Performance Optimizations and Bounds for Sparse Symmetric Matrix - Multiple Vector Multiply

Benjamin C. Lee *Richard W. Vuduc* *James W. Demmel*
Katherine A. Yelick *Michael de Lorimier* *Lijue Zhong*

Report No. UCB/CSD-03-1297
November 25, 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Contents

1 Introduction 3

2 Experimental Methodology 5
- 2.1 Platforms . . . 5
- 2.2 Matrices . . . 7
- 2.3 Timing . . . 7

3 Optimizations for Matrix Symmetry 8
- 3.1 Symmetric Storage . . . 8
- 3.2 Register Blocking . . . 9
- 3.3 Diagonal Block Alignment . . . 12
- 3.4 Applying the Transpose of the Stored Triangle . . . 14
- 3.5 Loop Unrolling . . . 14
- 3.6 Implementation Summary . . . 14

4 Optimizations for Multiple Vectors 15
- 4.1 Vector Storage . . . 16
- 4.2 Vector Blocking . . . 16
- 4.3 Vector Width Selection . . . 17

5 Bounds on Performance 18
- 5.1 Execution Time Model . . . 19
- 5.2 Load Model . . . 20
- 5.3 Load Model Validation . . . 21
- 5.4 Cache Miss Model . . . 22
- 5.5 Cache Miss Model Validation . . . 22

6 Evaluation 33
- 6.1 Performance Data . . . 33
- 6.2 Effects of Symmetry on Performance . . . 39
- 6.3 Effects of Symmetry on Storage . . . 41
- 6.4 Block Size Selection . . . 42
- 6.5 Evaluation of Performance Upper Bounds . . . 43
- 6.6 Peak Performance . . . 47

7 Related Work 48

8 Conclusions and Future Directions 48

A Non-symmetric Register Blocked Example 53

B Modifications to NIST Sparse BLAS 54

C Symmetric Register & Vector Blocked Example 55

D Performance Profile Example **61**

E Application of the Transpose **63**

F Block Alignment **64**

G Loop Unrolling **65**

H Load Data **70**

I Cache Miss Data **73**

J Performance for Optimum Register & Vector Block Sizes **75**

K Memory Usage **80**

Abstract

We present performance optimizations and detailed performance analyses of sparse matrix-vector multiply SpMV and its generalization to multiple vectors, SpMM, when the matrix is symmetric. In addition to the traditional benefit of reduced storage from symmetry, we show considerable performance benefits are possible on modern architectures where the cost of memory access dominates the cost of other operations. We analyze the effects of (1) symmetric storage, (2) register-level blocked matrix storage, and (3) register-level vector blocking for multiple vectors, when applied individually and applied in conjunction. Compared to the best register blocked implementations that ignore symmetry, we show 2.1× speedups for SpMV and 2.6× speedups for SpMM. Compared to the most naïve implementation in which none of the three optimizations are applied, our best implementations are up to 9.9× faster for a dense matrix in sparse format and up to 7.3× faster for a true sparse matrix.

We evaluate our implementations with respect to *upper bounds* on their absolute performance in Mflop/s. Our performance model, which extends our prior bounds for SpMV in the non-symmetric case, bounds performance by considering only the cost of memory operations, and using lower bounds on cache misses. On four different platforms and a suite of twelve symmetric matrices spanning a variety of applications, we find our implementations are within 60% of the upper bounds on average. This fraction is smaller than what we have previously observed for non-symmetric SpMV, suggesting two possible avenues for future work: (1) additional refinements to the performance bounds that explicitly model low-level code generation (*e.g.*, register pressure or instruction scheduling/selection/bandwidth issues), and (2) a new opportunity to apply automated low-level tuning techniques in the spirit of ATLAS/PHiPAC systems to symmetric sparse kernels.

1 Introduction

This report presents performance optimizations and detailed performance analyses for the sparse matrix-vector multiply (SpMV) operation, $y \leftarrow y + A \cdot x$, when A is a symmetric sparse matrix (*i.e.*, $A = A^T$), and x, y are dense column vectors. We refer to x as the *source* vector, and y as the *destination* vector. In addition, we also consider the generalization of SpMV to multiple column vectors: x, y are replaced by dense matrices X, Y, and we refer to this multiple vector kernel as SpMM. Symmetry has traditionally been exploited to conserve storage, but significant performance improvements are also possible since the cost of memory accesses dominates the cost of flops on most modern cache-based superscalar architectures.

We show that by combining symmetric storage with existing techniques that exploit other kinds of sparse matrix substructure, improvements in performance (Mflop/s) of up to 2.1× and 2.6× are possible for SpMV and SpMM, respectively, when compared to equivalent optimized implementations without symmetry. In particular, the maximum performance gains from symmetry were: 1.7× and 1.1× on the Sun Ultra 2i; 1.4× and 1.5× on the Itanium 1; 2.0× and 2.6× on the Itanium 2; and 2.1× and 1.8× on the Power 4 for SpMV and SpMM, respectively.

The central problem in efficient performance tuning for sparse computational kernels like SpMV and SpMM is the considerable variation in the best choice of sparse matrix data structure and code transformations across machines, compilers, and matrices, the latter of which may not be known until run-time. Our approach to automatic tuning builds on prior experience with the Sparsity system [16, 14] for SpMV and SpMM when A is non-symmetric: for each kernel, we (1) identify and generate a space of candidate implementations, any one of which might be the best, and (2) search using a combination of modeling and experimental profiling (*i.e.*, actually running the code) to find the fastest implementation in that space. This report describes a new implementation space for the symmetric case in which the following three optimizations are considered:

1. *symmetric storage* of the sparse matrix (Section 3),
2. register-level *blocked storage* of the sparse matrix, as proposed in Sparsity (Section 3),
3. register-level *vector blocking*, as proposed in Sparsity (Section 4).

Our experimental work considers the effect of each optimization when applied both individually and in conjunction.

To evaluate the achieved performance of our best implementations, we develop upper bounds on their absolute performance (Mflop/s) as described in Section 5. Our performance model is a bound primarily in two senses. First, we bound execution time from below by considering only the cost of memory accesses (loads and stores). Second, we model data placement at the various levels of cache using lower bounds on the number of cache misses. We account for specific machines in our bounds by using published and measured values for the effective access latencies at each level of the memory hierarchy and compare our models to detailed hardware counter data. In prior work, similar bounds for the non-symmetric case indicated that our implementations were often within 80% of the upper bound, suggesting that further performance improvements from low-level tuning of the code would be limited [2]. By contrast, our implementations in the symmetric case achieve a smaller fraction of the upper bound, achieving approximately 60% on average. This observation suggests two avenues for future work: (1) refining the optimistic assumptions of the upper bound model, possibly to include modeling of register pressure, or instruction scheduling, selection, and bandwidth issues, and (2) applying automated low-level tuning techniques developed for dense linear algebra operations as in the ATLAS/PHiPAC systems [5, 22].

The following summarizes the observations of our experimental work, conducted on four different computing platforms (Table 1) and over a test suite of twelve sparse symmetric matrices (Table 2):

- We present a variant of the SPARSITY register blocking optimization for SpMV that exploits matrix symmetry. This optimization for symmetry can improve SpMV performance by as much as 2.1× over a non-symmetric implementation with register blocking and as much as 2.6× over a non-symmetric implementation with register blocking and vector blocking.

- We present a variant of the SPARSITY vector blocking optimization for SpMV that exploits multiple vectors. This optimization for multiple vectors can improve SpMV performance by as much as 4.9× over a single vector implementation when the matrix is stored in a non-symmetric format (full storage) and as much as 3.3× over a single vector implementation when the matrix is stored in a symmetric format (half storage).

- Optimizing the SpMV kernel with symmetry, register blocking, and multiple vectors improves performance by as much as 9.9× for a dense matrix in sparse format and 7.3× for a true sparse matrix when compared to a naïve implementation (no symmetry, no register blocking, single vector).

- Optimizing the Ultra 2i and Itanium 1 for symmetry yields performance comparable to the performance of a non-symmetric implementation (*i.e.* limited performance gains) when considered with register blocking and vector blocking. The case for symmetry exploiting optimization still holds, however, since tuning is necessary to achieve this performance along with savings in storage. In contrast, the previously mentioned maximum performance speedups were observed on the Itanium 2 and Power 4.

Collectively, these results show that significant gains are possible in practice by consideration of higher-level matrix structure (*e.g.*, symmetry) and kernels beyond SpMV (*e.g.*, SpMM) that inherently possess more opportunities for reuse.

2 Experimental Methodology

2.1 Platforms

We conducted our experimental evaluations on machines based on the microprocessors shown in Table 1. This table summarizes each platform's hardware and compiler configurations, and performance results on key dense kernels. Latency estimates were obtained from a combination of published sources and experimental measurements using the Saavedra-Barrera memory system microbenchmark [24] and MAPS benchmarks [29].

In addition, Table 1 shows performance data for a variety of related dense matrix kernels available from the best hand-tuned or automatically tuned Basic Linear Algebra Subroutine (BLAS) libraries. DGEMM is the double-precision dense matrix multiply routine, and DGEMV is the dense matrix-vector multiply routine. Both routines are a guide to the best possible performance of the corresponding sparse kernels when the input matrices are non-symmetric.

The BLAS also includes matrix-vector and matrix-matrix multiply routines for the symmetric case. In Table 1, we show the performance of DSYMV and DSYMM, which are matrix-vector and matrix-matrix multiply when the symmetric matrix is stored as if it were a dense matrix, with half the matrix entries ignored by the routine (*i.e.*, the routines assume symmetry). DSPMV is a symmetric matrix-vector multiply routine in which the symmetric matrix is stored symmetrically by storing either the upper or lower triangle of the matrix (effectively storing only half the total matrix entries). On our four evaluation platforms, DSYMM generally approaches the performance of DGEMM. The similarity in performance of DSYMM and DGEMM is to be expected since it is natural to implement DSYMM by invoking DGEMM on large, rectangular subblocks of the stored triangle. In contrast, DSYMV is nearly up to twice as fast as DGEMV. The differences in performance of DSYMV and DGEMV is also expected since DSYMV requires approximately half as many memory references as DGEMV. Except on the Power4, DSPMV performance is much lower than DSYMV. This effect is most likely due to a lack of tuning for DSPMV since we would not expect the performance of these routines to differ significantly.

Property	Sun Ultra 2i	Intel Itanium 1	Intel Itanium 2	IBM Power4
Clock Rate	333 MHz	800 MHz	900 MHz	1.3 GHz
Peak Main Memory Bandwidth	664 MB/s	2.1 GB/s	6.4 GB/s	8 GB/s
Peak Flop Rate	667 Mflop/s	3.2 Gflop/s	3.6 Gflop/s	5.2 Gflop/s
DGEMM ($n = 1000$)	425 Mflop/s	2.2 Gflop/s	3.5 Gflop/s	3.5 Gflop/s
DGEMV ($n = 1000$)	58 Mflop/s	345 Mflop/s	740 Mflop/s	915 Mflop/s
STREAM Triad Bandwidth [25]	250 MB/s	1.1 GB/s	3.8 GB/s	2.1 GB/s
DSYMV ($n = 1000$)	92 Mflop/s	625 Mflop/s	1.4 Gflop/s	1.6 Gflop/s
DSPMV ($n = 2000$)	62 Mflop/s	115 Mflop/s	356 Mflop/s	1.7 Gflop/s
DSYMM ($n = 2000$)	383 Mflop/s	1.9 Gflop/s	3.4 Gflop/s	3.5 Gflop/s
L1 data cache size	16 KB	16 KB	32 KB	32 KB
L1 line size	16 B	32 B	128 B	128 B
L1 latency	2 cy	2 cy (int)	0.34 cy	0.7 cy
L2 cache size	2 MB	96 KB	256 KB	1.5 MB
L2 line size	64 B	64 B	128 B	128 B
L2 latency	7 cy	6 cy (int) 9 cy (double)	0.5 cy	12 cy
L3 cache size	N/A	2 MB	1.5 MB	16 MB
L3 line size		64 B	128 B	512 B
L3 latency		21 cy (int) 24 cy (double)	3 cy	45 cy
TLB entries	64	32 (L1 TLB) 96 (L2 TLB)	32 (L1 TLB) 128 (L2 TLB)	1024
Page size	8 KB	16 KB	16 KB	4 KB
Memory latency (≈)	36 cy	36 cy	11 cy	167 cy
sizeof(double)	8 B	8 B	8 B	8 B
sizeof(int)	4 B	4 B	4 B	4 B
Compiler	Sun C v6.1	Intel C v5.0.1	Intel C v7.0	IBM XLC

Flags	`-dalign`
	`-xtarget=native`
	`-xO5`
	`-xarch=v8plusa`
	`-xrestrict=all`
		`-O3`	`-O3`	`-O5, -qhot`
				`-qalias=allp`
				`-qcache=auto`
				`-qarch=pwr4`
				`-qtune=pwr4`
				`-qnoipa`

Table 1: **Evaluation platforms**. We list the basic configuration data for the machines and compilers used in our experiments. For the BLAS routines (DGEMM, DGEMV, DSYMV, DSPMV, and DSYMM), we show the best performance between hand-tuned and automatically tuned libraries (ATLAS 3.4.1 [22], Sun Performance Library v6.0, Intel Math Kernel Library v5.2, and Goto's BLAS library [30]). The leading dimension (LDA) is set to be equal to the matrix dimension.

	Name	Application Area	Dimension	Nonzeros
1	dense1600	Dense Matrix	1600	1280800
2	bcsstk35	Stiff matrix automobile frame	30237	1450163
3	crystk02	FEM Crystal free vibration	13965	968583
4	crystk03	FEM Crystal free vibration	24696	1751178
5	nasasrb	Shuttle rocket booster	54870	2677324
6	3dtube	3-D pressure tube	45330	3213332
7	ct20stif	CT20 Engine block	52329	2698463
8	gearbox	ZF aircraft flap actuator	153746	4617075
9	finan512	Financial portfolio optimization	74752	596992
10	pwt	Structural engineering problem	36519	326107
11	vibrobox	Structure of vibroacoustic problem	12328	342828
12	gupta1	Linear programming matrix	31802	2164210

Table 2: **Matrix benchmark suite.** Matrices are categorized roughly as follows: 1 is a dense matrix stored in sparse format; 2–8 arise in finite element applications; 9–11 come from assorted applications; 12 is a linear programming example. For each matrix, we show the number of non-zeros in the upper-triangle.

2.2 Matrices

We evaluate the SpMV implementations on a subset of the matrix benchmark suite used by Im [14]. Table 2 summarizes the size and application of each matrix. Most of the matrices are available from either the collections at NIST (MatrixMarket [26]) or the University of Florida [27].

The matrices in Table 2 all possess symmetry. They are arranged in four groups. Matrix 1 is a dense matrix stored in sparse format; matrices 2–8 arise in finite element method (FEM) applications; matrices 9–11 come from assorted applications ; matrix 12 is a linear programming example [1].

2.3 Timing

We use the PAPI v2.1 library for access to hardware counters on all platforms [23] except for Power 4; we use the cycle counters as timers. Counter values reported are the median of 25 consecutive trials.[2]

PAPI load instruction counters are unavailable for the Power 4 platform. Alternatively, we employed the HPM Tool Kit from IBM to record hardware statistics, such as loads and cache misses. However, the version of HPM available to us significantly undercounted the number of cache misses. For this reason, hardware cache miss data is unavailable for the Power 4.

---

[1] This linear programming matrix is symmetric because it represents the explicit product $A^T A$.

[2] The standard deviation of these trials is typically less than 1% of the median.

The largest cache on some machines is large enough to contain some of the matrices. To avoid inflated findings, we report performance results only on the subset of out-of-cache matrices for each platform. Figures will always use the numbering scheme shown in Table 2.

For SpMV, reported performance in Mflop/s always uses "ideal" flop counts. That is, if a transformation of the matrix requires filling in explicit zeros (as with register blocking, described in Section 3), arithmetic with these extra zeros are *not* counted as flops when determining performance.

3 Optimizations for Matrix Symmetry

This section provides an overview of the optimizations for matrix symmetry and discusses the implementation space of a symmetric SpMV kernel. In particular, we examine the *symmetric storage* of a matrix as a technique to achieve increased storage efficiency by storing only half of the matrix, while reducing the number of accesses to the matrix. We then describe the *register blocking* optimization, designed to exploit naturally occuring dense blocks by reorganizing the matrix data into a sequence of small (small enough to fit in registers) dense blocks to further register reuse. Furthermore, combining register blocking and symmetric storage requires special attention to *diagonal block alignment* and the application of floating point operations for the *transpose* of the stored triangle. We then mention *loop unrolling* as a mechanism for reducing loop overhead within the register blocks. Finally, we conclude this section with a summary of the implementation space for symmetry optimizations and identify the subspace examined in the experimental work of this report.

The baseline implementation in this report uses full storage of the matrix (*i.e.*ignores symmetry) in compressed sparse row (CSR) format.[3] This baseline implementation computes symmetric SpMV using a non-symmetric kernel, accessing each matrix element once.

3.1 Symmetric Storage

Matrix symmetry makes it possible to store just half of the matrix and, without loss of generality, our implementation stores the upper-triangle. Although the symmetric implementation requires the same number of floating point operations as the baseline implementation, symmetric storage reduces the number of memory accesses to the matrix by half.

However symmetric storage also potentially changes the access pattern to the destination vector, y. First, observe that the baseline implementation processes each row in turn, thus requiring only one store to each element of y. In contrast, for each non-zero matrix element, a symmetric implementation simultaneously applies that element and its transpose, meaning one store for every non-zero. Furthermore, each of these stores is indirect and thus potentially irregular. From this view, it is not obvious that symmetric storage alone has a clear performance advantage over the baseline.

---

[3] One survey of common formats appears in Barrett, et al.[3].

Figure 1 shows the standard implementation of SpMV in C assuming CSR storage in the non-symmetric (full-storage) case. Note the overhead of extra storage for the data structure (`row_idx` and `col_ptr`) as well as the potential for irregular memory accesses to `x` in line 4. Also note that there is only one store per element of `y`.

Figure 2 shows a standard implementation for the symmetric case. As is evident in the code, we perform the same number of flops as the non-symmetric code while reducing the number of accesses to A (through `*value`). However, we have introduced an extra branch to handle the diagonal elements (lines 4–8), and we have also increased the number of stores to the destination vector (line 13).

Both implementations are slight modifications of the routines from the NIST Sparse BLAS library [20]. In particular, the original NIST routines compute $y = Ax$, whereas our implementations compute $y = y + Ax$. For this reason, the original routine sets all elements in y to zero on entry (Appendix B). For our accumulation version, we eliminate this operation. The original routines also use two sets of row pointers for the beginning and end of a row while our reference implementation uses one set of row pointers for the beginning of a row. Other differences include the condition to terminate the main loop, instantiation of new pointers to arrays passed into the routine, and the syntax used to access matrix and vector elements. Despite these differences, the performance of our reference implementation (Figure 1) and the performance of our register blocked implementation (Appendix A) with $(1, 1)$ register blocks are both comparable to that of the CSR implementation from the NIST Sparse BLAS [20]. We will use these routines as our reference implementations in subsequent performance comparisons.

3.2 Register Blocking

SPARSITY's *register blocking* optimization is a technique for improving register reuse over that of a conventional implementation [16]. Register blocking is designed to exploit naturally occuring dense blocks by reorganizing the matrix data structure into a sequence of small (enough to fit in registers) dense blocks. Register blocking reduces loop overhead, reduces indexing overhead, reduces irregular access, and increases temporal locality to the source vector.

In the register blocked implementation, consider an $m \times n$ matrix, divided logically into $\frac{m}{r} \times \frac{n}{c}$ submatrices, where each submatrix is of size $r \times c$. Assume for simplicity that r divides m and that c divides n. For sparse matrices, only

```
void spmv( int m, const double* value,
           const int* col_idx, const int* row_start,
           const double* x, double* y )
{
    int i;

    /* loop over rows */
    for( i = 0; i < m; i++ ) {
        int jj;
        double y_i = y[i];

        /* loop over non-zero elements in row i */
        for( jj = row_start[i]; jj < row_start[i+1]; jj++ ) {
            y_i += (*value++) * x[*col_idx++];
        }
        y[i] = y_i;
    }
}
```

Figure 1: A standard C implementation of reference SpMV for $y = y + Ax$, assuming CSR storage and C-style 0-based indexing. A is an $m \times n$ matrix. This is a modification of the corresponding NIST routine to compute $y = y + Ax$ instead of $y = Ax$.

```
    void spmv_symm( int m, const double* value,
                    const int* col_idx, const int* row_start,
                    const double* x, double* y )
    {
        int     i;

        /* loop over rows */
 1      for( i = 0; i < m; i++, row_start++ ) {
            double y_i = y[i];
 2          double x_i = x[i];
 3          int    jj  = row_start[i];

            /* special handling of the diagonal */
 4          if ( i == *col_idx ) {
 5              y_i += x[i] * (*value++);
 6              col_idx++;
 7              jj++;
            }
 8          else  y_i  = 0;

            /* loop over non-zeros in row i */
 9          for( ; jj < row_start[i+1]; jj++ ) {
10            int j = *col_idx++;       /* column index j */
11            double a_ij = *value++;   /* matrix element A(i,j) */

12            y_i  += a_ij * x[j];
13            y[j] += a_ij * x_i;
            }

14          y[i] = y_i;
        }
    }
```

Figure 2: A standard C implementation of SpMV for $y = y + Ax$ assuming CSR storage, where A is symmetric. The code assumes that only the upper-triangle is stored. This code differs from the non-symmetric case by the extra branch for the diagonal (lines 4–8) and the additional updates to `y` (line 13).

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

$\texttt{b\_row\_start} = \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}$

$\texttt{b\_col\_idx} = \begin{pmatrix} 0 & 4 & 2 & 4 \end{pmatrix}$

$\texttt{b\_value} =$

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

Figure 3: **Block compressed sparse row (BCSR) storage format**. BCSR format uses three arrays. The elements of each dense 2 × 2 block are stored contiguously in the `b_value` array. Only the first column index of the (1,1) entry of each block is stored in `b_col_idx` array; the `b_row_start` array points to block row starting positions in the `b_col_idx` array. In SPARSITY, blocks are stored in row-major order. (Figure taken from Im [14].)

those blocks which contain at least one non-zero are stored. The computation of SpMV proceeds block-by-block. For each block, we can reuse the corresponding c elements of the source vector and r elements of the destination vector by keeping them in registers, assuming a sufficient number are available.

SPARSITY's implementation of register blocking uses the blocked variant of compressed sparse row (BCSR) storage format. Blocks within the same block row are stored consecutively, and the elements of each block are stored consecutively in row-major order.[4] A 2 × 2 example of BCSR is shown in Figure 3. When $r = c = 1$, BCSR reduces to CSR.[5] Refer to Appendix A for an example of a register blocked implementation.

Note that BCSR potentially stores fewer column indices than CSR implementation (one per block instead of one per non-zero). The effect is to reduce memory traffic by reducing storage overhead. Furthermore, SPARSITY implementations fully unroll the $r \times c$ submatrix computation, reducing loop overheads and exposing scheduling opportunities to the compiler.
However, Figure 3 also shows that the imposition of a uniform block size may require filling in explicit zero values, resulting in extra computation. We define the *fill ratio* to be the number of stored values (original non-zeros plus explicit zeros) divided by the number of non-zeros in the original matrix. Whether conversion to a register blocked format is profitable depends highly on the fill and, in turn, the non-zero pattern of the matrix. Furthermore, the performance gains from register blocking for SpMV given a block size varies across platforms, suggesting that performance is a strong function of platform characteristics, such

---

[4] Row-major is SPARSITY's convention; column-major or other layouts are possible.

[5] The performance of this code is comparable to that of the CSR implementation from the NIST Sparse BLAS [20].

as architecture and compiler, in addition to block size.

The SPARSITY system includes a heuristic for selecting the register block size given a performance profile of the machine. The profile characterizes the performance of block sizes up to 12x12 on a very regular "sparse" problem: a dense $n \times n$ matrix stored in sparse BCSR format. The experimental work for this report does not include modifications to the SPARSITY heuristic for symmetry and instead presents performance based on exhaustive search for block sizes up to 8x8 for each matrix. The development of a heuristic for the symmetric case is future work.

3.3 Diagonal Block Alignment

The alignment of diagonal blocks refers to the manner in which the diagonal elements are register blocked. The alignment chosen will affect the complexity of a symmetric SpMV kernel as well as the alignment and blocking of non-diagonal elements. We describe two alignment schemes, *uniform blocking* and *square diagonal blocking*, and mention possible variations on these schemes.

Uniform blocking is an alignment scheme in which all blocks, diagonal and non-diagonal, are the same size (Figure 4, left). Employing a uniform block size is a natural extension of SPARSITY's alignment scheme with the exception that any block strictly below the diagonal is omitted. Furthermore, matrix elements below the diagonal are not included in the diagonal block. Although uniform blocking is conceptually simple, the implementation is subtly difficult since diagonal blocks must be handled specially. In particular, a diagonal element could intersect a diagonal block at any number of positions within the block, requiring conditional statements to handle each possible configuration.

Square diagonal blocking is an alignment scheme in which the diagonal blocks are square (Figure 4, right). As in the uniform blocking scheme, elements below the diagonal are not included in the diagonal block. Given a row-oriented storage scheme and $r \times c$ register blocks, the diagonal blocks are implemented as square $r \times r$ blocks, simplifying implementation and analysis since all diagonal blocks may be considered in the same manner. However, in order to keep the register blocks aligned as uniformly as possible, we choose to align the register blocks from the right edge of the matrix. For some rows, aligning these register blocks requires small degenerate $r \times c'$ blocks to the right of the diagonal blocks, where $c' < c$ and c' depends on the block row. Analyses of the distribution of block types (diagonal square, degenerate, non-diagonal register) when implementing square diagonal blocking, however, indicate that degenerate blocks account for at most 7.7% of the total number of blocks in a symmetrically stored matrix. In most cases, no degenerate blocks were necessary.

Variations on uniform blocking and square diagonal blocking are also possible. One variation on uniform blocking fills diagonal blocks with modified elements

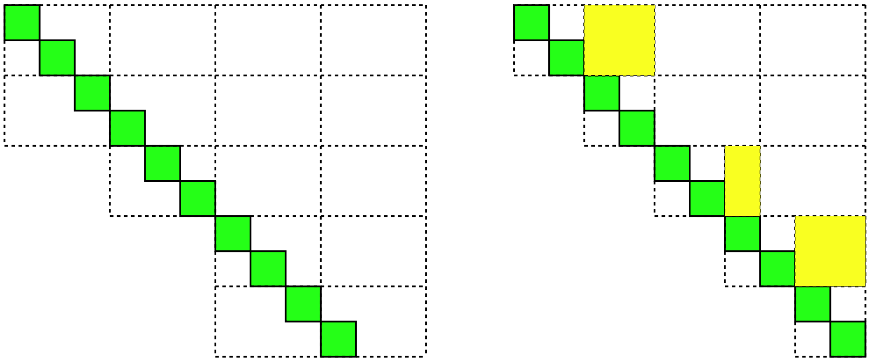


Figure 4: **Diagonal block alignment schemes**. Two diagonal block alignment schemes for a 10×10 matrix with 2×3 register blocks: uniform blocking(*left*) and square diagonal blocking (*right*). In uniform blocking, all blocks are 2×3. The matrix is logically expanded to a multiple of the block size. In square diagonal blocking, there is a single diagonal block size ($r \times r$). The non-diagonal blocks are aligned from the right edge of the matrix with smaller degenerate blocks to the right of the diagonal blocks, if necessary. The diagonal elements are highlighted in green and the degenerate blocks in yellow.

from below the diagonal. Filling the diagonal blocks enables all blocks to be considered in the same manner, but the filled elements actually duplicate storage since they represent values below the diagonal. Furthermore, the fill elements must be modified from their true value to account for duplicate operations on the same matrix element. Although filled elements complicate matrix manipulations, the cost of such complications could be amortized over many uses of the symmetric SpMV kernel for a given matrix.

A variation on square diagonal blocking uses large diagonal square blocks of size $l \times l$ where $l = \mathrm{lcm}(r, c)$, the least common multiple of r and c. Assuming $\mathrm{lcm}(r, c)$ divides the matrix dimension, the size of these blocks eliminates the need for degenerate blocks in the $r \times r$ square diagonal blocking scheme. The performance gains from this optimization, however, would most likely be relatively insignificant given the relatively small number of diagonal blocks in the distribution of block types as discussed above.

The experimental work for this report implements *square diagonal blocking*. Our preliminary analyses of uniform blocking and square diagonal blocking suggest no obvious advantage to either alignment scheme. Refer to Appendix F for a preliminary performance analysis of the two schemes.

3.4 Applying the Transpose of the Stored Triangle

In the blocked symmetric SpMV implementation, we apply both the block and its transpose for each stored register block. We consider two possible implementations: *simultaneous* and *sequential* application of the transpose.

The simultaneous application interleaves the floating point operations of the stored elements and those of the corresponding transpose elements within a register block. Thus, the simultaneous application of the transpose stores computed values into the destination vector element by element. In contrast, the sequential application of the transpose completes the operations for all stored elements within a given register block before performing those in the corresponding transposed register block. Thus, the sequential application of the transpose stores computed values into the destination vector block by block.

This report studies the *sequential* application of the transpose. In principle, the simultaneous scheme reduces register pressure since an entire block does not need to be stored in registers. However, preliminary evidence in Appendix E suggests that employing the sequential scheme is faster in practice. Note that our implementations are written in C and we rely on the compiler to schedule the code. Thus, although we would expect the simultaneous scheme to make more efficient use of the registers, this scheme does not necessarily yield more efficient schedules than the schedules for the sequential scheme. Fully understanding this phenomenon is possible future work.

3.5 Loop Unrolling

The floating point operations for the elements in a register block may be unrolled to reduce loop overhead. This report considers two possible implementations of loop unrolling, *row-wise* and *column-wise*. Given a register block, the elements may be unrolled across the block row or the block column. There is no obvious advantage to either unrolling scheme. This report studies *column-wise* unrolling since initial experimental work shows higher performance from column-wise unrolling, especially for Itanium 1 and Itanium 2 (Appendix G).

3.6 Implementation Summary

Numerous alternative implementations for a symmetric SpMV kernel were considered, but not all implemented in the experimental work of this report. The first of these alternatives is an option to multiply the diagonal blocks separately, instead of multiplying them when considering each block row. Let $A = D + A_1$ where D consists of the diagonal blocks from the matrix and A_1 is the matrix with the diagonal blocks excluded. Thus, the SpMV operation $y = y + A \cdot x$ is equivalent to $y = y + D \cdot x + A_1 \cdot x$ where the terms are evaluated separately. We do not consider such an implementation in this report.

Another option interleaves the source and destination vectors. In particular, this option lays out x and y in memory such that $x[i]$ and $y[i]$ reside at consecutive memory locations. Interleaving the vectors takes advantage of the locations of data in cache since elements are needed from both the source and destination vectors to calculate each result in the destination. We do not implement vector interleaving since the data must be explicitly organized in this fashion, potentially complicating the user interface.

In summary, the implementation space for a symmetric SpMV kernel includes the following factors. The italicized option indicates the option considered in this report.

- Storage: Non-symmetric; *Symmetric*
- Blocking: None; *Register*
- Block Size Selection: Heuristic; *Exhaustive*
- Diagonal Block Alignment: Uniform; *Square diagonal*
- Application of the Transpose: Simultaneous; *Sequential*
- Loop Unrolling: Row-wise; *Column-wise*

Refer to Appendix C for an example of an implementation of SpMM with the italicized options above.

4 Optimizations for Multiple Vectors

This section provides an overview of the optimizations for the sparse matrix-multiple vector (SpMM) kernel, $Y = Y + A \cdot X$, where A is a symmetric sparse $n \times n$ matrix, and X, Y are dense $n \times k$ matrices. We can view X and Y equivalently as collections of k dense column vectors of length n: $X = (x_1, \ldots, x_k)$ and $Y = (y_1, \ldots, y_k)$. Here, we examine the *vector blocking* optimization, designed to reduce loop overhead and increase temporal locality to A. In a vector blocked implementation of SpMM, the k vectors are processed in groups of the *vector width* v, and multiplication of each element of A (or block, if A is register blocked) is unrolled by v—conceptually, the vector width is the vector blocking analogue of the register block size $r \times c$.

The baseline implementation assumed in considering multiple vectors takes a symmetric matrix and, given k vectors, applies the unblocked symmetric SpMV kernel once for each vector. This implementation requires k accesses to the entire matrix and, for large matrices, requires bringing the entire matrix through the memory hierarchy once for each vector. Thus, the vector blocking optimization has the potential to decrease the number of memory accesses by as much as a factor of v.

4.1 Vector Storage

Vector storage has implications for kernel implementation and performance. The two obvious storage options are storage by *columns* and storage by *rows*. Column storage places the multiple vectors end to end. Thus, given a vector length of n, $x_1[i]$ and $x_2[i]$ are separated by $n-1$ elements in memory. Similarly, $y_1[i]$ and $y_2[i]$ are separated by $n-1$ elements in memory. In contrast, row storage places the multiple vectors side to side. In this case, $x_1[i]$ and $x_2[i]$ are placed in adjacent memory locations. Similary $y_1[i]$ and $y_2[i]$ are adjacent in memory. The access patterns to the x and y imply that row storage may be more efficient due to spatial locality. When A is stored by rows, also known as field interlacing [28], the natural access pattern to the vectors x and y is by rows. However, the user must explictly organize the data in this fashion. Elements within the same row of x and y are likely to straddle multiple pages when the vectors are stored by columns and n is large, thus applying pressure to the TLBs, which are typically small. The experimental work of this report implements vector storage by *columns*, since this is the most common user interface.

4.2 Vector Blocking

Vector blocking is a technique for reducing memory traffic during a sparse computation with multiple vectors over a conventional implementation. An implementation of SpMM with vector blocking exploits a set of multiple vectors by amortizing the cost of accessing a matrix element across a subset of these vectors, where the size of the subset is defined as the *vector width*. This technique to reduce loop overhead and increase temporal locality to a matrix may be applied to sparse matrix-vector multiply with multiple vectors.

In the vector blocked implementation, consider a set of k vectors divided into $\left\lceil \frac{k}{v} \right\rceil$ vector blocks, where each block has a vector width of at most v vectors. In the case with no register blocking, the computation of SpMM proceeds sequentially across matrix elements. For each matrix element, the multiple vector kernel computes results for the corresponding elements in each of the v destination vectors in the vector block before continuing with the computation for the next matrix element. Throughout the computation of an element for each of the v destination vectors in the vector block, we can reuse the corresponding element in the matrix by keeping it stored in a register, assuming a sufficient number are available. An analogous computational sequence applies for the register blocked case, with computation proceeding across register blocks instead of individual elements. Refer to Appendix C for an implementation of register blocking combined with vector blocking.

The implementation of vector blocking is dependent on the vector width. Given a vector width v as a tuning parameter, the SpMM kernel implements v subroutines. Each subroutine, SR_i for $1 \leq i \leq v$, is an implementation of SpMM for a fixed number of vectors, i. Given a set of k vectors, the optimized kernel

- `dispatch to` SR_v $\left\lfloor \frac{k}{v} \right\rfloor$ `times`

- `if` $k\%v > 0$`, dispatch to` $SR_{k\%v}$ `once`

Figure 5: **Multiple vector dispatch algorithm.** The optimized kernel dispatches to the appropriate subroutine. Refer to routine bssmvm_m_2x3_2 in Appendix C for a dispatch example for two subroutines.

dispatches to the appropriate subroutine (Figure 5). When $v = 1$, the subroutine is effectively a single vector implementation of SpMV. Such a code sample would differ from Appendix C in that the kernel always dispatches to routine bssmvm_m_2x3_1_once.

Note that each subroutine accesses each matrix element only once. Thus, given a set of k vectors and a vector width of v, the matrix is accessed at most $\left\lceil \frac{k}{v} \right\rceil + 1$ times in contrast to the k times required by the baseline implementation. The effect is to reduce memory traffic by reducing matrix accesses. Furthermore, the SpMV implementations for multiple vectors fully unroll the computations for the v vectors in the subset, thereby reducing loop overhead and exposing scheduling opportunities to the compiler.

4.3 Vector Width Selection

The selection of the vector width is highly dependent on the platform, matrix, and application. An implementation of SpMV for multiple vectors is usually register intensive, since the elements from each source and destination should be stored in registers, assuming a sufficient number of registers are available. Although increasing the vector width reduces memory traffic and achieves performance gains, it also increases the number of registers that the kernel seeks to use.

A simple analysis of register pressure that ignores compiler scheduling and optimizations provides a conservative estimate of register usage for the computation of a single register block. In the symmetric single vector case, the elements of the stored transpose require rc registers. The c source vector elements and r destination vector elements corresponding to the register block should be stored in registers. Similarly, the computation for the transpose block will also require $r + c$ registers. In a sequential application of the transpose, however, the registers for the stored block and the transpose block are not needed simultaneously. Thus, a conservative estimate of register usage in the computation for one register block and its transpose is $r + c + rc$ registers when the transpose is applied sequentially and $2(r+c)+rc$ when the transpose is applied simultaneously.

A similar estimate for the multiple vector case is obtained by scaling the number of registers used for the source and destination vectors by a factor of v. An estimate of register usage in the computation with multiple vectors is $v(r+c)+rc$ registers when the transpose is applied sequentially and $2v(r+c)+rc$ for when the transpose is applied simultaneously. Thus, a simple analysis of register usage indicates that register pressure increases with vector width. In the case where the estimated number of registers exceeds the actual number available, register spilling occurs resulting in a greater number of loads. The ultimate effect is an increase in execution time and a decrease in performance.

Increasing the vector width also raises possible issues in instruction cache misses. Instruction caches tend to be relatively small and the number of instructions in an implementation of a fully unrolled $r \times c \times v$ computation scales like $r \cdot c \cdot v$. For this reason, larger vector widths tend to decrease performance, reflecting the effects of register spilling. Refer to Appendix D for a representative plot of performance as a function of vector width.

This report presents performance data from an exhaustive search that would enable a user to select an optimal vector width for a particular platform and matrix. Furthermore, an analysis of the performance data for varying vector widths often indicate the existence of an optimal width. Continuing work with a non-symmetric SpMV kernel optimized with register blocking and vector blocking has indicated potential for a comprehensive search heuristic for register block sizes and vector widths, suggesting a similar heuristic is possible for a symmetric kernel.

5 Bounds on Performance

We present performance upper bounds to estimate the best possible performance given a matrix and a data structure, but independent of any particular instruction mix or ordering. In related work, code generators in automatic tuning systems for dense linear algebra, such as ATLAS or PHiPAC [5, 22], vary the instruction schedule during the tuning process. In our work on sparse kernel tuning, we have focused on data structure transformations, relying on the compiler (and eventually on a system like ATLAS or PHiPAC [5, 22]) to produce efficent schedules. An upper bound allows us to estimate the probable payoff from low-level tuning.

Our bounds for the register blocked, vector blocked implementations of symmetric SpMM, as described in Section 3 and Section 4, are based on bounds previously developed for non-symmetric SpMV [2]. In particular, we make the following assumptions in the derivation of the upper bound on performance:

1. SpMV and SpMM are memory bound since most of the time is spent streaming through the matrix data. Thus, we form a lower bound on execution time by considering only the cost of memory operations. Furthermore, we assume write-back caches (a valid assumption for each platform

considered in Table 1) and sufficient store buffer capacity such that we are able to consider only the contribution of loads to execution time and ignore the cost of stores.

2. The execution time model accounts 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*. When data resides in the internal cache (L1 on these machines), we assume that all accesses to this data can be fully pipelined, and therefore commit at the maximum load/store commit rate. Table 1 shows this effective L1 access latency [6]. Refer to Section 5 of our prior paper [2] for a detailed analysis of the STREAM benchmarks that justifies this assumption.

3. We are able to get a lower bound on memory costs by computing a lower bound on cache misses. We, therefore, consider only compulsory and capacity misses, ignoring conflict misses. Furthermore, we account for cache capacity and cache line size.

4. We do not consider the cost of TLB misses. Unless the matrix dimension is so large that the source vector no longer fits in cache, operations such as SpMV, SpMM, SpA^TA [7], and SpTS [8] spend most of the execution time streaming through the matrix using stride one accesses, so very few TLB misses will occur [9].

5.1 Execution Time Model

Let the total execution time of SpMM be T seconds. The corresponding performance P in Mflop/s is

$$P = \frac{4kv}{T} \times 10^{-6} \tag{1}$$

where k is the number of stored non-zeros in the $n \times n$ sparse matrix A (excluding any fill) and v is the vector width in the vector blocked implementation. To get an upper bound on performance, we require a lower bound on T.

Consider a machine with κ cache levels where the access latency at cache level i is α_i (in cycles or seconds) and the memory access latency is α_{mem}. Let h_i be the number of hits and m_i be the number of misses at cache level i. Assuming that we can overlap the latencies due to a perfect nesting of the caches, the execution time T is

---

[6] For example, the Sun Ultra 2i has a L1 load latency of 2 cycles [31]. This processor can commit, however, 1 load per cycle. We therefore charge a 1 cycle latency for L1 accesses in the bound.

[7] $y = y + A^TAx$ [1]

[8] sparse triangular solve

[9] This observation has been verified experimentally using hardware counters.

$$T = \sum_{i=1}^{\kappa-1} h_i \alpha_i + m_\kappa \alpha_{\text{mem}}, \tag{2}$$

Given the number of loads Loads(r,c,v) as a function of block size (r, c) and vector width v, the number of L1 hits h_1 is given by $h_1 = \text{Loads}(r, c, v) - m_1$. Assuming a perfect nesting of the caches, such that a miss at level i is an access at level $i+1$, $h_{i+1} = m_i - m_{i+1}$ for $i \geq 1$. Thus, to get an estimate of the upper bound on performance given a lower bound $\text{M}^{(\text{i})}_{\text{lower}}(r, c, v)$ on misses in the i-th cache as a function of block size and vector width, let $m_i = \text{M}^{(\text{i})}_{\text{lower}}(r, c, v)$ in Equation (2), and convert to MFlop/s using Equation (1). The following sections derive expressions for Loads(r,c,v) and $\text{M}^{(\text{i})}_{\text{lower}}(r, c, v)$.

5.2 Load Model

The following performance model assumes an implementation of an SpMV kernel optimized for symmetry, register blocking, and vector blocking, as described in Section 3 and Section 4. The performance model counts the number of loads and stores required for symmetric SpMM as follows.

Let A be an $m \times m$ symmetric matrix with k stored non-zeros. Let D_r be the number of $r \times r$ non-zero diagonal blocks and B_{rc} be the number of $r \times c$ non-zero register blocks required to store the matrix in symmetric BCSR format. Let $\|D_r\|$ and $\|B_{rc}\|$ denote the total number of matrix elements (including filled zeros) stored in the diagonal and non-diagonal blocks, respectively. Also denote the fill ratio, given $r \times r$ diagonal blocks and $r \times c$ register blocks, as f_{rc}.

The upper bound on D_r is $\lceil \frac{m}{r} \rceil$ with at most $\|D_r\| = \lceil \frac{m}{r} \rceil \cdot \frac{r(r+1)}{2} \approx \frac{m(r+1)}{2}$ diagonal blocked elements in the matrix. This follows from the observation that a diagonal block has at most $\frac{r(r+1)}{2}$ elements since elements strictly below the diagonal are not stored. Furthermore, we estimate the number of non-diagonal blocks B_{rc} by counting the stored elements excluded from the diagonal blocks so that $B_{rc} = \frac{\|B_{rc}\|}{rc}$ where $\|B_{rc}\| \approx k f_{rc} - \frac{m(r+1)}{2}$. Each register block has exactly rc elements comprised of non-zeros and explicitly filled zeros. In the case of 1×1 register blocking, $\|D_r\| + \|B_{rc}\| = k$.

The matrix requires storage of $\|D_r\| + \|B_{rc}\|$ double precision values, $D_r + B_{rc}$ integers for the column indices, and $\lceil \frac{m}{r} \rceil + 1$ integers for the row indices. Since the fill ratio is defined as the number of stored elements (fill values included) divided by the number of non-zeros (fill values excluded), we find that $f_{rc} \approx \frac{\|D_r\| + \|B_{rc}\|}{k}$, and is always at least 1.

Every matrix element, row index, and column index must be loaded once. We assume that SpMM iterates over block rows in the stored upper triangle and that all vr entries of the v destination vectors can be kept in registers for the duration of the block row multiply. Thus, we only need to load each element of the destination vector once for computations on the upper triangle. We also assume that all c destination vector elements can be kept in registers during the multiplication of a given transpose block, thereby requiring $B_{rc}vc$ additional loads from the destination vector. Each load from the destination vector is followed by a store to the destination vector.

Furthermore, we assume that the kernel iterates over block columns in the transpose of the stored triangle, and that all vr entries of the v source vectors can be kept in registers for the duration of the block column multiply, requiring $D_r vr$ loads from the source vector. Finally, we assume that all c source vector elements can be kept in registers during the multiplication of a given register block in the upper triangle, thus requiring $B_{rc}vc$ additional loads from the source vector. Refer to the code subsections labeled *Handle register blocks* in Appendix C for the code performing the multiplication and addition within a register block.

In terms of the number of non-zeros and the fill ratio, the total number of loads and stores of floating point and integer data is

$$\begin{aligned}\text{Loads}(r,c,v) &= \underbrace{B_{rc}rc + D_r\left(\tfrac{r^2+r}{2}\right) + B_{rc} + D_r + \left\lceil \tfrac{m}{r} \right\rceil + 1 +}_{\text{matrix}} \\ &\quad \underbrace{vrD_r + vcB_{rc}}_{\text{source vector}} + \underbrace{vrD_r + vcB_{rc}}_{\text{destination vector}} \qquad (3)\end{aligned}$$

$$\text{Stores}(r,c,v) = vrD_r + vcB_{rc} \qquad (4)$$

where D_r and B_{rc} can be represented in terms of f_{rc}, m, r, c, and k.

5.3 Load Model Validation

We present results from data collected for all register block sizes up to 8x8 across all matrices and platforms, measuring execution time, loads, and cache misses using PAPI (except for Power4, refer to discussion below). For each matrix, we compare the number of loads as measured by PAPI to the number estimated by the Equation (3). We validate our load model in Figures 6–9, showing data from a fully optimized implementation (symmetry, register blocking, vector blocking) for the block size and vector with that maximizes performance, chosen by exhaustive search (Appendix J). The data shows that load model estimates are a good match to the actual number of load instructions issued, especially for the Sun Ultra 2i and the IBM Power 4 (Appendix H). Differences in the estimated and the actual number of loads may reflect register spilling that results from vector blocking, requiring extra loads (Section 4.3).

PAPI load instruction counters are unavailable for the Power 4 platform. Alternatively, we employed the HPM Tool Kit from IBM to record hardware statistics. The HPM loads counted, however, count only floating point loads.

5.4 Cache Miss Model

The analytic upper bound on execution time as modeled in this report is derived from specifying an analytic lower bound on the number of cache misses.

Beginning with the L1 cache, let l_1 be the L1-cache line size, in double-precision words. One compulsory L1 read miss per cache line is incurred for every matrix element (value and index) and each of the mv destination vector elements. In considering the source vector for the lower bound, we assume the source vector size is less than the L1 cache size, so that in the best case, only one compulsory miss per cache line is incurred for each of the mv source vector elements. Thus, a lower bound $\mathrm{M}^{(1)}_{\mathrm{lower}}$ on L1 misses is

$$\mathrm{M}^{(1)}_{\mathrm{lower}}(r,c,v) \quad = \quad \tfrac{1}{l_1}\left[kf_{rc} + \tfrac{1}{\gamma}\left(D_r + B_{rc} + \lceil \tfrac{m}{r} \rceil + 1\right) + 2mv\right] \tag{5}$$

where the size of one double precision floating point value equals γ integers. In this paper, we use 64-bit double-precision floating point data and 32-bit integers, so that $\gamma = 2$. The factor of $\frac{1}{l_1}$ accounts for the L1 line size. An analogous expression applies at other cache levels by simply substituting the appropriate line size.

In the worst case, we will miss on every access to a source and destination vector element due to capacity and conflict (both self-interference and cross-interference) misses. Thus, an upper bound on misses is [10]

$$\begin{aligned}\mathrm{M}^{(1)}_{\mathrm{upper}}(r,c,v) \quad = \quad & \tfrac{1}{l_1}\left[kf_{rc} + \tfrac{1}{\gamma}\left(D_r + B_{rc} + \lceil \tfrac{m}{r} \rceil + 1\right)\right] + \\ & 2vrD_r + 2vcB_{rc}\end{aligned} \tag{6}$$

5.5 Cache Miss Model Validation

We present results from data collected for all register block sizes up to 8 × 8 across all matrices and platforms, measuring execution time, loads, and cache misses using PAPI (except Power 4, refer to discussion below). For each matrix, we compared the cache misses as measured by PAPI to the cache miss lower and upper bounds given by Equation (5) and Equation (6), respectively. We validate our cache miss models in Figures 10–14, showing data on a fully optimized implementation (symmetry, register blocking, and vector blocking) for the block size and vector width that maximizes performance, chosen by exhaustive search (Appendix J). The data shows that the miss bounds are a good match to the true misses (Appendix I). In particular, the vector lengths in our matrix suite are small enough that the lower miss bounds, which assume no capacity and conflict misses, count the true misses rather accurately in the off-chip caches (*i.e.*, the L2 cache on the Ultra 2i platform and the L3 cache on the Itanium 1, Itanium 2, and Power 4 platforms).

---

[10] Equation (6) is a loose upper-bound because it essentially ignores any spatial locality in accesses to the source vector. In principle, we can refine this bound by using the matrix non-zero pattern to identify spatial locality when present, but the experimental work of this report does not do so for simplicity.

The Itanium 1 and Itanium 2 architectures have L1 caches that do not cache floating point values. Therefore, the data presented excludes the L1 cache for these platforms.

The PAPI hardware counters used to validate the cache miss models were unavailable for the IBM Power 4. The HPM Tool Kit from IBM was used as an alternative source of hardware data, but the version available to us significantly undercounted the number of cache misses. For this reason, actual cache misses are omitted for the Power 4 platform in Figures 13–14.

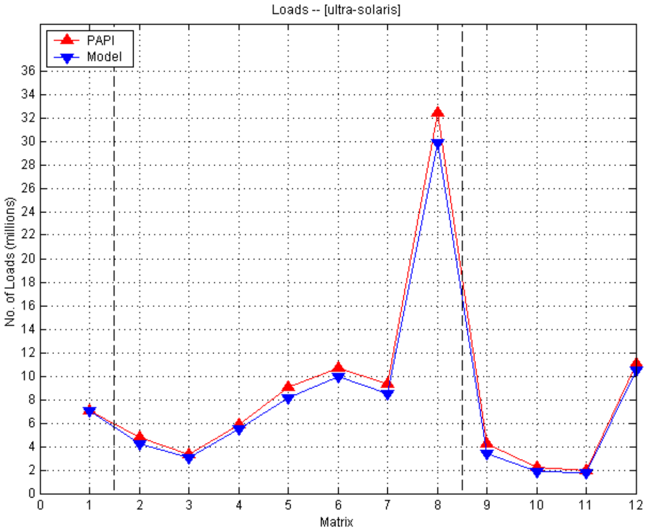


Figure 6: **Load Model Validation – Sun Ultra 2i.** Model for the number of issued load instructions on the Sun Ultra 2i compared to PAPI measurements for the block size and vector width that maximizes performance in a fully optimized implementation.

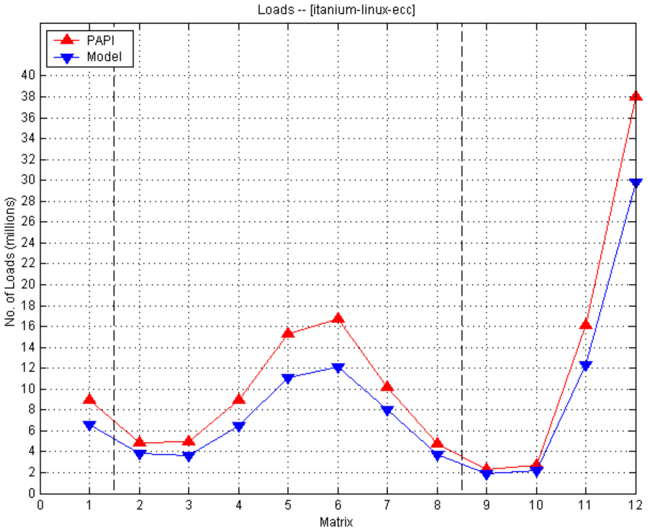


Figure 7: **Load Model Validation – Intel Itanium 1.** Model for the number of issued load instructions on the Intel Itanium 1 compared to PAPI measurements for the block size and vector width that maximizes performance in a fully optimized implementation.

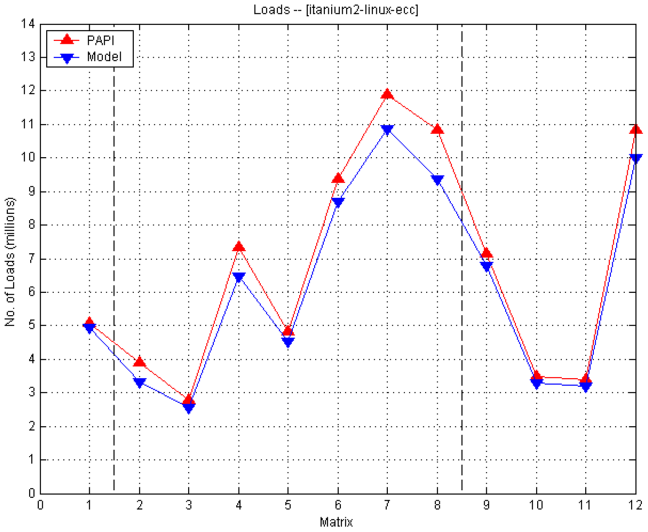


Figure 8: **Load Model Validation – Intel Itanium 2.** Model for the number of issued load instructions on the Intel Itanium 2 compared to PAPI measurements for the block size and vector width that maximizes performance in a fully optimized implementation.

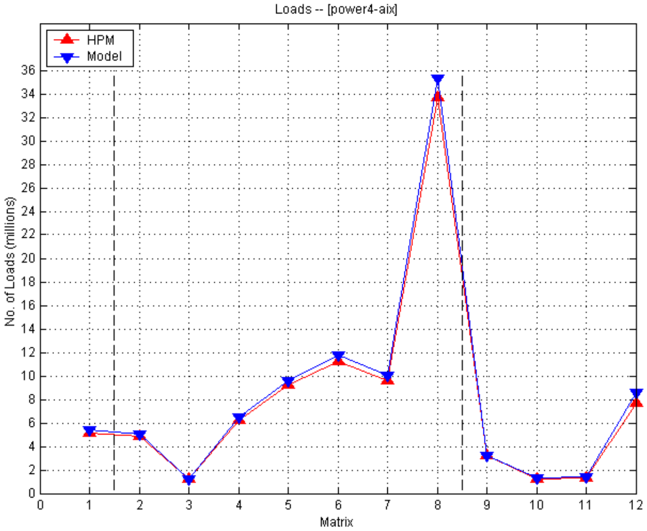


Figure 9: **Load Model Validation – IBM Power 4.** Model for the number of issued load instructions on the IBM Power 4 compared to HPM measurements for the block size and vector width that maximizes performance in a fully optimized implementation. The HPM measurements presented reflect only floating point load instructions.

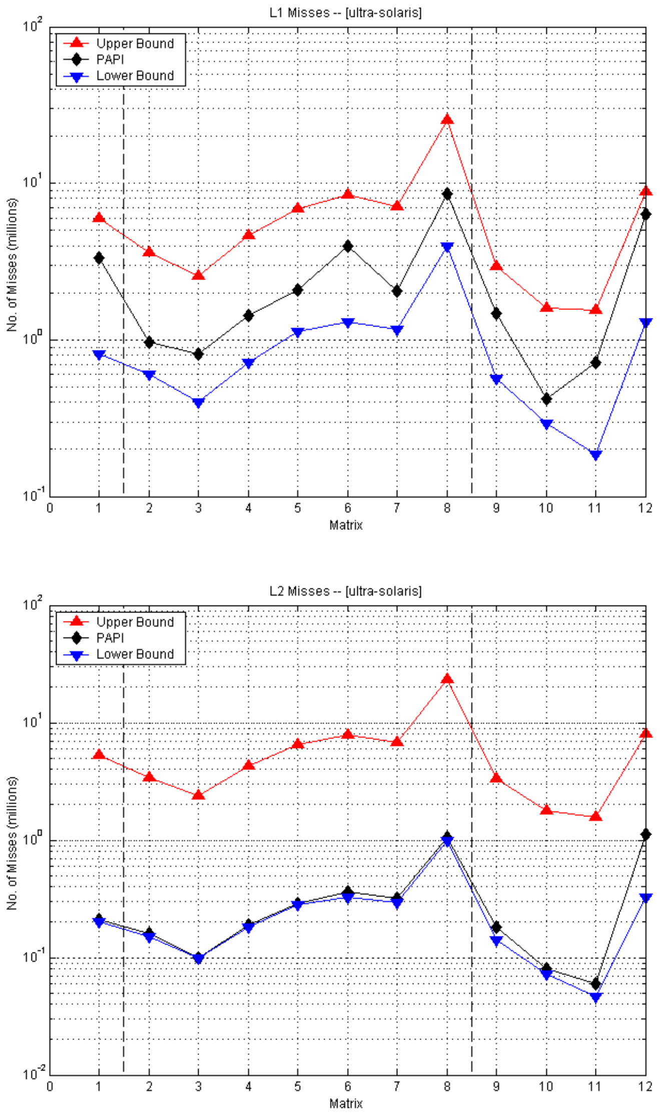
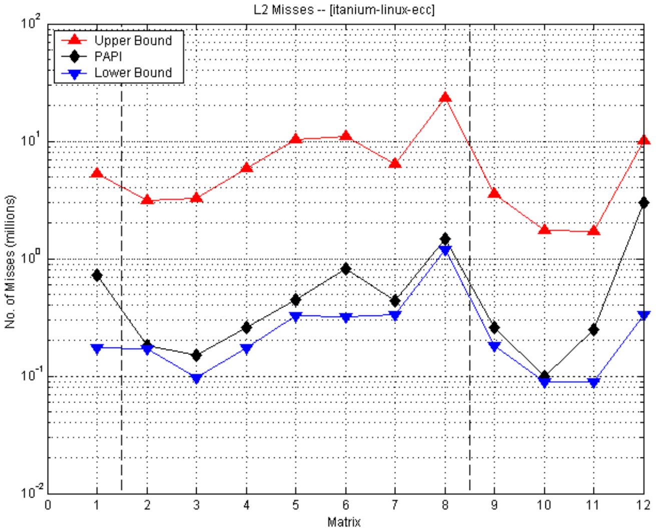


Figure 10: **Cache Miss Model Validation – Sun Ultra 2i.** Upper and lower bounds on L1 and L2 cache misses on the Sun Ultra 2i compared to PAPI measurements for the block size and vector that maximizes performance in a fully optimized implementation. The number of L2 misses match the lower bound well in the larger L2 cache, suggesting the vector sizes are small enough that conflict misses play a relatively small role.



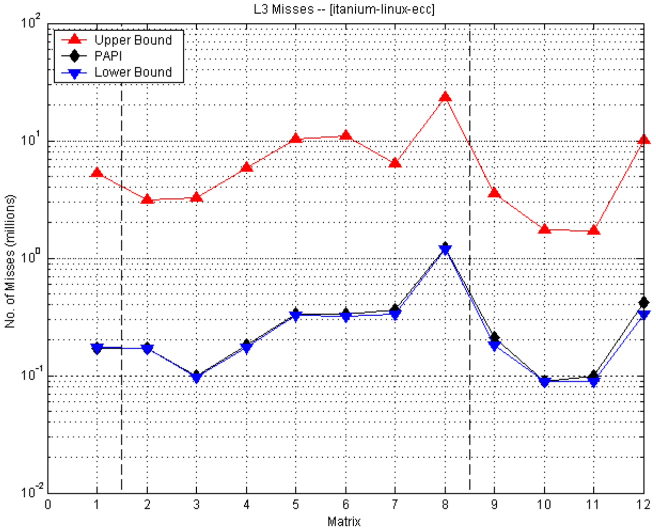
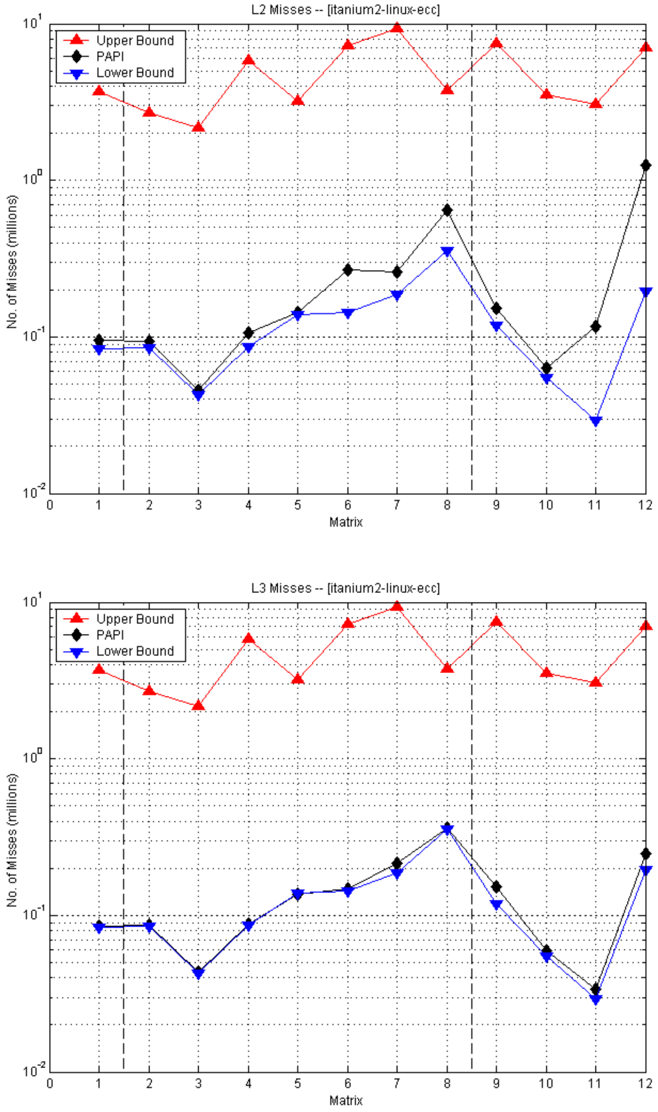
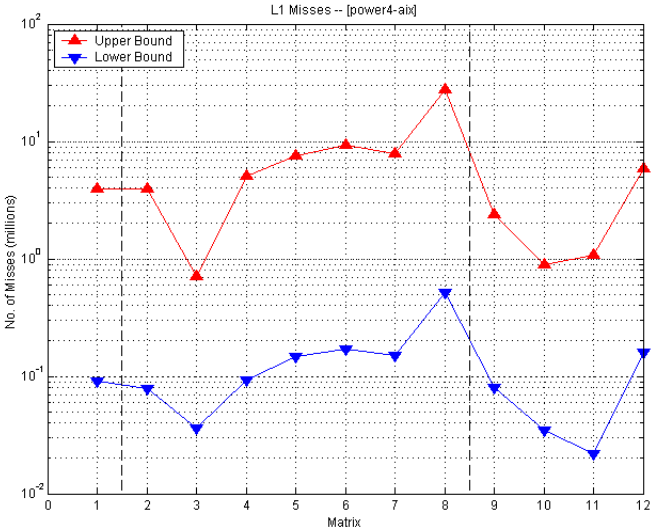


Figure 11: **Cache Miss Model Validation – Intel Itanium 1**. Upper and lower bounds on L2 and L3 cache misses on the Intel Itanium 1 compared to PAPI measurements. As with Figure 10, the lower bounds are a good match in the largest (L3) cache.


L2 Misses -- [itanium2-linux-ecc]

L3 Misses -- [itanium2-linux-ecc]

Figure 12: **Cache Miss Model Validation – Intel Itanium 2**. Upper and lower bounds on L2 and L3 cache misses on the Intel Itanium 2 compared to PAPI measurements. As with Figure 10, the lower bounds are a good match in the largest (L3) cache.



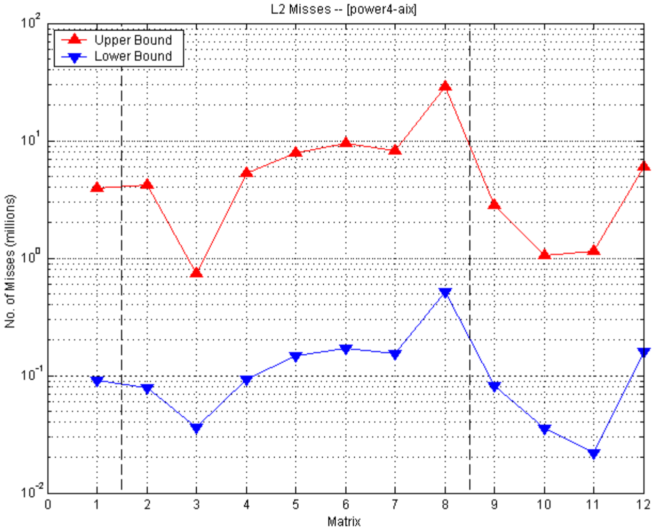


Figure 13: **Cache Miss Model Validation – IBM Power 4.** Upper and lower bounds on L1 and L2 cache misses on the IBM Power 4. The actual number of cache misses are omitted due to PAPI counter unavailibility and significant undercounting with the HPM Tool Kit.

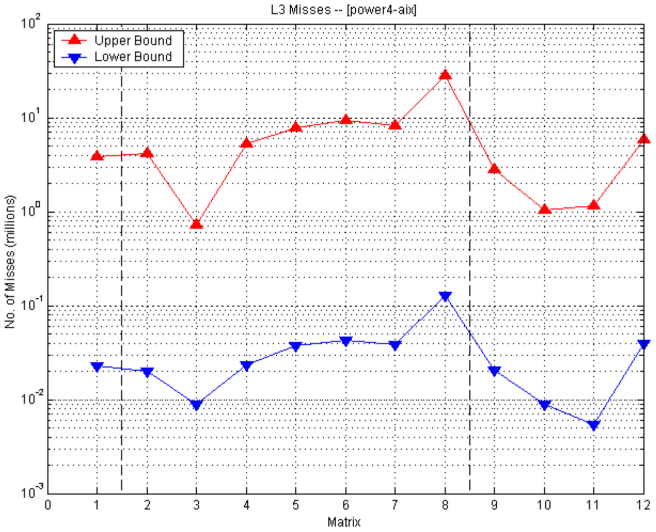


Figure 14: **Cache Miss Model Validation – IBM Power 4.** Upper and lower bounds on L3 cache misses on the IBM Power 4 compared to HPM measurements. As in Figure 13, the actual number of cache misses are omitted due to PAPI counter unavailibility and significant undercounting with the HPM Tool Kit.

6 Evaluation

This section evaluates the effects of various optimizations in the optimization space considered in this report. In Section 6.2 we examine the payoff in performance from exploiting symmetry. In Section 6.3 we examine the payoff in storage space from exploiting symmetry. In Section 6.4, we examine experimental data for any dependence between the register block size that maximizes performance and the matrix, the number of vectors to be multiplied, and the platform. In Section 6.5, we evaluate our performance models as upper bounds on optimized performance. In Section 6.6, we present optimized performance as a fraction of peak performance on each platform.

6.1 Performance Data

We summarize the performance results for various combinations of the optimizations discussed in Section 3 and Section 4. Figures 15–18 summarize the performance of our optimizations on the four hardware platforms in Table 1 and the matrix test suite in Table 2. In particular, we compare the following six implementations:

- **Non-Symmetric Unoptimized Reference**: The unblocked $(1, 1)$ single vector implementation with non-symmetric storage. Refer to Figure 1 for a code sample. Represented by *crosses.*

- **Symmetric Reference**: The unblocked $(1, 1)$ single vector implementation with symmetric storage. Refer to Figure 2 for a code sample. Represented by *five-pointed stars.*

- **Non-Symmetric Register Blocked**: The blocked single vector implementation with non-symmetric storage where r and c are chosen by exhaustive search to maximize performance. Refer to Appendix J for the optimal values of r and c given a matrix and platform. Refer to Appendix A for a code sample. Represented by *asterisks.*

- **Symmetric Register Blocked**: The blocked single vector implementation with symmetric storage where r and c are chosen by exhaustive search to maximize performance. Refer to Appendix J for the optimal values of r and c given a matrix and platform. Refer to Appendix C (bssmvm_m_2x3_1_once) for a code sample. Represented by *plus signs.*

- **Non-Symmetric Register Blocked with Multiple Vectors**: The blocked multiple vector implementation with non-symmetric storage where r, c, and v are chosen by exhaustive search to maximize performance. Refer to Appendix J for the optimal values of r, c, and v given a matrix and platform. Represented by *upward pointing triangles.*

- **Symmetric Register Blocked with Multiple Vectors**: The blocked multiple vector implementation with symmetric storage where r, c, and

v are chosen by exhaustive search to maximize performance. We refer to these parameters as r_{opt}, c_{opt}, and v_{opt}. Refer to Appendix J for the values of r_{opt}, c_{opt}, and v_{opt} given a matrix and platform. Refer to Appendix C (bssmvm_m_2x3_2_once) for a code sample. Represented by *six-pointed stars.*

In addition to the measured performance values for these six implementations, Figures 15–18 also present upper bounds on performance as computed in Section 5. The bounds differ only in the number of loads and cache misses. In particular, we present two upper bounds on optimized performance:

- **Analytic Upper Bound**: The analytic upper bound on performance for the $r_{opt} \times c_{opt}$ blocked implementation with v_{opt} vectors. The number of loads and cache misses are determined from Equation (3) and Equation (5), respectively. Refer to Appendix J for the values of r_{opt}, c_{opt}, and v_{opt} given a matrix and platform. Represented by *solid lines.*
- **PAPI Upper Bound**: An upper bound on performance for the $r_{opt} \times c_{opt}$ blocked implementation with v_{opt} vectors. The number of loads and cache misses are obtained from PAPI event counters. Refer to Appendix J for the values of r_{opt}, c_{opt}, and v_{opt} given a matrix and platform. Represented by *dashed lines.*

We further analyze our results in subsequent tables and figures. In particular, we examine the following effects of our optimizations:

- **Performance Effects of Symmetry:** Table 3 assesses the effect of symmetric storage on performance. The performance speedups are computed for symmetric storage when combined with three levels of optimization: (1) Unoptimized Reference; (2) Register Blocking; (3) Register Blocking and Vector Blocking.
- **Performance Effects of Optimizations with Symmetry:** Table 4 assesses the effect of optimizations, such as register blocking and vector blocking, assuming symmetric storage. The performance speedups are computed for varying combinations of register and vector blocking.
- **Storage Effects of Symmetry:** Table 5 assesses the effect of symmetry optimizations on matrix storage. The savings in storage are not only affected by symmetry, but also by subtleties in register blocking fill and integer storage of matrix row and column indices.
- **Performance as a Percentage of Peak:** Figures 19–20 presents the optimized performance as a percentage of peak machine performance for the four platforms in Table 1.

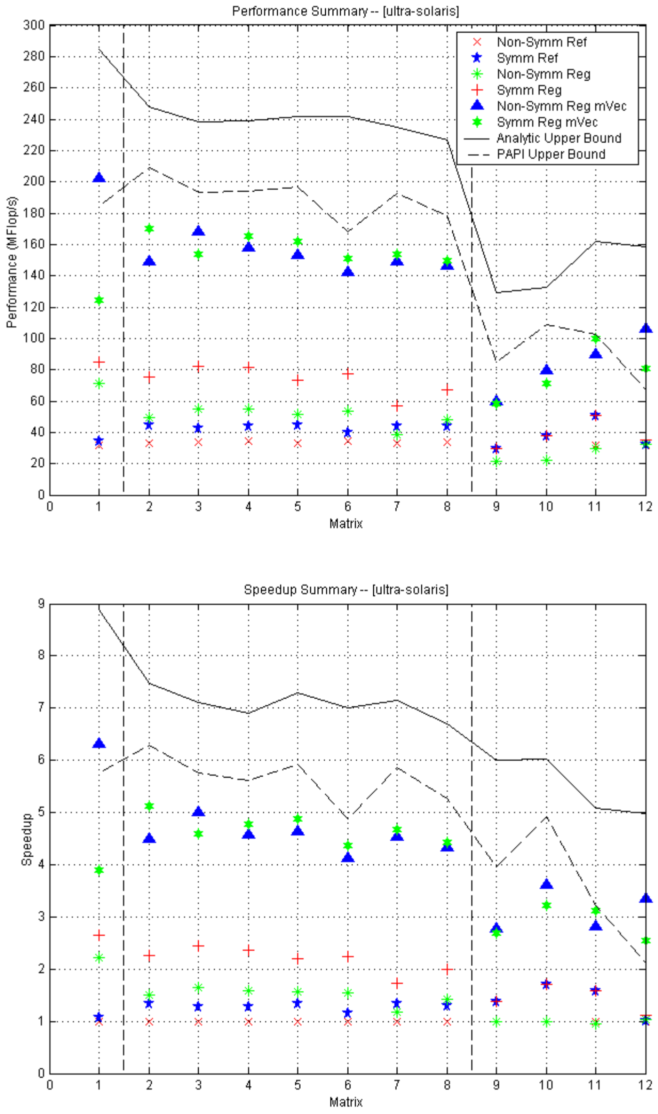


Figure 15: **Performance Summary – Sun Ultra 2i**. Performance (MFlop/s) of various optimized implementations with the optimization parameters chosen by exhaustive search. The performance data is compared to the upper bounds on execution time for the symmetric, register blocked, multiple vector implementation (fully optimized). The data shown is collected on the Sun Ultra 2i. The fully optimized implementation achieves improvements by as much as 5.1× over the non-symmetric reference for this platform.

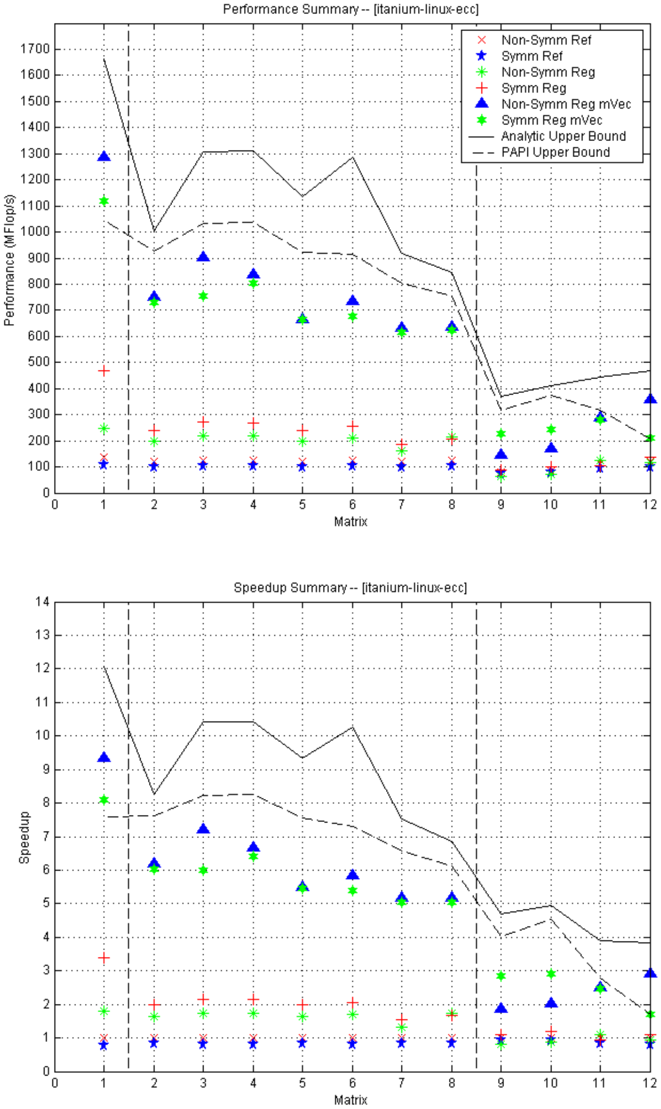


Figure 16: **Performance Summary – Intel Itanium 1.** Performance data and upper bounds shown in a format analogous to the format in Figure 15. The data shown is collected on the Intel Itanium. The fully optimized implementation achieves improvements by as much as 8.1× over the non-symmetric reference for this platform.

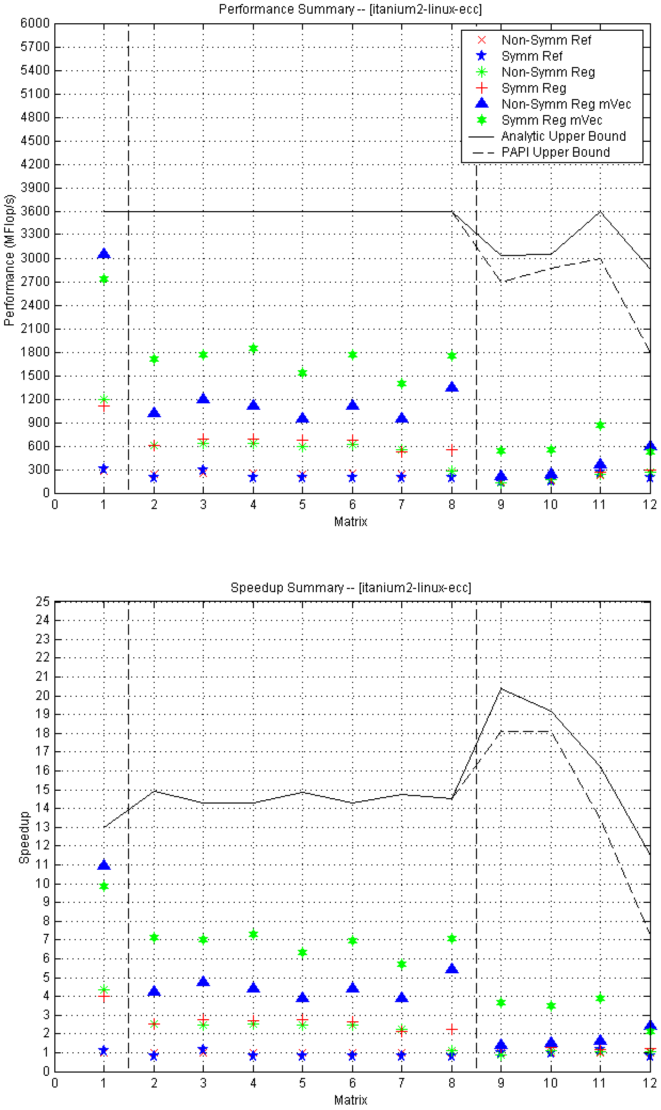


Figure 17: **Performance Summary – Intel Itanium 2.** Performance data and upper bounds shown in a format analogous to the format in Figure 15. The data shown is collected on the Intel Itanium 2. The constant bounds of 3.6 GFlop/s for matrices 1-7 indicate that there is sufficient memory bandwidth to achieve peak speed for matrix multiply operations. The fully optimized implementation achieves improvements by as much as 9.9× over the non-symmetric reference for this platform.

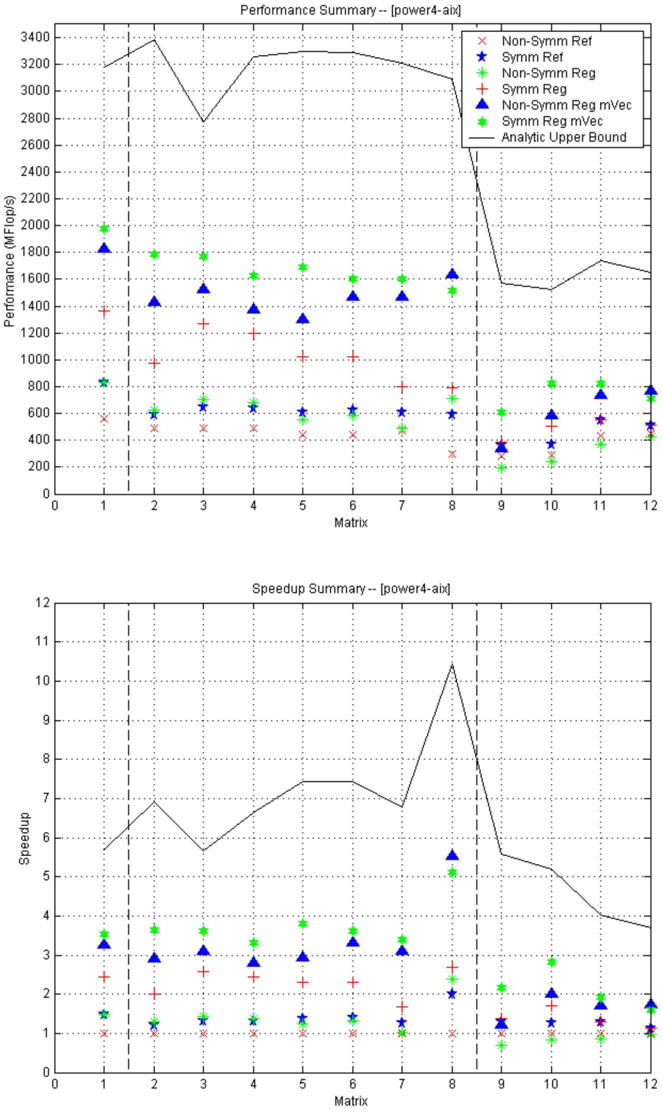


Figure 18: **Performance Summary – IBM Power 4.** Performance data and upper bounds shown in a format analogous to the format in Figure 15. The data shown is collected on the IBM Power 4. The fully optimized implementation achieves improvements by as much as 5.1× over the non-symmetric reference for this platform. The analytic upper bound is omitted due to the unavailability of hardware counters.

	Overall	Ultra 2i	Itanium 1	Itanium 2	Power4
I. Optimizations: None					
minimum	0.80	1.02	0.82	0.80	1.14
median	1.15	1.35	0.83	0.81	1.28
maximum	2.01	1.70	0.95	1.17	2.01
II. Optimizations: Register Blocking					
minimum	0.85	1.09	0.85	0.95	1.12
median	1.34	1.47	1.22	1.11	1.76
maximum	2.08	1.71	1.35	2.01	2.08
III. Optimizations: Register and Vector Blocking					
minimum	0.59	0.76	0.59	0.88	0.93
median	1.09	1.03	0.98	1.62	1.16
maximum	2.58	1.14	1.53	2.58	1.78

Table 3: **Performance Effects of Symmetry**. The table shows the effects of symmetry with other optimizations such as register blocking and vector blocking. The values shown reflect performance speedups of the symmetric implementation of the optimizations specified over the non-symmetric implementation of the same optimizations. Values are reported for each platform and over all platforms.

6.2 Effects of Symmetry on Performance

Symmetry has varying effects on performance depending on the presence of other optimizations such as register blocking and vector blocking. Furthermore, symmetry yields varying results for different platforms and different matrices. Table 3 presents the effects of symmetry as a ratio of optimized performance with and without symmetry. This table effectively represents the advantages of symmetry from the perspective of execution time regardless of storage requirements. Furthermore, the table allows for performance comparisons across platforms and lends itself to several notable observations.

Considering all four platforms, the maximum performance gains from exploiting symmetry at each level of optimization are 2.01×, 2.08×, and 2.58× for the unoptimized reference, register blocking implementation, and register blocking with multiple vector implementation, respectively. However, the overall median speedups of 1.15×, 1.34×, and 1.09× are appreciably lower. Furthermore, the worst case performance gains are all less than one, indicating reduced performance due to symmetry. The register block sizes in Appendix J suggest that these performance decreases are due to block sizes in the symmetric case that lead to significantly more fill than those in the non-symmetric case. Although rare, it possible for symmetry to yield little or no performance gains.

	Overall	Ultra 2i	Itanium 1	Itanium 2	Power4
I. Symmetry **Register** Blocking					
minimum	1.00	1.00	1.08	1.00	1.00
median	1.64	1.54	2.03	2.63	1.36
maximum	3.40	1.93	2.64	3.40	1.96
II. Symmetry Register and **Vector** Blocking					
minimum	1.36	1.87	1.54	1.79	1.36
median	2.28	2.03	2.77	2.71	1.57
maximum	3.34	2.71	3.29	3.34	2.01
III. Symmetry **Register** and **Vector** Blocking					
minimum	1.40	1.89	2.09	2.63	1.40
median	3.44	3.48	6.11	7.07	2.55
maximum	9.19	3.81	7.82	9.19	3.01
IV. Symmetry Register Blocking vs. Naïve					
minimum	0.93	1.11	0.93	1.17	1.14
median	1.78	2.00	1.68	2.24	2.00
maximum	2.76	2.45	2.16	2.76	2.68
V. Symmetry Register and Vector Blocking vs. Naïve					
minimum	1.60	2.56	1.71	2.14	1.60
median	4.15	4.42	5.05	6.36	3.39
maximum	7.32	5.13	6.40	7.32	5.12

Table 4: **Effects of Optimizations in Symmetric Case**. The table shows the effects of symmetry with other optimizations such as register blocking and vector blocking. The values shown reflect performance speedups attributed to the bold optimization (*e.g.* Symmetry **Register** Blocking refers to the performance speedups of the symmetric register blocked implementation over a symmetric implementation). The last two sets of values show the performance speedup of a symmetric register blocked implementation and a fully optimized implementation over the non-symmetric reference implementation.

The effectiveness of register blocking and vector blocking when combined with symmetry is assesed in Table 4. The performance improvements presented in this table indicate the effectiveness of the our combined optimizations. In particular, symmetry and register blocking achieve a maximum speedup of 2.76× and a median speedup of 1.78× over a non-symmetric unoptimized reference (Group IV). In the worst case, performance decreases by 0.93×. However, this case occurs only once for matrix 11 on the Itanium 1. Upon examining Appendix J, we find that the loss in performance is caused by a large fill ratio of 2.49. This fill ratio implies that the symmetric data structure contains almost 60% explicitly stored zeros and is nearly as large as the non-symmetric data structure. This anomaly does not exist for other platforms where the performance speedup is at least 1.11×. Thus, it is possible, although rare, for little or no performance advantage from symmetry.

If we also exploit multiple vectors by implementing vector blocking, significant performance gains are possible. An implementation optimized for symmetry, register and vector blocking achieves maximum, median, and minimum performance gains of 7.32×, 4.15×, and 1.60× over all platforms and matrices.

Given symmetric storage, register and vector blocking almost always yield performance improvements and never reduce performance. This conclusion is apparent from the overall performance gains of groups I-III in Table 4. The increasing performance gains as optimizations are incrementally applied suggest cumulative performance effects of these optimizations. Furthermore, Itanium 2 speedups of 3.40×, 3.34×, and 9.19× for Groups I, II, and III, respectively, suggest multiplicative effects on performance from our optimizations may be possible.

6.3 Effects of Symmetry on Storage

The effects of symmetry optimizations on storage are summarized in Table 5. Assuming floating point data takes twice as much space as integer indices, a large register block size in the symmetric case can reduce storage requirements by as much as a factor of three when compared to a non-symmetric reference implementation. This is possible because the memory needed for matrix indices decreases by up to a factor of $r \times c$, complementing the memory savings from storing only half the matrix elements. However, an increase in space requirements from symmetric storage is also possible, however, if the chosen register block size results in significant fill of explicitly stored zeros.

Table 5 presents the storage savings of a symmetric, register blocked implementation in group I. In particular, we show maximum, median, and minimum savings of 2.84×, 2.30×, and 0.91×, respectively. Symmetry usually saves significantly more than a factor of 2 in space, but may also use almost 10% more memory in the case of matrix 12 on the Itanium 1 and 2. We also present the storage savings of a symmetric, register and vector blocked implementation in

	Overall	Ultra 2i	Itanium 1	Itanium 2	Power4
I. Symmetry Register Blocking					
minimum	0.91	1.55	0.91	0.91	1.47
median	2.40	2.56	1.91	2.38	2.57
maximum	2.84	2.84	2.84	2.84	2.84
II. Symmetry Register and Vector Blocking					
minimum	0.96	1.40	0.96	1.14	1.40
median	2.16	2.16	2.01	2.47	2.16
maximum	2.84	2.33	2.84	2.84	2.84

Table 5: **Memory Usage.** The table shows the effects of symmetry for storage requirements. The values shown reflect reductions in matrix storage requirements as a ratio of storage for an optimized symmetric and non-symmetric implementations. The numerator approximates the storage requirements of a non-symmetric implementation while the denominator approximates the requirements of a symmetric implementation. These approximations are derived in Appendix K.

group II. Adding multiple vectors to group I, we show maximum, median, and minimum savings of 2.84×, 2.16×, 0.96×, respectively.

6.4 Block Size Selection

The experimental data of this report is the result of exhaustive search across all register block sizes up to 8×8 and vector widths up to 10. This report does not propose a scheme for predicting the optimal (r, c, v) based on the matrix, number of vectors multiplied, and platform. The complexity of the dependences between these performance factors motivates the need for heuristic search to efficiently determine optimization parameters that yield acceptable performance (perhaps defined as performance within 10% of the best optimized performance). We discuss any simple patterns, or lack thereof, in the optimal block sizes from Appendix J.

Examining the block sizes for the Ultra 2i in Appendix J, we find that the optimal register block sizes are identical for non-symmetric register blocked code in both the single and multiple vector case. This observation suggests that the block sizes depend on the sparse matrix, but not on the presence of vector blocking. Performing a similar examination for the Itanium 1, Itanium 2, and Power 4 will yield similar observations though not all block sizes are identical.

In contrast, the block sizes for symmetric code are dependent on the presence of vector blocking. In the multiple vector case for the Ultra 2i, the optimal block size is $(2, 1)$ in nine of the eleven matrices and $(1, 1)$ for the other two matrices. However, the optimal block size is never $(2, 1)$ in the single vector case. This pattern also holds true for the Itanium 1, Itanium 2, and Power 4.

Comparing the block sizes for the non-symmetric and symmetric implementations of register blocking for the Ultra 2i, show five matrices for which the block sizes are different. These differences, however, are usually cases where one block dimension (*i.e.* row or column) is an integer multiple of the same block dimension for the differing block size. We refer to this difference as *commensurate*. For example, block sizes (6,6) and (3,3) are commensurate, but block sizes (2,2) and (4,3) are not. Performing a similar comparison for non-symmetric and symmetric implementations of register blocking with multiple vectors for the Ultra 2i, we find differing block sizes for nine of the eleven matrices. Five of these nine differences are commensurate. Similar comparisons for the Itanium 1, Itanium 2, and Power 4 will yield similar patterns where the difference between block sizes are often commensurate.

To summarize our observations and their implications on block size selection, we note the optimal block size (r, c) does not depend strongly on the vector width in a non-symmetric implementation. In contrast, the addition of symmetry reveals an empirical correlation between block size and multiplication by multiple vectors. Despite fewer differing block sizes between symmetric and non-symmetric implementations in the single vector case, these differences occur frequently enough to preclude safe prediction of the optimal symmetric register block size from the non-symmetric register block size.

Searching for patterns in the optimal block sizes chosen across the four platforms, we find the block sizes tend to be small. Let us define acceptable performance as performance within 10% of the maximum optimized performance achieved from a block size chosen with exhaustive search. For the symmetric, register and vector blocked implementation, it is almost always possible to find a block size smaller than $(3, 3)$ to yield acceptable performance. Furthermore, it always possible to achieve acceptable performance from a block size with a row dimension equal to 1 and a column dimension less than 3 for the Ultra 2i, Itanium 2, and Power 4. Despite these patterns, however, we find the performance profile and register block sizes that yield acceptable performance for the dense matrix are not representative of those for sparse matrices. The dense register blocks tend to have larger block sizes that would be ineffective for sparse matrices. Nonetheless, these patterns suggest that a heuristic for selecting register block sizes could effectively limit the search space to smaller register blocks without significantly compromising performance when optimizations for symmetry and multiple vectors have been implemented.

6.5 Evaluation of Performance Upper Bounds

To evaluate the effectiveness of our performance models, we consider the proximity of the analytic and PAPI upper bounds to the measured performance of the symmetric, register and vector blocked implementation. In our evaluation of the upper bounds, we will distinguish between FEM matrices (matrices 2-8 in our test suite) and non-FEM matrices.

The measured performance of FEM matrices on the Ultra 2i and Itanium 1 is within 80% to 90% and 72% to 83% of the PAPI upper bound, respectively. In contrast, the measured performance of FEM matrices on these platforms only achieve 64% to 70% and 53% to 73% of the analytic upper bound, respectively. This difference suggests that further performance improvements on these platforms will require understanding and reducing the gap between the number of predicted and measured cache misses.

For the Itanium 2, the analytic and PAPI upper bounds predict that our optimized routines achieve machine peak of 3.6 Gflops (*i.e.*the code is not memory bound) for matrices 2 through 7. However, Figure 19 indicates the true performance of these matrices is between 38% to 51% of peak. The measured performance of matrix 8 is slightly higher than the PAPI upper bound (for reasons we do not understand) and within 88% of the analytic upper bound.

For the Power 4, the measured performance of FEM matrices is between 48% and 63% of the analytic upper bound. No PAPI data was available to calculate a PAPI upper bound for this platform.

The non-FEM matrices on the Ultra 2i and Itanium 1 achieve lower measured performance relative to the FEM matrices. In particular, measured performance of non-FEM matrices was within 65% to 120% and 65% to 105% of the PAPI upper bound, respectively. On both platforms, the measured performance for matrix 12 is higher than the PAPI upper bound on both platforms (for reasons we do not understand). The measured performance of non-FEM matrices on these platforms is within 44% to 61% and 44% to 62% of the analytic upper bound, respectively.

There is significant potential for non-FEM performance improvement on the Itanium 2 and Power 4. The measured performance of non-FEM matrices on the Itanium 2 is only within 29% to 38% of the PAPI upper bound. No PAPI data was available to calculate a PAPI upper bound for the Power 4. The measured performance for the Itanium 2 and Power 4 is only within 23% to 32% and 38% to 54% of the analytic upper bound, respectively.

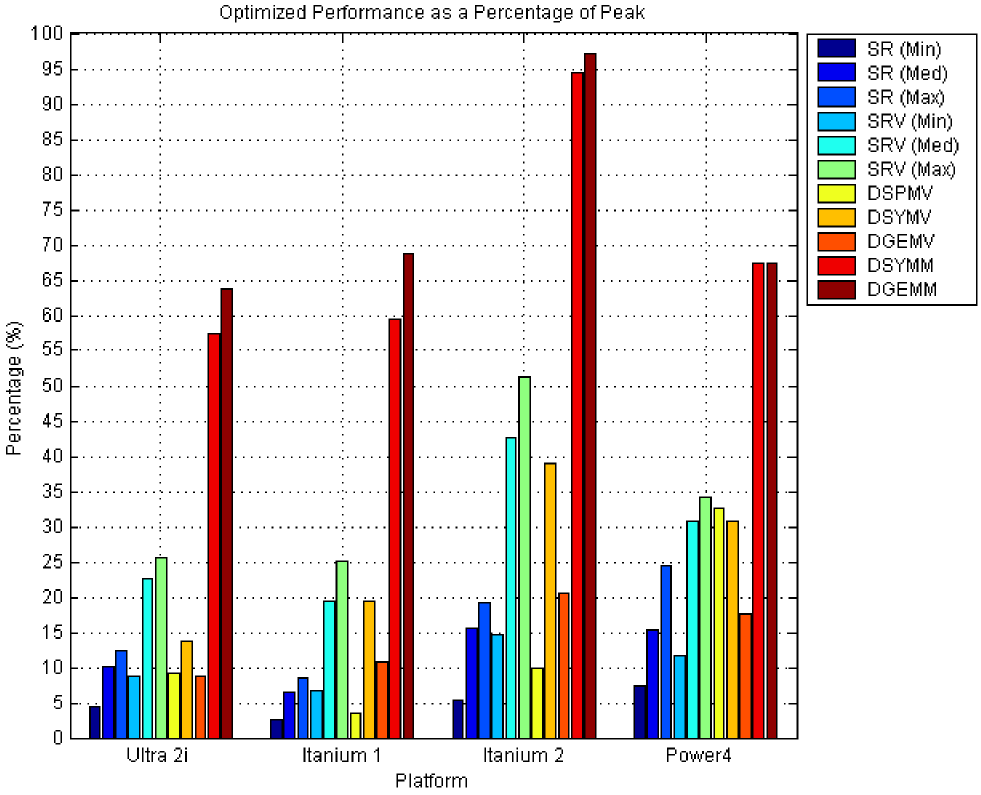


Figure 19: **Performance as a Percentage of Peak.** The figure summarizes the performance of our optimized SpMM implementation across all matrices in the test suite as a percentage of each platform's peak FLOP rate (Table 1). Performance is presented for the following implementations: symmetry and register blocking (SR); symmetry, register and vector blocking (SRV); BLAS implementations (DSPMV, DSYMV, DGEMV, DSYMM, DGEMM).

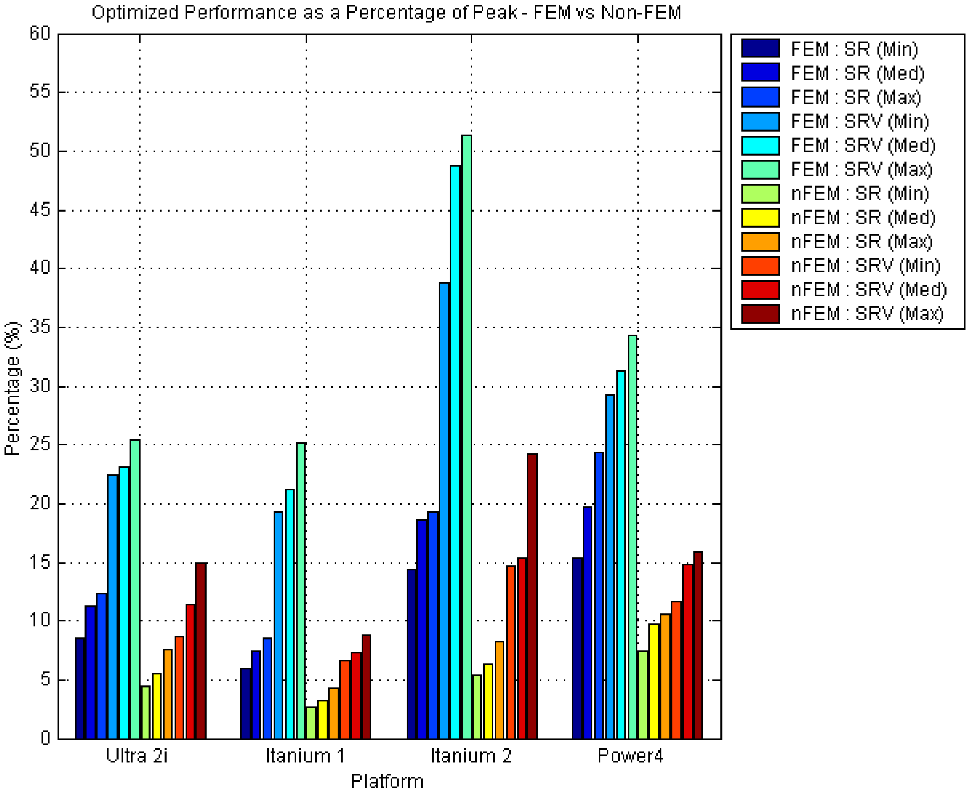


Figure 20: **Performance as a Percentage of Peak (FEM vs. non-FEM).** The figure compares the performance of our optimized SpMM implementation for matrices from finite element applications (FEM) and all other matrices (nFEM). Performance is presented as a percentage of each platform's peak FLOP rate (Table 1). Performance is presented for the following implementations: symmetry and register block (SR); symmetry, register and vector blocking (SRV).

6.6 Peak Performance

The overall performance of our matrix test suite is presented in Figure 19 as a percentage of the peak Mflop rate for each of the four platforms. This representation of the performance data is interesting because the percentages of peak achieved by the dense Level 2 BLAS matrix-matrix multiply (DSYMM and DGEMM) are implicit upper bounds on our optimized performance.

In general, Figure 19 indicates the maximum and median percentages of peak performance for the Itanium 2 and the Power 4 are significantly higher than the corresponding values for the Ultra 2i and the Itanium 1. This observation is true for both the symmetric register blocked implementation (SR) and the symmetric, register and vector blocked implementation (SRV).

The median percentage of peak for the symmetric register blocked implementation (SR) actually exceeds the percentage of peak for the dense Level 2 BLAS DSPMV on all platforms except the Power 4. Furthermore, the median percentage of peak for the symmetric, register and vector blocked implementation (SRV) is significantly greater than the percentage of peak for all the dense Level 2 BLAS codes (DSPMV, DSYMV, DGEMV). It is remarkable that sparse codes perform as well as dense codes.

Figure 20 distinguishes between the performance of matrices from finite element applications (FEM) and all other matrices (non-FEM). The overall performance of FEM matrices is higher than the overall performance of the non-FEM matrices in our test suite. Register blocking for FEM matrices tends to yield larger performance gains because of the natural dense structure of these matrices.

Figure 20 also indicates a combination of symmetry, register blocking, and vector blocking will generally perform better than a combination of only symmetry and register blocking. This is observed by comparing the percentages of peak for the symmetric, register and vector blocked implementation (SRV) and those for the symmetric register blocked implementation (SR). For both FEM and non-FEM matrices across all platforms, the percentages for optimizations that include vector blocking are significantly larger than those for optimizations for the single vector case. In particular, the median percentages of SRV are almost always at least a factor of two larger than those for SV for all four platforms.

7 Related Work

For dense algorithms, a variety of sophisticated static models for selecting transformations and tuning parameters have been developed, each with the goal of providing a compiler with sufficently precise models for selecting memory hierarchy transformations and parameters such as tile sizes [32, 33, 34, 35]. 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 for SpMV. Temam and Jalby [37], Heras, *et al.* [38], and Fraguela, *et al.* [39] have developed sophisticated probablistic cache miss models for SpMV, but assume uniform distribution of non-zero entries. These models vary in their ability to account for self-interference and cross-interference misses. To obtain lower bounds, we account only for compulsory and capacity misses.

Gropp, *et al.*, use bounds like the ones we develop to analyze and tune a computation fluid dynamics code [28] on Itanium 1. However, we are interested in tuning for matrices that come from a variety of domains and machine architectures. Furthermore, our bounds explicitly model execution time (instead of only modeling cache misses) in order to evaluate the extent to which our tuned implementations achieve optimal performance.

Work in sparse compilers, e.g., Bik *et al.* [40], Pugh and Spheisman [41], and the Bernoulli compiler [42], 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, architecture-specific model for selecting transformations (tuning parameters).

8 Conclusions and Future Directions

The simultaneous application of symmetric storage, register blocking, and vector blocking for SpMM yields substantial improvements in performance of up to 7.3× for a sparse matrix compared to a baseline implementation in which none of these optimizations are applied and up to 2.6× compared to a non-symmetric (full-storage), register and vector blocked implementation. Furthermore, the performance effects of each optimization in our optimization space (symmetry, register blocking, vector blocking) appear to be cumulative, making the case for the benefits of implementing all three techniques in conjunction. Future work will lie in four broad areas: (1) heuristic block size selection, (2) performance model refinement, (3) code generation, and (4) related optimizations.

Previous work has included the development of heuristics to select the optimization parameters. Such work includes the SPARSITY heuristic (Version 2.0) [2] that selects the register block size, given the performance profile of a particular platform, independently of vector width. We are currently pursuing a comprehensive heuristic that selects the register block size and vector width simultaneously, maximizing performance as function of r, c, and v.

The modest accuracy achieved by the performance models presented in this report reflect a relatively optimistic performance upper bound. We believe that the high bounds reflect issues arising from the multiple vector implementation of the kernel and the assumptions made in the development of the performance models (assumptions 1 and 3 from Section 5). As the number of vectors increases, conflict misses may occur as the number of vector elements exceeds the number of available registers. Furthermore, not all vectors may fit in cache as the vector width increases. Model refinements could potentially account for the additional memory traffic incurred as the number of vectors increases and more accurately predict execution time spent in the memory hierarchy.

Closing the gap between the realized performance and the upper bound may require automatic tuning techniques to explore a larger optimization space to maximize performance for a given platform. Future work to improve realized performance may include techniques used in ATLAS and PhiPAC [22, 5] and to produce parameterized code generators whose parameters are relevant to the resulting machine performance. The generated code would also follow machine-specific coding optimizations such as manually unrolling loops, explicitly removing unnecessary dependencies in code blocks, and using machine sympathetic C constructs. These optimizations would augment the current performance gains achieved by individually tuning the SpMM routines for a particular platform.

Future work in optimization techniques may involve variations of those described in this report. In particular, sparse kernels may be optimized to exploit other forms of symmetry:

- **Structural**: Structural symmetry refers to a matrix in which the position of non-zero elements are symmetric about the diagonal, but the non-zero values themselves are not. Assuming that structural symmetry provides a performance advantage, a conceivable optimization would fill an "almost symmetric" matrix with zeros to make the matrix structurally symmetric.
- **Skew**: Skew symmetry refers to a matrix in which each transpose element equals the negative stored element.
- **Hermitian**: Hermitian symmetry refers to a matrix in which each transpose element equals the complex conjugate of the stored element.
- **Skew Hermitian**: Skew Hermitian symmetry refers to a matrix in which each tranpose element equals the negative complex conjugate of the stored element.

The increasing size of matrices and vectors reflects growing storage requirements in cache that could limit the effectiveness of the multiple vector optimizations.

In particular, the assumption that all vectors fit in cache is invalid for large vector widths. These effects may be mitigated by cache blocking, a technique to group the matrix into larger blocks whose sizes are determined by the cache size. Each cache block would be considered separately and only portions of the matrix and the corresponding vectors would need to be considered at any given time, thereby reducing the vector storage requirements in cache.

Lastly, a variation on the vector blocking optimization is the technique of field interlacing [28], equivalent to vector row storage described in Section 4. This optimization involves interleaving the source vectors and interleaving the destination vectors to create spatial locality for successive vector accesses. The primary advantages of this technique include improved data reuse for data retrieved from cache and reduced TLB misses.

Acknowledgements

We would like to thank Tuyet-Linh Phan for experimental work and data on non-symmetric multiple vector optimizations that were used for baseline comparisons in this report.

References

[1] R. Vuduc, A. Gyulassy, J. Demmel, K. Yelick. *Memory Hierarchy Optimizations and Performance Bounds for Sparse Sp*$A^T A$. In *ICCS 2003: Workshop on Parallel Linear Algebra.*

[2] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, B. Lee. *Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply.* In *Proceedings of the IEEE/ACM Conference on Supercomputing, Baltimore, MD*, November 2002.

[3] 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, 1994.

[4] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. *Intel Technology Journal*, 5(Q1), 2001.

[5] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proc. of the Int'l Conf. on Supercomputing, Vienna, Austria*, July 1997.

[6] R. Biswas, X. Li, P. Husbands, and L. Oliker. Ordering sparse matrices for cache-based systems. In *Proc. of the SIAM Conf. on Parallel Processing*, 2001.

[7] 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. `www.netlib.org/blast`.

[8] T. Davis. University of Florida sparse matrix collection. `www.cise.ufl.edu/~davis/sparse`.

[9] J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[10] J. Dongarra, J. D. Croz, I. Duff, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[11] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of the Int'l Conf. on Acoustics, Speech, and Signal Processing*, May 1998.

[12] G. Haentjens. An investigation of recursive FFT implementations. Master's thesis, Carnegie Mellon University, 2000.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5(Q1), 2001.

[14] E.-J. Im. *Optimization the Performance of Sparse Matrix–Vector Multiplication.* PhD thesis, University of California, Berkeley, May 2000.

[15] E.-J. Im and K. Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Proc. of the 9th SIAM Conf. on Parallel Processing for Sci. Comp.*, March 1999.

[16] E.-J. Im and K. A. Yelick. Optimizing sparse matrix–vector multiplication for register reuse. In *Proceedings of the International Conference on Computational Science*, May 2001.

[17] E.-J. Im and K. A. Yelick. Optimizing sparse matrix kernels for data mining. In *SIAM International Conference on Data Mining*, April 2001.

[18] Intel. Math Kernel Library, v5.1. `developer.intel.com`.

[19] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

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

[21] S. Toledo. Improving instruction-level parallelism in sparse matrix-vector multiplication using reordering, blocking, and prefetching. In *Proc. of the 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

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

[23] 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.

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

[25] J. D. McCalpin. STREAM: Measuring sustainable memory bandwidth in high performance computers, 1995. `http://www.cs.virginia.edu/stream`.

[26] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London, 1997. Chapman and Hall. `math.nist.gov/MatrixMarket`.

[27] T. Davis. UF Sparse Matrix Collection. www.cise.ufl.edu/research/sparse/matrices.

[28] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. High performance parallel implicit CFD. *Parallel Computing*, 27(4), March 2001.

[29] A. Snavely, N. Wolter, and L. Carrington. *Modeling Application Performance by Convolving Machine Signatures with Application Profiles.* In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization, Austin, TX*, December, 2001.

[30] K. Goto and R. van de Geijn. *On reducing TLB misses in matrix multiplication.* Technical Report(TR-2002-55), University of Texas at Austin, November, 2002.

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

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

[33] 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 Progarmming Language Design and Implementation*, pages 286-297, Snowbird, UT, USA, June 2001.

[34] 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.

[35] 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.

[36] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 9́1 Conference on Programming Language Design and Implementation*, June 1991.

[37] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing 9́2*, 1992.

[38] 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.

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

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

[41] W. Pugh and T. Spheisman. Generation of efficient code for sparse matrix computations. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing*, LNCS, August 1998.

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

A Non-symmetric Register Blocked Example

The following is a C implementation of a 2x2 register blocked code. Here, `bm` is the number of block rows, i.e., the number of rows in the matrix is `2*bm`. The dense sub-blocks are stored in row-major order.

```
void smvm_regblk_2x2( int bm, const int *b_row_start,
               const int *b_col_idx, const double *b_value,
               const double *x, double *y )
{
    int i, jj;

    /* loop over block rows */
    for( i = 0; i < bm; i++, y += 2 )
    {
        register double d0 = y[0];
        register double d1 = y[1];
        for( jj = b_row_start[i]; jj < b_row_start[i+1];
                jj++, b_col_idx++, b_value += 2*2 )
        {
            d0 += b_value[0] * x[*b_col_idx+0];
            d1 += b_value[2] * x[*b_col_idx+0];
            d0 += b_value[1] * x[*b_col_idx+1];
            d1 += b_value[3] * x[*b_col_idx+1];
        }
        y[0] = d0;
        y[1] = d1;
    }
}
```

B Modifications to NIST Sparse BLAS

The following is an adaptation of the NIST Sparse BLAS routine. The original routine implements $y = Ax$ whereas the routine below implements $y = y + Ax$.

```
void CSR_VecMult_CAB_double_accum(
                 const int m,  const int k,
                 const double *val, const int *indx,
                 const int *pntrb, const int *pntre,
                 const double *b,
                 double *c,
                 const int ind_base)
{
    double t;
    const double *pval;
    double *pc=c;
    int i,j,jb,je;
    b-=ind_base;
    indx-=ind_base;

    pval = val;
    for (i=0;i!=m;i++) {
        t = 0;
        jb = pntrb[i];
        je = pntre[i];
        for (j=jb;j!=je;j++)
            t +=  b[indx[j]] * (*pval++);
        c[i] += t;
    }
}
```

C Symmetric Register & Vector Blocked Example

The following is an implementation of a symmetric, register blocked, vector blocked code with 2 × 3 register blocks and vector width of 2. Here, `bm` is the number of block rows, i.e., the number of rows in the matrix is `2*bm`. The dense sub-blocks are stored in row-major order.

```
void bssmvm_m_2x3_2 (int m, int *b_row_start, int *b_col_idx, double *b_value,
               int src_len, double *src, int dest_len, double *dest, int vec_width)
{
  int l;
  int full_loops, short_loops;
  full_loops = vec_width / 2;
  short_loops = vec_width % 2;

  /* Dispatch to subroutine -- depends on the length of destination vectors */
  /* Handle vector blocks */
  for (l=0; l<full_loops; l++, src+=2*src_len, dest+=2*dest_len)
  {
    bssmvm_m_2x3_2_once (m, b_row_start, b_col_idx, b_value, src_len, src, dest_len, dest, vec_width);
  }

  /* Handle remaining vectors */
  switch (short_loops)
  {
    case 1:
      bssmvm_m_2x3_1_once (m, b_row_start, b_col_idx, b_value, src_len, src, dest_len, dest, vec_width);
      break;
  }
}
```

```
/* Routine that performs the SMVM for one vector */
void bssmvm_m_2x3_1_once (int m, int *b_row_start, int *b_col_idx, double *b_value,
               int src_len, double *src, int dest_len, double *dest, int vec_width)
{
  int ri, rimax, ci, bri, bci, brimax, idx;
  int blocks, degenerate, diag;
  int *b_max;
  int degen_max;
  double *dest_p;
  brimax = m/2;
  dest_p = dest;

  /* Loop over block rows */
  for (bri=0; bri<brimax; bri++,dest+=2)
  {
    register double d0_0;
    register double d1_0;
    register double s0_0;
    register double s1_0;

    bci = b_row_start[bri+1] - b_row_start[bri]; /* Number of block columns */
    if (bci == 0)  continue; /* No elements in this block row */

    d0_0 = dest[0*dest_len+0];
    d1_0 = dest[0*dest_len+1];
    s0_0 = src[(0*src_len)+(2*bri+0)];
    s1_0 = src[(0*src_len)+(2*bri+1)];

    /* Handle diagonal block, if necessary */
    b_max = b_col_idx + bci;
    if (*b_col_idx == 2*bri)
    {
      d0_0 += b_value[0] * s0_0;
      d1_0 += b_value[1] * s0_0;
      d0_0 += b_value[1] * s1_0;
      d1_0 += b_value[2] * s1_0;
      b_value+=3;
      b_col_idx++;

      bci--;
      if (bci == 0)
      {
        dest[0*dest_len+0] = d0_0;
        dest[0*dest_len+1] = d1_0;
        continue;
      }
    }

    /* Handle degenerate block, if necessary */
    if (*b_col_idx == 2*(bri+1))
    {
      degen_max = (m-*b_col_idx)%3;
      if (degen_max != 0)
      {
        for (ci=0;ci<degen_max; b_value++, ci++)
        {
          dest_p[0*dest_len + (*b_col_idx+ci)] += b_value[0*degen_max] * s0_0;
          d0_0 += src[0*src_len + (*b_col_idx+ci)] * b_value[0*degen_max];
          dest_p[0*dest_len + (*b_col_idx+ci)] += b_value[1*degen_max] * s1_0;
          d1_0 += src[0*src_len + (*b_col_idx+ci)] * b_value[1*degen_max];
        }
        b_col_idx++;
        b_value+=degen_max*1;
      }
    }

    /* Handle register blocks */
    for (; b_col_idx<=b_max-1; b_col_idx+=1,b_value+=6)
    {
      register double t0_0 = dest_p[0*dest_len + *b_col_idx];
      register double t1_0 = dest_p[0*dest_len + *b_col_idx+1];
      register double t2_0 = dest_p[0*dest_len + *b_col_idx+2];

      /* Handle stored register blocks */
      t0_0 += b_value[0] * s0_0;
      t0_0 += b_value[3] * s1_0;
      t1_0 += b_value[1] * s0_0;
      t1_0 += b_value[4] * s1_0;
      t2_0 += b_value[2] * s0_0;
      t2_0 += b_value[5] * s1_0;

      /* Handle transpose register blocks */
      d0_0 += src[0*src_len + *b_col_idx] * b_value[0];
      d1_0 += src[0*src_len + *b_col_idx] * b_value[3];
      d0_0 += src[0*src_len + *b_col_idx+1] * b_value[1];
      d1_0 += src[0*src_len + *b_col_idx+1] * b_value[4];
      d0_0 += src[0*src_len + *b_col_idx+2] * b_value[2];
      d1_0 += src[0*src_len + *b_col_idx+2] * b_value[5];

      dest_p[0*dest_len + *b_col_idx] = t0_0;
      dest_p[0*dest_len + *b_col_idx+1] = t1_0;
      dest_p[0*dest_len + *b_col_idx+2] = t2_0;
    }
    dest[0*dest_len+0] = d0_0;
    dest[0*dest_len+1] = d1_0;
  } /* End loop over rows */

  /* Handle any remaining, unblocked rows of the matrix */
  rimax = m - 2*brimax;
  idx = 0;
  for(ri=0; ri<rimax; ri++)
  {
    dest_p[0*dest_len+(*b_col_idx+ri)] += b_value[idx] * src[0*src_len+(*b_col_idx+ri)];
    idx++;
    for(ci=ri+1; ci<rimax; ci++, idx++)
    {
      dest_p[0*dest_len+(*b_col_idx+ci)] += b_value[idx] * src[0*src_len+(*b_col_idx+ri)];
      dest_p[0*dest_len+(*b_col_idx+ri)] += src[0*src_len+(*b_col_idx+ci)] * b_value[idx];
    }
  }
}
```

```
/* Routine that performs the SMVM for two vectors */
void bssmvm_m_2x3_2_once (int m, int *b_row_start, int *b_col_idx, double *b_value,
               int src_len, double *src, int dest_len, double *dest, int vec_width)
{
  int ri, rimax, ci, bri, bci, brimax, idx;
  int blocks, degenerate, diag;
  int *b_max;
  int degen_max;
  double *dest_p;
  brimax = m/2;
  dest_p = dest;

  /* Loop over block rows */
  for (bri=0; bri<brimax; bri++,dest+=2)
  {
    register double d0_0, d0_1;
    register double d1_0, d1_1;
    register double s0_0, s0_1;
    register double s1_0, s1_1;

    bci = b_row_start[bri+1] - b_row_start[bri]; /* Number of block columns */
    if (bci == 0)  continue; /* No elements in this block row */

    d0_0 = dest[0*dest_len+0];
    d0_1 = dest[1*dest_len+0];
    d1_0 = dest[0*dest_len+1];
    d1_1 = dest[1*dest_len+1];
    s0_0 = src[(0*src_len)+(2*bri+0)];
    s0_1 = src[(1*src_len)+(2*bri+0)];
    s1_0 = src[(0*src_len)+(2*bri+1)];
    s1_1 = src[(1*src_len)+(2*bri+1)];

    /* Handle diagonal block, if necessary */
    b_max = b_col_idx + bci;
    if (*b_col_idx == 2*bri)
    {
      d0_0 += b_value[0] * s0_0;
      d1_0 += b_value[1] * s0_0;
      d0_0 += b_value[1] * s1_0;
      d1_0 += b_value[2] * s1_0;
      d0_1 += b_value[0] * s0_1;
      d1_1 += b_value[1] * s0_1;
      d0_1 += b_value[1] * s1_1;
      d1_1 += b_value[2] * s1_1;
      b_value+=3;
      b_col_idx++;

      bci--;
      if (bci == 0)
      {
        dest[0*dest_len+0] = d0_0;
        dest[0*dest_len+1] = d1_0;
        dest[1*dest_len+0] = d0_1;
        dest[1*dest_len+1] = d1_1;
        continue;
      }
    }

    /* Handle degenerate block, if necessary */
    if (*b_col_idx == 2*(bri+1))
    {
      degen_max = (m-*b_col_idx)%3;
      if (degen_max != 0)
      {
        for (ci=0;ci<degen_max; b_value++, ci++)
        {
          dest_p[0*dest_len + (*b_col_idx+ci)] += b_value[0*degen_max] * s0_0;
          d0_0 += src[0*src_len + (*b_col_idx+ci)] * b_value[0*degen_max];
        dest_p[0*dest_len + (*b_col_idx+ci)] += b_value[1*degen_max] * s1_0;
        d1_0 += src[0*src_len + (*b_col_idx+ci)] * b_value[1*degen_max];
        dest_p[1*dest_len + (*b_col_idx+ci)] += b_value[0*degen_max] * s0_1;
        d0_1 += src[1*src_len + (*b_col_idx+ci)] * b_value[0*degen_max];
        dest_p[1*dest_len + (*b_col_idx+ci)] += b_value[1*degen_max] * s1_1;
        d1_1 += src[1*src_len + (*b_col_idx+ci)] * b_value[1*degen_max];
      }
      b_col_idx++;
      b_value+=degen_max*1;
    }
  }

  /* Handle register blocks */
  for (; b_col_idx<=b_max-1; b_col_idx+=1,b_value+=6)
  {
    register double t0_0 = dest_p[0*dest_len + *b_col_idx];
    register double t0_1 = dest_p[1*dest_len + *b_col_idx];
    register double t1_0 = dest_p[0*dest_len + *b_col_idx+1];
    register double t1_1 = dest_p[1*dest_len + *b_col_idx+1];
    register double t2_0 = dest_p[0*dest_len + *b_col_idx+2];
    register double t2_1 = dest_p[1*dest_len + *b_col_idx+2];

    /* Handle stored register blocks */
    t0_0 += b_value[0] * s0_0;
    t0_1 += b_value[0] * s0_1;
    t0_0 += b_value[3] * s1_0;
    t0_1 += b_value[3] * s1_1;
    t1_0 += b_value[1] * s0_0;
    t1_1 += b_value[1] * s0_1;
    t1_0 += b_value[4] * s1_0;
    t1_1 += b_value[4] * s1_1;
    t2_0 += b_value[2] * s0_0;
    t2_1 += b_value[2] * s0_1;
    t2_0 += b_value[5] * s1_0;
    t2_1 += b_value[5] * s1_1;

    /* Handle transpose register blocks */
    d0_0 += src[0*src_len + *b_col_idx] * b_value[0];
    d0_1 += src[1*src_len + *b_col_idx] * b_value[0];
    d1_0 += src[0*src_len + *b_col_idx] * b_value[3];
    d1_1 += src[1*src_len + *b_col_idx] * b_value[3];
    d0_0 += src[0*src_len + *b_col_idx+1] * b_value[1];
    d0_1 += src[1*src_len + *b_col_idx+1] * b_value[1];
    d1_0 += src[0*src_len + *b_col_idx+1] * b_value[4];
    d1_1 += src[1*src_len + *b_col_idx+1] * b_value[4];
    d0_0 += src[0*src_len + *b_col_idx+2] * b_value[2];
    d0_1 += src[1*src_len + *b_col_idx+2] * b_value[2];
    d1_0 += src[0*src_len + *b_col_idx+2] * b_value[5];
    d1_1 += src[1*src_len + *b_col_idx+2] * b_value[5];

    dest_p[0*dest_len + *b_col_idx] = t0_0;
    dest_p[1*dest_len + *b_col_idx] = t0_1;
    dest_p[0*dest_len + *b_col_idx+1] = t1_0;
    dest_p[1*dest_len + *b_col_idx+1] = t1_1;
    dest_p[0*dest_len + *b_col_idx+2] = t2_0;
    dest_p[1*dest_len + *b_col_idx+2] = t2_1;
  }
  dest[0*dest_len+0] = d0_0;
  dest[0*dest_len+1] = d1_0;
  dest[1*dest_len+0] = d0_1;
  dest[1*dest_len+1] = d1_1;
} /* End loop over rows */

/* Handle any remaining, unblocked rows of the matrix */
rimax = m - 2*brimax;
idx = 0;
for(ri=0; ri<rimax; ri++)
{
    dest_p[0*dest_len+(*b_col_idx+ri)] += b_value[idx] * src[0*src_len+(*b_col_idx+ri)];
    dest_p[1*dest_len+(*b_col_idx+ri)] += b_value[idx] * src[1*src_len+(*b_col_idx+ri)];
    idx++;
    for(ci=ri+1; ci<rimax; ci++, idx++)
    {
      dest_p[0*dest_len+(*b_col_idx+ci)] += b_value[idx] * src[0*src_len+(*b_col_idx+ri)];
      dest_p[0*dest_len+(*b_col_idx+ri)] += src[0*src_len+(*b_col_idx+ci)] * b_value[idx];
      dest_p[1*dest_len+(*b_col_idx+ci)] += b_value[idx] * src[1*src_len+(*b_col_idx+ri)];
      dest_p[1*dest_len+(*b_col_idx+ri)] += src[1*src_len+(*b_col_idx+ci)] * b_value[idx];
    }
  }
}
```

D Performance Profile Example

The experimental work of this report collected performance data exhaustively for block sizes up to 8 × 8 and vector widths up to 10. Figures 21–22 show the performance profiles of the optimal tuning parameters for the dense matrix (matrix 1) on the Intel Itanium 2. These profiles are representative of the profiles obtained for all 12 matrices in our test suite on all platforms.

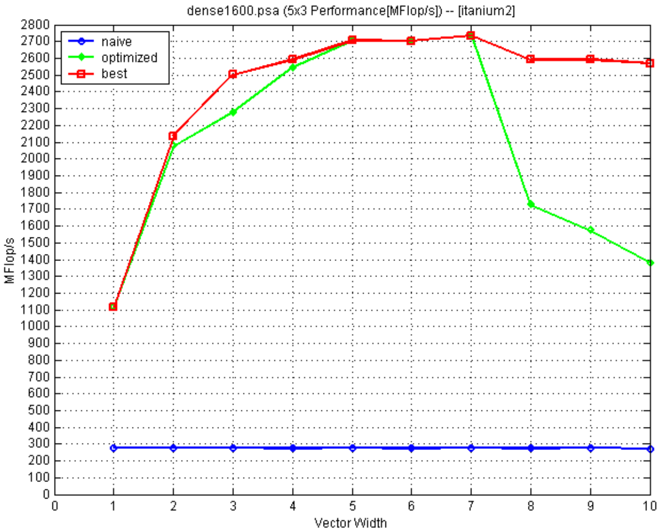


Figure 21: **Performance – Varying Vector Width, Intel Itanium 2**. Performance (MFlop/s) presented as a function of the vector width. The naïve implementation refers to the non-symmetric reference. The optimized implementation refers to the register blocked performance varying with vector width, given a 5×3 register block size chosen because 5×3 is fastest when $v = 1$. The best implementation refers to the register blocked performance varying with vector width, given the best register block size for each particular width. Note that performance gains from vector blocking peaks at $v = 7$ and falls drastically for high vector widths.

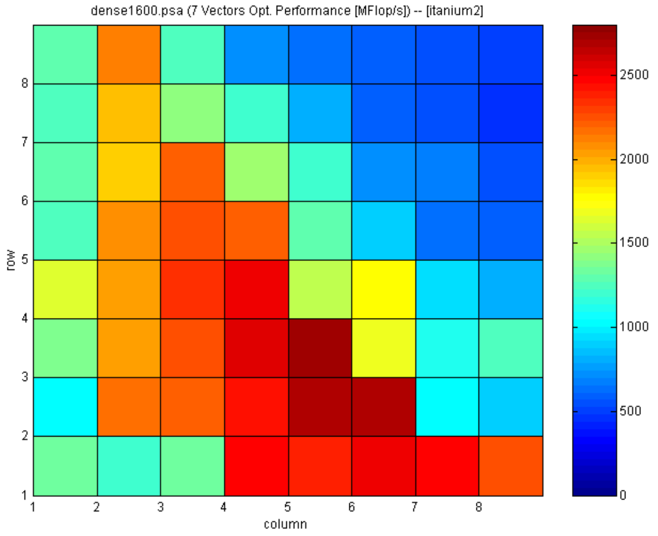


Figure 22: **Performance – Varying Block Size, Intel Itanium 2**. Performance (Mflop/s) presented as a function of the register block size, given a vector width of 7. Note that the performance gains for register blocking peaks at 5 × 3 and falls drastically with larger register block sizes.

E Application of the Transpose

This report considered the sequential and simultaneous application of the transpose. Initial studies indicate a performance advantage for the sequential application over the simultaneous application. Figure 23 presents the performance (MFlop/s) of the two implementations for the Ultra2i.

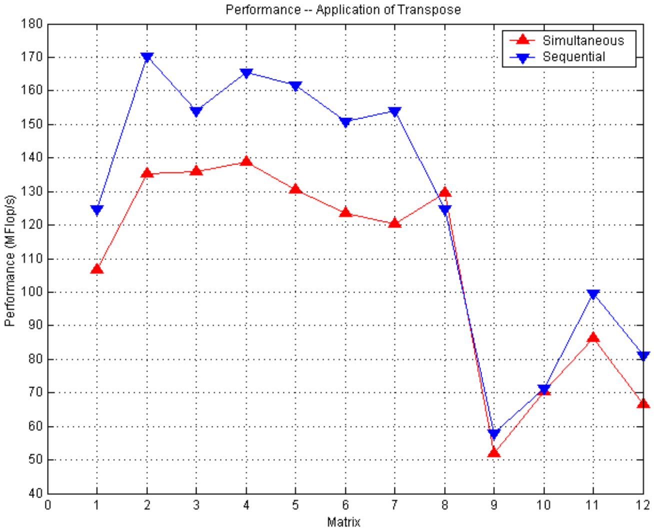


Figure 23: **Simultaneous vs. Sequential Application of the Transpose, Sun Ultra2i**. Performance (Mflop/s) is presented for preliminary studies in comparing the simultaneous and sequential application of the transpose (Section 3.4). The sequential application improved performance by as much as 1.3× over the simultaneous application.

F Block Alignment

This report considered uniform blocking and square diagonal blocking in the alignment of the diagonal blocks (Section 3.3). The performance results of initial studies comparing the two alignment schemes is presented in Figure 24. There is no obvious advantage to any particular alignment scheme.

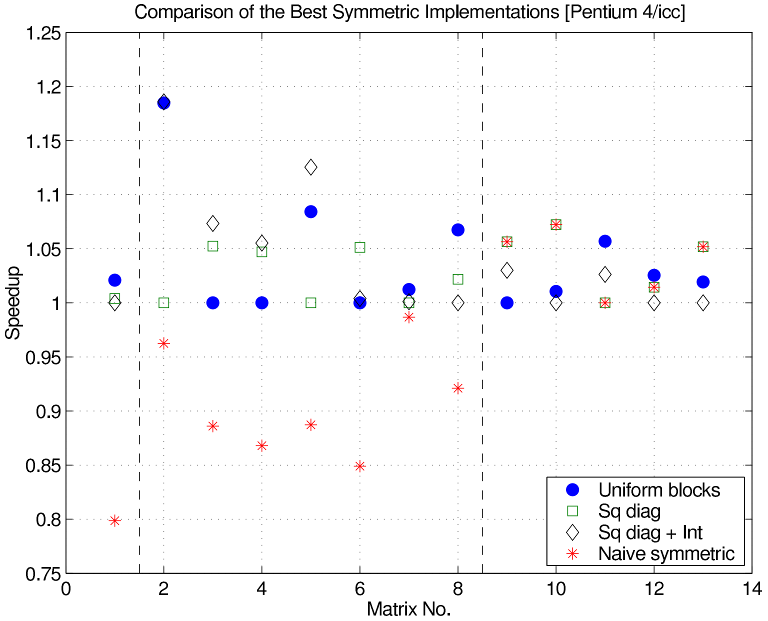


Figure 24: **Diagonal Block Alignment, Intel Pentium 4.** Performance (MFlop/s) presented for uniform blocking, square diagonal blocking, and square diagonal blocking with interleaved source and destination vectors. For each matrix, performance is normalized to the slowest of the three implementations. The performance of the naïve symmetric implementation is included for reference.

G Loop Unrolling

This report considered row-wise and column-wise loop unrolling of the register block elements. Initial studies indicate a performance advantage for the column-wise implementation, especially for Itanium 1 and Itanium 2. Figures 25–28 present the performance (MFlop/s) of the two implementations for the four platforms considered in this report.

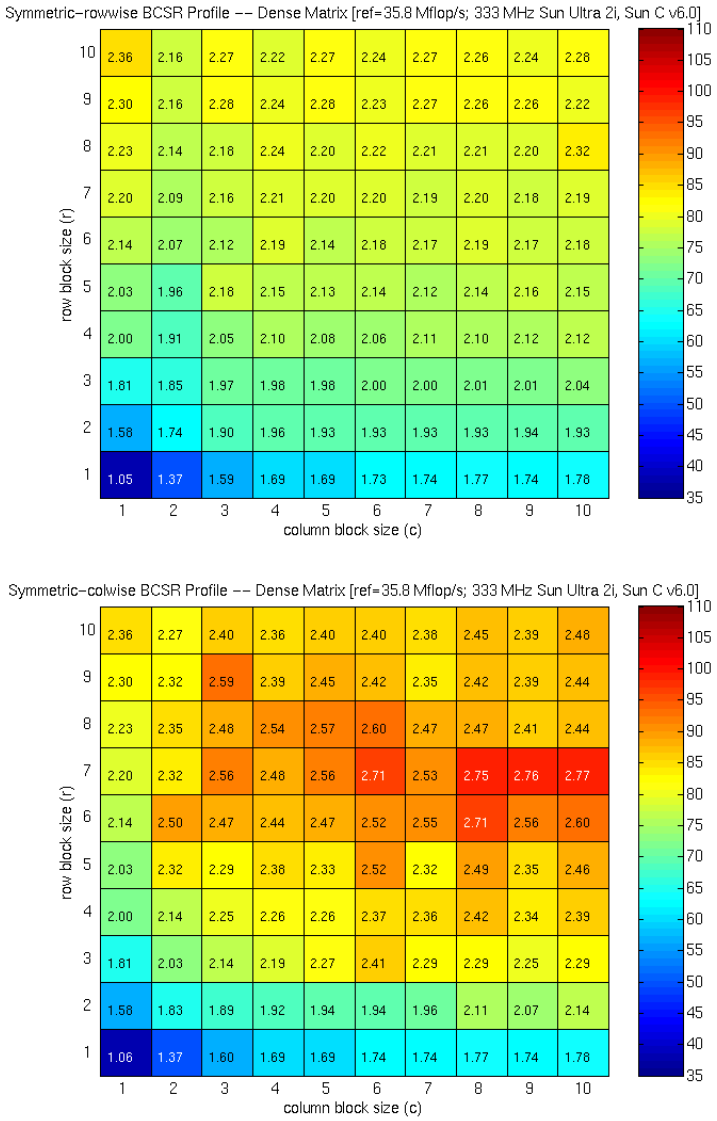


Figure 25: **Row-wise vs. Column-wise Unrolling, Sun Ultra2i.** Performance (MFlop/s) for the row-wise (*top*) and col-wise (*bottom*) implementation on the Sun Ultra2i. The numbers within each cell in the register profile is the speedup factor over the reference (35.8 MFlop/s).

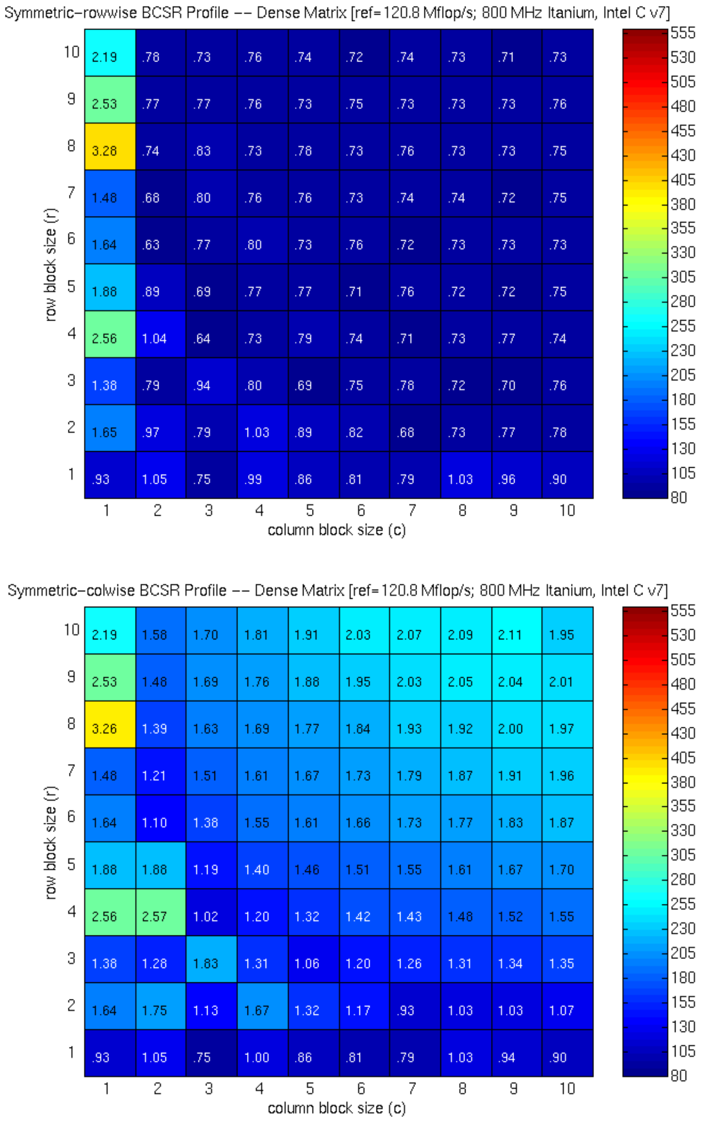
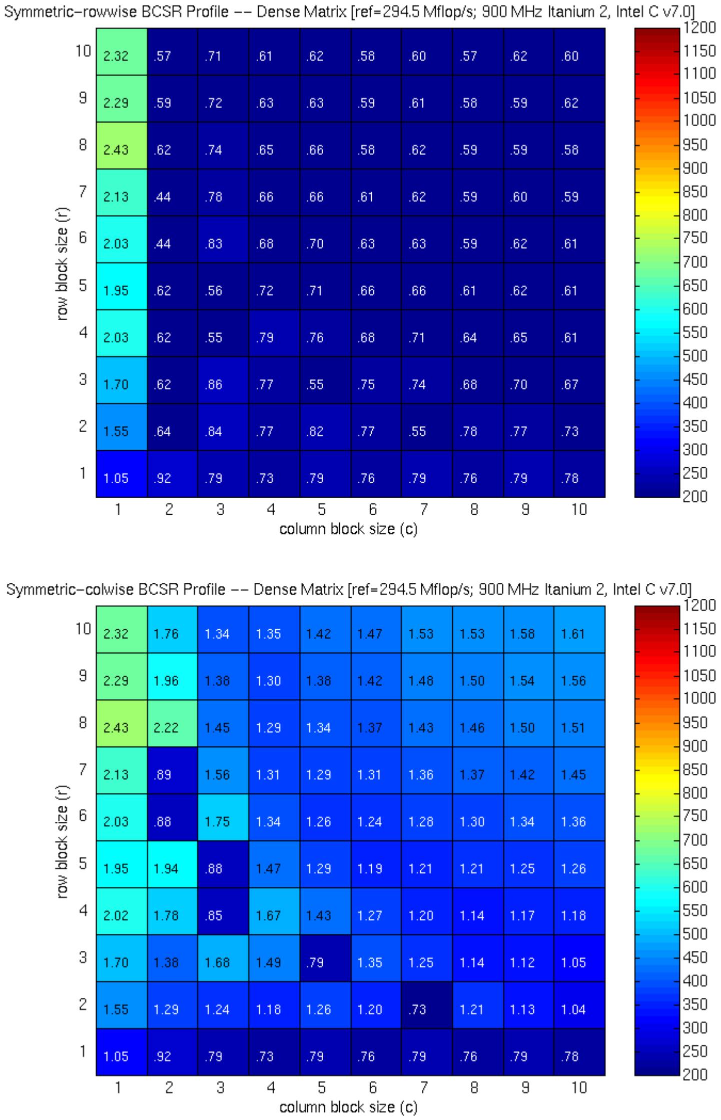


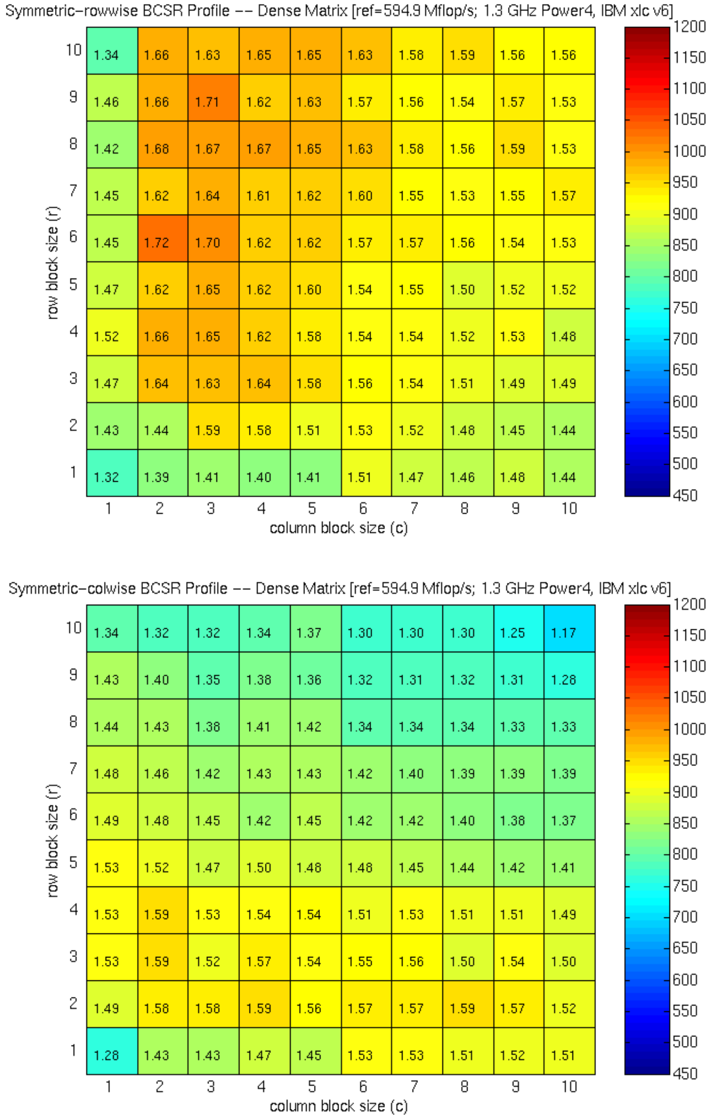
Figure 26: **Row-wise vs. Column-wise Unrolling, Intel Itanium 1.** Performance (MFlop/s) for the row-wise (*top*) and col-wise (*bottom*) implementation on the Intel Itanium 1. The numbers within each cell in the register profile is the speedup factor over the reference (120.8 MFlop/s).



	1	2	3	4	5	6	7	8	9	10
10	2.32	.57	.71	.61	.62	.58	.60	.57	.62	.60
9	2.29	.59	.72	.63	.63	.59	.61	.58	.59	.62
8	2.43	.62	.74	.65	.66	.58	.62	.59	.59	.58
7	2.13	.44	.78	.66	.66	.61	.62	.59	.60	.59
6	2.03	.44	.83	.68	.70	.63	.63	.59	.62	.61
5	1.95	.62	.56	.72	.71	.66	.66	.61	.62	.61
4	2.03	.62	.55	.79	.76	.68	.71	.64	.65	.61
3	1.70	.62	.86	.77	.55	.75	.74	.68	.70	.67
2	1.55	.64	.84	.77	.82	.77	.55	.78	.77	.73
1	1.05	.92	.79	.73	.79	.76	.79	.76	.79	.78

	1	2	3	4	5	6	7	8	9	10
10	2.32	1.76	1.34	1.35	1.42	1.47	1.53	1.53	1.58	1.61
9	2.29	1.96	1.38	1.30	1.38	1.42	1.48	1.50	1.54	1.56
8	2.43	2.22	1.45	1.29	1.34	1.37	1.43	1.46	1.50	1.51
7	2.13	.89	1.56	1.31	1.29	1.31	1.36	1.37	1.42	1.45
6	2.03	.88	1.75	1.34	1.26	1.24	1.28	1.30	1.34	1.36
5	1.95	1.94	.88	1.47	1.29	1.19	1.21	1.21	1.25	1.26
4	2.02	1.78	.85	1.67	1.43	1.27	1.20	1.14	1.17	1.18
3	1.70	1.38	1.68	1.49	.79	1.35	1.25	1.14	1.12	1.05
2	1.55	1.29	1.24	1.18	1.26	1.20	.73	1.21	1.13	1.04
1	1.05	.92	.79	.73	.79	.76	.79	.76	.79	.78

Figure 27: **Row-wise vs. Column-wise Unrolling, Intel Itanium 2.** Performance (MFlop/s) for the row-wise (*top*) and col-wise (*bottom*) implementation on the Intel Itanium 2. The numbers within each cell in the register profile is the speedup factor over the reference (294.5 MFlop/s).



r \ c	1	2	3	4	5	6	7	8	9	10
10	1.34	1.66	1.63	1.65	1.65	1.63	1.58	1.59	1.56	1.56
9	1.46	1.66	1.71	1.62	1.63	1.57	1.56	1.54	1.57	1.53
8	1.42	1.68	1.67	1.67	1.65	1.63	1.58	1.56	1.59	1.53
7	1.45	1.62	1.64	1.61	1.62	1.60	1.55	1.53	1.55	1.57
6	1.45	1.72	1.70	1.62	1.62	1.57	1.57	1.56	1.54	1.53
5	1.47	1.62	1.65	1.62	1.60	1.54	1.55	1.50	1.52	1.52
4	1.52	1.66	1.65	1.62	1.58	1.54	1.54	1.52	1.53	1.48
3	1.47	1.64	1.63	1.64	1.58	1.56	1.54	1.51	1.49	1.49
2	1.43	1.44	1.59	1.58	1.51	1.53	1.52	1.48	1.45	1.44
1	1.32	1.39	1.41	1.40	1.41	1.51	1.47	1.46	1.48	1.44

r \ c	1	2	3	4	5	6	7	8	9	10
10	1.34	1.32	1.32	1.34	1.37	1.30	1.30	1.30	1.25	1.17
9	1.43	1.40	1.35	1.38	1.36	1.32	1.31	1.32	1.31	1.28
8	1.44	1.43	1.38	1.41	1.42	1.34	1.34	1.34	1.33	1.33
7	1.48	1.46	1.42	1.43	1.43	1.42	1.40	1.39	1.39	1.39
6	1.49	1.48	1.45	1.42	1.45	1.42	1.42	1.40	1.38	1.37
5	1.53	1.52	1.47	1.50	1.48	1.48	1.45	1.44	1.42	1.41
4	1.53	1.59	1.53	1.54	1.54	1.51	1.53	1.51	1.51	1.49
3	1.53	1.59	1.52	1.57	1.54	1.55	1.56	1.50	1.54	1.50
2	1.49	1.58	1.58	1.59	1.56	1.57	1.57	1.59	1.57	1.52
1	1.28	1.43	1.43	1.47	1.45	1.53	1.53	1.51	1.52	1.51

Figure 28: **Row-wise vs. Column-wise Unrolling, IBM Power4.** Performance (MFlop/s) for the row-wise (*top*) and col-wise (*bottom*) implementation on the IBM Power4. The numbers within each cell in the register profile is the speedup factor over the reference (594.9 MFlop/s).

H Load Data

In Tables Tables 6–9, we present for each platform and matrix, the number of load instructions (integer and floating point loads) issued as reported by PAPI hardware counters and as estimated by our load model. The exception is the Power 4 platform for which HPM hardware counters were used to count only floating point loads.

	Loads Issued	
Matrix	PAPI Loads (millions)	Model Loads (millions)
1	7.083	7.048
2	4.772	4.269
3	3.273	3.041
4	5.903	5.491
5	9.027	8.118
6	10.723	9.969
7	9.337	8.462
8	32.426	29.854
9	4.241	3.433
10	2.252	1.850
11	1.955	1.788
12	11.049	10.459

Table 6: **Load Instructions Issued, Sun Ultra 2i**.

Matrix	Loads Issued	
	PAPI Loads (millions)	Model Loads (millions)
1	5.079	4.962
2	3.902	3.310
3	2.782	2.573
4	7.335	6.473
5	4.827	4.531
6	9.374	8.696
7	11.872	10.865
8	10.849	9.355
9	7.130	6.792
10	3.482	3.300
11	3.387	3.209
12	10.832	10.011

Table 7: **Load Instructions Issued, Intel Itanium 1**.

Matrix	Loads Issued	
	PAPI Loads (millions)	Model Loads (millions)
1	8.971	6.574
2	4.908	3.870
3	4.954	3.584
4	8.952	6.473
5	15.337	11.146
6	16.754	12.111
7	10.184	8.003
8	38.027	29.773
9	4.741	3.777
10	2.325	1.852
11	2.746	2.154
12	16.088	12.363

Table 8: **Load Instructions Issued, Intel Itanium 2**.

Matrix	Loads Issued PAPI Loads (millions)	 Model Loads (millions)
1	5.127	5.447
2	4.849	5.048
3	1.196	1.226
4	6.184	6.492
5	9.226	9.599
6	11.236	11.786
7	9.575	10.005
8	33.685	35.296
9	3.196	3.190
10	1.259	1.276
11	1.308	1.433
12	7.678	8.554

Table 9: **Load Instructions Issued, IBM Power 4**.

I Cache Miss Data

In Tables 10–12, we present for each platform and matrix, the number of cache misses as reported by PAPI hardware counters and as estimated by our lower bound on the number of cache misses.

	L1 Cache Misses		L2 Cache Misses	
Matrix	PAPI Misses (millions)	Lower Bound Misses (millions)	PAPI Misses (millions)	Lower Bound Misses (millions)
1	3.342	0.807	0.211	0.202
2	0.955	0.600	0.157	0.151
3	0.811	0.399	0.102	0.100
4	1.420	0.718	0.188	0.180
5	2.103	1.131	0.293	0.284
6	3.970	1.305	0.357	0.327
7	2.044	1.161	0.315	0.291
8	8.566	3.978	1.050	0.997
9	1.472	0.570	0.178	0.142
10	0.416	0.291	0.079	0.073
11	0.722	0.186	1.058	0.046
12	6.377	1.310	1.128	0.328

Table 10: **Cache Misses, Sun Ultra 2i.**

	L2 Cache Misses		L3 Cache Misses	
Matrix	PAPI Misses (millions)	Lower Bound Misses (millions)	PAPI Misses (millions)	Lower Bound Misses (millions)
1	0.715	0.173	0.174	0.173
2	0.183	0.169	0.170	0.169
3	0.147	0.097	0.097	0.097
4	0.257	0.173	0.175	0.173
5	0.447	0.325	0.326	0.325
6	0.815	0.322	0.334	0.322
7	0.443	0.337	0.359	0.337
8	1.463	1.196	1.211	1.196
9	0.258	0.182	0.212	0.182
10	0.100	0.089	0.092	0.089
11	0.247	0.088	0.096	0.088
12	3.032	0.336	0.422	0.336

Table 11: **Cache Misses, Intel Itanium 1.**

	L2 Cache Misses		L3 Cache Misses	
Matrix	PAPI Misses (millions)	Lower Bound Misses (millions)	PAPI Misses (millions)	Lower Bound Misses (millions)
1	0.095	0.084	0.086	0.084
2	0.094	0.085	0.087	0.085
3	0.046	0.043	0.043	0.043
4	0.106	0.087	0.088	0.087
5	0.142	0.138	0.137	0.138
6	0.269	0.144	0.147	0.144
7	0.261	0.188	0.215	0.188
8	0.641	0.358	0.359	0.358
9	0.153	0.118	0.153	0.118
10	0.063	0.055	0.059	0.055
11	0.117	0.029	0.034	0.029
12	1.257	0.197	0.248	0.197

Table 12: **Cache Misses, Intel Itanium 2.**

J Performance for Optimum Register & Vector Block Sizes

In Tables 13–16, we present for each platform and matrix, the performance (Mflop/s) of the following register block sizes:

- **Non-Symm Reg**: The values of $r \times c$ yielding the best performance in the non-symmetric, register blocked, single vector implementation.
- **Symm Reg**: The values of $r \times c$ yielding the best performance in the symmetric, register blocked, single vector implementation.
- **Non-Symm Reg mVec**: The values of $r \times c$ and v yielding the best performance in the non-symmetric, register blocked, vector blocked implementation.
- **Symm Reg mVec**: The value of the $r \times c$ and v yielding the best performance in the symmetric, register blocked, vector blocked implementation.

	Non-Symm Reg				Symm Reg			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	8×7	1	71.0	1.00	4×3	1	84.8	1.00
2	6×6	1	49.7	1.52	3×3	1	75.4	1.06
3	3×3	1	55.2	1.00	3×3	1	82.4	1.00
4	3×3	1	54.9	1.00	3×3	1	81.5	1.00
5	6×6	1	51.8	1.15	3×3	1	73.0	1.11
6	3×3	1	53.3	1.02	3×3	1	77.3	1.02
7	2×2	1	38.7	1.25	4×3	1	56.9	1.63
8	3×3	1	48.0	1.49	3×1	1	67.5	1.00
9	1×1	1	21.5	1.00	1×1	1	29.6	1.00
10	1×1	1	22.0	1.00	1×1	1	37.6	1.00
11	1×1	1	30.0	1.00	1×1	1	50.7	1.00
12	1×1	1	32.5	1.00	1×3	1	35.3	1.66

	Non-Symm Reg mVec				Symm Reg mVec			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	8×3	10	202.0	1.00	2×1	4	124.8	1.00
2	6×6	10	148.9	1.52	2×1	4	170.4	1.03
3	3×3	10	167.8	1.00	2×1	4	154.1	1.11
4	3×3	10	157.9	1.00	2×1	4	165.5	1.11
5	6×6	10	153.4	1.15	2×1	4	161.8	1.06
6	3×3	5	142.3	1.02	2×1	4	150.9	1.10
7	2×2	10	149.1	1.25	2×1	4	154.0	1.10
8	3×3	10	146.0	1.49	2×1	4	149.5	1.16
9	1×1	10	59.6	1.00	2×1	4	58.0	1.64
10	1×1	10	79.8	1.00	1×1	4	71.2	1.00
11	1×1	6	89.4	1.00	1×1	4	99.7	1.00
12	1×1	8	106.2	1.00	2×1	4	81.1	1.72

Table 13: **Block Sizes, Sun Ultra 2i.**

	Non-Symm Reg				Symm Reg			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	4 × 1	1	249.0	1.00	2 × 4	1	486.6	1.00
2	3 × 1	1	197.5	1.04	3 × 3	1	241.3	1.06
3	3 × 1	1	217.5	1.00	3 × 3	1	270.9	1.00
4	3 × 1	1	217.4	1.00	3 × 3	1	269.8	1.00
5	3 × 1	1	197.5	1.05	2 × 2	1	239.9	1.10
6	3 × 1	1	212.3	1.01	3 × 3	1	257.6	1.02
7	2 × 1	1	162.8	1.10	2 × 4	1	187.5	1.48
8	2 × 1	1	214.0	1.16	2 × 4	1	207.5	1.68
9	1 × 1	1	65.2	1.00	2 × 1	1	86.9	1.64
10	1 × 1	1	73.0	1.00	2 × 1	1	98.9	1.49
11	1 × 1	1	123.8	1.00	2 × 2	1	105.8	2.49
12	1 × 1	1	115.7	1.00	2 × 4	1	136.2	3.10

	Non-Symm Reg mVec				Symm Reg mVec			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	2 × 2	10	1285.0	1.00	4 × 2	8	1116.9	1.00
2	3 × 1	8	752.4	1.04	4 × 1	6	732.2	1.19
3	3 × 1	9	902.4	1.00	3 × 3	9	753.4	1.00
4	3 × 1	9	836.8	1.00	3 × 3	9	804.8	1.00
5	3 × 1	8	666.0	1.05	3 × 3	9	664.3	1.11
6	3 × 1	8	734.0	1.01	3 × 3	9	677.0	1.02
7	3 × 1	9	632.8	1.27	4 × 1	6	617.4	1.33
8	2 × 2	10	637.0	1.34	4 × 1	6	622.8	1.48
9	2 × 1	9	147.0	1.64	2 × 1	5	225.2	1.64
10	1 × 1	7	168.1	1.00	2 × 1	5	241.6	1.49
11	1 × 1	7	287.0	1.00	4 × 1	6	281.3	2.78
12	2 × 1	8	358.0	1.72	2 × 1	5	209.8	1.72

Table 14: **Block Sizes, Intel Itanium 1**.

	Non-Symm Reg				Symm Reg			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	4 × 2	1	1200.0	1.00	5 × 3	1	1114.3	1.00
2	4 × 2	1	611.6	1.34	3 × 3	1	612.0	1.06
3	3 × 2	1	629.2	1.11	3 × 3	1	695.3	1.00
4	3 × 2	1	631.8	1.11	3 × 3	1	682.3	1.00
5	6 × 1	1	593.1	1.12	6 × 2	1	668.4	1.13
6	3 × 2	1	623.0	1.13	3 × 3	1	671.9	1.02
7	4 × 2	1	546.2	1.49	4 × 2	1	519.2	1.49
8	3 × 2	1	277.0	1.16	3 × 2	1	557.4	1.16
9	2 × 1	1	133.4	1.64	2 × 2	1	194.3	2.29
10	4 × 1	1	172.0	2.37	2 × 2	1	198.4	1.93
11	1 × 1	1	239.4	1.00	1 × 1	1	260.2	1.00
12	1 × 1	1	259.2	1.00	2 × 4	1	296.7	3.10

	Non-Symm Reg mVec				Symm Reg mVec			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	4 × 2	10	3041.0	1.00	5 × 3	7	2735.9	1.00
2	4 × 2	10	1022.1	1.34	6 × 1	8	1714.8	1.10
3	3 × 2	10	1198.5	1.11	3 × 3	6	1771.6	1.00
4	3 × 2	10	1112.2	1.11	3 × 3	9	1846.0	1.00
5	6 × 1	10	948.3	1.12	6 × 1	5	1538.3	1.12
6	3 × 2	10	1114.5	1.13	3 × 3	6	1761.9	1.02
7	4 × 2	10	949.6	1.49	4 × 1	9	1396.2	1.33
8	3 × 2	5	1346.0	1.16	3 × 1	10	1755.0	1.00
9	2 × 1	10	210.6	1.64	1 × 1	9	542.9	1.00
10	4 × 1	6	242.3	2.37	1 × 1	8	558.0	1.00
11	6 × 1	10	361.0	3.68	1 × 1	8	869.6	1.00
12	1 × 1	9	599.8	1.00	3 × 1	4	530.3	2.26

Table 15: **Block Sizes, Intel Itanium 2.**

	Non-Symm Reg				Symm Reg			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	7 × 1	1	835.0	1.00	4 × 4	1	1368.0	1.00
2	3 × 3	1	628.0	1.06	3 × 3	1	978.3	1.06
3	3 × 3	1	706.3	1.00	3 × 3	1	1270.2	1.00
4	3 × 3	1	681.4	1.00	3 × 3	1	1197.1	1.00
5	3 × 1	1	552.6	1.05	6 × 3	1	1021.8	1.13
6	3 × 3	1	581.2	1.02	3 × 3	1	1022.2	1.02
7	2 × 1	1	487.9	1.10	2 × 1	1	796.4	1.10
8	3 × 3	1	708.4	1.49	3 × 1	1	794.8	1.00
9	1 × 1	1	197.3	1.00	2 × 1	1	389.3	1.64
10	1 × 1	1	241.1	1.00	2 × 1	1	501.7	1.49
11	1 × 1	1	366.8	1.00	1 × 1	1	552.6	1.00
12	1 × 1	1	428.8	1.00	1 × 1	1	510.9	1.00

	Non-Symm Reg mVec				Symm Reg mVec			
Matrix	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio	Block Size	Vector Width	Perf. (MFlop/s)	Fill Ratio
1	7 × 1	4	1828.0	1.00	2 × 2	3	1973.2	1.00
2	3 × 3	3	1424.6	1.06	2 × 1	5	1784.9	1.03
3	3 × 3	4	1521.9	1.00	3 × 3	2	1769.3	1.00
4	3 × 3	4	1374.3	1.00	2 × 1	5	1628.0	1.11
5	3 × 1	6	1303.2	1.05	2 × 1	5	1691.3	1.06
6	3 × 3	4	1467.2	1.02	2 × 1	5	1603.0	1.10
7	2 × 1	9	1465.5	1.10	2 × 1	5	1600.1	1.10
8	3 × 3	4	1634.7	1.49	2 × 1	5	1517.7	1.16
9	1 × 1	2	340.7	1.00	2 × 1	4	605.2	1.64
10	1 × 1	8	587.5	1.00	2 × 1	3	825.6	1.49
11	1 × 1	5	732.2	1.00	1 × 1	3	826.6	1.00
12	1 × 1	7	768.0	1.00	2 × 1	3	713.0	1.72

Table 16: **Block Sizes, IBM Power4**.

K Memory Usage

In Tables 17–20, we present for each platform and matrix, an approximation of the storage requirements of a matrix with and without symmetric storage.

The approximation for naïve storage requirements accounts for double precision floating point matrix elements (8 bytes), row indices (4 bytes), column indices (4 bytes). Let k be half the total number of non-zero elements in the matrix and m be the matrix dimensions for a square symmetric matrix. Naïve storage of the matrix would store all $2k$ matrix elements for $8 \times 2k = 16k$ bytes. Each element would also store a corresponding column index for $4 \times 2k = 8k$ bytes. Lastly, the matrix dimension is a conservative estimate for the number of row indices for $4m$ bytes. Thus, the total naïve storage requirements is $24k + 4m$ bytes.

The approximation for symmetric storage accounts for double precisions floating point matrix elements and indices, but also depends on the optimal register block size. Let r and c be the row and column block size for a particular platform, matrix, and optimization. Symmetric storage of the matrix wold store only half the elements in the matrix (k) scaled for any fill from register blocking (kf_{rc}) for $8kf_{rc}$ bytes. The number of register blocks is given by $\frac{kf_{rc}}{rc}$. Each register block would also store a corresponding column index for $4\frac{kf_{rc}}{rc}$ bytes. Lastly, the number of blocked rows is approximated by $\lceil \frac{m}{r} \rceil$ for $4\lceil \frac{m}{r} \rceil$ bytes. Thus, the total symmetric storage requirements is $8kf_{rc} + 4\frac{kf_{rc}}{rc} + 4\lceil \frac{m}{r} \rceil$ bytes.

I. Symmetry Register Blocking						
	1	2	3	4	5	6
k	1280800	1450163	968583	1751178	2677324	3213332
m	1600	30237	13965	24696	54870	45330
Naïve Storage (MB)	29.32	33.31	22.22	40.18	61.49	73.72
Ultra 2i						
Row Size	4	3	3	3	3	3
Column Size	3	3	3	3	3	3
Fill Ratio	1.00	1.06	1.00	1.00	1.11	1.02
Symm Storage (MB)	10.18	12.42	7.82	14.13	24.00	26.45
Naïve-Symm Ratio	2.88	2.68	2.84	2.84	2.56	2.79
Itanium 1						
Row Size	2	3	3	3	2	3
Column Size	4	3	3	3	2	3
Fill Ratio	1.00	1.06	1.00	1.00	1.10	1.02
Symm Storage (MB)	10.39	12.42	7.82	14.13	25.38	26.45
Naïve-Symm Ratio	2.82	2.68	2.84	2.84	2.42	2.79
Itanium 2						
Row Size	5	3	3	3	6	3
Column Size	3	3	3	3	2	3
Fill Ratio	1.00	1.06	1.00	1.00	1.13	1.02
Symm Storage (MB)	10.10	12.42	7.82	14.13	24.08	26.45
Naïve-Symm Ratio	2.90	2.68	2.84	2.84	2.55	2.79
Power 4						
Row Size	4	3	3	3	6	3
Column Size	4	3	3	3	3	3
Fill Ratio	1.00	1.06	1.00	1.00	1.13	1.02
Symm Storage (MB)	10.08	12.42	7.82	14.13	23.76	26.45
Naïve-Symm Ratio	2.91	2.68	2.84	2.84	2.59	2.79

Table 17: **Memory Usage.** The table shows the effects of symmetry for storage requirements. The values shown reflect reductions in matrix storage requirements as a ratio of storage for symmetric and non-symmetric register blocking. Storage ratios for matrices 1–6 are presented.

II. Symmetry Register Blocking						
	7	8	9	10	11	12
k	2698463	4617075	596992	326107	342828	2164210
m	52329	153746	74752	36519	12328	31802
Naïve Storage (MB)	61.96	106.26	13.95	7.60	7.89	49.66
Ultra 2i						
Row Size	4	3	1	1	1	1
Column Size	3	1	1	1	1	3
Fill Ratio	1.63	1.00	1.00	1.00	1.00	1.66
Symm Storage (MB)	35.01	41.29	7.12	3.87	3.97	32.10
Naïve-Symm Ratio	1.77	2.57	1.96	1.96	1.99	1.55
Itanium 1						
Row Size	2	2	2	2	2	2
Column Size	4	4	1	1	2	4
Fill Ratio	1.48	1.68	1.64	1.49	2.49	3.10
Symm Storage (MB)	32.47	63.17	9.48	4.70	7.35	54.45
Naïve-Symm Ratio	1.91	1.68	1.47	1.62	1.07	0.91
Itanium 2						
Row Size	4	3	2	2	1	2
Column Size	2	2	2	2	1	4
Fill Ratio	1.49	1.16	2.29	1.93	1.00	3.10
Symm Storage (MB)	32.64	44.46	11.88	5.47	3.97	54.45
Naïve-Symm Ratio	1.90	2.39	1.17	1.39	1.99	0.91
Power 4						
Row Size	2	3	2	2	1	1
Column Size	1	1	1	1	1	1
Fill Ratio	1.10	1.00	1.64	1.49	1.00	1.00
Symm Storage (MB)	28.41	41.29	9.48	4.70	3.97	24.89
Naïve-Symm Ratio	2.18	2.57	1.47	1.62	1.99	2.00

Table 18: **Memory Usage.** The table shows the effects of symmetry for storage requirements. The values shown reflect reductions in matrix storage requirements as a ratio of storage for symmetric and non-symmetric register blocking. Storage ratios for matrices 7–12 are presented.

III. Symmetry Register & Vector Blocking						
	1	2	3	4	5	6
k	1280800	1450163	968583	1751178	2677324	3213332
m	1600	30237	13965	24696	54870	45330
Naïve Storage (MB)	29.32	33.31	22.22	40.18	61.49	73.72
Ultra 2i						
Row Size	2	2	2	2	2	2
Column Size	1	1	1	1	1	1
Fill Ratio	1.00	1.03	1.11	1.11	1.06	1.10
Symm Storage (MB)	12.22	14.30	10.28	18.58	27.17	33.80
Naïve-Symm Ratio	2.40	2.33	2.16	2.16	2.26	2.18
Itanium 1						
Row Size	4	4	3	3	3	3
Column Size	2	1	3	3	3	3
Fill Ratio	1.00	1.19	1.00	1.00	1.11	1.02
Symm Storage (MB)	10.38	14.84	7.82	14.13	24.00	26.45
Naïve-Symm Ratio	2.82	2.24	2.84	2.84	2.56	2.79
Itanium 2						
Row Size	5	6	3	3	6	3
Column Size	3	1	3	3	1	3
Fill Ratio	1.00	1.10	1.00	1.00	1.12	1.02
Symm Storage (MB)	10.10	13.20	7.82	14.13	24.82	26.45
Naïve-Symm Ratio	2.90	2.52	2.84	2.84	2.48	2.79
Power 4						
Row Size	2	2	3	2	2	2
Column Size	2	1	3	1	1	1
Fill Ratio	1.00	1.03	1.00	1.11	1.06	1.10
Symm Storage (MB)	11.00	14.30	7.82	18.58	27.17	33.80
Naïve-Symm Ratio	2.67	2.33	2.84	2.16	2.26	2.18

Table 19: **Memory Usage**. The table shows the effects of symmetry for storage requirements. The values shown reflect reductions in matrix storage requirements as a ratio of storage for symmetric and non-symmetric register and vector blocking. Storage ratios for matrices 1–6 are presented.

IV. Symmetry Register & Vector Blocking

	7	8	9	10	11	12
k	2698463	4617075	596992	326107	342828	2164210
m	52329	153746	74752	36519	12328	31802
Naïve Storage (MB)	61.96	106.26	13.95	7.60	7.89	49.66
Ultra 2i						
Row Size	2	2	2	1	1	2
Column Size	1	1	1	1	1	1
Fill Ratio	1.10	1.16	1.64	1.00	1.00	1.72
Symm Storage (MB)	28.41	51.37	9.48	3.87	3.97	35.56
Naïve-Symm Ratio	2.18	2.07	1.47	1.96	1.99	1.40
Itanium 1						
Row Size	4	4	2	2	4	2
Column Size	1	1	1	1	1	1
Fill Ratio	1.33	1.48	1.64	1.49	2.78	1.72
Symm Storage (MB)	30.85	58.80	9.48	4.70	8.19	35.56
Naïve-Symm Ratio	2.01	1.81	1.47	1.62	0.96	1.40
Itanium 2						
Row Size	4	3	1	1	1	3
Column Size	1	1	1	1	1	1
Fill Ratio	1.33	1.00	1.00	1.00	1.00	2.26
Symm Storage (MB)	30.85	41.29	7.12	3.87	3.97	43.58
Naïve-Symm Ratio	2.01	2.57	1.96	1.96	1.99	1.14
Power 4						
Row Size	2	2	2	2	1	2
Column Size	1	1	1	1	1	1
Fill Ratio	1.10	1.16	1.64	1.49	1.00	1.72
Symm Storage (MB)	28.41	51.37	9.48	4.70	3.97	35.56
Naïve-Symm Ratio	2.18	2.07	1.47	1.62	1.99	1.40

Table 20: **Memory Usage**. The table shows the effects of symmetry for storage requirements. The values shown reflect reductions in matrix storage requirements as a ratio of storage for symmetric and non-symmetric register and vector blocking. Storage ratios for matrices 7–12 are presented.