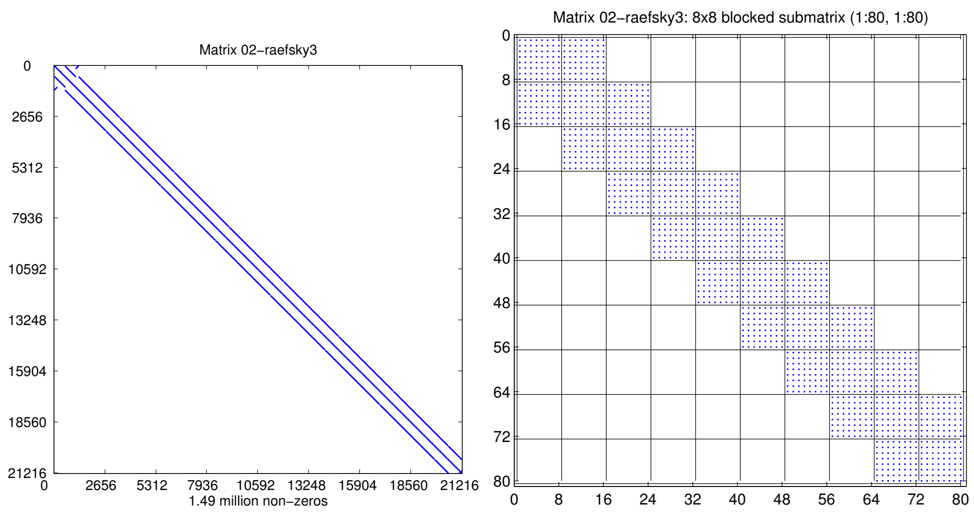


Figure 3: **Sparse matrix example.** A macroscopic view of the non-zero structure of matrix #2, raefsky3 (*left*). This matrix is structurally, but not numerically, symmetric. Furthermore, the matrix consists entirely of uniformly aligned 8×8 blocks as shown by the 80×80 leading submatrix (*right*).

3.2 Choosing the Register Block Size

The idea of register *blocking* or *tiling* for dense matrix operations is well-known (e.g., [17]), but the sparse matrix transformation is quite different, since it involves filling in zeros, which add both storage and computation overhead. Blocked sparse matrix formats are commonly used in applications where the matrices are constructed one dense block at a time. The critical question in our work is how to select the block dimensions r and c for optimal performance. This is quite different than the block size selection problem for dense matrices, because it depends on the nonzero structure of the matrix. We also find that the best blocking factor is not always the one naturally chosen by a user, since it may vary across machines for a given matrix. We illustrate how the choice of block size can be surprising using the following experimental example. Figure 3 shows the non-zero structure of matrix #2 (raefsky3) in Table 2. This matrix consists entirely of 8×8 blocks, uniformly aligned as shown in the figure. A user is likely to choose a square block size such as 8×8, or possibly even 4×4 if register pressure is a known problem. Indeed, PETSc, a standard packages in scientific computing [3], only allows square block sizes at the time of this writing.

For this example, consider an experiment in which we measure the performance of sixteen $r \times c$ implementations where $r, c \in \{1, 2, 4, 8\}$. All of these block sizes are "natural" in that they evenly divide the largest, natural block size of 8×8, and therefore require no fill. We show the results of such an experiment on the four evaluation machines in Figure 4. Each plot shows all 16 implementations, each shaded by its performance in MFLOPS and labeled by its speedup over the reference (1×1) code. On the UltraSPARC IIi, we see the expected behavior: performance increases smoothly as r and c increase, and indeed 8×8 blocking is the best choice. By contrast, 4×4 and 8×8 are good choices on the Itanium 2 but nevertheless only half as fast as the optimal choice of 4×2 blocking. The best block size is also non-square on the other platforms: 2×8 on the Pentium III and 4×1 on the Power4. On the Power4, 4×4 is actually not much slower than the optimal 4×1 block size, but it is not clear why 8×8 should be 10% slower than either, since the Power4 has twice as many double-precision floating point registers as the UltraSPARC IIi (32 vs. 16 registers). Thus, even in this simple example, we see that the choice of block size is not always obvious.

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 model is used to select a good block size. 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, the index structure is more compact, because there is only one index stored per block, rather than per nonzero. However, some zero elements are filled in to make a complete $r \times c$ block, and these extra zero values not only consume storage, but increase the number of floating point operations, because they are involved in the sparse matrix computation. The number of added zeros in the blocked representation are referred to as *fill*, and the ratio of entries before and after fill is the *fill overhead*. Our performance model for selecting register block size 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 *fill overhead*.

These two components differ in the amount of information they require: the first needs the target machine but not the matrix, whereas the second needs the matrix structure but not the machine.

3.2.1 Machine Profile

The first component cannot be exactly determined without running the resulting blocked matrix on each machine of interest. To avoid the cost of running all these for each matrix structure of interest, we use an approximation for this Mflop rate, which is the performance of a dense matrix stored in the blocked sparse format. The second component could be computed exactly for a given matrix, but is quite expensive to compute for multiple block sizes. Instead, we develop an approximation that can be done in a single pass over only a part of the matrix.

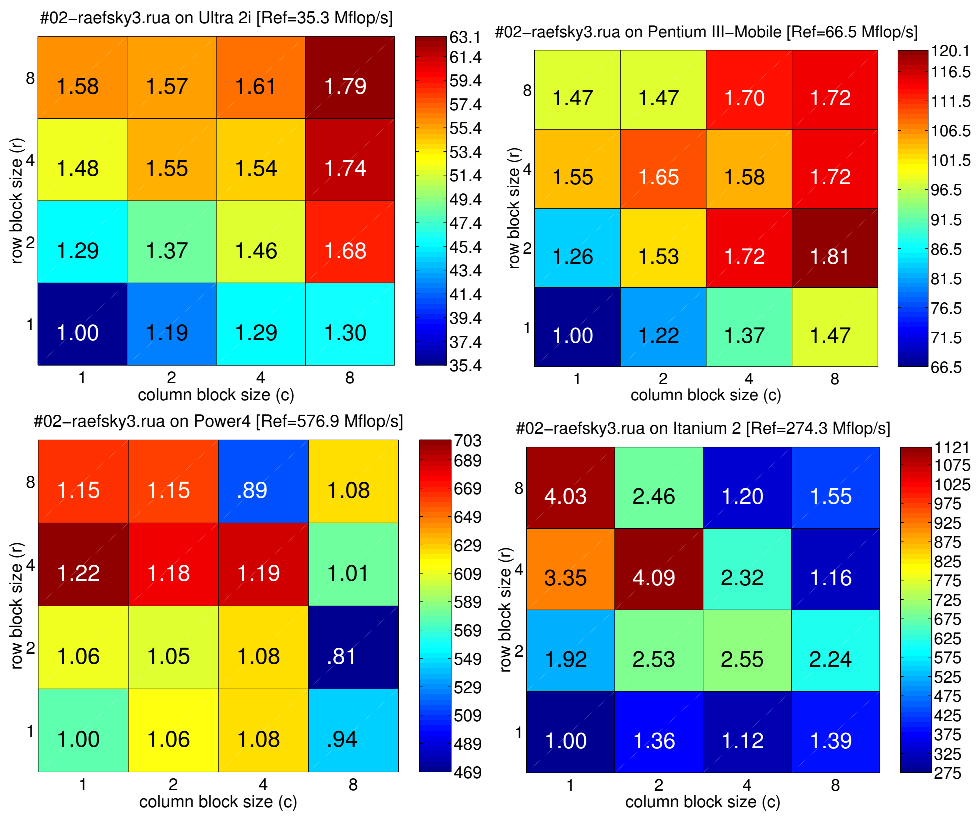
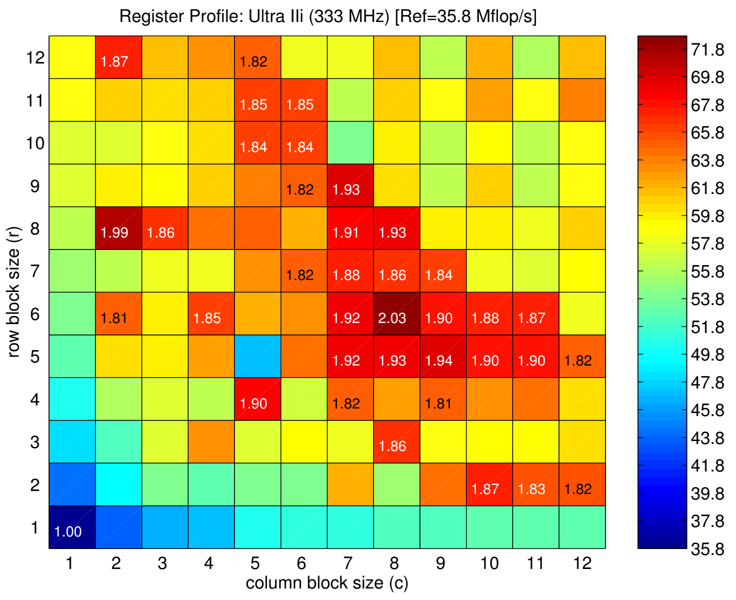


Figure 4: **Register blocking performance: Matrix #2, raefsky3.** We show the results of register blocking the matrix shown in Figure 3 on four platforms (clockwise from upper-left): UltraSPARC IIi, Intel Pentium III, Itanium 2, and Power4. On each platform, each square is an $r \times c$ implementation shaded by its performance in MFLOPS and labeled by its speedup relative to the unblocked CSR implementation (1×1). Sixteen implementations are shown for $r, c \in \{1, 2, 4, 8\}$. The optimal block sizes are 8×8 on the UltraSPARC IIi (1.79x speedup, 63 MFLOPS), 2×8 on the Pentium III (1.81x, 120 MFLOPS), 4×2 on the Itanium 2 (4.09x, 1.1 GFLOPS), and 4×1 on the Power4 (1.22x, 700 MFLOPS).



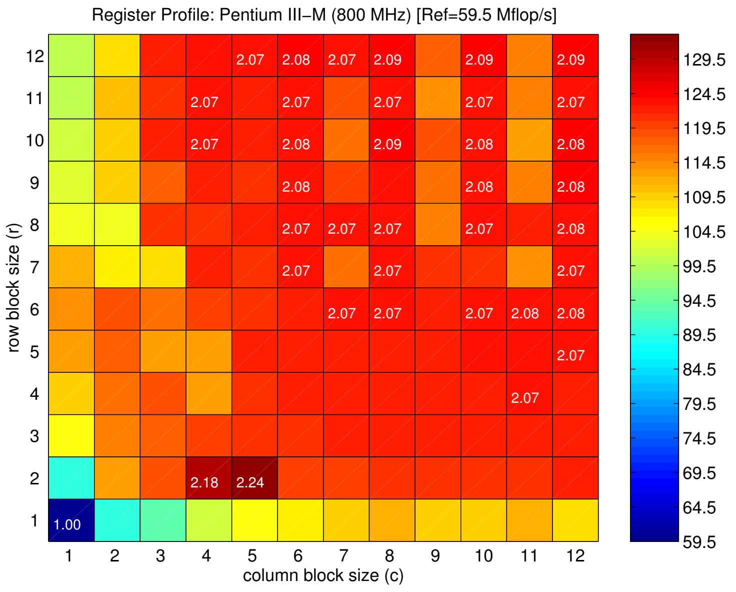
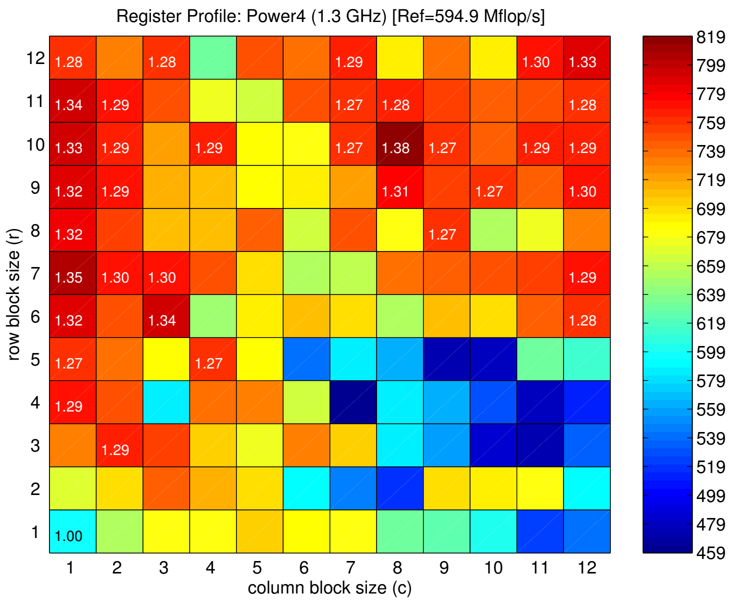


Figure 5: **Performance profile of register-blocked code on an UltraSPARC IIi (top) and a Pentium III (bottom).** Each $r \times c$ implementation is shaded by its performance (MFLOPS); the top 25% of the implementations are labeled by their speedup relative to the unblocked (1×1) case (lower-leftmost implementation, labeled 1.0). The largest observed speedup is 2.03 at 6×8 on the UltraSPARC IIi, and is 2.24 on the Pentium III at 2×5. The baseline performance is 36 MFLOPS on the UltraSPARC IIi and 60 MFLOPS on the Pentium III. The best performance is 72 MFLOPS on the UltraSPARC IIi and 130 MFLOPS on the Pentium III.



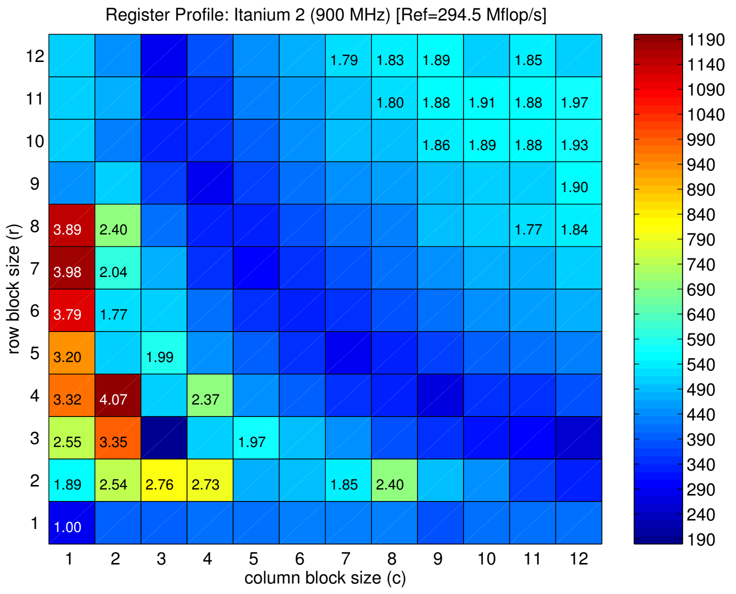


Figure 6: **Performance profile of register-blocked code on a Power4 (top), and an Itanium 2 (bottom).** The largest observed speedup is 1.38 at 10×8 on the Power4, and 4.07 at 4×2 on the Itanium 2. The baseline performance is 595 MFLOPS on the Power4 and 300 MFLOPS on the Itanium 2; the best observed performance is 820 MFLOPS on the Power4 and nearly 1.2 GFLOPS on the Itanium 2.

Figures 5 and 6 show the performance of sparse matrix vector multiplication for a dense matrix using register-blocked sparse format. Specifically, each square represents an implementation at a particular row block size (varying along the y-axis) and column block size (x-axis). Each implementation is shaded by its performance in MFLOPS. We call these plots the *machine profile* because they provide the machine-specific component of our performance model. The data was collected using a 2000×2000 dense matrix in sparse format, although the performance is relatively insensitive to the total matrix size as long as the matrix is small enough to fit in memory but too large to fit in cache. We vary the block size within a range of $r \times c$ values from 1×1 up to 12×12. This limit is likely to be reasonable in practice, since none of the application matrices in Table 2 have many naturally occuring dense blocks larger than 8×8.

From these profiles, we can see some interesting characteristics of the machines with respect to sparse matrix operations. First, the difference between the graphs shows the need for machine-specific tuning, and with the exception of the Pentium III (and to some extent the UltraSPARC IIi), the performance curves are far from smooth, so a small change in block size can make a large difference in performance. While the reasons for this erratic behavior are not clear, both the memory system structure and the compiler are significant factors. The code is C code generated automatically by the SPARSITY system, and for register blocked code, the basic blocks can be quite large. This may, for example, explain the noticeable drop in performance for large block sizes on the UltraSPARC. Overall, these graphs show the difficulty of choosing the register block size fully automatically, even for the simplest case of a dense matrix in sparse format. These observations motivate our use of the machine profiles for optimization.

Second, the difference between the 1×1 performance and the best case will give us a rough idea of the kind of speedups we may expect from register blocking real matrices. For the UltraSPARC and Pentium III, there is roughly a $2\times$ difference between the 1×1 performance and the best case. The Itanium 2 has a wider range of $4\times$, whereas the Power4 maximum is $1.4\times$. Note that the baseline on the Power4 is significantly faster than the baseline on the other machines—nearly 600 MFLOPS compared to the second fastest of about 300 MFLOPS on the Itanium 2.

3.2.2 Estimating Fill Overhead

To approximate the unnecessary computation that would result from register blocking, we estimate the fill overhead. For each r, we select 1% of the block rows uniformly at random and count the number of zeros which would be filled in for all c simultaneously. Currently, we limit our estimate to sizes up to 12×12, though on the matrix benchmark suite we have not observed optimal sizes greater than 8×8. Also, we perform the 1% scan independently for each r, though this could obviously be improved by simultaneously scanning r and its factors (*e.g.*, while scanning $r = 12$, simultaneously search $r = 1$, 2, 3, 6, and 12). As described and implemented, we scan up to 12% of the matrix.

Nevertheless, the cost of this procedure for all r, c is in practice less than the cost of converting the matrix from CSR to register blocked format. Furthermore, our procedure typically estimates the fill ratio to within 1% on FEM matrices, and to typically within 5–10% on the other matrices in our benchmark suite.

We use this estimate of fill overhead to predict the performance of an $r \times c$ blocking of a particular matrix A as:

$$\frac{\textit{performance of a dense matrix in } r \times c \textit{ sparse blocked format}}{\textit{estimated fill overhead for } r \times c \text{ blocking of } A}$$

We choose the r and c that maximizes this performance estimate.

The heuristic is imperfect, and to ensure that SPARSITY never produces code that is slower than the naive implementation, we run the transformed code with the selected block size against the unblocked code, and select whichever is fastest. A more aggressive optimization approach would be to search exhaustively over a set of possible block sizes using the matrix of interest, but that may be too expensive for most application programmers. For example, the cost of converting the matrix to any given block size is at least an order of magnitude more expensive than performing a single matrix-vector multiplications. Our approach of using a heuristic to select block size, followed by a single search gives good performance with much less overhead than exhaustive search.

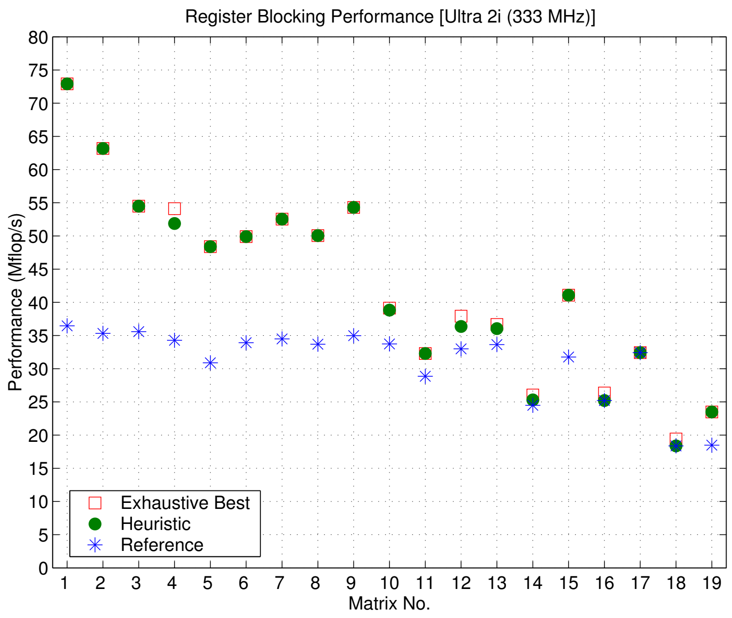
3.3 Performance of Register Optimizations

We ran SPARSITY on all of the benchmark matrices in Section 2. The optimizations are quite effective on the more structured matrices across all machines. On the less sturctured matrices, numbered 20-44, we see no benefit from register blocking on most machines.[1] We therefore present performance results only for matrices 1 through 19. Note that when reporting a Mflop rate, we do not count the extra operations with explicitly filled in zeros as floating point operations. Thus, for a given matrix, comparing performance is equivalent to comparing inverse time.

Figures 7–8 compare the performance of the following three implementations:

- **Exhaustive best**—The implementation in which we choose the block size for each matrix by exhaustive search over 1×1 to 12×12. We denote this block size by $r_{\mathrm{o}} \times c_{\mathrm{o}}$.
- **Heuristic**—The implementation in which we choose the block size using our heuristic. We denote this block size by $r_{\mathrm{h}} \times c_{\mathrm{h}}$.
- **Reference**—The baseline implementation, our unblocked 1×1 CSR code. This baseline code is comparable to that of other unblocked implementations, such as the NIST Sparse BLAS Toolkit [22].

[1] On Itanium 2, register blocking pays off even on the unstructured matrices. We will present those results in Section 5.



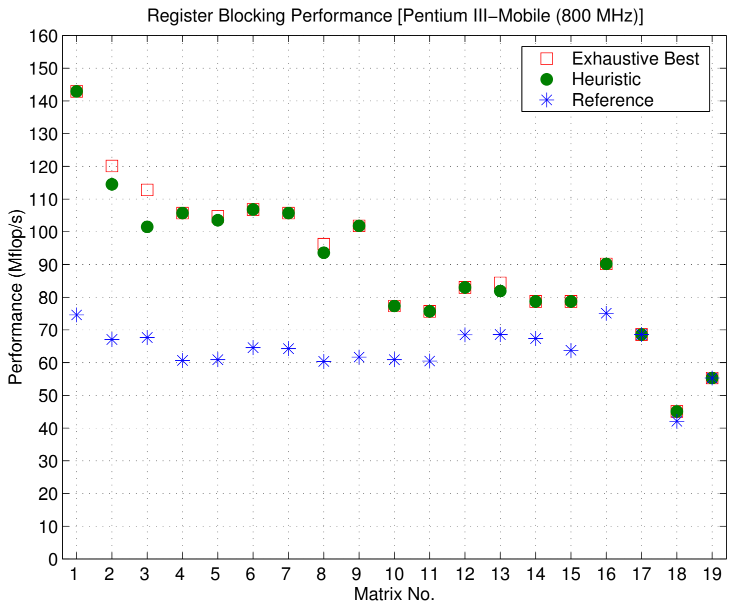
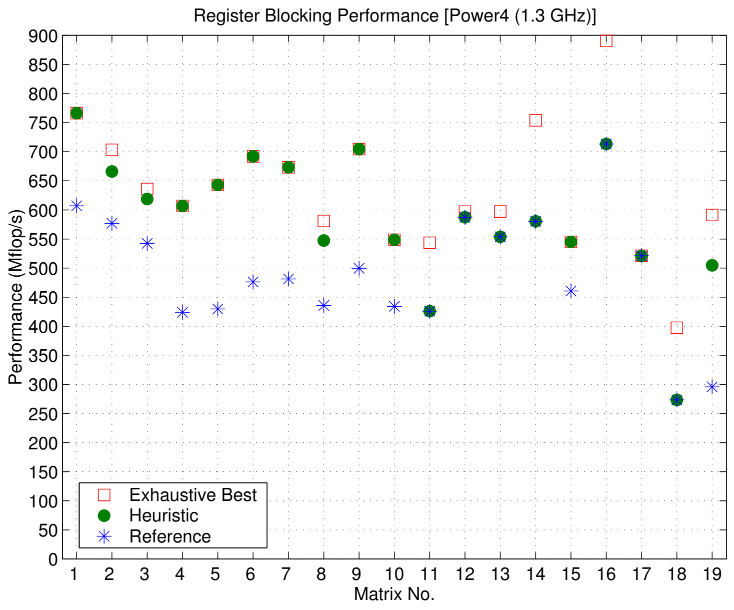


Figure 7: **Register blocking performance on the UltraSPARC IIi (top) and Pentium III (bottom) platforms.** For each matrix (x-axis), we compare the performance (MFLOPS, y-axis) of three implementations: (1) the best performance when r and c are chosen by exhaustive search, (2) the performance when r and c are chosen using our heuristic, and (3) the unblocked reference code.



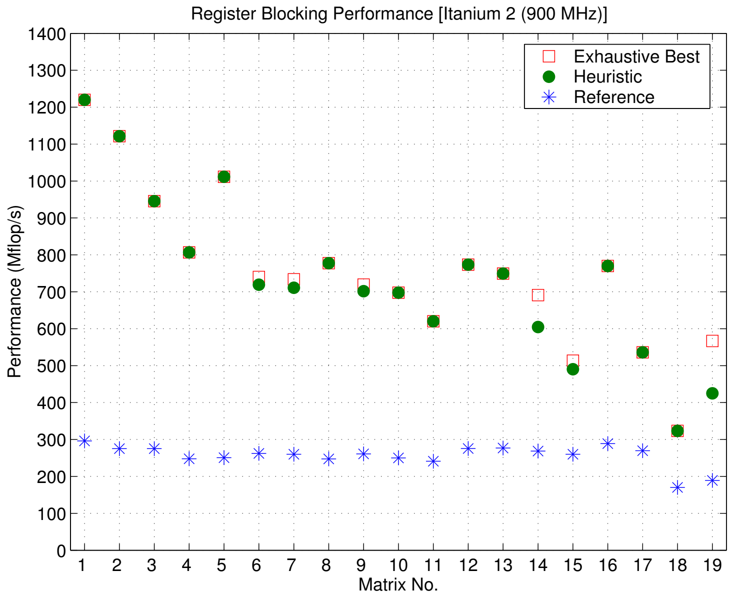


Figure 8: **Register blocking performance on the Power4 (top) and Itanium 2 (bottom) platforms.** For each matrix (x-axis), we compare the performance (MFLOPS, y-axis) of three implementations: (1) the best performance when r and c are chosen by exhaustive search, (2) the performance when r and c are chosen using our heuristic, and (3) the unblocked reference code.

In Appendix A, we show the values of $r_o \times c_o$ and $r_h \times c_h$, as well as the resulting fill overheads. As noted earlier the best register block sizes differ across machines as well as matrices. For example, the best block sizes for matrix 4 (bcsstk35) are 6×2 on the UltraSPARC IIi, 3×3 on the Pentium III and Power4, and 4×2 on the Itanium 2. The higher fill overhead required for the larger blocks are not justifed by the performance profiles on the Pentium III and Power4.

Comparing just the exhaustive best implementation and the reference, we see that register blocking shows significant speedups with a maximum of 4× for this set of machines and matrices. As expected from the register profiles, the Itanium 2 shows the highest performance and speedups relative to the baseline, while the Power4 has the lowest speedups. The benefits are also highest on the the lower numbered matrices, which are finite element matrices with relatively large natural blocks. Even in these cases, the blocks are not uniform throughout the matrix, so there is some noticeable fill overhead. In some cases a surprisingly large number of zeros can be filled in while still obtaining a speedup: on the Itanium 2, a number of matrices have a fill overhead of more than 1.5, while still being a factor of 2x or more than the baseline.

In addition, we see that our heuristic usually selects the optimal or near-optimal block size. On the UltraSPARC IIi and Pentium III platforms, the performance of the heuristically chosen implementation is never more than 10% slower than the exhaustive best code. On Itanium 2, the code chosen by heuristic does somewhat worse than this for matrices 14 and 19, and on Power 4, the heuristic performance is more than 10% slower in 5 instances (matrices 11, 14, 16, 18, and 19). We note matrices 14 and 19 have the fewest number of non-zeros of matrices 1–19. Indeed, the 5 matrices for which our heuristic made a sub-optimal choice on the Power4 also have the fewest number of non-zeros of matrices 1–19. This suggests that the machine profile, which we chose to be large relative to the caches, is not a good match to these small matrices. We have performed some preliminary experiments in which we use a machine profile based on an in-L2 cache workload (i.e., a dense matrix which fits in the L2 cache), and the new predicted block sizes yield performance within 10% of the exhaustive best. Therefore, future work could consider collecting multiple profiles and matching profiles to a given matrix.

4 Cache Optimizations for Sparse Matrices

In this section we describe an optimization technique 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. 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 vectors. In cache 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 [12]. Because

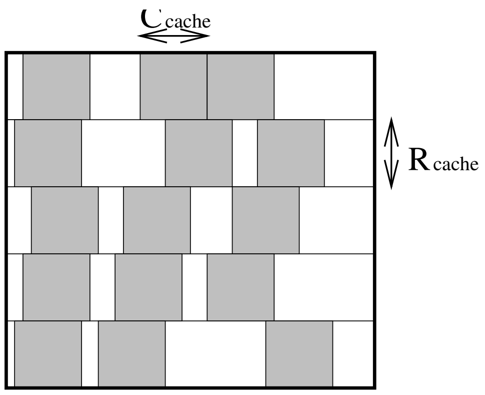


Figure 9: **Cache-blocks in a sparse matrix:** The gray areas are sparse matrix blocks that contain nonzero elements in the $r_{cache} \times c_{cache}$ rectangle. The white areas contain no nonzero elements, and are not stored.

the caches can hold thousands of values, it is not practical to fill in an entire cache block with zeros. Instead, we preserve the sparse structure but rearrange the order of computation to improve cache reuse. (In contrast, register blocking avoids some of the indexing and loop overhead by filling in dense subblocks to make them uniform, but this is not practical for cache blocking due to the big size of cache block and to low density of such block.)

4.1 Description of Cache Optimizations

The idea of cache blocking optimization is to keep c_{cache} elements of the source vector x in the cache 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 because this decision is under hardware control, interference between elements of the matrix and the two vectors can be a problem.

One of the difficulties with cache blocking is determining the block sizes, r_{cache} and c_{cache}. To simplify the code generation problem and to limit the number 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 the 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 9 shows a matrix with fixed size cache blocks. Note that the blocks need not begin at the same offsets in each row.

We have considered two strategies for cache blocking: The first implementation, referred to as *static cache blocking*, involves a preprocessing step to reorganize the matrix so that each block is stored contiguously in main memory. In the second implementation, referred to as *dynamic cache blocking*, 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. Although dynamic cache blocking avoids any preprocessing overhead, it incurs significantly more runtime overhead than static cache blocking [14], so SPARSITY uses static cache blocking.

4.2 Performance of Cache Optimizations

Only matrices with very large dimensions will benefit from cache blocking, i.e., if the source vector easily fits in a cache and still leaves room for the matrix and destination elements to stream through without significant conflicts, there is no benefit to blocking. Matrices 40–44 are large enough to cause conflicts of this kind, since the source vectors are large—in particular, the source vectors are all at least 2 MB in size, which is at least as large as the L2 cache on all four machines.

We applied cache blocking to these matrices and measured the speedups on the four machines used in the previous section. We also applied combination of register blocking and cache blocking to these matrices. However, since the register block sizes for these matrices were chosen to be 1×1, the optimization was reduced to a simple cache blocking. The results of cache blocking are shown in Figure 10. We show, for each platform, raw performance in MFLOPS before and after cache blocking. Cache blocking performance is also labeled by speedup over the unblocked code.

For these matrices, the benefits are significant: we see speedups of up to 2.2x. Cache blocking appears to be most effective on matrix 40, which sees the largest speedups, and least effective on matrix 41, which did not see any speedup on two machines. The selected block sizes are shown in Table 3. The unoptimized performance is relatively poor: the unblocked code runs at only 15–24 MFLOPS on the Ultra IIi, 25–42 MFLOPS on the Pentium III, 100–280 MFLOPS on the Power4, and 170–220 MFLOPS on the Itanium 2. Roughly speaking, if we order the matrices by increasing density—matrix 41, 42, 40, 43, and 44—we see that the cache blocked performance also tends to increase. Note that while cache blocking is only of interest on only a few of these matrices in the benchmark suite, we believe this more of a reflection on the age of these matrices, which came primarily from standard benchmarks suites. On modern machines much larger matrices are likely to be used.

The cache block sizes are chosen automatically by the SPARSITY system after measuring the performance for rectangular block sizes between 32×32 and $128K \times 128K$ that are powers of two. We may miss optimal block size by searching only for block sizes of powers of 2, but this choice is made because it is practically impossible to search for all possible block sizes since the range of block sizes are enormous.

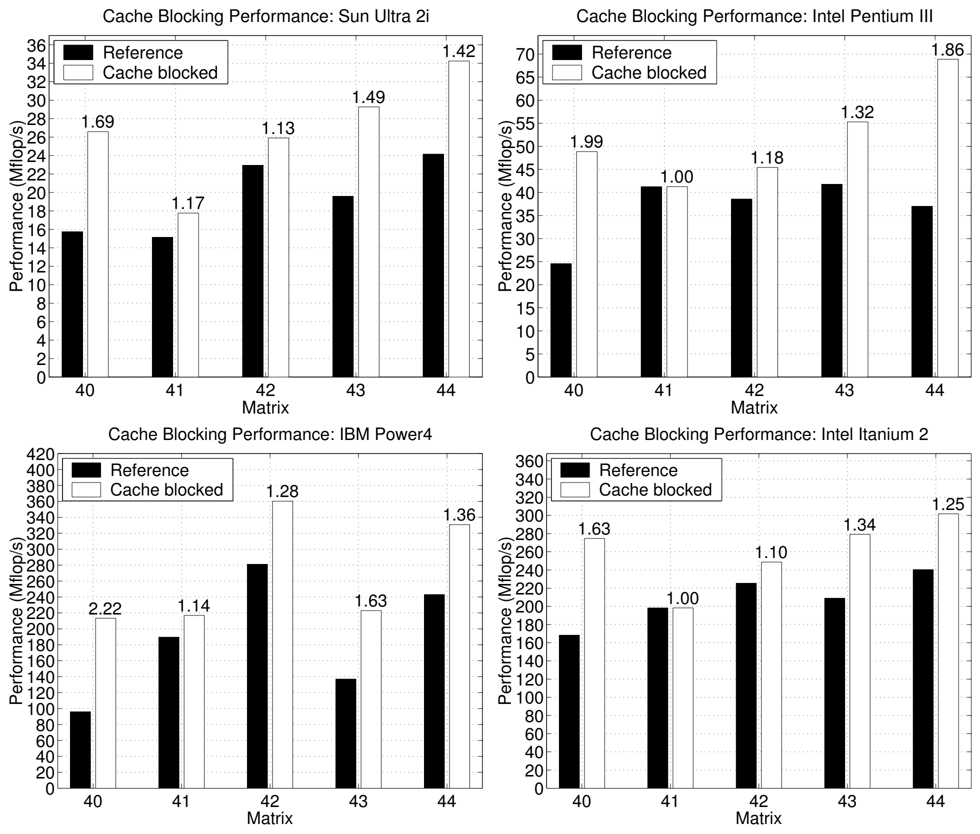


Figure 10: **Speedup from cache blocking.** We show performance (MFLOPS, y-axis) of the reference implementation compared to a cache-blocked implementation for Matrices 40–44 (x-axis). Data from four platforms are shown (clockwise from top-left): UltraSPARC IIi, Pentium III, Itanium 2, and Power4. Each of the bars corresponding to cache-blocked performance is labeled above by its speedup over the reference performance.

Matrix	Ultra IIi	Pentium III	Power4	Itanium 2
`40-webdoc`	10000×32768	10000×8192	10000×32768	10000×65536
`41-lp_nug30`	2048×32768	—	32768×65536	—
`42-lp_osa60`	10280×32768	4096×8192	8192×65536	4096×65536
`43-rail4284`	4284×16384	4284×8192	4284×32768	4284×32768
`44-bibd_22_8`	231×1024	231×4096	231×4096	231×4096

Table 3: **Chosen Cache Block Sizes** A dash "—" indicates that it was faster to leave the code unblocked. See Figure 10 for corresponding performance results.

5 Optimizing for Multiple Vectors

Both register blocking and cache blocking improve memory locality so that a sparse matrix might approach the performance of a matrix-vector operation in dense format. Neither can turn these operations into a BLAS-3 operation like matrix-matrix multiplication, which has a much higher computation to memory ratio. To see these benefits, we look at a variation on the sparse matrix-vector multiplication, which is multiplying a sparse matrix times a set of dense vectors (or equivalently times a dense matrix, but usually one that is very tall and thin). This type of operation occurs in practice when there are multiple right-hand sides in an iterative solver, in recently proposed blocked iterative solvers [2], and in blocked eigenvalue algorithms, such as block Lanczos [8, 10, 11, 19, 1] or block Arnoldi [25, 24, 18, 1], It also occurs in image segmentation algorithm in video, where a set of vectors is used as the starting guess for a subsequent frame in the video [26].

5.1 Description of Multiple Vector Optimizations

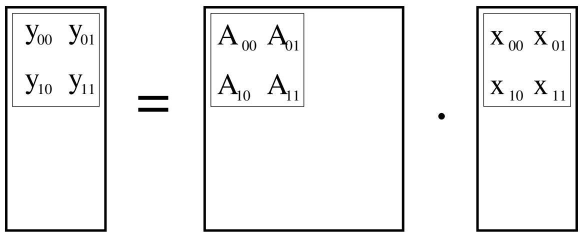
When multiplying a sparse matrix by a set of vectors, the code for multiplication by a single vector can be repeatedly used, but the extra locality advantages are not likely to be exhibited under such conditions. Figure 11 illustrates the sequence of steps for the algorithm, showing that two uses of the same matrix element are nz steps apart, where nz is the number of nonzeros in the matrix. Multiplication can be optimized for the memory hierarchy by moving those operations together in time, as shown in figure 12.

The code generator of Sparsity produces register-blocked multiplication codes for a fixed number of vectors. The number of vectors, v, is fixed and the loops across v are fully unrolled. Because of full unrolling, different code is generated for each value of v. The strategy is used because we view these as inner loops of a larger stripmined code.

5.2 Choosing the Number of Vectors

The question of how to choose the number of vectors v 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 or the convergence of an iterative algorithm may slow as the number of vector increases. If there are a large number of vectors available, and the only concern is performance, the optimization space is still quite complex because there are three parameters to consider: the number of rows and columns in register blocks, and the number of vectors.

Here 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 dependent on the compiler. We will simplify the discussion by looking at



(1) $y_{00} = A_{00} x_{00} + A_{01} x_{10}$

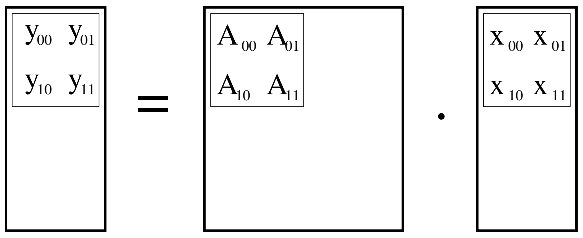
(2) $y_{10} = A_{10} x_{00} + A_{11} x_{10}$

$\cdots$

(nz+1) $y_{01} = A_{00} x_{01} + A_{01} x_{11}$

(nz+2) $y_{11} = A_{10} x_{01} + A_{11} x_{11}$

Figure 11: **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.



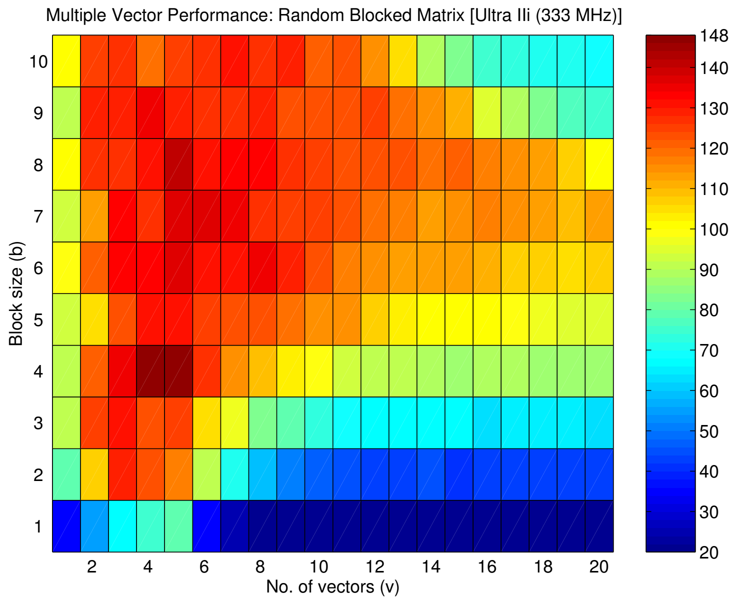
(1) $y_{00} = A_{00} x_{00} + A_{01} x_{10}$

(2) $y_{10} = A_{10} x_{00} + A_{11} x_{10}$

(3) $y_{01} = A_{00} x_{01} + A_{01} x_{11}$

(4) $y_{11} = A_{10} x_{01} + A_{11} x_{11}$

Figure 12: **Sequence of multiple vector code:** This example is the same as that in figure 11, except that the code has been reorganized to use each element twice (once per vector) before moving to the next element.



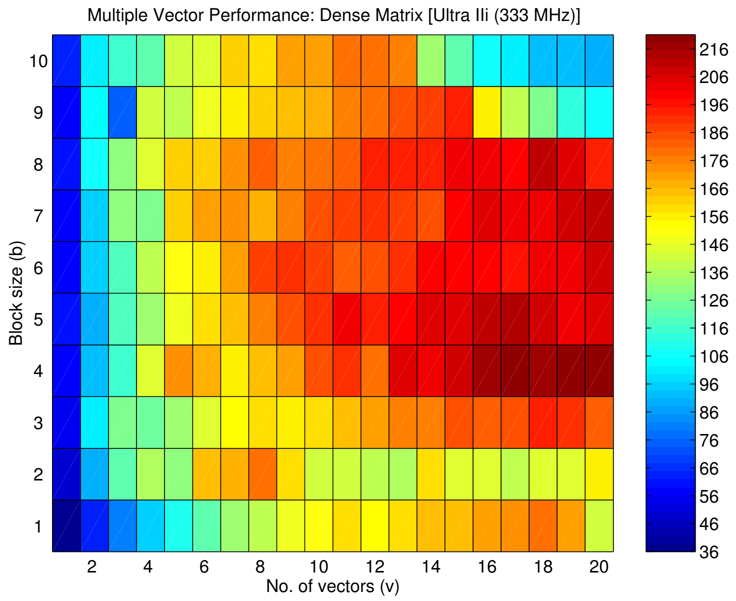


Figure 13: **Register-blocked, multiple vector performance on an UltraSPARC IIi, varying the number of vectors.**

two extremes in the space of matrix structures: a dense $1K \times 1K$ matrix in a sparse format, and a 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 are clustered differently to exactly match the block size. We will also limit our data to square block sizes from 1×1 up to 10×10.

Figure 13 shows the effect of changing the block size and the number of vectors on UltraSPARC IIi. The figure shows the performance of register-blocked code optimized for multiple vectors, with the top figure showing the randomly structured matrix and the bottom figure showing the dense matrix.

Multiple vectors typically pay off for matrices throughout the regularity and density spectrum. We can get some sense of this by looking at the dense and random matrices. For most block sizes, 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. The random matrix tends to have a peak with some relatively small number of vectors (2-5), whereas the dense matrix tends to continue increase in speedup for larger number of vectors.

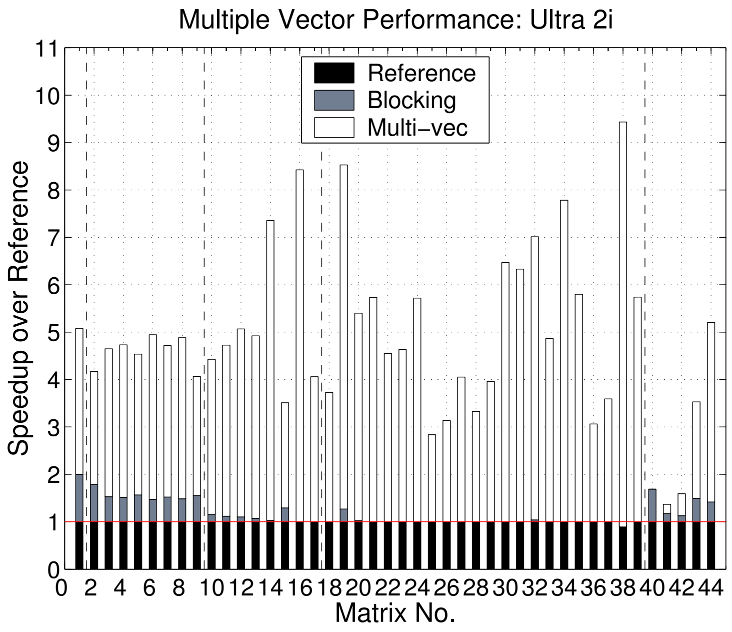
5.3 Performance of Multiple Vector Optimizations

Figures 14 and 15 show the speedup of the multiple vector optimization on the whole matrix set introduced in Table 2. The speedup is computed relative to the performance of naive code without any optimization. We applied the multiple vector optimizations combined either with register blocking or cache blocking for all of the matrices, and show the best speedup in the graph. The number of vectors was fixed at 9 in this experiment.

For reference, Figures 14 and 15 show (1) the speedup due to blocking (either register or cache blocking) alone, and (2) the speedup when blocking and the multiple vector optimization are combined. We see tremendous speedups of up to a factor of 10.5x, with fairly consistent speedups of 2x or more. It is notable that even matrices 20 through 44 speed up. We also tried combining register blocking and cache blocking, but it was not effective for any matrix in the test set.

For matrices 1 to 39, the optimization that exhibits the speedup shown in the graph was register blocking unrolled for multiple vectors, while the optimization for the matrices 40–44 was cache blocking unrolled for multiple vectors. In fact, often for the matrices 20 to 44, the register block sizes are chosen to be 1×1, which means the multiplication code is unrolled for multiple vectors, but not for a particular block size. And still, performance improvement is good.

On the Itanium 2, most of the benefit comes from blocking, while on the other three platforms, most of the benefit comes from having multiple vectors. On the Pentium III and Itanium 2 machines, the benefit tends to decrease as the matrix number increases. The overall benefits are much more uniform across matrices than in a register blocking for a single vector.



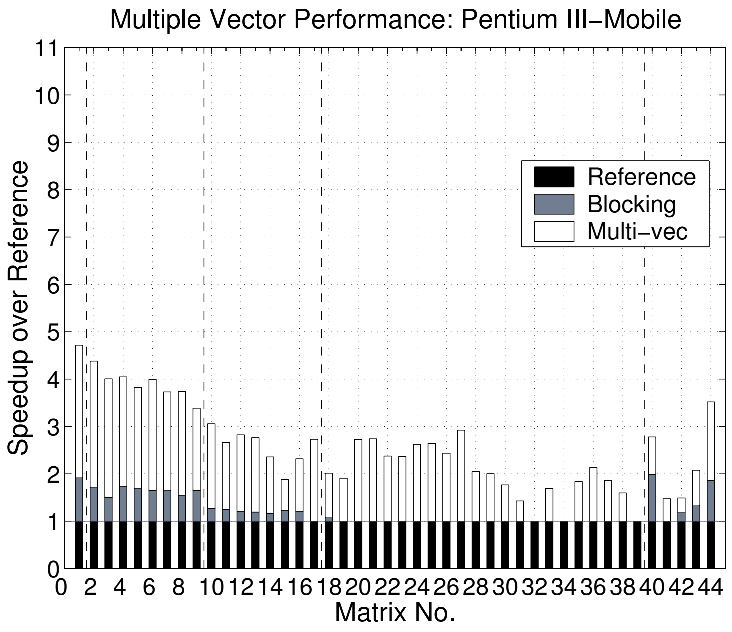
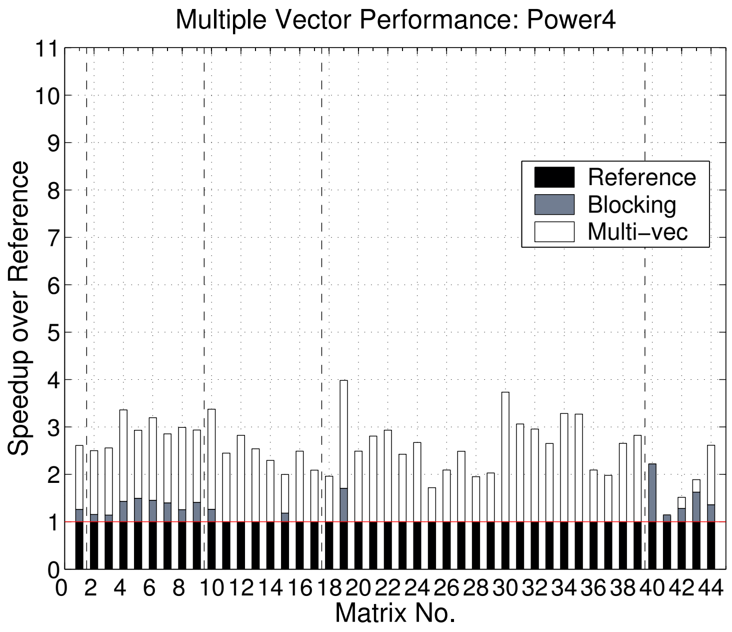


Figure 14: **Speedup of register-blocked multiplication on the Ultra-SPARC IIi (top) and Pentium III (bottom) platforms.** We show the best speedup from blocking only (either register or cache) and from blocking combined with the multiple vector optimization. The number of vectors is fixed at 9. The baseline implementation is an unblocked CSR code which multiplies by each vector separately.



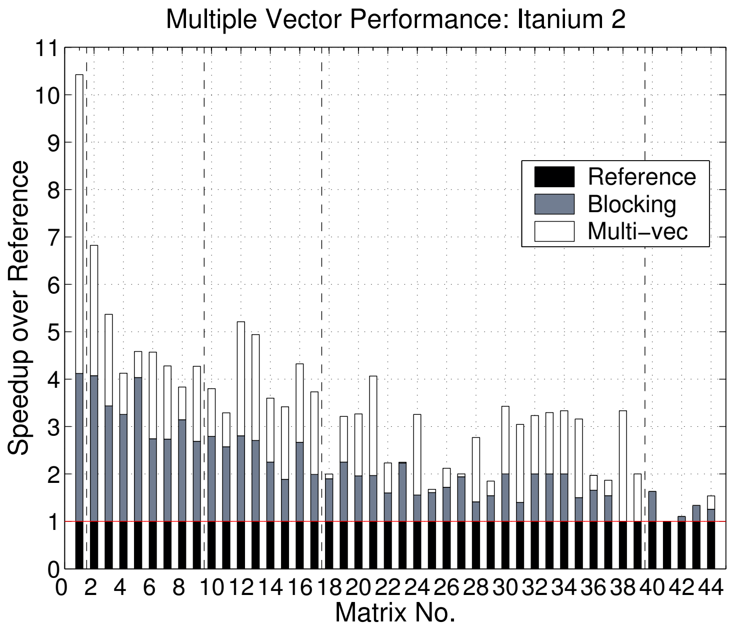


Figure 15: **Speedup of register-blocked multiplication on the Power4 (top) and Itanium 2 (bottom) platforms.** We show the best speedup from blocking only (either register or cache) and from blocking combined with the multiple vector optimization. The number of vectors is fixed at 9. The baseline implementation is an unblocked CSR code which multiplies by each vector separately.

A multiplication code for cache blocked matrices was also unrolled for a given number of vectors. For the same reason matrices 1 to 39 did not speed up by cache blocking, cache blocking for multiple vectors does not make a difference for those matrices. Also, the performance of multiple vector optimization with no register blocking yielded mixed results on matrices 40–44 because, as discussed earlier, the source vectors are so long that elements are rarely in cache. We should note that each of the vectors are stored contiguously in memory, because that seems to reflect the most likely application order; if, instead, the i^{th} elements of all vectors were stored contiguously, the multiple vector optimization by itself would probably be more significant.

6 The Sparsity System

As a result of our study on optimization techniques for sparse matrix-vector multiplication, we demonstrated that register blocking, cache blocking, and use of multiple vectors can significantly improve performance. We showed 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 paper. 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 other optimization tasks, including data structure reorganization, insertion of explicit zeros, and compiler-style loop optimizations. SPARSITY generated C code for portability, allowing the C compiler to perform machine-specific intsruction scheduling and register allocation.

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 Itanium 2. We therefore believe that exhaustive search over some range of register block size would be necessary under search-based register blocking. However, the overhead of running an exhaustive search for every input matrix is very 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 [6, 28]. 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's optimization phase.

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 over those that did not. However, the choice of block size still requires search, so SPARSITY does not employ this heuristic.

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 than 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.) 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

The general structure of the Sparsity system is illustrated in figure 16. 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. 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 vectors is selected.

For the single vector case, the matrix is first 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

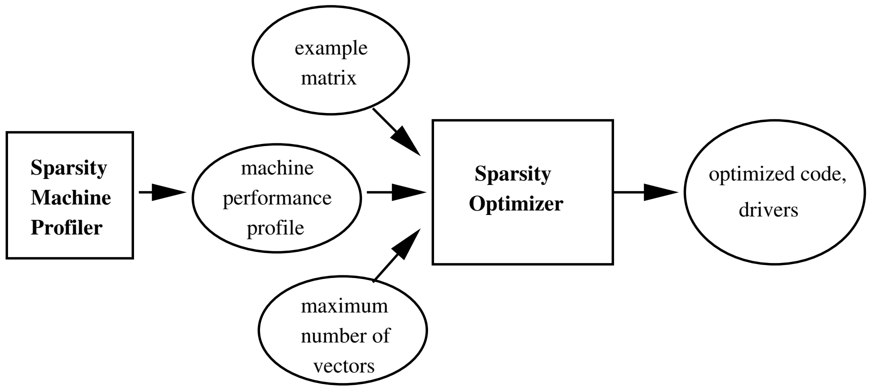

example matrix
Sparsity Machine Profiler
machine performance profile
Sparsity Optimizer
optimized code, drivers
maximum number of vectors

Figure 16: Sparsity **system**

the machine. 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 section 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.

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.

7 Related Work

Sparsity is related to several other projects that automatically tune the performance of algorithmic kernels for specific machines. In the area of sparse matrices, these systems include the *sparse compiler* that takes a dense matrix program as input and generates code for a sparse implementation [5]. As in Sparsity, the matrix is examined during optimization, although the sparse compiler looks for higher level structure, such as bands or symmetry. This type of analysis is orthogonal to ours, and it is likely that the combination would prove useful. The Bernoulli compiler also takes a program written for dense matrices and compiles it for sparse ones, although it does not specialize the code to a particular matrix structure [16]. Finally, Pugh and Shpeisman propose a sparse intermediate program representation (SIPR) for use inside a sparse compiler [23]. They augment their representation with a high-level, machine and matrix independent cost model to make high-level transformation decisions. However, these models are not sufficiently fine-grained to choose block sizes, and therefore complement the transformations and heuristics which we propose.

Toledo [27] demonstrated some of the performance benefits or register blocking, including a scheme that mixed multiple block sizes in a single matrix, but his optimizations were done by hand and there was no general approach to finding a good block size. PETSc (Portable, Extensible Toolkit for Scientific Computation) is a library for Finite Element Methods, which also uses a matrix format with small dense blocks [3], although the block sizes are chosen by the application programmer based on what is natural in the algorithm, rather than optimal for a particular machine. This avoids expensive model evaluation or searching through the parameter space, but as we have shown, selecting based on machine parameters is often important.

Many researchers have explored the benefits of reordering sparse matrices, usually for parallel machines or when the natural ordering of the application has been destroyed [20]. In particular, Pinar and Heath show that reordering based on a heuristic for the Travelling Salesman Problem (TSP) can be combined with register blocking to improve performance on uniprocessors [21]; Heras, *et al.*, propose a similar TSP-based reordering [13]. Reordering could be incorporated into Sparsity, and in prior work we used it to optimize sparse matrix-vector multiplication for shared memory multiprocessors, but we found little benefit on uniprocessors [15]. The difference is the reordering strategy, and both groups found that simple bandwidth reduction orderings are not useful.

Finally, we note that the BLAS Technical Forum has already identified the need for runtime optimization of sparse matrix routines. This is an ongoing effort aimed at expanding the BLAS in a number of ways to reflect developments in hardware, software, and languages, including a specification for BLAS operations on sparse matrices. An early draft of this standard contained a parameter in the matrix creation routine to indicate how frequently matrix-vector multiplication will be performed [7].

8 Conclusions

In this paper, we have described optimization techniques to improve memory efficiency in sparse matrix-vector multiplication for one or more vectors. Our optimizations showed significant payoffs, with up to a $4\times$ improvement for register blocking, $2\times$ for cache blocking, and nearly $10\times$ for register blocking combined with multiple vectors. 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 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. 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. The application of multiple vectors provides large opportunities for performance gains, because it allows for reuse of matrix elements that is not possible with a single vector. We believe this is an important area for further investigation, requiring work on both blocked algorithms and selection of the optimal number of vectors.

Our performance studies showed that all of these optimization have significant performance improvements on some matrices and some machines, but the performance is highly depedent on both. Register optimizations are most effective for sparse matrices arising in scientific simulations, especially Finite Element Methods, whereas cache optimizations are suitable for matrices arising in information retrieval applications. In general, the effects of these optimizations are more pronounced for machines with deeper memory higherachies, and difficulty of selecting optimization parameters increases with hardware complexity. Current trends in hardware indicate that both the penalty for accessing main memory and the complexity of the memory system are likely to increase, making systems like SPARSITY even more important.

Acknowledgement

We would like to thank Osni Marques for providing us a web document matrix, Tuyet-Linh Phan for her help with the data collection, and Jim Demmel for discussions on algorithms that use multiple vectors and other application-level issues.

References

[1] 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.

[2] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted GMRES. Technical Report CU-CS-045-03, University of Colorado, Dept. of Computer Science, January 2003.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.28, Argonne National Laboratory, 2000.

[4] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

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

[6] 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.

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

[8] 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.

[9] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, University of Texas at Austin, November 2002.

[10] 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.

[11] 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.

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

[13] D. B. Heras, V. B. Perez, J. C. C. Dominguez, and F. F. Rivera. Modeling and improving locality for irregular problems: sparse matrix-vector product on cache memories as a case study. In *HPCN Europe*, pages 201–210, 1999.

[14] E.-J. Im. *Optimizing the Performance of Sparse Matrix - Vector Multiplication.* PhD thesis, University of California at Berkeley, May 2000.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing*, 1999.

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

[23] W. Pugh and T. Shpeisman. Generation of efficient code for sparse matrix computations. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, August 1998.

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

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

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

[27] 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.

[28] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project, Sept. 2000. http://math-atlas.sourceforge.net.

	Exhaustive Best			Heuristic			Reference
	$r_\mathrm{o}\times c_\mathrm{o}$	Fill	MFLOPS	$r_\mathrm{h}\times c_\mathrm{h}$	Fill	MFLOPS	MFLOPS
1	8×5	1.00	72.9	8×5	1.00	72.9	36.5
2	8×8	1.00	63.2	8×8	1.00	63.2	35.3
3	6×6	1.12	54.5	6×6	1.12	54.5	35.6
4	6×2	1.13	54.1	3×3	1.06	51.9	34.3
5	4×4	1.00	48.4	4×4	1.00	48.4	30.9
6	3×3	1.00	49.9	3×3	1.00	49.9	33.9
7	3×3	1.00	52.5	3×3	1.00	52.5	34.5
8	6×6	1.15	50.1	6×6	1.15	50.1	33.7
9	3×3	1.02	54.3	3×3	1.02	54.3	35.0
10	2×1	1.10	39.1	2×2	1.21	38.8	33.7
11	2×2	1.23	32.3	2×2	1.23	32.3	28.9
12	2×2	1.24	37.9	2×3	1.36	36.4	33.0
13	2×1	1.14	36.7	2×2	1.28	36.0	33.6
14	2×1	1.17	26.0	1×2	1.15	25.3	24.5
15	2×1	1.00	41.1	2×1	1.00	41.1	31.8
16	2×1	1.17	26.3	1×1	1.72	25.2	25.2
17	1×1	1.00	32.4	1×1	1.00	32.4	32.4
18	2×1	1.36	19.4	1×1	1.00	18.4	18.4
19	2×1	1.01	23.5	2×1	1.01	23.5	18.5

Table 4: **Register blocking performance on the Sun UltraSPARC IIi.**

A Tabulated Register Blocking Data

In Tables 4–7, we show data for the following implementations:

- **Exhaustive best**: Block size, fill overhead, and performance when the block size, $r_\mathrm{o}\times c_\mathrm{o}$, is chosen by exhaustive search.
- **Heuristic**: Block size, fill overhead, and performance when the block size, $r_\mathrm{h}\times c_\mathrm{h}$, is chosen by our heuristic. In addition, if the performance is less than 10% of the exhaustive best performance, we mark the heuristic performance by an asterisk (*).
- **Reference**: Performance of the unblocked (1×1) CSR implementation.

	Exhaustive Best			Heuristic			Reference
	$r_{\mathrm{o}}\times c_{\mathrm{o}}$	Fill	MFLOPS	$r_{\mathrm{h}}\times c_{\mathrm{h}}$	Fill	MFLOPS	MFLOPS
1	3×11	1.00	142.9	3×11	1.00	142.9	74.6
2	2×8	1.00	120.1	8×8	1.00	114.5	67.1
3	6×1	1.10	112.8	3×6	1.12	101.5	67.7
4	3×3	1.10	105.7	3×3	1.10	105.7	60.7
5	4×2	1.00	104.7	2×4	1.00	103.5	60.9
6	3×3	1.03	106.8	3×3	1.03	106.8	64.6
7	3×3	1.03	105.7	3×3	1.03	105.7	64.3
8	6×6	1.25	96.2	3×3	1.15	93.6	60.4
9	3×3	1.05	101.8	3×3	1.05	101.8	61.7
10	2×2	1.23	77.3	2×2	1.23	77.3	60.9
11	2×2	1.23	75.7	2×2	1.23	75.7	60.5
12	2×2	1.24	83.0	2×2	1.24	83.0	68.5
13	3×2	1.40	84.4	2×2	1.28	81.9	68.6
14	2×2	1.33	78.7	2×2	1.33	78.7	67.4
15	2×1	1.00	78.7	2×1	1.00	78.7	63.8
16	3×3	1.69	90.2	4×1	1.43	90.2	75.1
17	1×1	1.00	68.6	1×1	1.59	68.6	68.6
18	2×1	1.36	45.1	2×1	1.36	45.1	42.1
19	2×1	1.01	55.3	2×1	1.01	55.3	55.3

Table 5: **Register blocking performance on the Intel Pentium III.**

	Exhaustive Best			Heuristic			Reference
	$r_{\mathrm{o}} \times c_{\mathrm{o}}$	Fill	MFLOPS	$r_{\mathrm{h}} \times c_{\mathrm{h}}$	Fill	MFLOPS	MFLOPS
1	8×1	1.00	766.4	8×1	1.00	766.4	607.1
2	4×1	1.00	703.2	8×1	1.00	666.0	576.9
3	3×2	1.12	636.0	6×1	1.10	618.7	542.4
4	3×3	1.10	606.7	3×3	1.10	606.7	424.1
5	4×1	1.00	642.9	4×1	1.00	642.9	429.8
6	3×3	1.03	691.9	3×3	1.03	691.9	476.1
7	3×3	1.03	673.3	3×3	1.03	673.3	481.3
8	6×2	1.23	581.0	3×1	1.09	547.4	435.8
9	3×3	1.05	704.6	3×3	1.05	704.6	499.7
10	2×1	1.12	548.7	2×1	1.12	548.7	434.2
11	2×1	1.23	543.6	1×1	1.00	425.9∗	425.9
12	3×1	1.24	597.0	1×1	1.00	587.5	587.5
13	2×1	1.14	597.1	1×1	1.00	553.7	553.7
14	3×1	1.31	754.0	1×1	1.00	580.4∗	580.4
15	2×1	1.00	545.1	2×1	1.00	545.1	460.7
16	2×1	1.17	890.6	1×1	1.00	713.2∗	713.2
17	1×1	1.00	521.2	1×1	1.00	521.2	521.2
18	2×1	1.36	397.3	1×1	1.00	273.5∗	273.5
19	4×1	1.87	591.3	2×1	1.01	504.7∗	295.7

Table 6: **Register blocking performance on the IBM Power4.**

	Exhaustive Best			Heuristic			Reference
	$r_o \times c_o$	Fill	MFLOPS	$r_h \times c_h$	Fill	MFLOPS	MFLOPS
1	4×2	1.00	1219.8	4×2	1.00	1219.8	296.1
2	4×2	1.00	1121.6	4×2	1.00	1121.6	275.3
3	6×1	1.10	945.6	6×1	1.10	945.6	275.2
4	4×2	1.28	806.8	4×2	1.28	806.8	247.8
5	4×2	1.00	1011.5	4×2	1.00	1011.5	250.7
6	4×2	1.50	740.2	3×2	1.16	719.1	262.2
7	4×2	1.49	733.7	3×2	1.16	710.8	259.9
8	6×1	1.22	777.6	6×1	1.22	777.6	247.3
9	6×1	1.38	719.6	3×2	1.17	701.5	260.9
10	4×2	1.50	697.6	4×2	1.50	697.6	249.9
11	4×2	1.70	620.2	4×2	1.70	620.2	240.9
12	4×2	1.48	773.5	4×2	1.48	773.5	275.6
13	4×2	1.54	749.3	4×2	1.54	749.3	276.8
14	4×1	1.49	690.9	3×2	1.47	604.6*	268.7
15	4×1	1.78	513.5	2×1	1.00	490.1	259.8
16	4×1	1.43	769.8	4×2	1.66	769.8	288.7
17	4×1	1.75	536.0	6×1	1.98	536.0	269.4
18	4×1	2.44	323.1	4×2	2.97	323.1	170.1
19	4×1	1.87	566.7	2×1	1.01	425.0*	188.9

Table 7: **Register blocking performance on the Intel Itanium 2.**