Optimization of Sparse Matrix Kernels for Data Mining

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

ejim@cs.Berkeley.EDU, yelick@cs.Berkeley.EDU
Tel: (510)642-1266
Fax: (510)642-5775

Abstract Number: 193

Optimization of Sparse Matrix Kernels for Data Mining

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

Abstract

Many data mining algorithms rely on eigenvalue computations or iterative linear solvers in which the running time is dominated by sparse matrix-vector products. Sparse matrix-vector multiplication on modern machines often runs one to two orders of magnitude slower than peak hardware performance, and because of their lack of structure, the worst performance is often observed for matrices from text retrieval and other data mining applications. In this paper we explore a set of memory hierarchy optimizations for sparse matrix-vector multiplication, concentrating on matrices that arises in text and image retrieval. We also consider algorithms that multiply the sparse matrix by a set of vectors, and show that reorganizing the code to take advantage of multiple vectors can significantly speed up the running time. These optimization are supported by a code generation and optimization system called SPARSITY, which automatically tunes sparse matrix-vector multiplication for a given matrix structure and machine.

1 Introduction

Many data mining algorithms rely on eigenvalue computations or iterative linear solvers in which the running time is dominated by sparse matrix-vector products. Sparse matrix operations are slower than their dense matrix counterparts due to irregular memory access patterns and to indirection overhead in the sparse data structure. As a result, sparse matrix-vector multiplication often runs one to two orders of magnitude below the peak hardware performance. Sparse matrix performance strongly depends on the nonzero structure of the matrix, with the worst performance often seen from data mining matrices. We expect this gap between hardware peak and sparse matrix performance to worsen as the relative speed of processors and memory continues to diverge and the size of data sets to be mined increases.

We have developed a system called SPARSITY to automatically generate an optimized sparse matrix-vector multiplication routine for a given matrix structure and machine [Im00]. The optimization include register level blocking, cache blocking, and blocking across multiple vectors when they exist in the higher level algorithm. The absolute performance as well as the relative speedup each optimization is highly dependent on the matrix structure, which in turn depends on the application domain. Matrices from physical simulation often contain small dense subblocks or dense bands that can be used to improve performance. For these matrices, we have shown that register blocking is the most effective optimization method. In contrast, the matrices that arise in data mining applications are quite irregular, and are instead closer to a random pattern.

In this paper we explore memory hierarchy optimizations for sparse matrix-vector multiplication on data mining matrices. We will concentrate on matrices that have been used in text retrieval and face recognition for algorithms like Latent Semantic Indexing [BDO95], Concept Decomposition [DM99], and Eigenface Approximation [Li98, Li99]. In particular, we will show how cache level blocking can be used to significantly improve the performance of matrices from web document retrieval and face recognition, while register level blocking, which is highly effective in matrices from physical modeling, proves to be relatively ineffective. Some of these algorithms can be organized to multiply the sparse matrix by a set of vectors, and we will show that reorganizing the code to take advantage of multiple vectors can also significantly improve the running time. These optimization are supported by Sparsity, which performs automatic optimization selection and code generation based on a given matrix structure and machine.

The rest of this paper is organized as follows. In section 2, we show three examples of data mining applications that use sparse matrix-vector multiplication and we describe two of Sparsity's memory hierarchy optimizations techniques in section 3. These optimizations were selected because they prove most effective on data mining matrices. Then we show the performance improvement on those applications on the UltraSPARC II, Alpha 21164, MIPS R10000 and Pentium III in section 4 and conclude in section 5.

Collection	Applied Algorithm	Dimension	Nonzeros	Density	Avg. # of NZs per row
Web Document	LSI	10000x255943	3.7M	0.15 %	371
NSF Abstracts	CD	94481x 6366	7.0M	1.16 %	74
Face Images	EA	36000x 2640	5.6M	5.86 %	155

Figure 1: **Sparse Matrices in Data Mining Applications**

2 Data Mining Algorithms Using Sparse Matrix Computations

We first introduce three examples of data mining algorithms in which sparse matrix-vector multiplication is used: Latent Semantic Indexing, Concept Decomposition and Eigenface Approximation. The first two are used for text clustering and retrieval and the third is used for face recognition in images. In section 4, we use three matrices, one associated with each of the three algorithms, for measuring the effectiveness of our optimization techniques; the characteristics of those matrices are summarized in figure 1. It should be noted that these algorithms and matrices are chosen to be representative of the kinds of problems that use sparse matrix-vector multiplication for data mining of text and images. Many other algorithms and data sets exist for data mining, and it is not our intent to compare the algorithms for quality in this paper.

2.1 Text Retrieval

Document retrieved may be done by literally matching terms in documents with those of a query. This direct matching may be inaccurate, though, since there are usually many ways to express a given concept that the literal terms in a user's query may not match those of a relevant document. Two examples of algorithms that address this problems are Latent Semantic Indexing [BDO95] and concept decomposition. They address the problems of lexical matching by using statistically derived conceptual indices instead of individual words for retrieval.

In LSI, a term-by-document matrix is projected to a smaller dimensional space using by computing the truncated singular value decomposition SVD [GVL96] of the matrix, and retrieval is performed by projecting the query onto the same space. According to the literature [BDO95], the bulk of LSI processing time is spent in computing the truncated SVD of the sparse term-by-document matrix, in which the kernel is a sparse matrix-vector multiplication. In a blocked version of the SVD computation, a sparse matrix is multiplied by a number of vectors [BCD$^+$00].

Dhillon and Modka introduced a related idea of *concept decomposition* for document clustering [DM99]. Concept decomposition is a matrix approximation scheme that solves a least-squares problem. It is comparable to LSI, but has advantages over the truncated SVD in both memory and time. To calculate concept decomposition, sparse matrix-vector multiplication is also used, and in this case the multiplication is often performed on multiple vectors in the range of 5 to 100.

We have collected two term-by-document matrices for demonstration of our optimization techniques. The first is a was collected from documents on the web by Inktomi, Inc. The entire matrix (collected a few years ago) is $100K \times 2560K$ which does not fit in memory of a single processor. Since our interest in this paper is on uniprocessor performance, we use the first 10% of the rows and 10% of the columns and refer to this as a web document matrix. It is a $10K \times 256K$ sparse matrix with 3,712,489 nonzeros. The second matrix contains NSF award abstracts since 1990. This is an extended collection of the NSF matrix used in [DM99] and is a 94481×6366 sparse matrix with 6,979,420 nonzeros.

2.2 Eigenface Approximation (EA)

Eigenvector analysis is widely used for image processing, pattern matching and machine vision. When it is used for face recognition, the algorithm is referred as an "eigenface" computation. Li [Li98, Li99] proposed a multi-resolution algorithm for calculating primary eigenvectors of a large set of high resolution images. The algorithm systematically coarsens images to create a multi-resolution hierarchy of the image set and computes co-eigenvectors for the coarsest images; it works its way up, and finally recovers primary eigenvectors for the original images from their approximate co-eigenvectors. The algorithm gains substantial speedups over the more common SVD approach. The original matrix is a dense pixel-by-image matrix, in which each image is linearized to form a column of the matrix. However, the algorithm is combined with wavelet compression techniques to further speedup the eigenvector computation, and in the process, some values in the matrix that are below a given threshold are discarded. The resulting matrix is sparse, and the algorithm multiplies this sparse matrix times a set of vectors.

From the code and 2640 face images with resolution 200×180 we got from Ren-Cang Li, we have generated a face image matrix whose size is 36000×2640 and with 5,569,643 nonzeros. Unlike the other two matrices from text retrieval, the face image matrix shows a strong distribution pattern where the top rows are denser and bottom rows are sparser. This is because the higher resolution images are represented by the top rows and the lower resolution images are represented by bottom rows. Like the text retrieval matrices, the image matrix is devoid of dense sublocks or bands.

3 Optimization Methods

The optimization techniques used by the Sparsity system include register blocking, cache blocking, and blocking across multiple vectors. Register blocking is effective when the matrices contain a large number of small dense subblocks, which is not the case for any of the data mining matrices in this study. In this paper we describe the two optimizations that are effective, cache blocking and blocking across multiple vectors.

3.1 Cache Blocking Optimization

We first 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 vector. 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 [HP96]. Because the caches can hold thousands of values, we rearrange the computation so that a block of values in the matrix are accessed near each other in time, but retain the sparse structure of the matrix. (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.)

The only potential reuse of data within a matrix-vector computation is between the source and destination vectors – each matrix entry is used only once. For the purpose of this discussion, we assume the matrix is organized by rows, although a similar optimization could be done for a column-based layout. An obvious matrix-vector computation for a row-based layout is to reuse the destination vector across every element of the row (possibly leaving

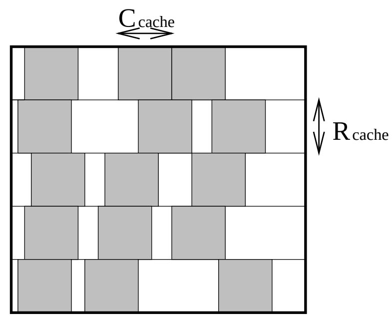

Figure 2: **Cache-blocks in a sparse matrix:** The gray areas are sparse matrix blocks that contain nonzero elements in the $r_{cache} \times c_{cache}$ rectangle. The white areas contain no nonzero elements, and are not stored.

it in a register) while one picks up the source elements on demand. For a large matrix, especially one with many columns, it is likely that the source element will not be in cache when it is needed.

The idea of cache blocking optimization is to keep c_{cache} elements of the source vector x in the cache along with r_{cache} elements of the destination vector y while an $r_{cache} \times c_{cache}$ block of matrix A is multiplied by this portion of the vector x. The entries of A need not be saved in the cache, but 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 for such an irregular problem is determining the block sizes, r_{cache} and c_{cache}. To simplify the code generation problem and to limit the range of experiments, we start with the assumption that cache blocks within a single matrix should have a fixed size. In other words, r_{cache} and c_{cache} are fixed for a particular matrix and machine. This means that 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 2 shows a matrix with fixed size cache blocks. Note that the blocks need not begin at the same offsets in each row.

We 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

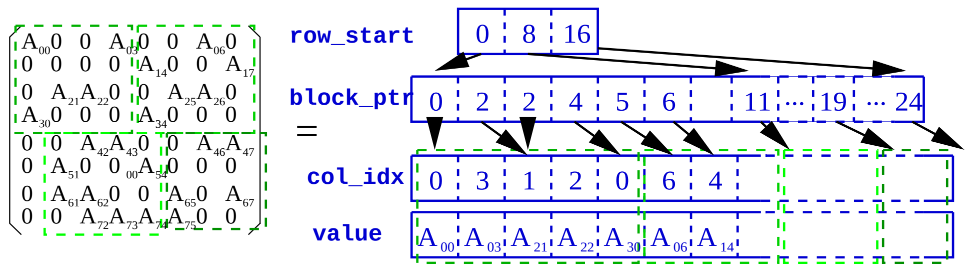

Figure 3: **Storage format of a cache-blocked sparse matrix:** In cache blocking, each block is stored in sparse format, similarly to CSR, using data structures *block_ptr*, *col_idx* and *value*. This example matrix has 4, 4×4 blocks. The *row_start* array points to the beginning of each row of blocks, while the *block_ptr* array keeps pointers to the beginnings of individual rows inside those blocks.

overhead, it incurs significantly more runtime overhead than static cache blocking [Im00], so use static cache blocking. The practical implication of this decision is that the matrix storage should be used either throughout an entire application or at least during a iterative solver to amortize the cost of reorganization.

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

The data structures used in a cache-blocked matrix are shown in figure 3. The top level array is called *row_start* and it points to the beginning of each row of blocks. In the figure, there are two rows of blocks, so the *row_start* matrix has three entries, the last pointing past the end of the *block_ptr* array. The *block_ptr* array points to the beginning of each row within a block, and the *col_idx* and *value* arrays store the column indices and values of each nonzero element. The main difference between this and the CSR format is the extra level of indirection for the blocks.

During multiplication, the nonzero elements are accessed in the order in which they are stored in memory, which is important for preserving spatial locality in the matrix. Referring back to figure 2, this means that while processing one gray block, the indices and values in the matrix are accessed in storage order, and the portions of the x and y vectors that correspond to that block are accessed repeatedly. The sub-arrays of x and y will sit in the cache during processing, as long as they both fit and there is no interference between the two sub-arrays and the matrix entries.

3.2 Optimization for Multiplication by Multiple Vectors

To improve the performance of sparse matrix operations, one can also take advantage of the fact that there are often multiple vectors being multiplied. All of the data mining algorithms in this paper multiply the sparse matrix times a set of vectors. In scientific computing, it also occurs in practice when there are multiple right-hand sides in an iterative solver, or in blocked eigenvalue algorithms, such as block Lanczos [GU77, GLS94, Mar95] or block Arnoldi [Sad93, LM97]. Another application is image segmentation in videos, where a set of vectors is used as the starting guess for a subsequent frame in the video [SM98].

The use of multiple vectors in these problems essentially turns the kernel into a matrix-matrix multiplication in which the second matrix is small, but dense. This admits much more potential for memory hierarchy optimizations than the single vector case, since it increases the number of floating point operations per matrix element. Matrix-vector multiplication accesses each matrix element only once, whereas a matrix times a set of k vectors will access each matrix element k times.

While there is much more potential for high performance with multiple vectors, the advantage will not be exhibited in straightforward implementations that organize the computation as single matrix-vector multiplies. We therefore change the multiplication code to access elements across the vectors, allowing the matrix elements to be reused.

Our code generator, developed for the Sparsity system, produces code specifically for register-blocked multiplication for a fixed set of vectors. The number of vectors is fixed at code generation time and all of the loops fully unrolled across the vectors. The code generator creates the inner kernels of a larger computation; if the number of vectors is very large, the loop over the vectors would be strip-mined, with the resulting inner loop becoming one of these unrolled loops.

Processor	Clock (MHz)	L2 cache size	DGEMV (MFLOPS)	DGEMM (MFLOPS)
MIPS R10000	200	2 MB	67	322
UltraSPARC II	250	1 MB	100	401
Pentium III	450	512 KB	87	328
Alpha 21164	533	96 KB	83	550

Figure 4: **Summary of Machines**

4 Performance Improvement of Sparse Matrix-Vector Multiplication on Matrices from Data Mining Applications

We have applied the two optimizations, cache blocking and blocking across multiple vectors, to the three sparse matrices described in section 2. We also applied a third optimization which is the combination of the first two. Our experiments are run on four modern microprocessors, a 200 MHz MIPS R10000, a 250MHz SUN UltraSPARC II,a 450MHz Intel Pentium III, and a 533 MHz Compaq Alpha 21164. The machines are summarized in figure 4. It shows processors' clock speed and L2 cache size along with performance of optimized dense BLAS routines for comparison. The BLAS routines in the table are dense matrix-vector multiplication (DGEMV) and dense matrix-matrix multiplication (DGEMM), both for double-precision floating point numbers. They are measured for dense 1000×1000 matrices. We have used vendor-supplied hand-optimized BLAS libraries (SCSL and Sun Performance libraries) for the R10000 and UltraSPARC, we have used automatically tuned BLAS routines for the Pentium III and Alpha 21164, based on the ATLAS BLAS-generation system [WD]. The DGEMV is an upper bound on the expected performance for sparse matrix-vector multiplication, while DGEMM is an upper bound for the multiple vector case.

Figures 6 – 8 show the performance of sparse matrix-vector multiplication on the web document matrix, the NSF abstract matrix, and the face recognition matrix. Each group of bars shows the performance of one machine, which are ordered from left to right as: R10000, UltraSPARC II, Pentium III and Alpha 21164. In each group, the leftmost bar shows the raw performance before optimization, and next two bars show the performance after cache blocking and with multiple vectors, respectively. The last bar shows the performance of combined optimizations. The speedups of combined optimizations are summarized in figure 5. Overall, the speedup

Matrix	MIPS R10000	UltraSPARC II	Pentium III	Alpha 21164
Web Document	3.8	5.9	2.0	2.7
NSF Abstract	2.9	1.3	1.6	1.3
Face Images	4.7	5.1	2.6	4.5

Figure 5: **Speedup of Sparse Matrix-Vector Multiplication**

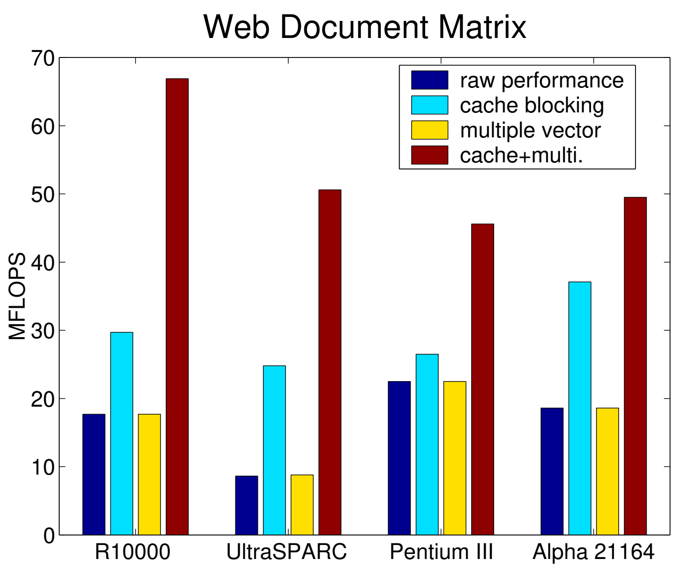

Figure 6: **Performance of Sparse Matrix Vector Multiplication on the Web Document Matrix**

is in the range of 1.3 – 5.9.

Note that the vertical scales of three graphs are different, the performance on a web document matrix being smallest, and the performance on a face recognition matrix being largest. From figure 1, we can see the density of those matrices are in the same order; the density of the web document matrix is smallest (0.15 %) and face recognition is largest (5.86 %). As the matrix becomes denser, there is more spatial/temporal locality of access in the operation, so even the raw performance is better on denser matrices.

The second observation is that cache blocking improves the performance on web document matrix, but does not exhibit noticeable speedup for the NSF abstract matrix and the face recognition matrix. This is clearly due to the matrix size, and in particular, the number of columns in the matrix. The web document matrix has almost 100x more columns than the other

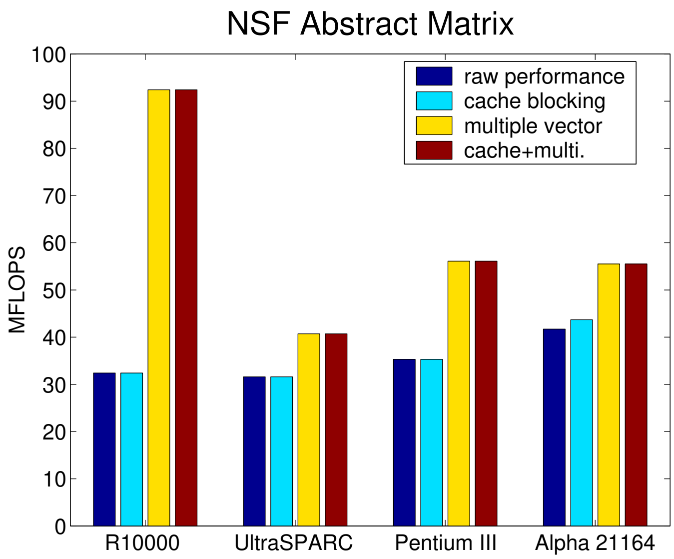

Figure 7: **Performance of Sparse Matrix Vector Multiplication on the NSF Abstract Matrix**

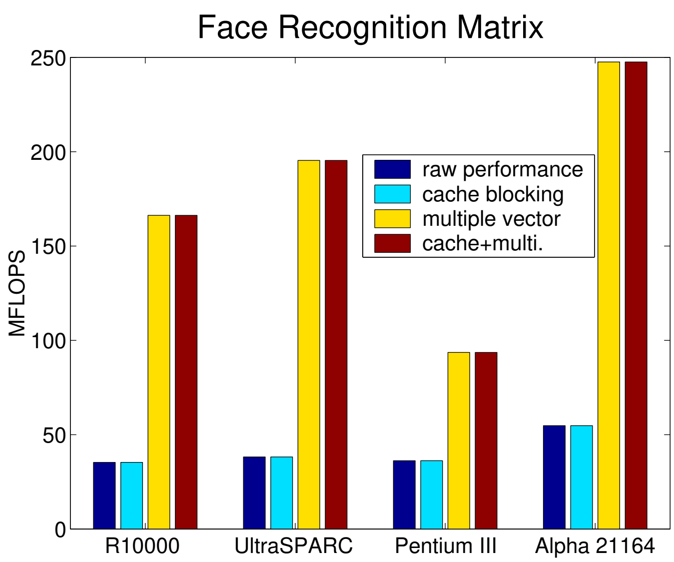

Figure 8: **Performance of Sparse Matrix Vector Multiplication on the Face Recognition Matrix**

Processor	L2 cache size	Block Size for Single Vector	Block Size for 10 vectors
MIPS R10000	2 MB	10000x 65536	10000x 4096
UltraSPARC II	1 MB	10000x 32768	10000x 2048
Pentium III	512 KB	10000x 16384	10000x 4096
Alpha 21164	96 KB	10000x 4096	10000x 2048

Figure 9: **Chosen Cache Block Sizes for Web Document Matrix**

two matrices, and combined with its low density, the likelihood that a source vector element will be in cache when it is needed is very low. Although these matrices are somewhat constrained in size, the importance of cache blocking is likely to increase when mining larger datasets. The number of rows in these matrices, which represents the number of keywords or pixels, is not likely to increase with the data set, but the number of columns, which is the number of documents or images, is very likely to grow.

The cache block sizes are chosen automatically by the Sparsity system after measuring the performance for block sizes between 32×32 and $128K \times 128K$ that are powers of two. Figure 9 shows the cache block sizes chosen for each matrix on each machine. Sparsity selects block sizes for which the number of rows in the cache blocks are larger than 10000, but since that is the total number of rows in the matrix, it is recorded as 10000 in the table. The L2 cache sizes are shown together in the graph, and we can see the relation between the size of cache and the number of columns in the cache block. It is roughly shown by the following expression.

$$Number\ of\ columns\ in\ cache\ block = Size\ of\ cache/(4 * sizeof(double))$$

The constant 4 may not be the exact number because we have not searched the space of cache block size exhaustively in the range, but the expression clearly shows that the width (the number of columns) of cache block is limited by cache size. This matches the intuition that the aim of cache blocking is to increase the reuse of source vector elements in the cache.

The table also shows the chosen cache block size for multiplication by 10 vectors with the loop-unrolled code for multiple vectors. The cache block sizes are chosen in the same way as in the multiplication for the single vector. The cache block sizes are smaller than that of single vector case, because $(number\ of\ columns\ in\ cache\ block) * (number\ of\ vectors\ multiplied)$ elements of the source vectors should be kept in cache while the computation is performed.

A separate parameter from the cache block size is the amount of unrolling across the vectors. We chose 10 total vectors for these experiments because that is a reasonable number for the algorithms that use multiple vectors. The multiplication code is unrolled 10 times for the MIPS R10000, UltraSPARC II and Pentium III, but the same code is unrolled only 3 times for the Alpha 21164. The loop-unrolling factor is also chosen automatically by Sparsity to optimize performance [Im00]. As discussed in section 3.2, the set of vectors are strip-mined to be multiplied if the number of vectors are larger than the loop-unrolling factor, as is the case on the Alpha. In the table, the widths of cache blocks for single vector are 8 times smaller than the widths of cache blocks for multiple vectors for those first three processors, and the ratio is 2 for Alpha 21164 which was unrolled 3 times. The ratio could only be the power of two's since we have chosen the cache block size of power of two's. The ratios 8 and 2 are close approximations of 10 and 3 which are loop-unrolling factors for each case.

For the other two matrices, NSF abstract matrix and face image matrix, the cache blocking did not improve the performance because the width of cache block sizes shown in figure 9 are not much larger than those matrices. So we have chosen cache block sizes to be matrix sizes for those two matrices. Using an unrolled loop for multiplication improved the performance noticeably for those two matrices. However, the performance of multiplication with multiple vectors did not speed up for the web document matrix, because, as discussed earlier, the source vectors are so long that elements are rarely in cache. We should note that the each of the vectors is 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. In the current layout, the best performance for the web matrix is obtained by combining both optimizations.

5 Conclusion

In this paper, we have introduced two optimization techniques for sparse matrix-vector multiplication, cache blocking and blocking across multiple vectors. These two optimizations are especially important for large data mining matrices. Cache blocking is important when one of the dimensions is very large, even if the other is significantly smaller. Optimizations like register blocking were not discussed in detail, because while they are important for certain scientific applications, they have little effect in data mining [Im00].

These optimizations were applied using our Sparsity toolbox, which automatically generates optimized sparse matrix-vector code. Sparsity chooses parameters such as cache block size by searching over a set of possible candidates and measuring their performance. We demonstrated the effectiveness of these optimizations for three example sparse matrices taken from data mining applications. Cache blocking is particularly useful for a sparse matrix with many columns. Unrolling across multiple vectors is effective as long as the number of columns is not too large, and if it is the combination is very effective. We also identified the relationship between the number of columns in cache block and size of cache. When the multiplication is performed on a set of vectors, the number of columns in the cache block decreased accordingly.

Overall, our approach produced speedups between 1.3 and 5.9 on the three matrices when measured across 4 machines. We believe these optimizations will be increasingly important as memory latency increases relative to clock rate and as the desired data set size increases. Because of the complexity of modern memory hierarchies and the difficulty of reorganizing sparse data structures and computation, we believe that the Sparsity approach of combining search with the kinds of analytical models derived for cache block size are a key to helping end users obtain high performance.

6 Acknowledgement

We would like to thank Inderjit Dhillon for providing us an NSF abstract matrix, Osni Marques for a web document matrix, and Ren-Cang Li for face database and code for generating wavelet-transformed matrix. And we also thank them for their insightful discussions.

References

[BCD+00] 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.

[BDO95] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37(4):573–595, 1995.

[DM99] Inderjit S. Dhillon and Dharmendea S. Modha. Concept decompositions for large sparse text data using clustering. Technical Report RJ 10147, IBM, July 1999. to appear in Machine Learning.

[GLS94] 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.

[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.

[GVL96] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[HP96] John L. Hennesy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, second edition, 1996.

[Im00] Eun-Jin Im. *Optimizing the Performance of Sparse Matrix - Vector Multiplication*. PhD thesis, University of California at Berkeley, May 2000.

[Li98] R.-C. Li. A multi-resolution approach for calculating primary eigenvectors of a large set of images. Technical Report 98-13, Department of Mathmatics, University of Kentucky, June 1998.

[Li99] R.-C. Li. Fast partial eigenvalue decomposition with wavelet transformation for large images, July 1999.

[LM97] R. Lehoucq and K. Maschhoff. Implementation of an implicitly restarted block Arnoldi method. Preprint MCS-P649-0297, Argonne National Lab, 1997.

[Mar95] Osni A. Marques. BLZPACK: Decsription and User's guide. Technical Report TR/PA/95/30, CERFACS, 1995.

[Sad93] M. Sadkane. A block Arnoldi-Chebyshev method for computing the leading eigenpairs of large sparse unsymmetric matrices. *Numer. Math.*, 64:181–193, 1993.

[SM98] Jianbo Shi and Jitendra Malik. Motion segmentation and tracking using normalized cuts. In *International Conference on Computer Vision*, January 1998.

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