Automatic Performance Tuning and Analysis of Sparse Triangular Solve

Richard Vuduc Shoaib Kamil Jen Hsu Rajesh Nishtala
James W. Demmel Katherine A. Yelick

Computer Science Division, U.C. Berkeley
Berkeley, California, USA
`{richie,skamil,jenhsu,rajeshn,demmel,yelick}@cs.berkeley.edu`

1 Introduction

We address the problem of building high-performance uniprocessor implementations of sparse triangular solve (SpTS) automatically. This computational kernel is often the bottleneck in a variety of scientific and engineering applications that require the direct solution of sparse linear systems. Performance tuning of SpTS—and sparse matrix kernels in general—is a tedious and time-consuming task, because performance depends on the complex interaction of many factors: the performance gap between processors and memory, the limits on the scope of compiler analyses and transformations, and the overhead of manipulating sparse data structures. Consequently, it is not unusual to see kernels such as SpTS run at under 10% of peak uniprocessor floating point performance.

Our approach to automatic tuning of SpTS builds on prior experience with building tuning systems for sparse matrix-vector multiply (SpM×V) [21, 22, 40], and dense matrix kernels [8, 41]. In particular, we adopt the two-step methodology of previous approaches: (1) we identify and generate a set of reasonable candidate implementations, and (2) search this set for the fastest implementation by some combination of performance modeling and actually executing the implementations.

In this paper, we consider the solution of the sparse lower triangular system $Lx = y$ for a single dense vector x, given the lower triangular sparse matrix L and dense vector y.[1] We refer to x as the *solution* vector and y as the *right-hand side* (RHS). Many of the lower triangular factors we have observed from sparse LU factorization have a large, dense triangle in the lower right-hand corner of the matrix; this *trailing triangle* can account for as much as 90% of the matrix non-zeros. Therefore, we consider both algorithmic and data structure reorganizations which partition the solve into a sparse phase and a dense phase. To the sparse phase, we adapt the *register blocking optimization*, previously proposed for sparse matrix-vector multiply (SpM×V) in the Sparsity system [21, 22], to the SpTS kernel; to the dense phase, we make judicious use of highly tuned BLAS routines by switching to a dense implementation (*switch-to-dense* optimization). We describe fully automatic hybrid off-line/on-line heuristics for selecting the key tuning parameters: the register block size and the point at which to use the dense algorithm. (See Section 2.)

We then evaluate the performance of our optimized implementations relative to the fundamental limits on performance. Specifically, we first derive simple models of the upper bounds on the execution rate (Mflop/s) of our implementations. Using hardware counter data collected with the PAPI library [10], we then verify our models on three hardware platforms (Table 1) and a set of triangular factors from applications (Table 2). We observe that our optimized implementations can achieve 80% or more of these bounds; furthermore, we observe speedups of up to 1.8x when both register blocking and switch-to-dense optimizations are applied. We also present preliminary results confirming that our heuristics choose reasonable values for the tuning parameters.

These results support our prior findings with SpM×V [40], suggesting two new directions for performance enhancements: (1) the use of higher-level matrix structures (*e.g.*, matrix reordering and multiple register block sizes), and (2) optimizing kernels with more opportunities for data reuse (*e.g.*, multiplication and solve with multiple vectors, multiplication of $A^T A$ by a vector).

[1] We focus on the lower triangular case for concreteness; these results apply in the upper triangular case as well.

Property	Sun Ultra 2i	IBM Power3	Intel Itanium
Clock rate	333 MHz	375 MHz	800 MHz
Peak Main Memory Bandwidth	500 MB/s	1.6 GB/s	2.1 GB/s
Peak Flop Rate	667 Mflop/s	1.5 Gflop/s	3.2 Gflop/s
DGEMM ($n = 1000$)	425 Mflop/s	1.3 Gflop/s	2.2 Gflop/s
DGEMV ($n = 2000$)	58 Mflop/s	260 Mflop/s	345 Mflop/s
DTRSV ($n = 2000$)	59 Mflop/s	270 Mflop/s	320 Mflop/s
STREAM Triad Bandwidth [27]	250 MB/s	715 MB/s	1.1 GB/s
No. of FP regs (double)	16	32	128
L1 size / line size / latency	16 KB/16 B/2 cy	64 KB/128 B/1 cy	16 KB/32 B/2 cy (int)
L2 size / line size / latency	2 MB/64 B/7 cy	8 MB/128 B/9 cy	96 KB/64 B/6-9 cy
L3 size / line size / latency	N/A	N/A	2 MB/64 B/21-24 cy
TLB entries / page size	64/8 KB	256/4 KB	32 (L1), 96 (L2)/16 KB
Memory latency ($\approx$)	36-66 cy	35-139 cy	36-85 cy
sizeof(double) / sizeof(int)	8 B/4 B	8 B/4 B	8 B/4 B
Compiler	Sun C v6.1	IBM C v5.0	Intel C v5.0.1

Table 1: **Evaluation platforms**: Basic characteristics of the computing platforms on which we performed our performance evaluations. Most of these data were obtained from processor manuals [23, 4, 38]; memory access latencies were measured using the Saavedra microbenchmark [35]. Dense BLAS numbers are reported for ATLAS 3.2.0 [41] on the Ultra 2i, IBM ESSL v3.1 on the Power3, and the Intel Math Kernel Library v5.2 on Itanium.

				Nnz	Dense Trailing Triangle		
	Name	Application Area	Dimension	in L	Dim.	Density	% Total Nnz
1	dense	Dense matrix	1000	500500	1000	100.0%	100.0%
2	memplus	Circuit simulation	17758	1976080	1978	97.7%	96.8%
3	wang4	Device simulation	26068	15137153	2810	95.0%	24.8%
4	ex11	Fluid flow	16614	9781925	2207	88.0%	22.0%
5	raefsky4	Structural mechanics	19779	12608863	2268	100.0%	20.4%
6	goodwin	Fluid mechanics	7320	984474	456	65.9%	6.97%
7	lhr10	Chemical processes	10672	368744	104	96.3%	1.43%

Table 2: **Matrix benchmark suite**: The LU factorization of each matrix was computed using the sequential version of SuperLU 2.0 [14] and Matlab's column minimum degree ordering. The number of non-zeros in the resulting lower triangular (L) factor is shown. We also show the size of the trailing triangle found by our switch-point heuristic (column 6: see Section 2.3), its density (column 7: fraction of the trailing triangle occupied by true non-zeros), and the fraction of all matrix non-zeros contained within the trailing triangle (column 8).

2 Optimizations for Sparse Triangular Solve

The triangular matrices which arise in sparse Cholesky and LU factorization frequently have the kind of structure shown in Figure 1, spy plots of two examples of lower triangular factors. The lower right-most dense triangle of each matrix, which we call the *dense trailing triangle*, accounts for a significant fraction of the total number of non-zeros. In Figure 1 (*left*), the dimension of the entire factor is 17758 and the dimension of the trailing triangle is 2268; nevertheless, the trailing triangle accounts for 96% of all the non-zeros. Similarly, the trailing triangle of Figure 1 (*right*), contains approximately 20% of all matrix non-zeros; furthermore, the remainder of the matrix (the *leading trapezoid*) appears to consist of many smaller dense blocks and triangles.

We exploit this structure by decomposing $Lx = y$ into sparse and dense components:

$$\begin{pmatrix} L_1 & \\ L_2 & L_D \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{1}$$

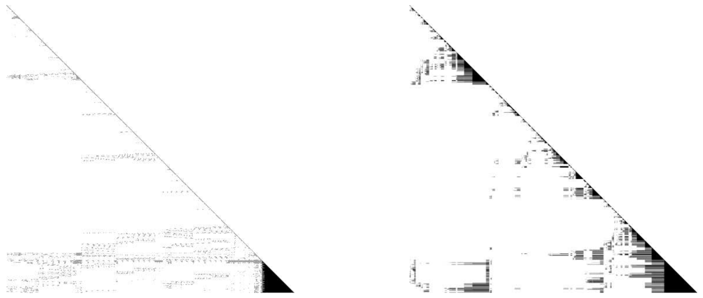


Figure 1: **Triangular matrix examples**: (*Left*) Matrix 2 (`memplus`) from Table 2 has a dimension of 17758. The dense trailing triangle, of size 1978, contains 96% of all the matrix non-zeros. (*Right*) Matrix 5 (`raefsky4`) from Table 2 has a dimension of 19779. The dense trailing triangle, of size 2268, accounts for 20% of all the matrix non-zeros.

where L_1 is a sparse $n_1 \times n_1$ lower-triangular matrix, L_2 is a sparse $n_2 \times n_1$ rectangular matrix, and L_D is a dense $n_2 \times n_2$ trailing triangle. We can solve for x_1 and x_2 in three steps:

$$L_1 x_1 = y_1 \tag{2}$$
$$\hat{y}_2 = y_2 - L_2 x_1 \tag{3}$$
$$L_D x_2 = \hat{y}_2 \tag{4}$$

Equation (2) is a SpTS, (2), equation (3) is a SpM×V, and equation (4) is a call to the tuned dense BLAS routine, DTRSV. (This process of splitting into sparse and dense components could be repeated for equation (2), but we do not consider this possibility in this paper.)

For reference, Figure 2 shows two common implementations of *dense* triangular solve: the row-oriented ("dot product") algorithm in Figure 2 (*left*), and the column-oriented ("axpy") algorithm in Figure 2 (*right*). The row-oriented algorithm is the basis for our register-blocked sparse algorithm; the column-oriented algorithm is essentially the reference implementation of the BLAS routine, DTRSV, and its details are important in our analysis in Section 3.

2.1 Improving register reuse: register blocking

The *register blocking* optimization [22] improves register reuse by reorganizing the matrix data structure into a sequence of "small" dense blocks; the block sizes are chosen to keep small blocks of the solution and RHS vectors in registers. An $n \times n$ sparse lower-triangular matrix in $b \times b$ register blocked format is divided logically into $\frac{n}{b} \times \frac{n}{b}$ submatrices, where each submatrix is of size $b \times b$. (In this paper, we consider only square block sizes.) For simplicity, assume that b divides n. For sparse matrices, only those blocks which contain at least one non-zero are stored. The computation of SpTS proceeds block-by-block. For each block, we can reuse the corresponding b elements of the RHS vector and b elements of the solution vector by keeping them in registers.

We use a blocked variant of the CSR storage format (BCSR).[2] Blocks within the same block row are stored consecutively, and the elements of each block are stored consecutively in row-major order. We store the diagonal blocks as full $b \times b$ blocks with explicit zeros above the diagonal.[3] A 2×2 example of BCSR is shown in Figure 3. When $r = c = 1$, BCSR reduces to CSR.[4] BCSR can store fewer column indices than CSR implementation (one per block instead of one per non-zero), reducing both storage and data structure manipulation overhead. Furthermore, we fully unroll the $b \times b$ submatrix computation, reducing loop overheads and exposing scheduling opportunities to the compiler. An example of the 2×2 code appears in Figure 4.

Figure 3 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 the explicit zeros) divided by the number of

[2]See Saad [34] or Barrett, *et al.* [6], for a survey of common sparse matrix formats.
[3]No computation is performed using these explicit zeros, however.
[4]The performance of our 1×1 code is comparable to that from the NIST Sparse BLAS [31].

true (or "ideal") non-zeros. We may trade-off extra computation (*i.e.*, fill ratio > 1) for improved efficiency in the form of uniform code and memory access.

2.2 Using the dense BLAS: switch-to-dense

To support the switch-to-dense optimization, we reorganize the sparse matrix data structure for L into two parts: a dense submatrix for the trailing triangle L_D, and a sparse component for the leading trapezoid. We store the trailing triangle in dense, unpacked column-major format as specified by the interface to DTRSV, and store the leading trapezoid in BCSR format as described above. We determine the column index at which to switch to the dense algorithm—the *switch-to-dense point* s (or simply, the *switch point*)—using the heuristic described below (Section 2.3).

2.3 Tuning parameter selection

In choosing values for the two tuning parameters—register block size b and switch point s—we first select the switch point, and then select the register block size. The switch point s is selected at run-time when the matrix is known. We choose s as follows, assuming the input matrix is stored in CSR format. Beginning at the diagonal element of the last row, we scan the bottom row until we reach *two consecutive* zero elements. The column index of this element marks the last column of the leading trapezoid.[5] Note that this method may select an s which causes additional fill-in of explicit zeros in the trailing triangle. As in the case of register blocking, tolerating some explicit fill can lead to some performance benefit. We are currently investigating a new selection procedure which evaluates the trade-off of gained efficiency versus fill to choose s.

To select the register block size b, we adapt the Sparsity v2.0 heuristic for SpM$\times$V [40] to SpTS. There are 3 steps. First, we collect a one-time *register profile* to characterize the platform. For SpTS, we evaluate the performance (Mflop/s) of the register blocked SpTS for all block sizes on a dense lower triangular matrix stored in sparse BCSR format. These measurements are independent of the sparse matrix, and therefore only need to be made once per architecture. Second, when the matrix is known at run-time, we estimate the fill for all block sizes. A recent paper [40] discusses a random sampling scheme for performing this step accurately and efficiently. Third, we select the block size b that maximizes

$$\text{Estimated Mflop/s} = \frac{\text{Mflop/s on dense matrix in BCSR for } b\times b \text{ blocking}}{\text{Estimated fill for } b\times b \text{ blocking}} \quad . \tag{5}$$

In principle, we could select different block sizes when executing the two sparse phases, equations (2) and (3); we only consider uniform block sizes in this paper.

The overhead of choosing the switch point, picking a register block size, and converting into our data structure, can all be performed once per triangular factor and amortized over multiple solves that use the same matrix non-zero pattern. This run-time overhead is the same amount of work as the analgous preprocessing step in Sparsity, whose cost is between 10–30 executions of naïve SpM$\times$V on the Itanium platform [40]. Thus, the optimizations we propose are most suitable when SpTS must be performed many times.

3 Performance Bounds and Evaluation

In this section, we derive upper bounds on the performance (in Mflop/s) of SpTS. We then evaluate both the accuracy of these bounds (Section 3.1) and the quality of our implementations with respect to these bounds (Section 3.2). We assume the notation of equations (1)–(4).

In deriving an analytic model of performance, we consider the sparse equations (2) and (3) separately from the dense solve, equation (4). We count the number of loads and stores required for

[5]Detecting the no-fill switch point is much easier if compressed sparse column (CSC) format is assumed. In fact, the dense trailing triangle can also be detected using symbolic structures (*e.g.*, the elimination tree) available during LU factorization. However, we do not assume access to such information. This assumption is consistent with the latest standardized Sparse BLAS interface [9], earlier interfaces [34, 31], and parallel sparse BLAS libraries [15].

```
    void dense_trisolve_dot( int n,
        const double* L, const double* y,
        double* x )
    {
      int i, j;
1     for( i = 0; i < n; i++ ) {
2       register double t = y[i];
3       for( j = 0; j < i; j++ )
4         t -= L[i+n*j]*x[j];
5       x[i] = t / L[i+n*i];
      }
    }
```

```
    void dense_trisolve_axpy( int n,
        const double* L, const double* y,
        double* x )
    {
      int i, j;
1     for( j = 0; j < n; j++ ) x[j] = 0;
2     for( j = 0; j < n; j++ ) {
3       register double t = (y[j] - x[j]) / L[j*n+j];
4       for( i = j+1; i < n; i++ )
5         x[i] = x[i] + L[i+n*j]*t;
6       x[j] = t;
      }
    }
```

Figure 2: **Dense triangular solve.** Reference implementations in C of (*left*) the row-oriented formulation, and (*right*) the column-oriented formulation of dense lower-triangular solve: $Lx = y$. In both routines, the matrix `L` is stored in unpacked column-major order. (For simplicity, the leading dimension is set to the matrix dimension, `n`, and the vectors are assumed to be unit-stride accessible.)

$$A = \begin{pmatrix} a_{00} & & & & & \\ a_{10} & a_{11} & & & & \\ a_{20} & 0 & a_{22} & & & \\ a_{30} & a_{31} & a_{32} & a_{33} & & \\ a_{40} & a_{41} & 0 & 0 & a_{44} & \\ a_{50} & a_{51} & 0 & 0 & 0 & a_{55} \end{pmatrix}$$

$\texttt{b\_row\_ptr} = \begin{pmatrix} 0 & 1 & 3 & 5 \end{pmatrix}$, $\texttt{b\_col\_ind} = \begin{pmatrix} 0 & 0 & 2 & 0 & 4 \end{pmatrix}$, $\texttt{b\_values} =$
$\begin{pmatrix} a_{00} & 0 & a_{10} & a_{11} & a_{20} & 0 & a_{30} & a_{31} & a_{22} & 0 & a_{32} & a_{33} & a_{40} & a_{41} & a_{50} & a_{51} & a_{44} & 0 & 0 & a_{55} \end{pmatrix}$

Figure 3: **Block compressed sparse row (BCSR) storage format.** Here, we see how a 6×6 sparse lower-triangular matrix is stored in 2×2 BCSR format. BCSR uses three arrays. The elements of each dense 2×2 block are stored contiguously in the `b_values` array. Only the first column index of the (0,0) entry of each block is stored in `b_col_ind` array; the `b_row_ptr` array points to block row starting positions in the `b_col_ind` array. Each 2×2 block of values is stored in row-major order.

```
     void sparse_trisolve_BCSR_2x2( int n, const int* b_row_ptr,
         const int* b_col_ind, const double* b_values,
         const double* y, double* x )
     {
       int I, JJ;     assert( (n\%2) == 0 );

1      for( I = 0; I < n/2; I++) { // loop over block rows
2        register double t0 = y[2*I], t1 = y[2*I+1];
3        for( JJ = b_row_ptr[I]; JJ < (b_row_ptr[I+1]-1); JJ++ ) {
4a         int j0 = b_col_ind[ JJ ];
4b         register double x0 = x[j0], x1 = x[j0+1];

4c         t0 -= b_values[4*JJ+0] * x0; t1 -= b_values[4*JJ+1] * x1;
4d         t0 -= b_values[4*JJ+2] * x0; t1 -= b_values[4*JJ+3] * x1;
         }
5a       x[2*I] = t0 / b_values[4*JJ+0];
5b       x[2*I+1] = (t1 - (b_values[4*JJ+2]*x[2*I])) / b_values[4*JJ+3];
       }
     }
```

Figure 4: **Register-blocked sparse triangular solve.** A 2×2 register-blocked example of sparse lower triangular solve, assuming BCSR format. For simplicity, the dimension `n` is assumed to be a multiple of the block size in this example. Note that the matrix blocks are stored in row-major order, and the diagonal block is assumed (1) to be the last block in each row, and (2) to be stored as an unpacked (2×2) block. Lines are numbered as shown to illustrate the mapping between this implementation and the corresponding dense implementation of Figure 2 (*left*).

$b{\times}b$ register blocking as follows. Let k be the total number of non-zeros in L_1 and L_2,[6] and let f_b be the fill ratio after register blocking. Then the number of loads required is

$$\text{Loads}_{\text{sparse}}(b) = \underbrace{kf_b + \frac{kf_b}{b^2} + \left\lceil \frac{n}{b} \right\rceil + 1}_{\text{matrix}} + \underbrace{\frac{kf_b}{b}}_{\text{soln vec}} + \underbrace{n}_{\text{RHS}} = kf_b\left(1 + \tfrac{1}{b^2} + \tfrac{1}{b}\right) + n + \left\lceil \tfrac{n}{b} \right\rceil + 1 \quad . \tag{6}$$

We include terms for the matrix (all non-zeros, one column index per non-zero block, and $\lceil n/b \rceil + 1$ row pointers), the solution vector, and the RHS vector. The number of stores is $\text{Stores}_{\text{sparse}} = n$.

To analyze the dense part, we first assume a column-oriented ("axpy") algorithm for DTRSV. We *model* the number of loads and stores required to execute equation (4) as

$$\text{Loads}_{\text{dense}} = \underbrace{\frac{n_2\,(n_2+1)}{2}}_{\text{matrix}} + \underbrace{\frac{n_2}{2}\left(\frac{n_2}{R}+1\right)}_{\text{solution}} + \underbrace{n_2}_{\text{RHS}} \quad , \quad \text{Stores}_{\text{dense}} = \underbrace{\frac{n_2}{2}\left(\frac{n_2}{R}+1\right)}_{\text{solution}}, \tag{7}$$

where the $1/R$ factors model register-level blocking in the dense code, assuming $R{\times}R$ register blocks.[7] In general, we do not know R if we are calling a proprietary vendor-supplied library; however, we estimate R by examining load/store hardware counters when calling DTRSV.

We model execution time as follows. First, we assume that the entire solve operation is memory-bound so that we can hide the cost of flops. Let h_i be the number of hits at cache level i during the entire solve operation, and let m_i be the number of misses. Then we model execution time T as

$$T = \sum_{i=1}^{\kappa-1} h_i \alpha_i + m_\kappa \alpha_{\text{mem}}, \tag{8}$$

where α_i is the access time (in seconds) at cache level i, κ is the level of the largest cache, and α_{mem} is the memory access time. This model assumes we must incur the cost of a full access latency for each load and store. The number of L1 hits h_1 is given by

$$h_1 = \text{Loads}_{\text{sparse}}(b) + \text{Stores}_{\text{sparse}}(b) + \text{Loads}_{\text{dense}} + \text{Stores}_{\text{dense}} - m_1 \quad . \tag{9}$$

Assuming a perfect nesting of the caches, we define $h_{i+1} = m_i - m_{i+1}$ for $1 \le i < \kappa$.

Counting each division as 1 flop, the total number of flops is $2(k-n)$ multiplies and adds plus n divisions, or $2k-n$ flops. The performance in Mflop/s is therefore $(2k-n)/T \cdot 10^{-6}$.

Next, we count the number of misses m_i, starting at L1. Let l_1 be the L1 line size, in doubles. We incur compulsory misses for every matrix line. The solution and RHS vector miss counts are more complicated. In the best case, these vectors fit into cache with no conflict misses; we incur only the $2n$ compulsory misses for the two vectors. Thus, a lower bound $M^{(1)}_{\text{lower}}$ on L1 misses is

$$M^{(1)}_{\text{lower}}(b) = \frac{1}{l_1}\left[kf_b\left(1 + \frac{1}{\gamma b^2}\right) + \frac{1}{\gamma}\left(\left\lceil \frac{n}{b} \right\rceil + 1\right) + \left(2n + \frac{n_2\,(n_2+1)}{2}\right)\right]. \tag{10}$$

where the size of one double-precision value equals γ integers. The factor of $1/l_1$ accounts for the L1 line size. To compute $M^{(i)}_{\text{lower}}(b)$ at cache levels $i > 1$, we simply substitute the right line size. In the worst case, we miss on every access to a line of the solution vector; thus, a miss upper bound is

$$M^{(1)}_{\text{upper}}(b) = \frac{1}{l_1}\left[kf_b\left(1 + \frac{1}{\gamma b^2}\right) + \frac{1}{\gamma}\left(\left\lceil \frac{n}{b} \right\rceil + 1\right) + \left(\frac{kf_b}{b} + n + \text{Loads}_{\text{dense}} + \text{Stores}_{\text{dense}}\right)\right]. \tag{11}$$

Finally, we calculate an *upper bound on performance*, in Mflop/s, by substituting the lower bound on misses, equation (10), into equation (8). Similarly, we compute a *lower bound on performance* by substituting equation (11) into equation (8). Interaction with the TLB complicates our performance model. We incorporate the TLB into our performance upper bound by letting α_{mem} be the *minimum* memory access latency shown in Table 1. This latency assumes a memory access but also a TLB hit. For the lower bound, we assume α_{mem} is the maximum memory access latency shown in Table 1, a memory access and a TLB miss.

[6]For a dense matrix stored in sparse format, we only need to call DTRSV; in the model, n_1 and k would be 0.

[7]The terms with R in them are derived by assuming R vector loads per register block. Assuming R divides n_2, there are a total of $\frac{n_2/R(n_2/R+1)}{2}$ blocks.

3.1 Validating the cache miss bounds

We used our heuristic procedure for selecting the switch point s. Keeping s fixed, we then performed an exhaustive search over all register block sizes up to $b = 5$, for all matrices and platforms, measuring execution time and cache hits and misses using PAPI. Figures 5–7 validate our bounds on misses, equations (10) and (11). In particular, for the larger cache sizes, the vector lengths are such that the true miss counts are closer to equation (10) than (11), implying that conflict misses can be ignored.

3.2 Evaluating optimized SpTS performance

Figures 8–10 compare the observed performance of various implementations to the bounds derived above (Section 3). In particular, we compare the following:

- Reference (1×1) implementation (shown as asterisks).
- The best implementation using only register blocking for all $1 \leq b \leq 5$ (solid squares).
- An implementation using only the switch-to-dense optimization (hollow triangles).
- The combined register blocking and switch-to-dense implementation (solid circles).
- Our analytic lower and upper performance bounds (Mflop/s) as computed in Section 3 (upper bound shown as dash-dot lines, lower bound as dashed lines). In particular, at each point we show the *best* bound over all b, with the switch point s fixed at the heuristic-selected value.
- A PAPI-based performance upper bound (solid triangles). This bound was obtained by substituting measured cache hit and miss data from PAPI into equation (8) and using the minimum memory latency for α_{mem}.

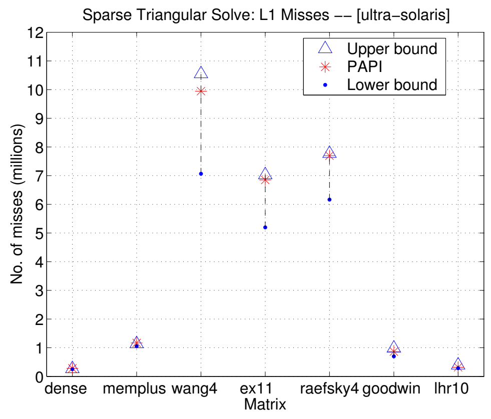
The switch points found by the heuristic are shown for each matrix in Table 2. The heuristic does select a reasonable switch point—yielding true non-zero densities of 85% or higher in the trailing triangle—in all cases except Matrix 6 (`goodwin`). Also, although Figures 8–10 show the performance using the best register block size, the heuristics described in Section 2.3 chose the optimal block size in all cases on all platforms *except* for Matrix 2 (`memplus`) running on the Ultra 2i and Itanium. Nevertheless, the performance (Mflop/s) at the sizes chosen in these cases was within 6% of the best.

The best implementations achieve speedups of up to 1.8 over the reference implementation. Furthermore, they attain a significant fraction of the upper bound performance (Mflop/s). On the Ultra 2i, the implementations achieve 75% up to 85% of the upper bound performance; on the Itanium, 80–95%; and about 80–85% on the Power3. On the Itanium in particular, we observe performance that is very close to the estimated bounds. The vendor implementation of DTRSV evidently exceeds our bounds. We are currently investigating this phenomenon. We know that the compiler (and, it is likely, the vendor DTRSV) uses prefetching instructions. If done properly, we would expect this to invalidate our charging for the full latency cost in equation (8), allowing us to move data at rates approaching memory bandwidth instead.

In two cases—Matrix 5 (`raefsky4`) on the Ultra 2i and Itanium platforms—the combined effect of register blocking and the switch-to-dense call significantly improves on either optimization alone. On the Ultra 2i, register blocking alone achieves a speedup of 1.29, switch-to-dense achieves a speedup of 1.48, and the combined implementation yields a speedup of 1.76. On the Itanium, the register blocking-only speedup is 1.24, switch-to-dense-only is 1.51, and combined speedup is 1.81.

However, register blocking alone generally does not yield significant performance gains for the other matrices. In fact, on the Power3, register blocking has almost no effect, whereas the switch-to-dense optimization performs very well. We observed that Matrices 3 (`wang4`), 4 (`ex11`), 6 (`goodwin`), and 7 (`lhr10`), none of which benefit from register blocking, all have register blocking fill ratios exceeding 1.35 when using the smallest non-unit block size, 2×2. The other matrices have fill ratios of less than 1.1 with up to 3×3 blocking. Two significant factors affecting the fill are (1) the choice of square block sizes and (2) imposition of a uniform grid. Non-square block sizes and the use of variable block sizes may be viable alternatives to the present scheme.

Note that our upper bounds are computed with respect to our particular register blocking and switch-to-dense data structure. It is possible that other data structures (*e.g.*, those that remove the uniform block size assumption and therefore change the dependence of f_b on b) could do better.



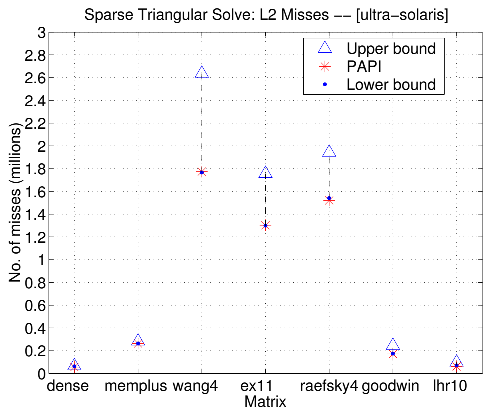
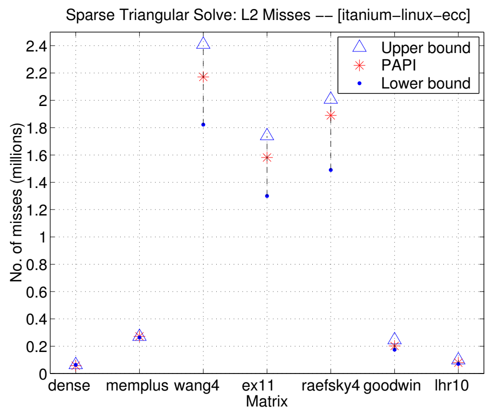


Figure 5: **Miss model validation (Sun Ultra 2i)**: Our upper and lower bounds on L1 and L2 cache misses compared to PAPI measurements. The bounds match the data well. The true L2 misses match the lower bound well in the larger (L2) cache, suggesting the vector sizes are small enough that conflict misses play a relatively minor role.



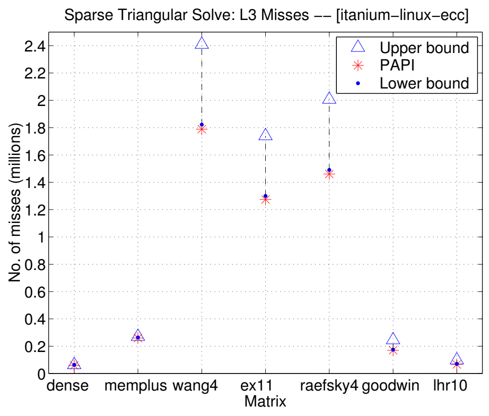
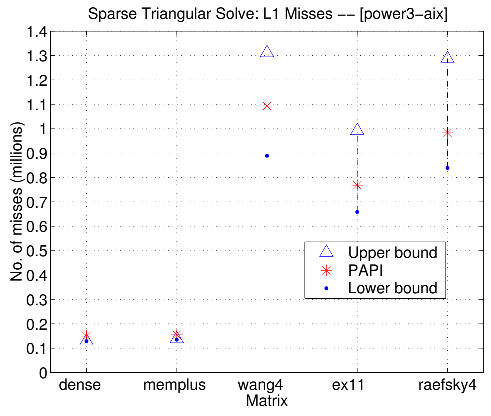


Figure 6: **Miss model validation (Intel Itanium)**: Our upper and lower bounds on L1 and L2 cache misses compared to PAPI measurements. The bounds match the data well. As with Figure 5, equation (10) is a good match to the measured misses for the larger (L3) cache.



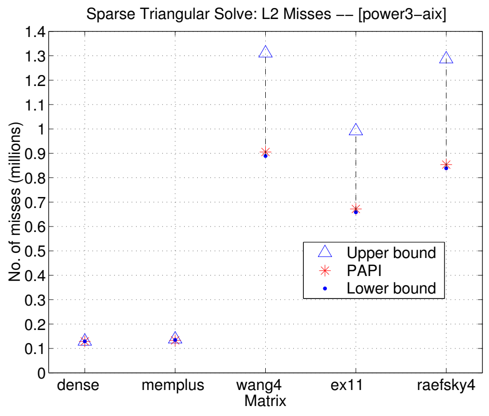


Figure 7: **Miss model validation (IBM Power3)**: Our upper and lower bounds on L1 and L2 cache misses compared to PAPI measurements. Note that two matrices have been omitted since they fit approximately within the large (8 MB) L2 cache.

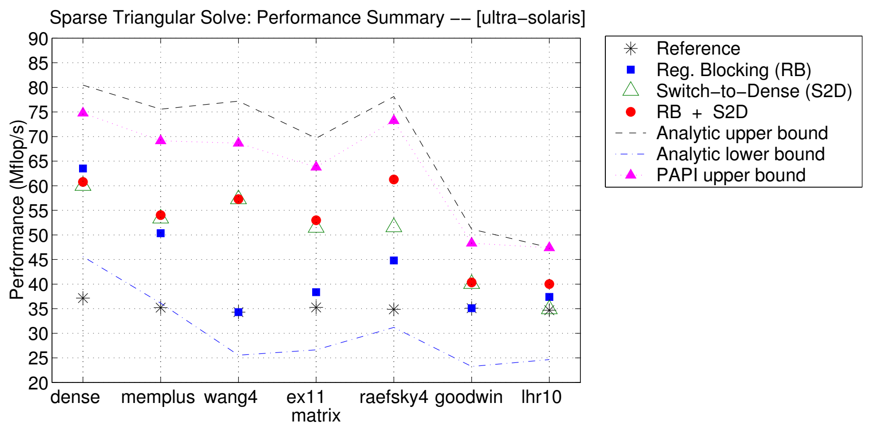


Figure 8: **Performance summary (Sun Ultra 2i)**: Performance (Mflop/s) shown for the seven items listed in Section 3.2. The best codes achieve 75–85% of the performance upper bound.

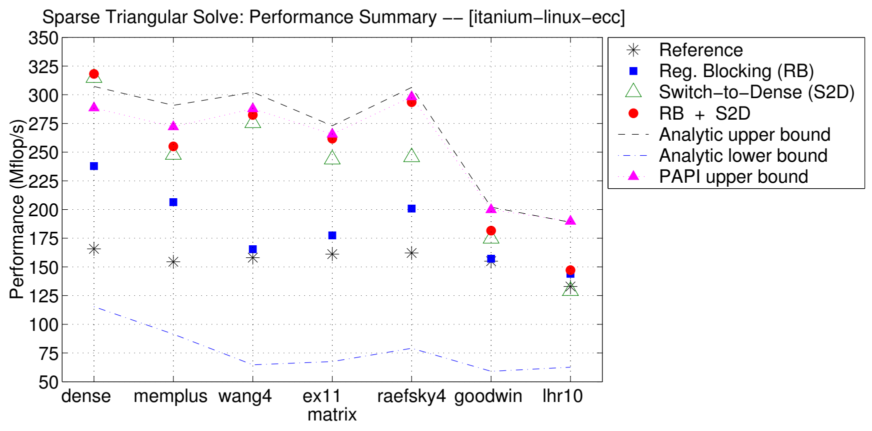


Figure 9: **Performance summary (Intel Itanium)**: Performance (Mflop/s) for the seven implementations listed in Section 3.2. The best implementations achieve 85–95% of the upper bound.

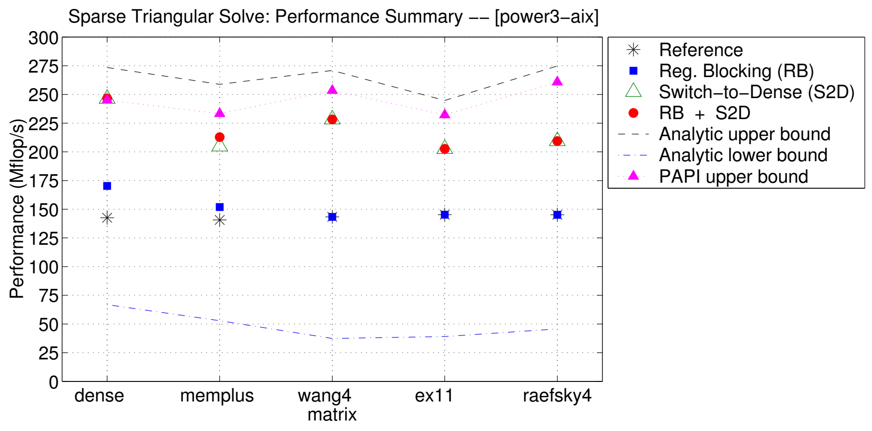


Figure 10: **Performance summary (IBM Power3)**: Performance (Mflop/s) of our SpTS implementations relative to various performance bounds (see Section 3.2). Register blocking evidently does not yield any significant performance improvements. The switch-to-dense optimization, however, tightens the performance gap. Note that two matrices have been omitted since they fit approximately within the large (8 MB) L2 cache.

4 Related work

Sparse triangular solve is a key component in many of the existing serial and parallel direct sparse factorization libraries (*e.g.*, SuperLU [14], MUMPS [2], UMFPACK [13], PSPASES [24], and SPOOLES [5], among others). These libraries have focused primarily on speeding up the factorization step, and employ sophisticated methods for creating dense structure. Efforts to speedup the triangular solve step in these and other work [33, 32, 25, 18, 26, 36, 19, 1, 30] have focused on improving parallel scalability, whereas we address uniprocessor tuning exclusively here.

For dense algorithms, a variety of sophisticated static models for selecting transformations and tuning parameters have been developed [12, 17, 28, 11, 42]. However, it is difficult to apply these analyses directly to sparse matrix kernels due to the presence of indirect and irregular memory access patterns. Nevertheless, there have been a number of notable modeling attempts in the sparse case. Temam and Jalby [39], Heras, *et al.* [20], and Fraguela, *et al.* [16] have developed sophisticated probabilistic cache miss models, but assume uniform distribution of non-zero entries. These models vary in their ability to account for self- and cross-interference misses. In this study, we see that on current and future machines, whose cache sizes continue to grow, conflict misses do not contribute significantly to accurate miss modeling.

Work in sparse compilers, *e.g.*, Bik *et al.* [7] and the Bernoulli compiler [37], complements our own work. These projects focus on the expression of sparse kernels and data structures for code generation, and will likely prove valuable to generating our implementations. One distinction of our work is our use of a hybrid off-line, on-line model for selecting transformations (tuning parameters).

5 Conclusions and Future Directions

The results of this paper raise a number of questions for future work in tuning sparse triangular solve, and future sparse matrix kernels.

- The performance of our implementations approaches upper-bounds on a variety of architectures, suggesting that additional gains from low-level tuning (*e.g.*, instruction scheduling) will be limited. Instead we are examining other algorithmic ways of improving reuse, for instance, via the use of multiple right-hand sides [29, 16, 22]. Preliminary results on Itanium for sparse matrix-multiple-vector mltiplication show speedups of 6.5 to 9 over single-vector code [40].
- Register blocking with square blocks on a uniformly aligned grid appears to be too limiting to see appreciable performance benefits. Encouraged by the gains from the switch-to-dense algorithm, we expect variable blocking or more intelligently guided use of dense structures (*e.g.*, using elimination trees) to be promising future directions. Existing direct solvers make use of such higher-level information in their forward and backward substitution phases; comparisons to these implementations is forthcoming.
- The success of the switch-to-dense optimization on our test problems may be particular to our choice of fill-reducing ordering. We are currently investigating other contexts—namely, under other ordering schemes, and in incomplete Cholesky and incomplete LU preconditioners.
- At present, we treat selection of the dense triangle as a property of the matrix only, and not of the architecture. We are investigating whether there is any benefit to making this platform-dependent as well, by off-line profiling of performance as a function of density, and incorporating that information into switch point selection.
- Our execution time model assumes that our solver does not run at peak memory bandwidth speeds, and therefore charges for the full latency. The performance results on Itanium, which are very close to the upper bound model, raise the issue of whether we can abandon this assumption in practice and approach memory bandwidth speeds, perhaps via prefetching.
- The model of operation in both this work and prior work on the Sparsity system assume one-time, off-line preprocessing, run-time searching, and amortizing the cost of run-time preprocessing over many solves. Furthermore, we make assumptions about the format of the input matrix. These issues raise general questions about what the interface between applications—either user code or existing solver libraries—and automatically tuned libraries should be.

References

[1] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal on Scientific Computing*, 14(2):446–460, March 1993.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] B. S. Andersen, F. Gustavson, A. Karaivanov, J. Wasniewski, and P. Y. Yalamov. LAWRA–Linear Algebra With Recursive Algorithms. In *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, Kazimierz Dolny, Poland, September 1999.

[4] S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo. *RS/6000 Scientific and Technical Computing: Power3 Introduction and Tuning*. International Business Machines, Austin, TX, USA, 1998. `www.redbooks.ibm.com`.

[5] C. Ashcraft and R. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, USA, 1994.

[7] A. J. C. Bik and H. A. G. Wijshoff. Automatic nonzero structure analysis. *SIAM Journal on Computing*, 28(5):1576–1587, 1999.

[8] 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*, Vienna, Austria, July 1997. ACM SIGARC. see `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[9] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, and J. W. von Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) standard: BLAS Technical Forum, 2001. `www.netlib.org/blast`.

[10] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, November 2000.

[11] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing*, pages 114–124, 1992.

[12] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297, Snowbird, UT, USA, June 2001.

[13] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse lu factorization. *SIAM Journal on Matrix Analysis and Applications*, 19(1):140–158, 1997.

[14] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[15] S. Filippone and M. Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software*, 26(4):527–550, December 2000.

[16] B. B. Fraguela, R. Doallo, and E. L. Zapata. Memory hierarchy performance prediction for sparse blocked algorithms. *Parallel Processing Letters*, 9(3), March 1999.

[17] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[18] A. Gupta and V. Kumar. Parallel algorithms for forward elimination and backward substitution in direct solution of sparse linear systems. In *Supercomputing*, 1995.

[19] M. T. Heath and P. Raghavan. The performance of parallel sparse triangular solution. In S. Schreiber, M. T. Heath, and A. Ranade, editors, *Proceedings of the IMA Workshop for Algorithms for Parallel Processing*, volume 105, pages 289–306. Springer-Verlag, 1998.

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

[21] E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication.* PhD thesis, University of California, Berkeley, May 2000.

[22] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 127–136. Springer, May 2001.

[23] Intel. Intel Itanium Processor Reference Manual for Software Optimization, November 2001.

[24] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and F. Gustavson. PSPASES: An efficient and scalable parallel sparse direct solver. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[25] M. V. Joshi, , A. G. G. Karypis, and V. Kumar. A high performance two dimensional scalable parallel algorithm for solving sparse triangular systems. In *Proceedings of the 4th International Conference on High Performance Computing*, December 1997.

[26] G. Li and T. F. Coleman. A parallel triangular solver for a distributed-memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, pages 485–502, May 1998.

[27] J. D. McCalpin. STREAM: Measuring sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream`.

[28] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[29] J. J. Navarro, E. García, J. L. Larriba-Pey, and T. Juan. Algorithms for sparse matrix computations on high-performance workstations. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 301–308, Philadelpha, PA, USA, May 1996.

[30] P. Raghavan. Efficient parallel triangular solution using selective inversion. *Parallel Processing Letters*, 8(1):29–40, 1998.

[31] K. Remington and R. Pozo. NIST Sparse BLAS: User's Guide. Technical report, NIST, 1996. `gams.nist.gov/spblas`.

[32] E. Rothberg. Alternatives for solving sparse triangular systems on distributed-memory multiprocessors. *Parallel Computing*, 21(7):1121–1136, 1995.

[33] E. Rothberg and A. Gupta. Parallel ICCG on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck. *Parallel Computing*, 18(7):719–741, July 1992.

[34] Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.

[35] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking.* PhD thesis, University of California, Berkeley, February 1992.

[36] E. E. Santos. Solving triangular linear systems in parallel using substitution. In *Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 553–560, October 1995.

[37] P. Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes.* PhD thesis, Cornell University, August 1997.

[38] Sun-Microsystems. UltraSPARC IIi: User's Manual, 1999.

[39] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing '92*, 1992.

[40] 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, 2002. *(To appear.)*.

[41] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. of Supercomp.*, 1998.

[42] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.