Memory Hierarchy Optimizations and Performance Bounds for Sparse A^TAx

Richard Vuduc, Attila Gyulassy, James W. Demmel, and Katherine A. Yelick

Computer Science Division, University of California, Berkeley
richie,jediati,demmel,yelick@cs.berkeley.edu

Abstract. This paper presents uniprocessor performance optimizations, automatic tuning techniques, and an experimental analysis of the sparse matrix operation, $y = A^TAx$, where A is a sparse matrix and x, y are dense vectors. We describe an implementation of this computational kernel which brings A through the memory hierarchy only once, and which can be combined naturally with the register blocking optimization previously proposed in the SPARSITY tuning system for sparse matrix-vector multiply. We evaluate these optimizations on a benchmark set of 44 matrices and 4 platforms, showing speedups of up to 4.2×. We also develop platform-specific upper-bounds on the performance of these implementations. We analyze how closely we can approach these bounds, and show when low-level tuning techniques (e.g., better instruction scheduling) are likely to yield a significant pay-off. Finally, we propose a hybrid off-line/run-time heuristic which in practice automatically selects near-optimal values of the key tuning parameters, the register block sizes.

1 Introduction

This paper considers automatic performance tuning of the sparse matrix operation, $y \leftarrow y + A^TAx$, where A is a sparse matrix and x, y are dense vectors. This computational kernel—SpA^TA hereafter—is the inner-loop of interior point methods for mathematical programming [17], algorithms for computing the singular value decomposition [5], and Kleinberg's HITS algorithm for finding hubs and authorities in graphs [10], among others. The challenge in tuning sparse kernels is choosing a data structure and algorithm that best exploits the non-zero structure of the matrix for a given memory hierarchy and microarchitecture: this task can be daunting and time-consuming because the best implementation will vary across machines, compilers, and matrices. Purely static solutions are limited since the matrix may not be known until run-time.

Our approach to automatic tuning of SpA^TA builds on experience with dense linear algebra [2,18], sparse matrix-vector multiply (SpM×V) [9,14], and sparse triangular solve (SpTS) [16]. Specifically, we apply the tuning methodology first proposed in the SPARSITY system for SpM×V [9]. We show how SpA^TA can be algorithmically cache-blocked to reuse A in a way that also allows register-level blocking to exploit dense subblocks (Section 2). The set of these implementations, parameterized by block size, defines an *implementation space*. We search

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2659, pp. 705–714, 2003.

Table 1. Evaluation platforms. Dense BLAS data reported using ATLAS 3.4.1 [18] on the Ultra 2i and Pentium III, ESSL v3.1 on the Power3, and MKL v5.2 on Itanium.

Property	Sun Ultra 2i	Intel Pentium III	IBM Power3	Intel Itanium
Clock rate (MHz)	333	500	375	800
Peak Main Memory Bandwidth (MB/s)	500	680	1600	2100
Peak Flop Rate (Mflop/s)	667	500	1500	3200
DGEMM, $n = 1000$ (Mflop/s)	425	331	1300	2200
DGEMV, $n = 2000$ (Mflop/s)	58	96	260	315
STREAM Triad Bandwidth (MB/s)	250	350	715	1100
No. of FP regs (double)	16	8	32	128
L1 size (KB), line size (B), latency (cy)	16,16,1	16,32,1	64,128,.5	16,32,1 (int)
L2 size (KB), line size (B), latency (cy)	2048,64,7	512,32,18	8192,128,9	96,64,6-9
L3 size (KB), line size (B), latency (cy)	n/a	n/a	n/a	2048,64,21-24
Memory latency (cycles, $\approx$)	36-66 cy	26–60	35-139 cy	36-85 cy
Vendor C compiler version	v6.1	v6.0	v5.0	v6.0

this space by (1) benchmarking the routines on a synethetic matrix *off-line* (*i.e.*, *once* per platform), and (2) predicting the best block size using estimated properties of the matrix non-zero structure and the benchmark data. Our experiments on four hardware platforms (Table 1) and 44 matrices show that we can obtain speedups between 1.5×–4.2× over a reference implementation which computes $t = Ax$ and $y = A^T t$ as separate steps.

We evaluate our SpA^TA performance relative to upper bounds on performance (Section 3). We used similar bounds for SpM×V to show that the performance (Mflop/s) of SPARSITY-generated code is often within 20% of the upper bound, implying that more low-level tuning (*e.g.*, better instruction scheduling) will be limited [14]. Here, we derive upper bounds for our SpA^TA implementations, and show that we typically achieve between 20%–40% of this bound. Since we rely on the compiler to schedule our fully unrolled code, this finding suggests that future work can fruitfully apply automatic low-level tuning methods, in the spirit of ATLAS/PHiPAC, to improve further the SpA^TA performance.

This paper summarizes the key findings of a recent technical report [15]. We refer the reader there for details omitted due to space constraints.

2 Memory Hierarchy Optimizations for Sparse A^TAx

We assume a baseline implementation of $y = A^TAx$ that first computes $t = Ax$ followed by $y = A^Tt$. For large matrices A, this implementation brings A through the memory hierarchy twice. However, we can compute A^TAx by reading A from main memory only once. Denote the rows of A by $a_1^T, a_2^T, \ldots, a_m^T$. Then,

$$y = A^TAx = (a_1 \ldots a_m) \begin{pmatrix} a_1^T \\ \ldots \\ a_m^T \end{pmatrix} x = \sum_{i=1}^{m} a_i(a_i^T x). \quad (1)$$

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

$$\texttt{b\_row\_ptr} = \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}, \texttt{b\_col\_idx} = \begin{pmatrix} 0 & 4 & 2 & 4 \end{pmatrix}$$
$$\texttt{b\_value} = \begin{pmatrix} a_{00} & a_{01} & a_{10} & a_{11} & a_{04} & a_{05} & a_{14} & a_{15} & a_{22} & 0 & a_{32} & a_{33} & a_{24} & a_{25} & a_{34} & a_{35} \end{pmatrix}$$

Fig. 1. 2×2 BCSR storage. Elements are stored in the `b_value` array. The column index of the (0,0) entry of each block is stored in `b_col_idx`. The `b_row_ptr` array points to block row starting positions in the `b_col_idx` array. (Figure taken from Im [9].)

Assuming sufficient cache capacity, each row a_i^T is read from memory into cache to compute the dot product $t_i = a_i^T x$, and reused for the vector scaling $t_i a_i$. We can also take each a_i^T to be a *block of rows* instead of just a single row, allowing us to combine the cache optimization of Equation (1) with a previously proposed *register blocking* optimization [9]. Below, we review register blocking, and describe our heuristic to choose the key tuning parameter, the block size.

Register blocking improves register reuse by reorganizing the matrix data structure into a sequence of "small" dense blocks, keeping small blocks of x and y in registers. An $m \times n$ sparse matrix in $r \times c$ register blocked format is divided logically into $\frac{m}{r} \times \frac{n}{c}$ submatrices, each of size $r \times c$. Only blocks containing at least one non-zero are stored. Multiplying by A proceeds block-by-block: for each block, we reuse the corresponding r elements of y and c elements of x. For simplicity, assume that r divides m and c divides n. We use the blocked compressed sparse row (BCSR) storage format [11], a 2×2 example of which is shown in Figure 1. When $r = c = 1$, BCSR reduces to compressed sparse row (CSR) storage. BCSR can store fewer column indices than CSR (one per block instead of one per non-zero). We fully unroll the $r \times c$ submatrix computation to reduce loop overhead and expose scheduling opportunities to the compiler.

Figure 1 also shows that creating blocks may require filling in explicit zeros. We define the *fill ratio* to be the number of stored values (*i.e.*, including zeros) divided by the number of true non-zeros. We may trade-off extra computation (*i.e.*, fill ratio > 1) for improved efficiency from uniform code and memory access.

To select r and c, we adapt the Sparsity v2.0 heuristic for SpM×V [14] to SpA^TA. First, we measure the speed (Mflop/s) of the blocked SpA^TA code for all $r \times c$ up to 8×8 for a dense matrix stored in sparse BCSR format. These measurements are made only once per architecture. Second, when the matrix is known at run-time, we estimate the fill ratio for all block sizes using a recently described sampling scheme [14]. Third, we choose the r and c that maximize

$$\text{Estimated Mflop/s} = \frac{\text{Mflop/s on dense matrix in } r \times c \text{ BCSR}}{\text{Estimated fill ratio for } r \times c \text{ blocking}} \quad . \tag{2}$$

The run-time overhead of picking a register block size and converting into our data structure is typically between 5–20 executions of naïve SpA^TA [14]. Thus,

the optimizations we propose are most suitable when SpA^TA must be performed many times (*e.g.*, in sparse iterative methods).

3 Upper Bounds on Performance

We use performance upper bounds to estimate the best possible performance given a matrix and data structure but *independent* of any particular instruction mix or ordering. In our work on sparse kernels, we have thus far focused on data structure transformations, relying on the compiler to produce good schedules. An upper bound allows us to estimate the likely pay-off from low-level tuning.

Our bounds for the SpA^TA implementations described in Section 2 are based on bounds we developed for SpM×V [14]. Our key assumptions are as follows:

1. We bound time from below by considering only the cost of *memory* operations. We also assume write-back caches (true of the evaluation platforms) and sufficient store buffer capacity so that we can consider only loads and ignore the cost of stores.
2. Our model of execution time charges for cache and memory *latency*, as opposed to assuming that data can be retrieved from memory at the manufacturer's reported peak main memory bandwidth. We also assume accesses to the L1 cache commit at the maximum load/store commit rate.
3. As shown below, we can get a lower bound on memory costs by computing a lower bound on cache misses. Therefore, we consider only compulsory and capacity misses, *i.e.*, we ignore conflict misses. Also, we account for cache capacity and cache line size, but assume full associativity.

We refer the reader to our prior paper [14] and our full SpA^TA technical report [15] for a more careful justification of these assumptions.

Let T be total time of SpA^TA in seconds, and P the performance in Mflop/s:

$$P = \frac{4k}{T} \times 10^{-6} \tag{3}$$

where k is the number of non-zeros in the $m \times n$ sparse matrix A, *excluding* any fill. To get an *upper bound on performance*, we need a *lower bound* on T. We present our lower bound on T, based on Assumptions 1 and 2, in Section 3.1. Our lower bounds on cache misses (Assumption 3) are described in Section 3.2.

3.1 A Latency-Based Execution Time Model

Consider a machine with κ cache levels, where the access latency at cache level i is α_i seconds, and the memory access latency is α_{mem}. Suppose SpA^TA executes H_i cache accesses (or cache hits) and M_i cache misses at each level i, and that the total number of loads is L. We take the execution time T to be

$$T = \sum_{i=1}^{\kappa} \alpha_i H_i + \alpha_{\text{mem}} M_\kappa = \alpha_1 L + \sum_{i=1}^{\kappa-1} (\alpha_{i+1} - \alpha_i) M_i + \alpha_{\text{mem}} M_\kappa \tag{4}$$

where we use $H_1 = L - M_1$ and $H_i = M_{i-1} - M_i$ for $2 \leq i \leq \kappa$. According to Equation (4), we can minimize T by minimizing M_i, assuming $\alpha_{i+1} \geq \alpha_i$.

3.2 A Lower Bound on Cache Misses

Following Equation (4), we obtain a lower bound on M_i for $\mathrm{Sp}A^TA$ by counting compulsory and capacity misses but ignoring conflict misses. The bound is a function of the cache configuration and matrix data structure.

Let C_i be the size of each cache i in double-precision words, and let l_i be the line size, in doubles, with $C_1 \leq \ldots \leq C_\kappa$, and $l_1 \leq \ldots \leq l_\kappa$. Suppose γ integer indices use the same storage as 1 double. (We used 32-bit integers; thus, $\gamma = 2$.) Assume full associativity and complete user-control over cache data placement.

We describe the BCSR data structure as follows. Let $\hat{k} = \hat{k}(r,c)$ be the number of stored values, so the fill ratio is $\hat{k}/k$, and the number of stored blocks is $\frac{\hat{k}}{rc}$. Then, the total number of loads L is $L = L_A + L_x + L_y$, where

$$L_A = 2\left(\hat{k} + \frac{\hat{k}}{rc}\right) + \frac{m}{r} \qquad L_x = \frac{\hat{k}}{r} \qquad L_y = \frac{\hat{k}}{r}. \tag{5}$$

L_A contains terms for the values, block column indices, and row pointers; the factor of 2 accounts for reading A twice (once to compute Ax, and once for A^T times the result). L_x and L_y are the total number of loads required to read x and y, where we load c elements of each vector for each of the $\frac{\hat{k}}{rc}$ blocks.

To correctly model capacity misses, we compute the amount of data, or *working set*, required to multiply by a block row and its transpose. For the moment, assume that all block rows have the same number of $r \times c$ blocks; then, each block row has $\frac{\hat{k}}{rc} \times \frac{r}{m} = \frac{\hat{k}}{cm}$ blocks, or $\frac{\hat{k}}{m}$ non-zeros per row. Define the *matrix working set*, $\hat{W}$, to be the size of matrix data for a block row, and the *vector working set*, $\hat{V}$, to be the size of the corresponding vector elements for x and y:

$$\hat{W} = \frac{\hat{k}}{m}r + \frac{1}{\gamma}\frac{\hat{k}}{cm} + \frac{1}{\gamma} \quad , \quad \hat{V} = 2\hat{k}/m \tag{6}$$

For each cache i, we compute a lower bound on the misses M_i according to one of the following 2 cases, depending on the relative values of C_i, $\hat{W}$, and $\hat{V}$.

1. $\hat{W} + \hat{V} \leq C_i$: *Entire working set fits in cache.* Since there is sufficient cache capacity, we incur only compulsory misses on the matrix and vector elements:

$$M_i \geq \frac{1}{l_i}\left(\frac{m}{r}\hat{W} + 2n\right) \quad . \tag{7}$$

2. $\hat{W} + \hat{V} > C_i$: *The working set exceeds the maximum cache capacity.* In addition to the compulsory misses shown in Equation (7), we incur capacity misses for each element of the total working set that exceeds the capacity:

$$M_i \geq \frac{1}{l_i}\left[\frac{m}{r}\hat{W} + 2n + \frac{m}{r}(\hat{W} + \hat{V} - C_i)\right] \quad . \tag{8}$$

The factor of $\frac{1}{l_i}$ optimistically counts only 1 miss per cache line. We refer the reader to our full report for detailed derivations of Equations (7)–(8) [15].

4 Experimental Results and Analysis

We measured the performance of various SpA^TA implementations on 4 platforms (Table 1) and the 44 matrices of the original SPARSITY benchmark suite. Matrix 1 is a dense matrix in sparse format. Matrices 2–17 come from finite element method (FEM) applications: 2–9 have a structure dominated by a single block size aligned uniformly, while 10–17 have irregular block structure. Matrices 18–39 come from non-FEM applications, including chemical process simulation and financial modeling, among others. Matrices 40–44 arise in linear programming.

We use the PAPI hardware counter library (v2.1) to measure cycles and cache misses [4]. Figures 2–5 summarize the results, comparing performance (Mflop/s; y-axis) of the following 7 bounds and implementations for each matrix (x-axis):

- **Upper bound** (shown as a solid line): The fastest (highest value) of our performance upper bound, Equation (3), over all $r \times c$ block sizes up to 8×8.
- **PAPI upper bound** (shown by triangles): An upper bound in which we set L and M_i to the true load and miss counts measured by PAPI. The gap between the two bounds indicates how well Equations (5)–(8) reflect reality.
- **Best cache optimized, register blocked** implementation (squares): Performance of the best code over all $r \times c$, based on an exhaustive search.
- **Heuristically predicted implementation** (solid circles): Performance of the implementation predicted by our heuristic.
- **Register blocking only** (diamonds): Performance without algorithmic cache blocking, where the block size is chosen by exhaustive search.
- **Cache optimization only** (shown by asterisks): Performance of the code with only the algorithmic cache optimization, (*i.e.*, with $r = c = 1$).
- **Reference** (×'s): No cache or register-level optimizations have been applied.

Matrices which are small relative to the cache size have been omitted.

We draw the following 5 high-level conclusions based on Figures 2–5. More complete and detailed analyses appear in the full report [15].

1. *The cache optimization leads to uniformly good performance improvements.* On all platforms, applying the cache optimization, even without register blocking, leads to speedups ranging from up to 1.2× on the Itanium and Power3 platforms, to just over 1.6× on the Ultra 2i and Pentium III platforms.

2. *Register blocking and the cache optimization can be combined to good effect.* When combined, we observe speedups from 1.2× up to 4.2×. Moreover, the speedup relative to the register blocking only code is still up to 1.8×.

3. *Our heuristic always chooses a near-optimal block size.* Indeed, the performance of heuristic's block size is within 10% of the exhaustive best in all but four instances—Matrices 17, 21, and 27 on the Ultra 2i, and Matrix 2 on the Pentium III. There, the heuristic performance is within 15% of the best.

4. *Our implementations are within 20–30% of the PAPI upper bound for FEM matrices, but within only about 40–50% on other matrices.* The gap between actual performance and the upper bound is larger than what we observed previously for SpM×V and SpTS [14,16]. This result suggests that a larger payoff is expected from low-level tuning by using ATLAS/PHiPAC techniques.

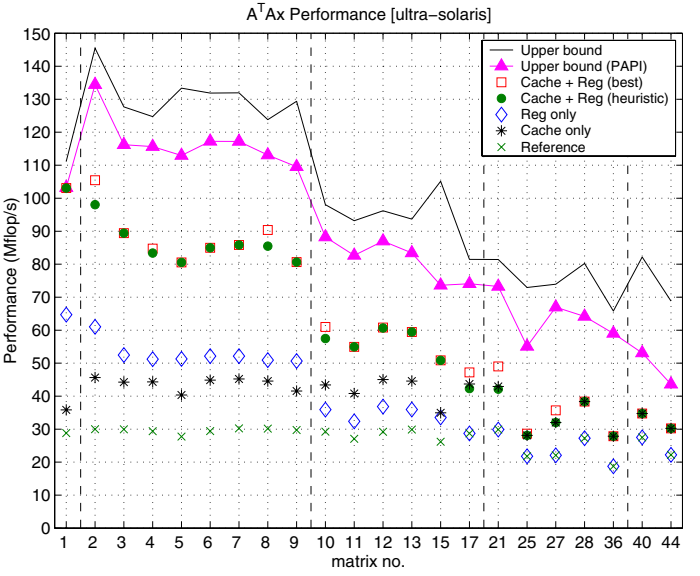

Fig. 2. SpA^TA performance on the Sun Ultra 2i platform.

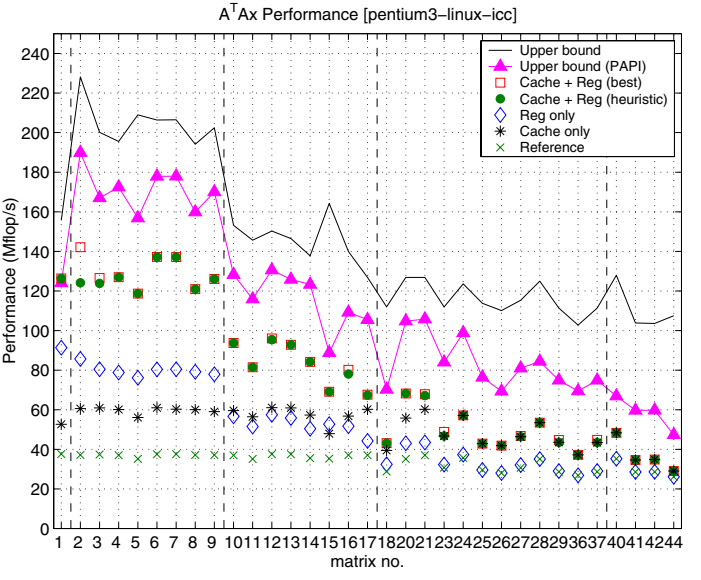

Fig. 3. SpA^TA performance on the Intel Pentium III platform.

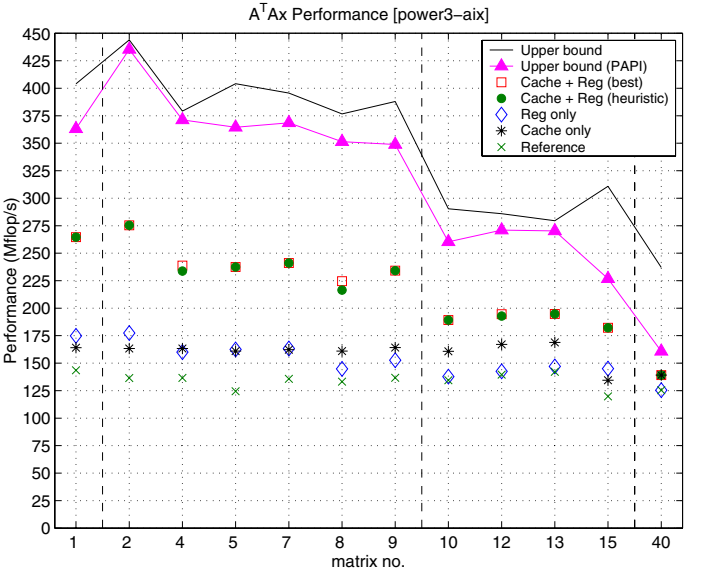

Fig. 4. SpA^TA performance on the IBM Power3 platform.

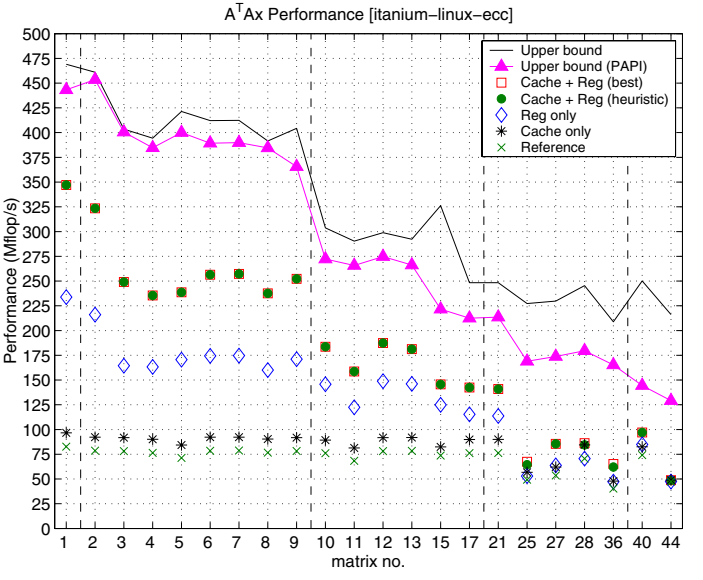

Fig. 5. SpA^TA performance on the Intel Itanium platform.

5. *Our analytic model of misses is accurate for FEM matrices, but less accurate for the others.* For the FEM matrices 1–17, the PAPI upper bound is typically within 10–15% of the analytic upper bound, indicating that our analytic model of misses is accurate in these cases. Matrices 18–44 have more random non-zero structure, so the gap between the analytic and PAPI upper bounds is larger due to our assumption of only 1 miss per cache line.

5 Related Work

There have been a number of sophisticated models proposed for analyzing memory behavior of SpM×V. Temam and Jalby [13], and Fraguela, *et al.* [6] have developed sophisticated probabilistic cache miss models for SpM×V, but assume uniform distribution of non-zero entries. To obtain lower bounds, we account only for conflict and capacity misses. Gropp, *et al.*, use bounds like the ones we develop to analyze and tune a computational fluid dynamics code [7]; Heber, *et al.*, analyze a sparse fracture mechanics code [8] on Itanium. We address matrices from a variety of applications, and furthermore, have proposed an explicit model execution time for sparse, streaming applications.

Work in sparse compilers, *e.g.*, Bik *et al.* [1] and the Bernoulli compiler [12], complements our own work. These projects address the expression of sparse kernels and data structures for code generation. One distinction of our work is our use of a hybrid off-line, on-line model for selecting transformations.

6 Conclusions and Future Directions

The speedups of up to 4.2× that we have observed indicate that there is tremendous potential to boost performance in applications dominated by SpA^TA. We are incorporating this kernel and these optimizations in an automatically tuned sparse library based on the Sparse BLAS standard [3].

Our upper bounds indicate that there is more room for improvement using low-level tuning techniques than with prior work on SpM×V and SpTS. Applying automated search-scheduling techniques developed in ATLAS and PHiPAC is therefore a natural extension. In addition, refined versions of our bounds could be used to study how performance varies with architectural parameters, in the spirit of SpM×V modeling work by Temam and Jalby [13].

Additional reuse is possible when multiplying by multiple vectors. Preliminary results on Itanium for sparse matrix-multiple-vector mltiplication show speedups of 6.5 to 9 [14]. This is a natural opportunity for future work with SpA^TA as well. We are exploring this optimization and other higher-level kernels with matrix reuse (*e.g.*, A^kx, matrix triple products).

Acknowledgements. We thank Michael de Lorimier for the SpATA initial implementations. This research was supported in part by the National Science Foundation under NSF Cooperative Agreement No. ACI-9813362, the Department of Energy under DOE Grant No. DE-FC02-01ER25478, and a gift from

Intel. The information presented here does not necessarily re.ect the position or the policy of the Government and no o.cial endorsement should be inferred.

References

1. A.J.C. Bik and H.A.G. Wijsho.. Automatic nonzero structure analysis. *SIAM Journal on Computing*, 28(5):1576–1587, 1999.
2. J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In Proceedings of the International Conference on Supercomputing, July 1997.
3. S. Blackford et al. Document for the Basic Linear Algebra Subprograms (BLAS) standard: BLAS Technical Forum, 2001. Chapter 3: `http://www.netlib.org/blast`.
4. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable crossplatform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, November 2000.
5. J.W. Demmel. Applied Numerical Linear Algebra. *SIAM*, 1997.
6. B.B. Fraguela, R. Doallo, and E.L. Zapata. Memory hierarchy performance prediction for sparse blocked algorithms. *Parallel Processing Letters*, 9(3), 1999.
7. W.D. Gropp, D.K. Kasushik, D.E. Keyes, and B.F. Smith. Towards realistic bounds for implicit CFD codes. In *Proceedings of Parallel Computational Fluid Dynamics*, pages 241–248, 1999.
8. G. Heber, A.J. Dolgert, M. Alt, K.A. Mazurkiewicz, and L. Stringer. Fracture mechanics on the intel itanium architecture: A case study. In *Workshop on EPIC Architectures and Compiler Technology* (ACM MICRO 34), Austin, TX, 2001.
9. E.-J. Im and K.A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of ICCS*, pages 127–136, May 2001.
10. J.M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.
11. Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.
12. P. Stodghill. A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes. PhD thesis, Cornell University, August 1997.
13. O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing '92*, 1992.
14. R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.
15. R. Vuduc, A. Gyulassy, J.W. Demmel, and K.A. Yelick. Memory hierarchy optimizations and performance bounds for sparse ATAx. Technical Report UCB/CS-03-1232, University of California, Berkeley, February 2003.
16. R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J.W. Demmel, and K.A. Yelick. Automatic performance tuning and analysis of sparse triangular solve. In *ICS 2002: POHLL Workshop*, New York, USA, June 2002.
17. W. Wang and D.P. O'Leary. Adaptive use of iterative methods in interior point methods for linear programming. Technical Report UMIACS-95-111, University of Maryland at College Park, College Park, MD, USA, 1995.
18. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. of Supercomp.*, Orlando, FL, 1998.