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] 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 $y$. 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 [10], 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.
<table>
<thead>
<tr>
<th>Property</th>
<th>Sun Ultra 2i</th>
<th>IBM Power3</th>
<th>Intel Itanium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock rate</td>
<td>333 MHz</td>
<td>375 MHz</td>
<td>800 MHz</td>
</tr>
<tr>
<td>Peak Main Memory Bandwidth</td>
<td>500 MB/s</td>
<td>1.6 GB/s</td>
<td>2.1 GB/s</td>
</tr>
<tr>
<td>Peak Flop Rate</td>
<td>667 Mflop/s</td>
<td>1.5 Gflop/s</td>
<td>3.2 Gflop/s</td>
</tr>
<tr>
<td>DGEMM ($n = 1000$)</td>
<td>425 Mflop/s</td>
<td>1.3 Gflop/s</td>
<td>2.2 Gflop/s</td>
</tr>
<tr>
<td>DGEMV ($n = 2000$)</td>
<td>58 Mflop/s</td>
<td>260 Mflop/s</td>
<td>345 Mflop/s</td>
</tr>
<tr>
<td>DTRSV ($n = 2000$)</td>
<td>59 Mflop/s</td>
<td>270 Mflop/s</td>
<td>320 Mflop/s</td>
</tr>
<tr>
<td>STREAM Triad Bandwidth</td>
<td>250 MB/s</td>
<td>715 MB/s</td>
<td>1.1 GB/s</td>
</tr>
<tr>
<td>No. of FP regs (double)</td>
<td>16</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>L1 size / line size / latency</td>
<td>16 KB/16 B/2 cy</td>
<td>64 KB/128 B/1 cy</td>
<td>16 KB/32 B/2 cy (int)</td>
</tr>
<tr>
<td>L2 size / line size / latency</td>
<td>2 MB/64 B/7 cy</td>
<td>8 MB/128 B/9 cy</td>
<td>96 KB/64 B/6-9 cy</td>
</tr>
<tr>
<td>L3 size / line size / latency</td>
<td>N/A</td>
<td>N/A</td>
<td>2 MB/64 B/21-24 cy</td>
</tr>
<tr>
<td>TLB entries / page size</td>
<td>64/8 KB</td>
<td>256/64 KB</td>
<td>32 (L1), 96 (L2)/16 KB</td>
</tr>
<tr>
<td>Memory latency ($\approx$)</td>
<td>36-66 cy</td>
<td>35-139 cy</td>
<td>36-85 cy</td>
</tr>
<tr>
<td>sizeof(double) / sizeof(int)</td>
<td>8 B/4 B</td>
<td>8 B/4 B</td>
<td>8 B/4 B</td>
</tr>
<tr>
<td>Compiler</td>
<td>Sun C v6.1</td>
<td>IBM C v5.0</td>
<td>Intel C v5.0.1</td>
</tr>
</tbody>
</table>

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, 13, 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.

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:

\[
\begin{pmatrix}
L_1 \\
L_2 \\
L_D
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2
\end{pmatrix} =
\begin{pmatrix}
y_1 \\
y_2
\end{pmatrix}
\]

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Application Area</th>
<th>Dimension</th>
<th>Nnz in L</th>
<th>Dense Trailing Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dense</td>
<td>1000</td>
<td>500500</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>memplus</td>
<td>17758</td>
<td>1976080</td>
<td>1978</td>
</tr>
<tr>
<td>3</td>
<td>wang4</td>
<td>26068</td>
<td>15137153</td>
<td>2810</td>
</tr>
<tr>
<td>4</td>
<td>ex11</td>
<td>16614</td>
<td>9781925</td>
<td>2207</td>
</tr>
<tr>
<td>5</td>
<td>raefsky4</td>
<td>19779</td>
<td>12608863</td>
<td>2268</td>
</tr>
<tr>
<td>6</td>
<td>goodwin</td>
<td>7320</td>
<td>984474</td>
<td>456</td>
</tr>
<tr>
<td>7</td>
<td>lhr10</td>
<td>10672</td>
<td>368744</td>
<td>104</td>
</tr>
</tbody>
</table>

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).
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.

$L_D$ is a dense trailing triangle of dimension $n_2$. We can solve for $x_1$ and $x_2$ in three steps:

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

Equation (2) is a SpTS, (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). 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. A 2×2 example of BCSR is shown in Figure 3. When $r = c = 1$, BCSR reduces to CSR. 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

---


3No computation is performed using these explicit zeros, however.

4The performance of our 1×1 code is comparable to that from the NIST Sparse BLAS [23].
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 the second zero element. 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 adopt the SPARSITY v2.0 heuristic for SpM×V\(^{10} \) 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

\[
\text{Estimated Mflop/s} = \frac{\text{Mflop/s on dense matrix in BCSR for } b \times b \text{ blocking}}{\text{Estimated fill for } b \times b \text{ blocking}}.
\]  

(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 analogous preprocessing step in SPARSITY, whose cost is between 10–30 executions of naïve SpM×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 3.2) and the quality of our implementations with respect to these bounds (Section 3.3). We assume the notation of equations (1)–(4).

3.1 Bounds based on modeling cache misses

In deriving an analytic model of performance, we consider the sparse equations (2) and (3) separately from the dense solve, equation (1). We count the number of loads and stores required for \( b \times b \) register

---

\(^5\)Of course, 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 that we have access to such information. This assumption is consistent with the latest standardized Sparse BLAS interface\(^4 \), similar earlier interfaces\(^{34, 31} \), and parallel sparse BLAS libraries\(^{17} \).
void dense_trisolve_dot( int n, 
    const double* L, const double* y, 
    double* x )
{
    int i, j;
    for( i = 0; i < n; i++ ) {
        register double t = y[i];
        for( j = 0; j < i; j++ )
            t -= L[i+n*j]*x[j];
        x[i] = t / L[i+n*i];
    }
}

void dense_trisolve_axpy( int n, 
    const double* L, const double* y, 
    double* x )
{
    int i, j;
    for( i = 0; i < n; i++ ) x[i] = 0;
    for( i = 0; i < n; i++ ) {
        register double t = (y[i] - x[i]) / L[i*n+i];
        for( j = i+1; j < n; j++ )
            x[j] = x[j] + L[i+n*j]*t;
        x[i] = 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.)

![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.)](image)

$$A = \begin{pmatrix}
    a_{00} & a_{10} & a_{11} & 0 & a_{22} \\
    a_{01} & a_{20} & 0 & a_{32} & a_{33} \\
    a_{02} & a_{11} & a_{31} & 0 & a_{44} \\
    a_{03} & a_{12} & a_{22} & a_{43} & a_{44} \\
    a_{04} & a_{13} & a_{23} & a_{34} & a_{44} & a_{55}
\end{pmatrix}$$

$b_{row\_ptr} = ( 0 \ 1 \ 3 \ 5 ), \quad b_{col\_ind} = ( 0 \ 0 \ 2 \ 0 \ 4 ), \quad b_{values} = \begin{pmatrix}
    a_{00} & 0 & a_{10} & a_{11} & a_{20} & 0 & a_{30} & a_{31} & a_{40} & a_{41} & a_{50} & a_{51} & a_{44} & 0 & 0 & a_{55}
\end{pmatrix}$

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 the $b_{col\_ind}$ 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.

![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 the $b_{col\_ind}$ 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.](image)

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

![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).](image)
blocking as follows. Let $k$ be the total number of non-zeros in $L_1$ and $L_2$ and let $f_b$ be the fill ratio after register blocking. Then the number of loads required is

$$\text{Loads}_{\text{sparse}}(b) = k f_b + \frac{k f_b}{b^2} + \left\lceil \frac{n}{b} \right\rceil + 1 + \frac{k f_b}{b} \left\lrfloor \frac{n}{RHS} = k f_b \left( 1 + \frac{1}{\gamma_2} + \frac{1}{b} \right) + n + \left\lceil \frac{n}{b} \right\rceil + 1 \ . \quad (6)$$

We include terms for the matrix (all non-zeros, one column index per non-zero block, and $\left\lceil n/b \right\rceil + 1$ 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

$$\text{Loads}_{\text{dense}} = \frac{n_2}{2} (\frac{n_2}{R} + 1) + n_2 \quad , \quad \text{Stores}_{\text{dense}} = \frac{n_2}{2} \left( \frac{n_2}{R} - 1 \right) \ . \quad (7)$$

where the $1/R$ factors model register-level blocking in the dense code, assuming $R \times R$ register blocks. 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

$$T = \sum_{i=1}^{\kappa-1} h_i \alpha_i + m_\kappa \alpha_{\text{mem}} \ . \quad (8)$$

where $\alpha_i$ is the access time (in seconds) at cache level $i$, $\kappa$ is the level of the largest cache, and $\alpha_{\text{mem}}$ is the memory access time. This model assumes we must incur the cost of a full access latency for each load and store.\footnote{For a dense matrix stored in sparse format, we only need to call DTRSV; in the model, $n_1$ and $k$ would be 0.}

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

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

$$M^{(1)}_{\text{lower}}(b) = \frac{1}{l_1} \left[ k f_b \left( 1 + \frac{1}{\gamma_2 b^2} \right) + \frac{1}{\gamma} \left( \left\lceil \frac{n}{b} \right\rceil + 1 \right) + \left( 2n + \frac{n_2 (n_2 + 1)}{2} \right) \right] \ . \quad (10)$$

where the size of one double-precision value equals $\gamma$ 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

$$M^{(1)}_{\text{upper}}(b) = \frac{1}{l_1} \left[ k f_b \left( 1 + \frac{1}{\gamma_2 b^2} \right) + \frac{1}{\gamma} \left( \left\lceil \frac{n}{b} \right\rceil + 1 \right) + \left( k f_b \frac{b}{b} + n + \text{Loads}_{\text{dense}} + \text{Stores}_{\text{dense}} \right) \right] \ . \quad (11)$$

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 $\alpha_{\text{mem}}$ be the minimum memory access latency shown in Table 2. This latency assumes a memory access but also a TLB hit. For the lower bound, we assume $\alpha_{\text{mem}}$ is the maximum memory access latency shown in Table 2, a memory access and a TLB miss.\footnote{When appropriate, we refine our time model to incorporate architecture-specific features. For instance, both the Power3 and the Itanium can commit two loads per cycle if they both hit in the L1 cache \cite{22, 23}. Thus, we reduce the L1 latency $\alpha_1$ by a factor of two to obtain a performance upper bound. Also, we take into account the fact that on Itanium, the cache hit times depend on whether the data is tied to integer or double-precision registers \cite{22}.}
3.2 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 4-6 validate our bounds on misses, equations (11) and (11). In particular, for the larger cache sizes, the vector lengths are such that the true miss counts are closer to equation (11) than (11), implying that conflict misses can be ignored.

3.3 Evaluating optimized SpTS performance

Figures 8-10 compare the observed performance of various implementations to the bounds derived above (Section 3.1). 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 3.1 (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 $\alpha_{mem}$.

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 (3), 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.

---

9In modeling the call to DTRSV, we used the empirically estimated register block sizes of $R = 4$ on the Power3 and Itanium platforms, and $R = 18$ on the Ultra 2i platform. We used the vendor-supplied DTRSV on the Power3 and Itanium. On the Ultra 2i, we used the ATLAS generated DTRSV, which uses a recursive implementation § and 4×8 blocking at the base case.
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 4, 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.
Figure 8: **Performance summary (Sun Ultra 2i):** Performance (Mflop/s) shown for the seven items listed in Section 3.3. 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 3.3. 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 3.3). 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 [13], MUMPS [2], UMFPACK [13], PSPASES [24], and SPOOKES [7], 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, 231, 29, 28, 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 have been developed, each with the goal of providing a compiler with sufficiently precise models for selecting memory hierarchy transformations and parameters such as tile sizes [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 are primarily distinguished from one another by 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 multiplication show speedups of 6.5 to 9 over single-vector code [30].

- 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 is, at this point, particular to our choice of fill-reducing ordering. We are currently investigating whether this optimization will prove fruitful in 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, but instead must tolerate the full memory latency of all loads and stores. The performance results on Itanium, which are very close to the upper bound model, raise the issue of whether we can in fact 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


