Model-Based Memory Hierarchy Optimizations for Sparse Matrices

Eun-Jin Im and Katherine Yelick *
Computer Science Division
University of California, Berkeley †

Abstract

Sparse matrix-vector multiplication is an important computational kernel used in numerical algorithms. It tends to run much more slowly than its dense counterpart, and its performance depends heavily on both the nonzero structure of the sparse matrix and on the machine architecture.

In this paper we address the problem of optimizing sparse matrix-vector multiplication for the memory hierarchies that exist on modern machines and how machine-specific or matrix-specific profiling information can be used to decide which optimizations should be applied and what parameters should be used. We also consider a variation of the problem in which a matrix is multiplied by a set of vectors. Performance is measured on a 167 MHz Ultrasparc I, 200 MHz Pentium Pro, and 450 MHz DEC Alpha 21164. Experiments show these optimization techniques to have significant payoff, although the effectiveness of each depends on the matrix structure and machine.

1 Introduction

Matrix-vector multiplication is an important computational kernel used in scientific computation, signal and image processing, document retrieval, and many other applications. In many cases, the matrices are sparse, meaning that most of elements are zero, and 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 tend to be much lower than their dense matrix counterparts, because the memory access patterns are irregular and the ratio of floating point operations to memory operations is lower than in some dense operations. Like dense matrices, optimizations depend heavily on details of the machine architecture, but unlike dense problems, they also depend on the structure of the matrix, because the distribution of nonzero elements in a sparse matrix determines the memory access pattern.

Our goal is to provide a toolbox to generate highly optimized sparse matrix kernels based on the knowledge of matrix structure and target architecture. In doing so, we make use of profiling information from the machine, data from the matrix structure, (possibly not known until runtime), and performance models to help choose from a wide set of possible optimization techniques. While this framework could be incorporated into a compiler framework with feedback, we do not address issues of dependence analysis or low-level code generation in this paper, but rather the issue of how to modify the data structure representation and to reorganize the computation to optimize for the memory hierarchy of a given machine and sparsity structure of a given matrix.

In this paper, we focus mainly on the problem of multiplying a sparse matrix by a dense vector. This operation is often used in iterative solvers for linear systems, in explicit methods, and in eigenvalue computations, just to name a few. Given a sparse matrix A, dense vectors x and y, the problem is to compute $y = A \times x + y$.

The nonzero structure of A directly determines the memory access pattern of the source vector x and destination vector y, although there is quite a bit of flexibility in reordering the computation to improve locality.

In this paper we present several design trade-offs in the optimization of sparse matrix-vector multiplication. In particular, we describe a technique called register blocking and perform a large set of experiments on the effectiveness of register blocking for various architectures and matrices. One of the challenges of reg-

*The contact addresses of authors are ejim@cs.Berkeley.EDU and yelick@cs.Berkeley.EDU.

†This work was supported in part by the Advanced Research Projects Agency of the Department of Defense (F30602-95-C-0136 and N00600-93C-2481), by the Department of Energy (DE-FG03-94ER25206 and W-7405-ENG-48), by the Army Research Office (DAAH04-96-1-0079 and DAA655-98-1-0153), and by the National Science Foundation (CDA-9401156). In addition, the equipment for this research was provided in part by Sun Microsystems, Intel, IBM, and NERSC/LBNL and Hewlett-Packard.

ister blocking for sparse matrices is the determination of block size; we introduce a new performance model for estimating the optimal block size, and evaluate it on a set of benchmark matrices. We also define and measure the overheads associated with register blocking and discuss the effectiveness of these techniques in a real compiler and application setting.

The remainder of this paper consists of the following sections. In section 2, the related work on matrix computation is reviewed. Register blocking is introduced in section 3, and in section 4, we describe how to model the performance to select a block size with better performance. Section 5 presents the performance of register blocked multiplication on various platforms and section 6 discusses the overhead of register blocking. The idea of register blocking is extended to the multiplication of a sparse matrix and a set of vectors in section 7. Other memory hierarchy optimizations are discussed with some preliminary results in section 8 and use of performance models in compilation is discussed in section 9. In section 10, we conclude and briefly overview some future work.

2 Related Work

The state of the art in optimizing matrix computation can be seen from two different points of views. One is from the compiler writer's perspective, where the focus is on how to automatically generate memory efficient code. The other is from the library developer's perspective, in which several optimized versions are provided and the user selects the routine to use.

For dense matrix operations, Wolf [Wol92] and Carr [Car94] formulated loop transformation theory for array-based loop nests; it was used to transform loops into blocked loops. The problem of determining the size of a cache block so that cache conflicts between different array elements is avoided is not easy even for dense matrices and Lam [MSLW91], Coleman [CM95], and Essenghir [Ess93] gave algorithms to determine blocking sizes using only cache characteristics and matrix size. PHiPAC [BAD+97] approaches this problem in a different way in that blocked code is generated on a target machine using a parameterized code generator and the block size is chosen by measuring the performance of those generated codes with different parameters.

For sparse matrices, Bik [Bik96] and Kotlyar's [KPS97] work generates a sparse code from dense code with the compiler, in order to relieve the programmer's effort in writing complex sparse code and to increase the flexibility of the code for various storage formats of sparse matrices. As a result of effort to provide a generic sparse matrix operation library, NIST sparseBLAS [PR97] provides generic routines and TNT (Template Numerical Toolkit) [Poz97] provides a generic matrix/vector classes. BlockSolve [PJ95] and Aztec [HST95] are parallel iterative solvers that include the implementation of optimized sparse matrix operations. Our research lies between the library and compiler approaches for sparse operations: we are building a system to automatically produce highly tuned libraries, using compilation techniques as well information about the input matrices to select optimizations; previous libraries are hand-tuned to a specific machine and sometimes a specific matrix structure; previous compilers automatically transforms the code, but they do not have ways of selecting the right set of transformation based on machine and matrix parameters.

3 Register Blocking

To optimize for register use at the highest level of the memory hierarchy, we reorganize the data structure and computation to improve the reuse of values. This optimization is commonly known as blocking or tiling for dense matrix operations [MSLW91, CM95].

Register blocking in sparse matrix codes requires information that is not available at compile time for several reasons. First, the nonzero structure of the sparse matrix is not known statically, so runtime-optimization techniques are needed to determine which transformations are legal [DHU+93]. Second, the problem of tile-size selection requires information about the machine characteristics which may not be captured in a simple model with the number of registers and memory costs. Determining block size even for dense codes requires feedback from benchmarking, because register level blocking interacts with instruction scheduling, especially the use of multiple functional units, prefetching and write buffers [BAD+97, WD]. Third, for sparse matrices, there is an additional problem of costs that depend on the sparsity structure of the matrix. Here, we focus on the latter two problems of selecting the block size for the specific problem of sparse matrix vector multiplication.

The storage format of sparse matrices varies across application domains and programming languages. One of the more common is the compressed sparse row format, shown in figure 1. A straightforward sparse matrix vector multiplication program for this format is shown in figure 2.

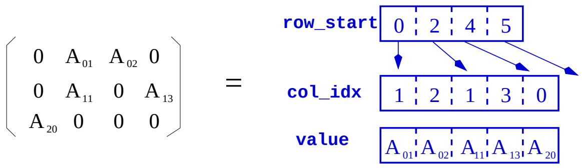


Figure 1: **Compressed sparse row format storage scheme of sparse matrix.** Each nonzero value and its column index are stored in *value* and *col_idx* arrays, respectively, row by row. The elements of *row_start* array point the starting indices of each rows into *value* and *col_idx* arrays.

This program is unlikely to exploit temporal locality of elements of vector x by saving them in registers across rows, in part because the compiler cannot determine the memory accesses 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 block contiguously, values in the source and destination vectors may be reused. The number of registers need to compute an $r \times c$ block of A is 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 block of A.

The code is not only easier to write, but admits more compile-time optimizations such as loop unrolling and software pipelining of the $r \times c$ computation if the values of r and c are fixed over the entire matrix. Figure 3 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 $r \times c$. Note, however, that there are some zeros stored in the blocked representation, since

```
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: **Basic sparse matrix-vector multiplication code**

$$A = \begin{pmatrix} 11 & 12 & 0 & 0 & 15 & 16 \\ 21 & 22 & 0 & 0 & 25 & 26 \\ 0 & 0 & 33 & 0 & 35 & 36 \\ 0 & 0 & 43 & 44 & 45 & 46 \end{pmatrix}$$

$row\_start = \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}$
$col\_idx = \begin{pmatrix} 0 & 4 & 2 & 4 \end{pmatrix}$
$value = (11\ 12\ 21\ 22\ 15\ 16\ 25\ 26\ 33\ 0\ 43\ 44\ 35\ 36\ 45\ 46)$

Figure 3: **Constant block compressed sparse row storage format.** The elements of each dense $r \times c$ 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 block row starting positions in the *col_idx* array.

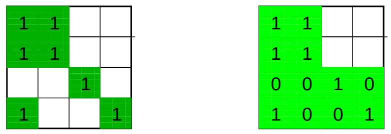


Figure 4: **Fill overhead of register blocking.** The BSR format in the right figure stores extra 5 zero values to make 2×2 block dense.

the original matrix did not have dense 2×2 blocks throughout.

4 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, zero elements are often filled in to make a complete $r \times c$ block as shown in figure 4. These extra zero values not only consume storage, but will increase the number of floating point operations since they are involved in the sparse matrix computation. (Placing branches in the code to avoid these extra operations proves to be worse than the computing them.)

We use a simple performance model to determine the block size for a given matrix and machines, 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.

The first component cannot be determined exactly 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 can be computed exactly, but is very expensive, given that we need to compute this for a range of block sizes under consideration. Multiple sweeps over the array or complex data structures must be made to compute the number of zero fills for a set of block sizes. Instead, we developed an approximation that can be done in a single pass over only a subset of the matrix.

4.1 Approximating the Performance of a Blocked Sparse Matrix

Figure 5 shows the performance of sparse matrix vector multiplication for a dense matrix using register blocked sparse format; i.e., a dense matrix in sparse format. 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). This graph is platform dependent, and is shown here for a 167 MHz Ultrasparc I. 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.

4.2 Approximating Fill Overhead

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*:

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

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

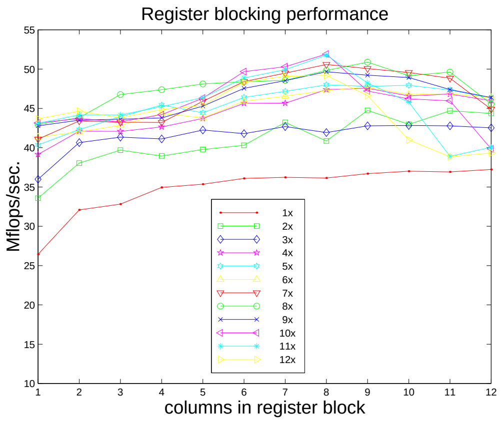


Figure 5: **Performance of register blocked multiplication** taken from 167 MHz Ultrasparc for 1000×1000 dense matrix represented in sparse format. Each line is for a fixed number of rows, varying the number of columns from 1 to 12.

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 at runtime. To approximate fill overhead, we separately compute a column blocking factor and row blocking factor by sampling a fraction of the matrix entries. First, we determine the column blocking factor by examining every one in k rows of the matrix. (In our implementation, k is chosen to be 100, which gives reasonably accurate fill overheads.) By dividing the column index of nonzero elements in the row with the block size in consideration, we compute a block column index for each element in the matrix. This gives us the number of total 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 algorithms, 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.

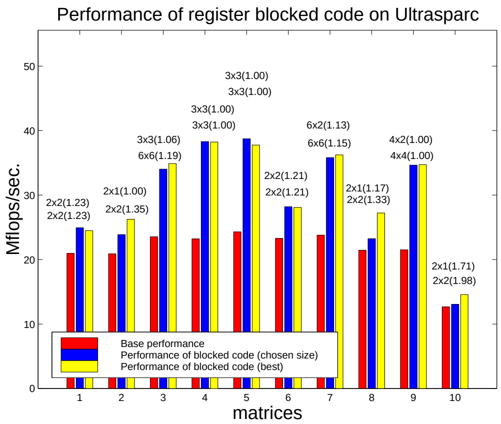
4.3 Instantiating the Model

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 we are measuring performance in Mflop/s, we calculate the real performance by including only those floating point operations that would have been needed in the unblocked code. Using our estimates, we predict this performance as:

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

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

We consider column blocking factors c in the range from 1 to c_{max}, where c_{max} is the column size with the maximum performance for a dense matrix in sparse blocked format. Within this range, we choose c by maximizing the predicted performance. We independently do the same to choose the row size r.



matrix	size(N)	nonzeros	Application area
1	23560	484K	Airfoil eigenvalue calculation
2	41092	1.7M	2D PDE problem
3	30237	1.5M	Automobile frame stiffness matrix
4	13965	1.0M	FEM stiffness matrix
5	24696	1.8M	FEM stiffness matrix
6	52329	2.7M	Engine block stiffness matrix
7	54870	2.7M	Structure from shuttle rocket booster
8	4134	94K	Chemical process separation
9	62424	1.7M	Unstructured Euler solver
10	26068	177K	Device simulation

Figure 6: **Performance of register blocked multiplication on 10 sparse matrices from real applications taken on a 167 MHz Ultrasparc.** The first bar is the base performance the second bar is the performance of the register blocked matrix for a block size chosen by our model, and the third bar is the best performance over all the block sizes considered. The numbers on the top of the bars are block size and fill overhead in the parenthesis for the second bars (bottom) and the third bars (top). The table shows the matrix dimension, number of nonzeros, and the application area of the matrix.

5 Performance

Figure 6 shows the performance of register blocked multiplication performance on sparse matrices that arise in numerical simulations. It compares the base performance without register blocking to the performance of register blocked multiplication chosen with the above heuristic and the best performance among all the register block sizes. In the figure, the block sizes are shown above the bars, with the block size from the heuristic on the bottom and the optimal block size on the top. In computing the performance in Mflop/s of the blocked code, we use the floating point operation count from the unblocked code – the machine executes at a faster rate than indicated, but the reported numbers reflects the improvement in running time that a user would see. Figure 7 shows the same performance data for two other machines, a 200 MHz Pentium Pro and a 450MHz DEC Alpha 21164. (Some bars are missing in the Alpha figure, because our platform did not have sufficient memory.)

On all of the platforms, register blocking shows a significant performance payoff. Our heuristic for choosing good block size does well on the Ultrasparc and Alpha, typically getting performance close to that of the optimal block size, even when it did not choose that block size. On the Pentium, although register blocking shows significant payoff, our heuristic does not do quite well. The reason is that our heuristic computes r and c independently, assuming square block sizes for each computation; it therefore favors larger block sizes. On the Pentium, with only 8 floating point registers, some of the small non-square block sizes such as 2×1 and 3×1 perform significantly better than the square sizes.

Note that the benchmark matrices are representative of numerical simulations, where some natural locality in the physical domains leads to patterns of dense sub-blocks in the matrix. Problems from linear programming and document retrieval, for example, have no naturally occurring sub-blocks, and register blocking them did not improve their performance.

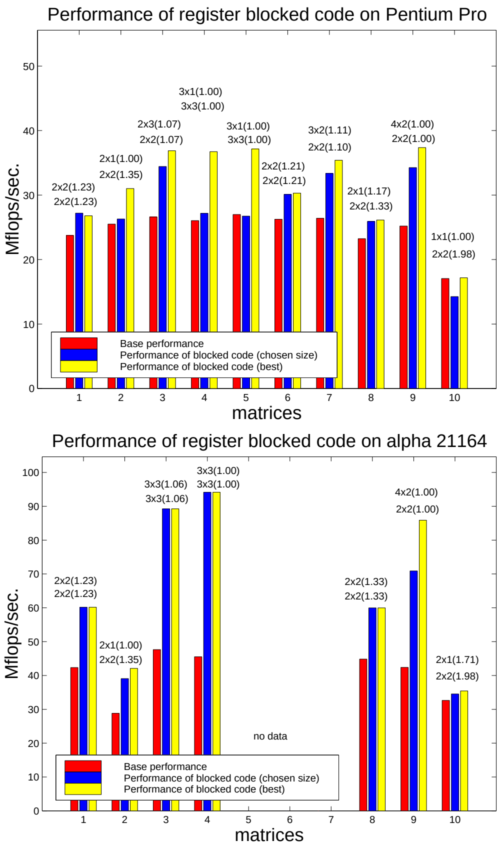


Figure 7: **Performance of register blocked sparse matrix-vector multiplication.** The top figure shows performance on 200 MHz Pentium Pro and the bottom figure shows performance on 450 MHz DEC Alpha 21164. Performances of matrix 5,6, and 7 were not measured on Alpha due to insufficient memory on our system.

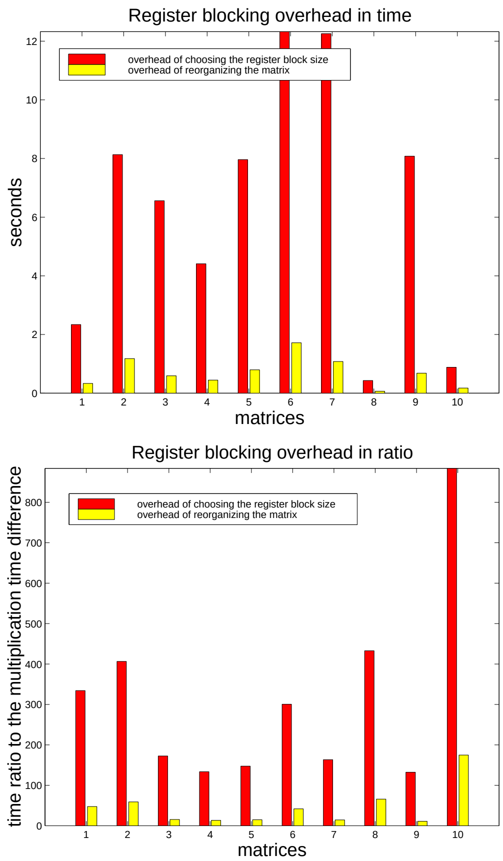


Figure 8: **Pre-computation overhead of register blocked multiplication on 10 sparse matrices taken on a 167 MHz Ultrasparc.** The top figure shows the time to perform each of the two pieces, determining the block size and reorganizing the matrix. The bottom figure shows the same values divided by the 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.

6 Precomputation Overheads of Register Blocking

There is precomputation overhead of applying register blocking that is separate from the overhead that appears during matrix-vector multiplication. The first of these is the price of determining the block size using some 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 to reorganize the matrix in the blocked format. Both of these are paid only once, while computations like matrix-vector multiplication may be performed many times on the same sparse matrix structure.

Figure 8 shows the amount of these two precomputation overheads, both in absolute time and in the ratio of overhead to savings from the optimization. If the block size selection and reorganization will be done at runtime, the bottom figure tells us the number of sparse matrix-vector multiplications that must be done before the optimization pays off. While these numbers are high, 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 structure (e.g., from the same problem domain) and the same number of degrees of freedom. We envision a system for generating optimized sparse matrix kernels in which there is a dialog between the application builder and the code generator.

7 Multiple Vectors

The performance of matrix-vector multiplication is inherently lower than that of matrix-matrix multiplication because there is no reuse of element of matrix A in matrix-vector multiplication. Algorithms such as block Lanczos [GU77], which compute a set of eigenvalues and associated eigenvectors, require multiplication of matrix to a set of vectors. Since there is potential for much higher performance, we extend the idea of register blocking to multiplication of sparse matrix and a set of vectors.

Figure 9 shows the performance of register blocked multiplication for a varying number of vectors from 1 to 30. The matrix used is the 5th FEM matrix in figure 6 for which the best block size was 3×3. The lines in figure 9 show the performance of the 1×1(unblocked), 1×3, 3×1 and 3×3 blocked codes for the multi-vector case. Each point was measured using different code which is automatically generated by an optimizing source code generator and then fed through a C compiler. The main optimization of the code generator is to unroll the loops in order to eliminate loop overheads and to increase the number of operations to be scheduled. The codes are compiled with *-xO2* options (performance shown in the top of figure 9) and with *-xO5* option (performance shown in the bottom of figure 9) in Sun Workshop C-compiler 4.2. The main difference between compiler optimization *-xO2* and *-xO5* is that the latter performs loop unrolling. This is in addition to the loop unrolling done by our C code generator.

In the top figure (compiled with option *-xO2*), we can see the performance improvement is quite large (from 28 Mflops/s up to 68 Mflops/s for 3×3 block) and the performance is best with 3×3, and then 3×1, 1×3, 1×1 in sequence for a fixed number of vectors. The number of floating point registers needed is modeled as $k \times r + r \times c + c$ where $r \times c$ is size of block and k is the number of vectors. In the bottom figure (compiled with option *-xO5*), the performance is even

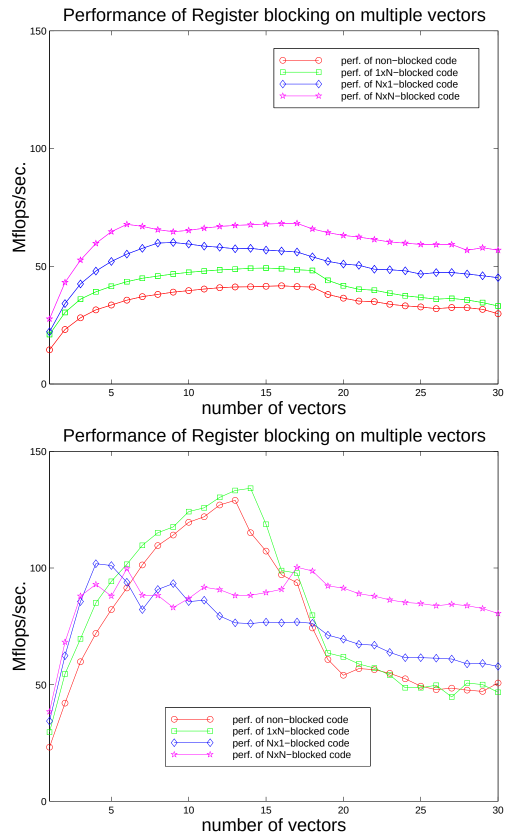


Figure 9: **Performance of register blocked sparse matrix-multiple vector multiplication** The multiplication codes for 1×1, 1×3, 3×1 and 3×3- blocked for different number of vectors are extensively generated by loop-unrolled multiplication source code generator and compiled with different compiler options (*-xO2* for top figure, and *-xO5* for bottom figure). The performance is shown in 4 lines in each figure.

better, but we observe that 1×3 and 1×1 block outperforms 3×1 and 3×3 blocks for 5-16 vectors. For a small number of vectors, the code is loop-unrolled further by the compiler with this compiler option *-xO5*, incurring register spills due to the demand for more registers. For 5 to 16 vectors, analysis of generated assembly code shows the ratio of the number of register spills (extra store operations) to the number of floating point operations (which is fixed for a matrix) is much smaller for 1×3 and 1×1 blocks. The ratios are 0–0.12 for 1×1, 0–0.02 for 1×3, 0.22–1.29 for 3×1, and 0.24–0.60 for 3×3 block. It was impossible to control the loop unrolling factor to keep it same across different block sizes and different number of vectors.

From these experiments, we can conclude that memory hierarchy optimizations for a set of vectors has even higher performance gains than the single vector case. However, our simple model would choose the 3×3 block size, which is not optimal for when the highest level of compiler optimizations are used. The graphs and our analysis of register spilling overhead indicate that the lower performance of the 3×3 is an anomaly due to lack of control over the final compilation phase. One of three approaches can be used to address this problem. First, we could adjust the model to more accurately reflect what the compiler is doing to the performance of the generated code; that would lead to a very complex model, and it is not clear we could capture the features of a large set of compilers in any reasonable model. Second, the compiler itself might do a better job of loop unrolling and register allocation, to avoid generating unnecessary spill code; a step in this direction would be better control over loop unrolling in the optimizer. Finally, we could integrate our code generation into the compiler to help control the optimization phases done by the compiler.

8 Other Optimizations

To address other levels of machine memory hierarchies below the registers, we have considered two other optimizations, cache blocking and matrix reordering. Matrix reordering is commonly used to reduce the number of computed nonzeros in direct solvers and to reduce communication costs. In our experience, reordering showed some benefit when running on a shared memory multiprocessor, but none for uniprocessors. Cache optimizations, however, we found to be extremely beneficial on a particular class of matrices, namely those with nearly random sparsity structure.

As an extension of the register blocking idea, we consider an optimization called *cache blocking*. The idea of this optimization is to keep c_{cache} elements of vector x in the cache while an $r_{cache} \times c_{cache}$ block of matrix A is multiplied to this portion of vector x. That is, we limit the vector products so that the elements of vector x can all be kept in cache and re-used for the vector product for the next row. A matrix with equal size cache blocks identified is illustrated in figure 10.

Unlike register blocking, creating dense $r_{cache} \times c_{cache}$ blocks by filling in zeros is not practical. Because the cache size is relatively large, expanding $r_{cache} \times c_{cache}$ sparse matrices to dense matrices will incur excessive storage and computation overhead. For that reason, the blocks in the cache blocked matrix are stored as sparse matrices in the implementation of *static cache blocking*. The sparse matrix is reorganized by changing the order of the nonzero elements of sparse matrix in the storage as shown in figure 11. We have also considered a variant of cache blocking which we call *dynamic cache blocking* in which the representation is left unchanged, but a set of r_{cache} pointers are used to keep track of blocks dynamically. However, the additional pointer manipulation and control required for dynamic cache blocking made it less interesting than the static case.

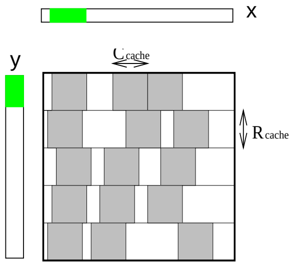


Figure 10: **Alignment of cache-blocks in sparse matrix** The grey areas are sparse matrix blocks that contain nonzero elements in the $c_{cache} \times r_{cache}$ rectangle. The white areas are the areas that contain no nonzero elements.

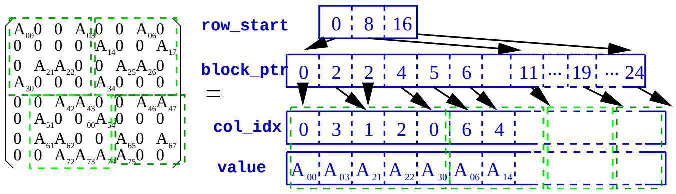


Figure 11: **Storage format of cache-blocked sparse matrix** The nonzero elements of $r_{cache} \times c_{cache}$ blocks are stored in contiguous locations. The cache blocks consist of *block_ptr*, *col_idx* and *value* arrays. The *block_ptr* array keeps track of starting points of each of r_{cache} rows, as *row_start* array did in figure 1. The *row_start* index array stores the indices into *block_ptr* array for every r_{cache}-th row.

The performance of cache blocked matrix-vector multiplication is measured for various matrices on 167 MHz Ultrasparc I, which has 512K bytes of off-chip L2 cache. The block size was chosen empirically as $16K \times 16K$. Unfortunately, cache blocking shows a slight degradation in performance for all of the matrices from numerical simulations on which register blocking has proven useful, but on one matrix, it showed an enormous payoff.

For a matrix that arises in a document retrieval algorithm called Latent Semantic Indexing (LSI), cache blocking dramatically improves performance. The unblocked code runs at 5.8 Mflops/s on an Ultrasparc, while the cache blocked code runs at 18 Mflops/s, giv-

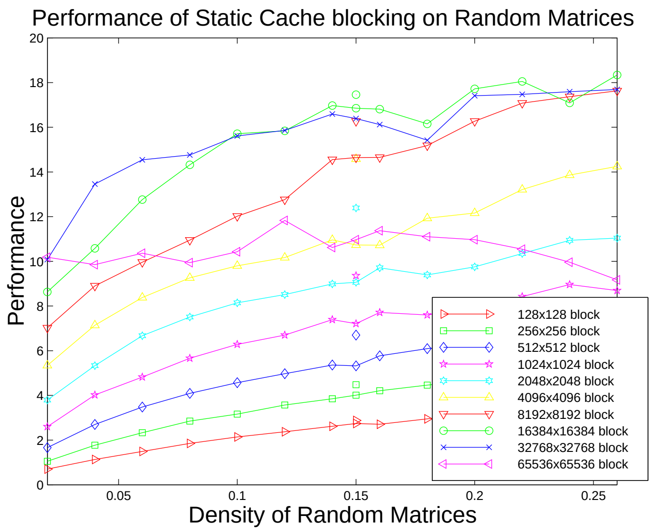


Figure 12: **Performance of cache-blocked multiplication on random matrices** measured on $64K \times 64K$ random matrices with different densities 0.02–0.26%. Each line represents different cache block sizes, and the separate points at 0.15% are the performance of LSI matrix.

ing a speedup of 3.1. It is interesting to note that register blocking for the LSI matrix showed no benefit.

The nonzero pattern of the LSI matrix is unusual compared to most scientific applications, in that it has little discernible structure.[1] Combined with the fact that the size of the matrix is very large, the performance of multiplication on LSI matrix before the optimization is very low (5.8 Mflops/sec.) relative to the other matrices (10–25 Mflops/sec.).

As further evidence that cache blocking is effective on matrices with evenly distributed nonzeros, we also measured the performance on random matrices. The results are shown in figure 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 very similar to those of a random matrix.

[1] Sparse matrix-vector multiplication is the kernel of the LSI algorithm. Our matrix came from NERSC/LBNL in collaboration with the Inktomi company, and is a subset of real data from the web. They use an algorithm different from LSI, which is also based on sparse matrix-vector multiplication.

9 Models Used in Compilation

The register blocking optimization uses profiling information specific to the target machine, namely the performance of a dense code in sparse format for various block sizes. Note that even for this fixed problem, the choice of optimal block size is not a simple function of the number of registers available on the machine. Similarly, although our current implementation of cache blocking does not make use of a performance model for determining block size, we showed that performance of a random matrix on the target machine could be the basis for a reasonable model. More importantly, information about matrix structure is needed to determine which optimizations are likely to be effective, and given that choice, how to select parameters like register block size based on the fill overhead. In a full compilation environment, we imagine this input-specific information to come from feedback from previous executions of the program. Although the actual matrix might change from one run to another, the characteristics, such as whether it contains dense sub-blocks or other regular patterns, are likely to persist over multiple runs. Note that decisions about register and cache block size are not only used to reorganize the matrix representation, but also to select the code from a set of automatically-generated, loop-unrolled codes for the various block sizes.

10 Conclusions

In this paper, we explored memory hierarchy optimizations for sparse matrix vector multiplication, primarily for register level optimizations. We demonstrated the need for profiling information based on the target machine and runtime information in the form of the nonzero structure of the matrix. We introduced a performance model for choosing register block size, and examined the performance advantages of blocking and the overheads associated with changing an unblocked representation to a blocked one. We also extended the register blocking idea to multiplication by a set of vectors and gave some preliminary performance results on cache-level blocking.

The conclusion is that register blocking and cache blocking can significantly improve the performance of sparse matrix-vector multiplication. There are different domains in which each technique is effective. Register blocking works well for matrices from numerical simulations that have some natural clustering of nonzeros, sometimes in the form of dense sub-blocks throughout the matrix. For matrices with less structure, in particular nearly random patterns of nonzeros, register blocking proves to be ineffective, but cache blocking showed extremely favorable results; for a matrix use in document retrieval, performance improved by a factor of 3 using cache blocking.

Because these optimizations depend on details of the memory hierarchy that are not captured by a simple set of parameters, profiling information was used to build a performance model that was specific to a machine and to the nonzero structure of a matrix. This model is used to select register block size, which in turn determines the optimized code that is generated for a given matrix/machine pair. We showed that our block size selection technique usually gave performance close to that of the optimal block size.

We plan to use these results in building a system to automatically generate highly tuned sparse matrix kernels. At the moment, some codes are automatically generated and the rest are written by hand. We envision a dialog between application developer and the code generation system to collect information about sparsity structure and the performance of certain operations (such as multiplication using a dense matrix in sparse format) for the machine of interest.

Generating optimized sparse matrix kernels is very difficult, especially given the importance of sparsity structure and the difficulty of predicting performance on modern machines. We plan to develop additional models for memory hierarchy optimizations on sparse matrix kernels, both to determine parameters of individual optimizations and to help select the best set of transformation from a larger set of possibilities. One of the keys to automatic optimization of sparse matrix codes is the development of performance models that can be evaluated quickly and are accurate enough to select good code transformations.

References

[ABB+95] 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*. SIAM, February 1995.

[BAD+97] Jeff A Bilmes, Krste Asanovic, Jim Demmel, CheeWhye Chin, and Dominic Lam. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, July 1997.

[Bik96] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden University, 1996.

[Car94] Steve Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, July 1994.

[CM95] 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.

[DCHH88] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14:1–17, March 1988.

[DHU+93] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 45–56. IEEE Computer Society Press, October 1993.

[Ess93] Karim Essenghir. Improving data locality for caches. Master's thesis, Rice University, September 1993.

[GU77] 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.

[HST95] 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.

[KPS97] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel code for sparse matrix applications. In *Supercomputing*, 1997.

[MSLW91] E. E. Rothberg M. S. Lam and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[PJ95] Paul 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.

[Poz97] Roldan Pozo. Template numerical toolkit (TNT), 1997. http://math.nist.gov/tnt.

[PR97] Roldan Pozo and Karin Remington. NIST Sparse BLAS, 1997. http://math.nist.gov/spblas.

[WD] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software (ATLAS). http://www.netlib.org/atlas.

[Wol92] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Computer Systems Laboratory, Stanford University, August 1992.