Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY

Eun-Jin Im[1] and Katherine Yelick[2]

[1] School of Computer Science, Kookmin University, Seoul, Korea
ejim@cs.kookmin.ac.kr,
[2] Computer Science Division, University of California, Berkeley, CA 94720, USA
yelick@cs.Berkeley.EDU

Abstract. Sparse matrix-vector multiplication is an important computational kernel that tends to perform poorly on modern processors, largely because of its high ratio of memory operations to arithmetic operations. Optimizing this algorithm is difficult, both because of the complexity of memory systems and because the performance is highly dependent on the nonzero structure of the matrix. The Sparsity system is designed to address these problem by allowing users to automatically build sparse matrix kernels that are tuned to their matrices and machines. The most difficult aspect of optimizing these algorithms is selecting among a large set of possible transformations and choosing parameters, such as block size. 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 for matrices arising in scientific simulations, register level optimizations are critical, and we focus here on the optimizations and parameter selection techniques used in Sparsity for register-level optimizations. We demonstrate speedups of up to 2× for the single vector case and 5× for the multiple vector case.

1 Introduction

Matrix-vector multiplication is used in scientific computation, signal and image processing, document retrieval, and many 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 for two reasons: 1) there is overhead to accessing the index information in the matrix structure and 2) the memory accesses tend to have little spatial or temporal locality. For example, on an 167 MHz UltraSPARC I, there is a 2x slowdown due to the data structure overhead (measured by comparing a dense matrix in sparse and dense format) and an additional 5x slowdown for matrices that have a nearly random nonzero structure.

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 several optimizations, including register blocking, cache blocking, loop unrolling, matrix reordering, and reorganization for multiple vectors [Im00]. The optimizations involve both code and data structure transformations, which can be quite expensive. Fortunately, sparse matrix-vector multiplication is often used in iterative solvers or other settings where the same matrix is multiplied by several different vectors, or matrices with different numerical entries but the same or similar nonzero patterns will be re-used. Sparsity therefore uses transformations that are specialized to a particular matrix structure, which we will show is critical to obtaining high performance.

In this paper we focus on register level optimizations, which include register blocking and reorganization for multiple vectors. The challenge is to select the proper block size and the right number of vectors to maximize performance. In both cases there are trade-offs which make the parameters selection very sensitive to both machine and matrix. We explore a large space of possible techniques, including searching over a set of parameters on the machine and matrix of interest and use of performance models to predict which parameter settings will perform well. For setting the register block size, we present a performance model based on some matrix-independent machine characteristics, combined with an analysis of blocking factors that is computed by a statistical sampling of the matrix structure. The model works well in practice and eliminates the need for a large search. For choosing the optimal number of vectors in applications where a large number or vectors are used, we present a heuristic for choosing the block size automatically, which works well on many matrices, but in some cases we find that searching over a small number of vectors produces much better results.

2 Register Optimizations for Sparse Matrices

In this section we describe two optimizations: register blocking and reorganization for multiple vectors. There are many popular sparse matrix formats, but to make this discussion concrete, assume we start with a matrix in 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 this case, the elements of A are accessed sequentially but not reused. The elements of y are also accessed sequentially, but more importantly they are re-used 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 re-use 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. To take advantage of the improved locality for register allocation, the block sizes need to be fixed 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 also 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 to add explicit zeros where necessary. While the idea of *blocking* or *tiling* for dense matrix operations is well-known (e.g., [LRW91]), the sparse matrix transformation is quite different, since it involves filling in zeros, and the choice of r and c will depend on the matrix structure as described in section 3.

We also consider a second register level optimization of matrix-vector multiplication when the matrix is going to be multiplied by a set of vectors. This is less common than the single vector case, but practical when there are multiple right-hand sides in an iterative solver, or in blocked eigenvalue algorithms, such as block Lanczos [Mar95] or block Arnoldi [BCD$^+$00]. 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. The basic optimization is to interchange loops so that for each matrix element, the source and destination values for all vectors are accessed before going to the next element.

Sparsity contains a code generator that produces loop-unrolled C code for given block sizes and for a fixed number of vectors. If the number of vectors is very large, the loop over the vectors is strip-mined, with the resulting inner loop becoming one of these unrolled loops. The optimized code removes some of the branch statements and load stalls by reordering instructions, all of which further improve the performance beyond simply interchanging loops.

3 Choosing the Register Block Size

Register blocking does not always improve performance if the sparse matrix does not have small dense blocks. Even when it has such blocks, the optimizer must pick a good block size for a given matrix and machine. We have developed a performance model that predicts the performance of the multiplication for various block sizes without actually blocking and running the multiplication. The 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, some zero elements are filled in to make a complete $r \times c$ block. 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 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*.

The first component cannot be exactly determined without running the resulting blocked matrix 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 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 subset of the matrix. 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.

Figure 1 show the performance of sparse matrix vector multiplication for a dense matrix using register-blocked sparse format, on an UltraSPARC I and a MIPS R10000. We vary the block size within a range of values for r and c until the performance degrades. The data in the figure uses a 1000×1000 dense matrix, but 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.

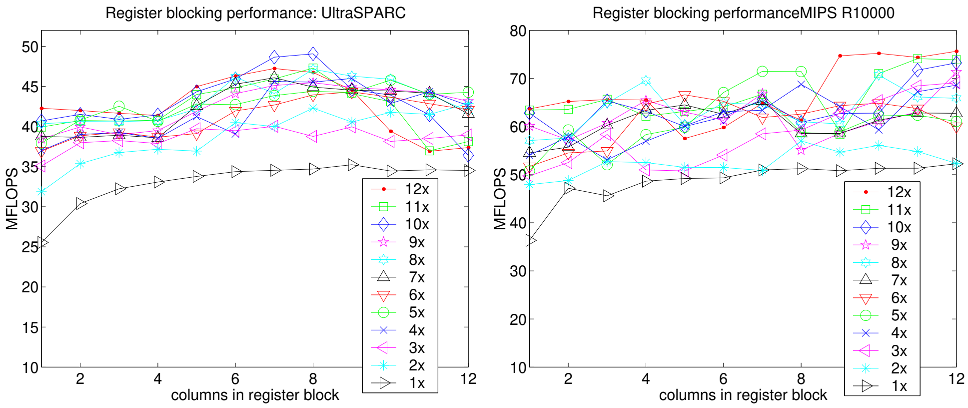

Fig. 1. Performance profile of register-blocked code on an UltraSPARC I (left) and a MIPS R10000 (right): These numbers are taken for a 1000×1000 dense matrix represented in sparse blocked format. Each line is for a fixed number of rows (r), varying the number of columns (c) from 1 to 12.

To approximate the unnecessary computation that would result from register blocking, we estimate the fill overhead. To keep the cost of this computation low, two separate computations are made over the matrix of interest for a column blocking factor (c) and a row blocking factor (r), each being done for a square block size and examining only a fraction of the matrix. For example, to compute r we sample every k^{th} row to compute the fill overhead for that row for every value of r being considered. We use this estimate of fill overhead to predict the performance of an $r \times r$ blocking of a particular matrix A as:

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

While k and the range of r can easily be adjusted, we have found that setting k to 100 and letting r range from 1 to r_{max} is sufficient, where r_{max} is the value of r for which the dense matrix demonstrates its best performance. The value of r is chosen to be the one that maximizes the above performance estimate for $r \times r$ blocks. The choice of c is computed independently by an analogous algorithm on columns. Note that while these two computations use square blocks, the resulting values of r and c may be different.

4 Choosing the Number of Vectors

The question of how many vectors to use when multiplying by a set of vectors is partly dependent on the application and partly on the performance of the multiplication operation. For example, there may be a fixed limit to the number of right-hand sides or convergence of an iterative algorithm may slow as the number of vector increases. If there is 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.

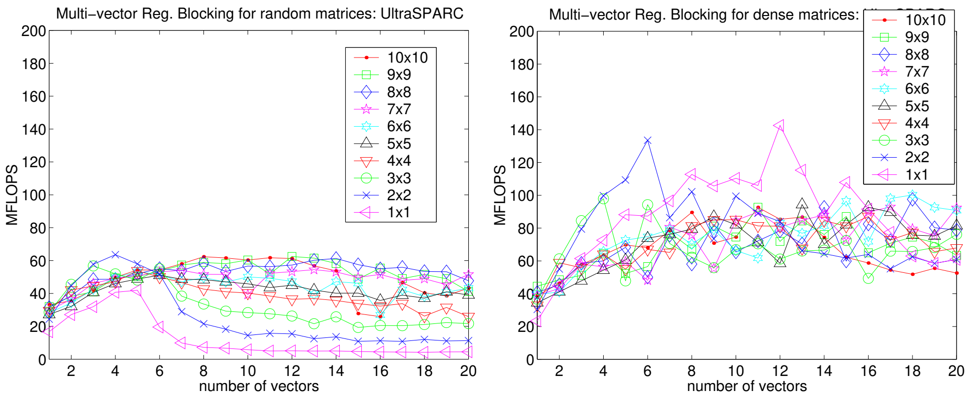

Fig. 2. Register-blocked, multiple vector performance on an UltraSPARC I, varying the number of vectors.

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

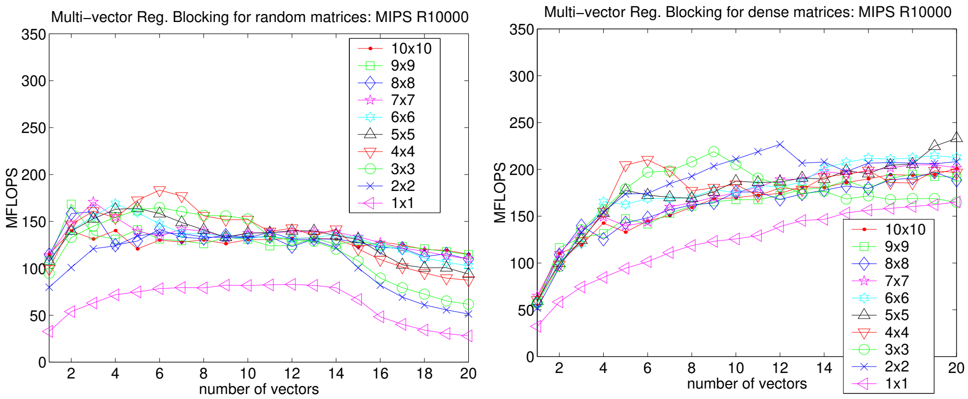

Fig. 3. Register-blocked, multiple vector performance on a MIPS R10000, varying the number of vectors.

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 two extremes in the space of matrix structures: a dense $1K \times 1K$ matrix in sparse format, and sparse $10K \times 10K$ randomly generated matrices with $200K$ (.2%) of the entries being nonzero. In both cases, the matrices are blocked for registers, which in the random cases means that the $200K$ nonzero entries will be clustered differently, depending on the block size. We will also limited our data to square block sizes from 1×1 up to 10×10.

Figures 2 and 3 show the effect of changing the block size and the number of vectors on an UltraSPARC I and MIPS R10000. (The shape of these graphs is different for other machines, but the basic observations below are the same.) The figures shows the performance of register-blocked code optimized for multiple vectors, with the left-hand side showing the randomly structured matrix and the right-hand side showing the dense matrix.

Multiple vectors typically pay off for matrices throughout the regularity and density spectrum, and we can get some sense of this but 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, and there is also quite a bit of variability across machines. There are two characteristics that appear common across both these two machines and others we have studied. First, the random matrix tends to have a peak with some relatively small number of vectors (2-5), whereas for the dense matrix it is at 12 (and generally in the range from 9 to 12 on other machines). For the dense matrix, all of these vectors consume register resources, so the optimal block size is relatively small compared to the that of the single vector code on the same matrix. The behavior of the R10000 is smoother than that of the UltraSPARC, which is probably a reflection of the more expensive memory system on the R10000.

5 Performance of Register Optimizations

We have generated register blocked codes for varying sizes of register blocks and varying numbers of vectors using Sparsity, and have measured their performance on several machines [Im00]. In this paper we will present the results for a set of 39 matrices on the UltraSPARC I and MIPS R10000. The matrices in the set are taken from fluid dynamics, structural modeling, chemistry, economics, circuit simulation and device simulation, and we include one dense matrix in sparse format for comparison. We have omitted matrices from linear programming and information retrieval, which have very little structure and therefore to not benefit from register blocking optimizations. Other optimizations such as cache blocking prove to be useful on some of those.

Figure 5 summarizes the 39 matrices. We have placed the matrices in the table according to our understanding of the application domain from which is was derived. Matrix 1 is a dense matrix. Matrices 2 through 17 are from Finite Element Method (FEM) applications, which in several cases means there are dense sub-locks within much of the matrix. Note however, that the percentage of nonzeros is still very low, so these do not resemble the dense matrix. Matrices 18 through 39 are from structural engineering and device simulation. All the matrices are square, and although some are symmetric, we do not try to take advantage of symmetry here. The matrices are roughly ordered by the regularity of nonzero patterns, with the more regular ones at the top.

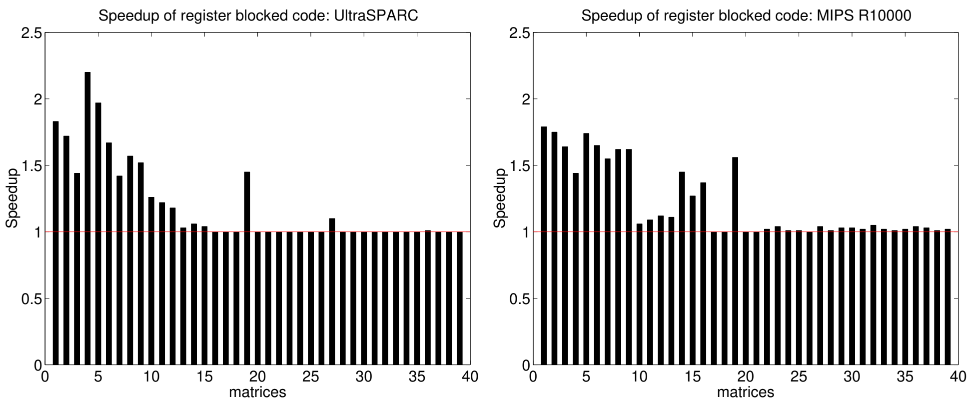

Fig. 4. Speedup of register-blocked multiplication on a 167 MHz UltraSPARC I (left) and a 200MHz MIPS R10000 (right).

Figure 4 shows the effect of register blocking with a single vector on the 39 matrices in table 5. (The Mflop rate was calculated using only those arithmetic operations required by the original representation, not those induced by fill from blocking.) The benefit is highest for the lower numbered matrices, which tend to have naturally occurring dense subblocks, although they are not uniform, so

	Name	Application Area	Dimension	Nonzeros	Sparsity
1	dense1000	Dense Matrix	1000x 1000	1000000	100
2	raefsky3	Fluid structure interaction	21200x21200	1488768	0.33
3	inaccura	Accuracy problem	16146x16146	1015156	0.39
4	bcsstk35	Stiff matrix automobile frame	30237x30237	1450163	0.16
5	venkat01	Flow simulation	62424x62424	1717792	0.04
6	crystk02	FEM Crystal free vibration	13965x13965	968583	0.50
7	crystk03	FEM Crystal free vibration	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 calculation	23560x23560	484256	0.09
12	raefsky4	buckling problem	19779x19779	1328611	0.34
13	ex11	3D steady flow calculation	16614x16614	1096948	0.40
14	rdist1	Chemical process separation	4134x 4134	94408	0.55
15	vavasis3	2D PDE problem	41092x41092	1683902	0.10
16	orani678	Economic modeling	2529x 2529	90185	1.41
17	rim	FEM fluid mechanics problem	22560x22560	1014951	0.20
18	memplus	Circuit Simulation	17758x17758	126150	0.04
19	gemat11	Power flow	4929x 4929	33185	0.14
20	lhr10	Light hydrocarbon recovery	10672x10672	232633	0.20
21	goodwin	Fluid mechanics problem	7320x 7320	324784	0.61
22	bayer02	Chemical process simulation	13935x13935	63679	0.03
23	bayer10	Chemical process simulation	13436x13436	94926	0.05
24	coater2	Simulation of coating flows	9540x 9540	207308	0.23
25	finan512	Financial portfolio optimization	74752x74752	596992	0.01
26	onetone2	Harmonic balance method	36057x36057	227628	0.02
27	pwt	Structural engineering problem	36519x36519	326107	0.02
28	vibrobox	Structure of vibroacoustic problem	12328x12328	342828	0.23
29	wang4	Semiconductor device simulation	26068x26068	177196	0.03
30	lnsp3937	Fluid flow modeling	3937x 3937	25407	0.16
31	lns3937	Fluid flow modeling	3937x 3937	25407	0.16
32	sherman5	Oil reservoir modeling	3312x 3312	20793	0.19
33	sherman3	Oil reservoir modeling	5005x 5005	20033	0.08
34	orsreg1	Oil reservoir simulation	2205x 2205	14133	0.29
35	saylr4	Oil reservoir modeling	3564x 3564	22316	0.18
36	shyy161	Viscous flow calculation	76480x76480	329762	0.01
37	wang3	Semiconductor device simulation	26064x26064	177168	0.03
38	mcfe	astrophysics	765x 765	24382	4.17
39	jpwh991	Circuit physics modeling	991x 991	6027	0.61

Fig. 5. Matrix benchmark suite: The basic characteristic of each matrix used in our experiments is shown. The sparsity column is the percentage of nonzeros.

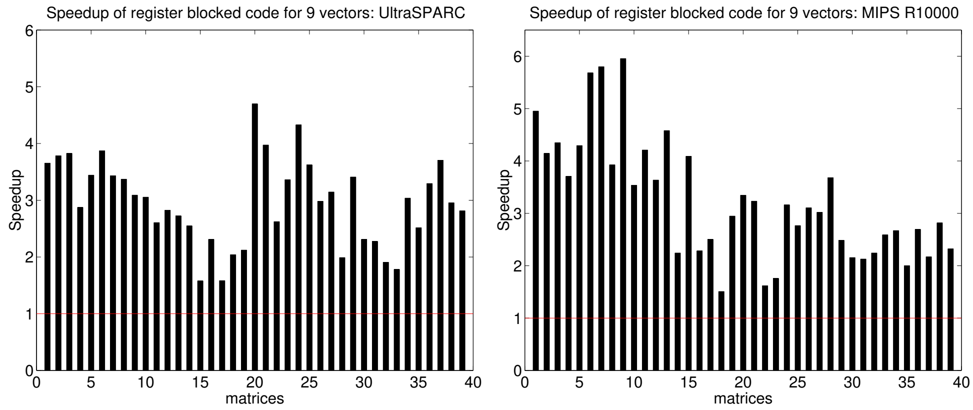

Fig. 6. Speedup of register-blocked, multiple vector code using 9 vectors.

there is fill overhead. Some of the matrices that have no natural subblocks still benefit from small blocks.

Figure 6 shows the speedup of register blocking for multiple vectors on a same matrix set. The number of vectors is fixed at 9, and it shows a tremendous payoff. On the MIPS R10000, the lower-number matrices have a slight advantage, and on the UltraSPARC, the middle group of matrices sees the highest benefit; these are mostly matrices from scientific simulation problems with some regular patterns, but without the dense sub-blocks that appear naturally in the lower-numbered FEM matrices. Overall, benefits are much more uniform across matrices than for simple register blocking.

6 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 [Bik96]. 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. Toledo [Tol97] demonstrated some of the performance benefits or register blocking, including a scheme that mixed multiple block sizes in a single matrix, and PETSc (Portable, Extensible Toolkit for Scientific Computation) [BGMS00] uses a application-specified notion of register blocking for Finite Element Methods. Toledo and many others have explored the benefits of reordering sparse matrices, usually for parallel machines or when the natural ordering of the application has been destroyed. Finally, we note that the BLAS Technical

Forum has already identified the need for runtime optimization of sparse matrix routines, since they include a parameter in the matrix creation routine to indicate how frequently matrix-vector multiplication will be performed [BLA99].

7 Conclusions

In this paper, we have described optimization techniques to increase register reuse in sparse matrix-vector multiplication for one or more vectors. We described some parts of the Sparsity system that generate code for fixed block sizes, filling in zeros as necessary. To select the register block size, we showed that a simple performance model that separately takes a machine performance profile and a matrix fill estimation worked very well. The model usually chooses the optimal block size, producing speedups of around 2× for some matrices. Even on matrices where the blocks were not evident at the application level, small blocks proved useful on some machines. We also extended the Sparsity framework to generate code for multiple vectors, where the benefits are are high as 5× on the machines and matrices shown here.[1]

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.

[BGMS00] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.28, Argonne National Laboratory, 2000.

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

[BLA99] BLAST Forum. *Documentation for the Basic Linear Algebra Subprograms (BLAS)*, October 1999. http://www.netlib.org/blast/blast-forum.

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

[LRW91] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

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

[Tol97] Sivan Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[1] This research is supported in part by U.S. Army Research Office, by the Department of Energy and by Kookmin University, Korea.