Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-Vector Multiply

Benjamin C. Lee, Richard W. Vuduc, James W. Demmel, Katherine A. Yelick
University of California, Berkeley
Computer Science Division
Berkeley, California, USA
{blee20, richie, demmel, yelick}@cs.berkeley.edu

Abstract

*We present **optimizations** for sparse matrix-vector multiply SpMV and its generalization to multiple vectors, SpMM, when the matrix is symmetric: (1) symmetric storage, (2) register blocking, and (3) vector blocking. Combined with register blocking, symmetry saves more than 50% in matrix storage. We also show performance speedups of $2.1\times$ for SpMV and $2.6\times$ for SpMM, when compared to the best non-symmetric register blocked implementation.*

*We present an approach for the selection of tuning parameters, based on **empirical modeling and search** that consists of three steps: (1) Off-line benchmark, (2) Run-time search, and (3) Heuristic performance model. This approach generally selects parameters to achieve performance with 85% of that achieved with exhaustive search.*

*We evaluate our implementations with respect to **upper bounds** on performance. Our model bounds performance by considering only the cost of memory operations and using lower bounds on the number of cache misses. Our optimized codes are within 68% of the upper bounds.*

1 Introduction

We present optimizations, an approach for tuning parameter selection, and performance analyses for sparse matrix-vector multiply (SpMV), $y \leftarrow y + A \cdot x$, where A is a symmetric, sparse matrix (*i.e.*, $A = A^T$) and x,y are dense column vectors called the *source* and *destination*. We also consider the generalization of SpMV to multiple vectors where x,y are replaced by matrices X,Y, referring to this kernel as SpMM. Symmetry is traditionally exploited to reduce storage, but performance gains are also possible since the cost of memory accesses dominates the cost of flops on most modern cache-based superscalar architectures.

The challenge in efficient performance tuning for sparse computational kernels 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. This paper describes a new implementation space for the symmetric case, considering symmetric storage (Section 3), register blocking (Section 3), and vector blocking (Section 4).

We present an empirical modeling and search procedure to select optimal tuning parameters that characterize a SpMM code for a given matrix and machine (Section 5). Our approach extends models of the SPARSITY system [7, 5]. To evaluate the measured performance of our best implementations, we formulate upper bounds on performance (Section 6). We bound execution time from below by considering only the cost of memory accesses and by modeling data placement in the memory hierarchy for a lower bound on cache misses.

The following summarizes experimental results from four different computing platforms (Table 1) and a test suite of twelve sparse symmetric matrices (Table 2):

1. **Symmetric register blocking** achieves up to $2.1\times$ speedups for SpMV and $2.6\times$ speedups for SpMM over non-symmetric register and vector blocking.
2. Combining **symmetry, register and vector blocking** achieves up to $9.9\times$ speedups for a dense matrix in sparse format and up to $7.3\times$ for a true sparse matrix when compared to a naïve code (no optimizations).
3. The **empirical modeling and search procedure** generally selects parameters that yield within 85% of the best performance achieved by an exhaustive search over all possible parameter values. [1]
4. Measured performance achieve 68% of the **performance upper bound**, on average.

[1] "All possible values" subject to constraints that consider the characteristics of practical applications and architectural parameters.

Property	Sun Ultra 2i	Intel Itanium 1	Intel Itanium 2	IBM Power 4
Clock rate (MHz)	333	800	900	1300
Peak Main Memory Bandwidth (MB/s)	664	2.1	6400	8000
Peak Flop Rate (Mflop/s)	667	3200	3600	5200
DGEMM $n = 1000$ (Mflop/s)	425	2200	3500	3500
DGEMV $n = 2000$ (Mflop/s)	58	345	740	915
DSYMV $n = 1000$ (Mflop/s)	92	625	1400	1600
DSPMV $n = 2000$ (Mflop/s)	62	115	356	1700
DSYMM $n = 2000$ (Mflop/s)	383	1900	3400	3500
STREAM Triad Bandwidth (MB/s)	250	1100	3800	2100
L1 total size (KB)	16	16	32	32
L1 line size (B)	16	32	128	128
L1 latency (cy)	2	2 (int)	0.34	0.7
L2 total size (KB)	2048	96	256	1536
L2 line size (B)	64	64	128	128
L1 latency (cy)	7	6-9	0.5	12
L3 total size (KB)	n/a	2	1.5	16
L3 line size (B)	n/a	64	128	512
L3 latency (cy)	n/a	21-24	3	45
Memory latency (cy, $\approx$)	36	36	11	167
Compiler	Sun C v6.1	Intel C v5.0.1	Intel C v7.0	IBM XLC

Table 1. Evaluation platforms.

This paper summarizes the key findings of a recent technical report [3]. We refer the reader to the report for further details. The empirical modeling and search procedure for tuning parameters does not appear in the report.

2 Experimental Methodology

We conducted experiments on machines based on the microprocessors in Table 1. Latency estimates were obtained from a combination of published sources and experimental measurements using the Saavedra-Barrera memory system microbenchmark [10] and MAPS benchmarks [11].

Table 2 summarizes the size and application of each symmetric matrix in the matrix benchmark suite used for evaluation. Most of the matrices are available from either the collections at NIST (MatrixMarket [12]) or the University of Florida [13]. The size of these matrices exceed the largest cache size for the evaluation platforms.

We use the PAPI v2.1 library for access to hardware counters on all platforms [14] except Power 4; not all PAPI counters are available for the Power 4 and HPM counters overcount memory traffic. We use the cycle counters, reported as the median of 25 consecutive trials, as timers.

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

	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. 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.

3 Optimizations for Matrix Symmetry

The baseline implementation stores the full matrix in compressed sparse row (CSR) format and computes SpMV using a non-symmetric kernel.

3.1 Symmetric Storage

Matrix symmetry enables storing only half of the matrix and, without loss of generality, our implementation stores the upper-triangle. Although the symmetric code requires the same number of floating point operations as the baseline, symmetry halves the number of memory accesses to the matrix: a symmetric implementation simultaneously applies each element and its transpose, processing only the stored half of the matrix. In both cases, stores to the destination are indirect and potentially irregular.

3.2 Symmetric Register Blocking

SPARSITY's *register blocking* is a technique for improving register reuse [7]. The sparse $m \times n$ matrix is logically divided into aligned $r \times c$ blocks, storing only those blocks containing at least one non-zero. SpMV computation proceeds block-by-block. For each block, we can reuse the corresponding c elements of the source by keeping them in registers to increase temporal locality to the source, assuming a sufficient number are available.

Register blocking uses the blocked variant of compressed sparse row (BCSR) storage format. Blocks within

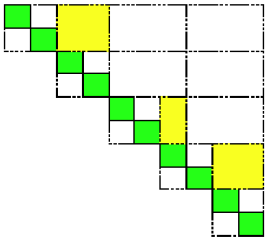

Figure 1. Square diagonal blocking. A 10×10 matrix with 2×3 register blocks.

the same block row are stored consecutively, and the elements of each block are stored consecutively in row-major order. When $r = c = 1$, BCSR reduces to CSR. 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.

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. The profitability of register blocking depends highly on the fill and the non-zero pattern of the matrix.

In our symmetric implementation of $r \times c$ register blocking, we use the following *square diagonal blocking* scheme (Figure 1). Given a row-oriented storage scheme and $r \times c$ register blocks, the diagonal blocks are implemented as square $r \times r$ blocks. We align the register blocks from the right edge of the matrix, which may require small degenerate $r \times c'$ blocks to the right of the diagonal blocks, where $c' < c$ and c' depends on the block row.

4 Optimizations for Multiple Vectors

The baseline implementation, 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, brings the entire matrix through the memory hierarchy once for each vector.

Vector blocking is a technique for reducing memory traffic for the 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. [2] The k vectors are processed in $\lceil \frac{k}{v} \rceil$ groups of the *vector width* v and multiplication of each element of A is unrolled by v. The computation of SpMM proceeds sequentially across matrix elements or register blocks, computing results for the corresponding elements in each of the v destinations in the vector block. When $v = 1$, the subroutine is effectively a single vector implementation of SpMV.

Thus, the matrix is accessed at most $\lceil \frac{k}{v} \rceil + 1$ times in contrast to the k times required by the baseline. The effect is to reduce memory traffic and increase temporal locality to A by amortizing the cost of accessing a matrix element for v vectors. Furthermore, unrolling the multiply for the v vectors reduces loop overhead and exposes scheduling opportunities to the compiler.

5 Automated Empirical Tuning

A register and vector blocked SpMM code is characterized by the parameters (r, c, v), indicating optimal register block size $r \times c$ and vector width v. The optimal parameters vary significantly across machines and matrices and are difficult to choose by purely analytic modeling [2].

Our approach to selecting (r, c, v) is based on *empirical modeling and search*, executed partly off-line and partly at run-time. Given a machine, a symmetric matrix A, and a number of vectors k, our tuning parameter selection procedure consists of the following 3 steps:

1. **Off-line benchmark**: Once per machine, measure the performance (in Mflop/s) of symmetric SpMM for a dense matrix stored in sparse format, for all (r, c, v) such that $1 \leq r, c \leq b_{\max}$ and $1 \leq v \leq v_{\max}$, where k is set equal to v. In practice, we use $b_{\max} = 8$ and $v_{\max} = 10$. Denote this *symmetric register profile* by $\{P_{rcv}\,(\text{dense}) \,|1 \leq r, c \leq b_{\max}, 1 \leq v \leq v_{\max}\}$.

2. **Run-time "search"**: When the matrix A is known at run-time, compute an estimate $\hat{f}_{rc}\,(A)$ of the true fill ratio for all $1 \leq r, c, \leq b_{\max}$. Estimating this quantity is a form of empirical search over possible block sizes, and depends only on the matrix non-zero structure.

3. **Heuristic performance model**: Choose (r, c, v) that maximizes the following *estimate* of register blocking performance $\hat{P}_{rcv}\,(A)$,

$$\hat{P}_{rcv}\,(A) = \frac{P_{rcv}\,(\text{dense})}{\hat{f}_{rc}\,(A)}, \tag{1}$$

   for all $1 \leq r, c \leq b_{\max}$ and $1 \leq v \leq \min\{v_{\max}, k\}$. Intuitively, $P_{rcv}\,(\text{dense})$ is an empirical estimate of expected performance, and $\hat{f}_{rc}\,(A)$ reduces this value according to the extra flops per non-zero due to fill.

The key idea is to decouple machine-specific aspects of performance which can be measured off-line (Step 1) from

[2] X, Y are collections of k dense column vectors of length n.

matrix-specific aspects determined at run-time (Step 2), combining these aspects with a heuristic model of performance evaluated at run-time (Step 3). This procedure adapts our prior technique for non-symmetric SpMV [1, 2, 4].

Trying all or even a subset of block sizes is infeasible if the matrix is known only at run-time, due to the cost of converting the matrix to blocked format. In contrast, the fill can be estimated accurately and cheaply [2], while the total run-time cost of executing Steps 2 and 3, followed by a single conversion to $r \times c$ blocked format, is not much greater than the cost of only the conversion [2].

6 Bounds on Performance

We present performance upper bounds to estimate the potential payoff from low-level tuning given a matrix and a data structure, but independent of instruction mix and ordering. These bounds extend prior bounds for non-symmetric SpMV [4]. The performance model consists of a lower bound model of execution time and a lower bound model on cache misses at each level in the memory hierarchy. The following are underlying assumptions for these bounds:

1. Our lower bound model of execution time considers only the cost of memory operations, taking SpMV and SpMM to be memory bound. Assuming write-back caches and sufficient store buffer capacity, we account only for the cost of loads and ignore the cost of stores.

2. For a hit in cache level i, we assign a cost α_i to access the data at that level, as determined by microbenchmarks on streaming workloads likely to represent the fastest memory access patterns.

3. We further bound time from below by obtaining a lower bound on cache misses that considers only compulsory misses, accounts for cache line sizes, and assumes full associativity.

4. We further bound time from below by ignoring TLB misses. This assumption is justified by the predominantly streaming behavior of SpMV [2], but may lead to an optimistic bound in the multiple vector case.

6.1 Lower Bound Execution Time Model

If the total execution time is T seconds, the performance P in Mflop/s is

$$P = \frac{4kv}{T} \times 10^{-6} \quad (2)$$

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, and the memory access latency is α_{mem}. Let H_i and M_i be the number of cache hits and misses at each level i, respectively. Also, let L be the total number of loads. The execution time T is

$$\begin{aligned} T &= \sum_{i=1}^{\kappa} \alpha_i H_i + \alpha_{\text{mem}} M_\kappa \\ &= \alpha_1 L + \sum_{i=1}^{\kappa-1} (\alpha_{i+1} - \alpha_i) M_i + (\alpha_{\text{mem}} - \alpha_\kappa) M_\kappa \end{aligned} \quad (3)$$

where $H_1 = L - M_1$ and $H_i = M_{i-1} - M_i$ for $2 \leq i \leq \kappa$. According to Equation (3), we can minimize T by minimizing M_i, assuming $\alpha_{i+1} \geq \alpha_i$.

6.2 Counting Load Operations

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, B_{rc} be the number of $r \times c$ non-zero register blocks, and f_{rc} be the fill ratio given these blocks. 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.

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, since a diagonal block has at most $\frac{r(r+1)}{2}$ elements. 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 $\|B_{rc}\| \approx kf_{rc} - \frac{m(r+1)}{2}$ and each register block contains rc elements. 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, $D_r + B_{rc}$ integer column indices, and $\lceil \frac{m}{r} \rceil + 1$ integer row indices. Since the fill ratio is defined as the number of stored elements (fill included) divided by the number of non-zeros (fill excluded), $f_{rc} \approx \frac{\|D_r\| + \|B_{rc}\|}{k}$ 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 r entries of the destinations can be loaded once for each of D_r block rows and kept in registers for the duration of the block row multiply. We also assume that all c destination elements can be kept in registers during the multiplication of a given transpose block, thereby requiring $B_{rc}c$ additional loads from the destination. A similar analysis for the block columns in the transpose of the stored triangle yields $rD_r + cB_{rc}$ loads. Thus, the number of loads, scaled for multiple vectors, is [3]

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

$$\text{Loads}(r,c,v) = \underbrace{B_{rc}rc + D_r\left(\frac{r^2+r}{2}\right) + B_{rc} + D_r + \left\lceil \frac{m}{r} \right\rceil + 1}_{\text{matrix}} + \underbrace{vrD_r + vcB_{rc}}_{\text{source}} + \underbrace{vrD_r + vcB_{rc}}_{\text{destination}} \quad (4)$$

6.3 Lower Bounds on 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 elements. In considering the vectors, we assume the 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 $2mv$ source and destination elements. Thus, a lower bound $\text{M}_{\text{lower}}^{(1)}$ on L1 misses is

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

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 substituting the appropriate line size.

7 Evaluation

Figures 2–5 summarize the performance of our optimizations on the four platforms in Table 1 and the matrices in Table 2. We compare the following nine implementations and bounds.

1. **Non-Symmetric Unoptimized Reference**: The unblocked $(1, 1)$ single vector implementation with non-symmetric storage. Represented by *crosses*.

2. **Symmetric Reference**: The unblocked $(1, 1)$ single vector implementation with symmetric storage. Represented by *five-pointed stars*.

3. **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. Represented by *asterisks*.

4. **Symmetric Register Blocked**: The blocked single vector implementation with symmetric storage where r and c are chosen by exhaustive search to maximize performance. Represented by *plus signs*.

5. **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. Represented by *upward pointing triangles*.

6. **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}. Represented by *six-pointed stars*.

7. **Tuning Parameter Selection Heuristic**: The blocked multiple vector implementation with symmetric storage where r, c, and v are chosen by the tuning parameter selection heuristic described in Section 5. We refer to these parameters as r_{heur}, c_{heur}, and v_{heur}. Represented by *hollow circles*.

8. **Analytic Upper Bound**: The analytic upper bound on performance implementation 6. We use Equation (4) and Equation (5) to compute the numbers of loads and cache misses. Represented by *solid lines*.

9. **PAPI Upper Bound**: An upper bound on performance for implementation 6. The number of loads and cache misses are obtained from PAPI event counters. Represented by *dashed lines*.

7.1 Effects of Symmetry on Performance

Considering all four platforms, the maximum performance gain from symmetry is $2.08\times$ for register blocked SpMV over the non-symmetric register blocked kernel (4 versus 3). The median speedup is $1.34\times$. The maximum performance gain from symmetry is $2.58\times$ for register and vector blocked SpMM over the non-symmetric register and vector blocked kernel (6 versus 5). However, the median speedup of $1.09\times$ is appreciably slower. Furthermore, symmetry can reduce performance in the rare worst case. The chosen register block sizes, shown in the appendices, suggest 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.

An implementation optimized for symmetry, register and vector blocking achieves maximum, median, minimum performance gains of $7.32\times$, $4.15\times$, and $1.60\times$ compared to the naïve code (6 versus 1) over all platforms and matrices. Given symmetric storage, the other two optimizations almost always improve and never reduce performance. The increasing performance gains as optimizations are incrementally applied suggest cumulative performance effects of these optimizations.

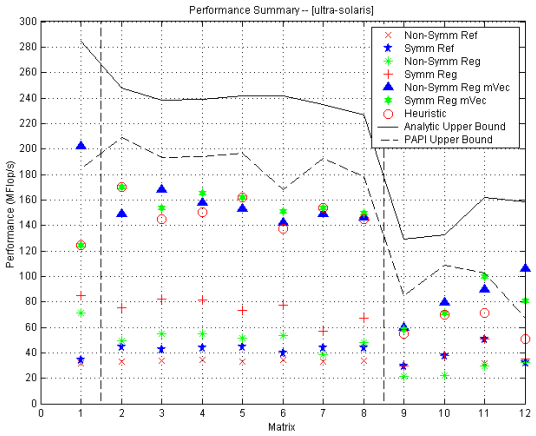

Figure 2. Performance Summary – Sun Ultra 2i. Performance (MFlop/s) of various optimized implementations compared to upper bounds on performance.

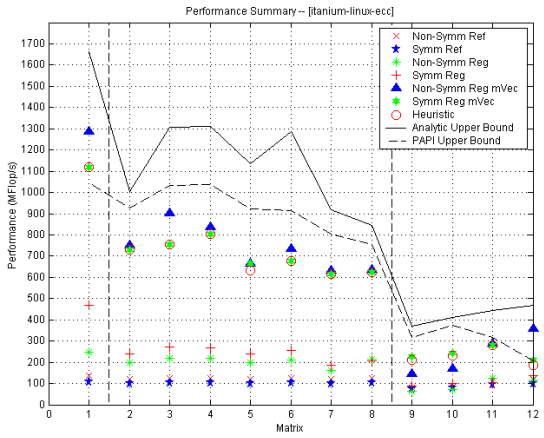

Figure 3. Performance Summary – Intel Itanium 1. Performance data and upper bounds shown in a format analogous to the format in Figure 2.

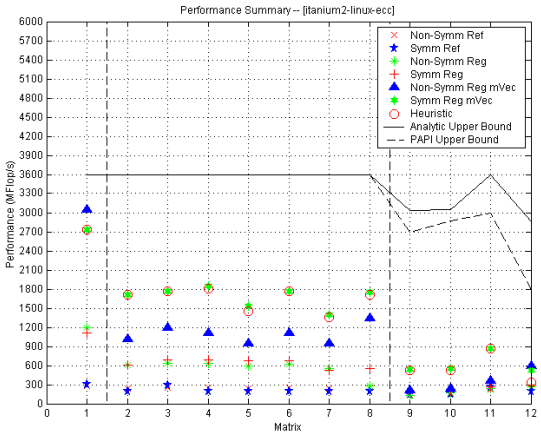

Figure 4. Performance Summary – Intel Itanium 2. Performance data and upper bounds shown in a format analogous to the format in Figure 2. The model predicts machine peak (3.6 Gflop/s) for matrices 1–7.

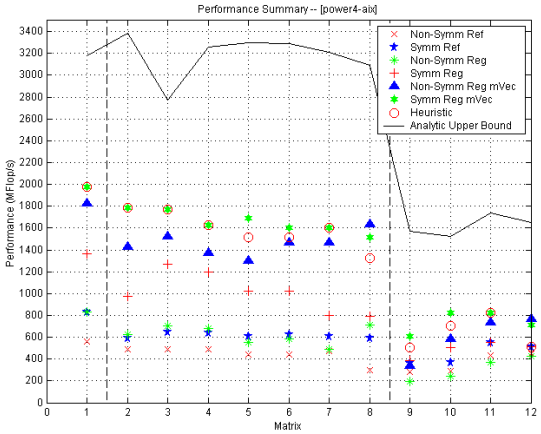

Figure 5. Performance Summary – IBM Power 4. Performance data and upper bounds shown in a format analogous to the format in Figure 2. The analytic upper bound is omitted due to the unavailability of hardware counters.

	Overall	Ultra 2i	Itanium 1	Itanium 2	Power4
I. Symmetry Register Blocking					
minimum	-9.89	35.48	-9.89	-9.89	31.97
median	58.33	60.94	47.64	57.98	61.09
maximum	64.79	64.79	64.79	64.79	64.79
II. Symmetry Register and Vector Blocking					
minimum	-4.17	28.57	-4.17	12.28	28.57
median	53.70	53.70	50.25	59.51	53.70
maximum	64.79	57.08	64.79	64.79	64.79

Table 3. Percentage savings in matrix storage.

7.2 Accuracy of Automatic Tuning Parameter Selection

On the Ultra 2i, Itanium 1, and Itanium 2, the block size selection procedure generally chooses (r, c, v) whose performance is 93% or more of the best by exhaustive search. On Power4, the predictions are somewhat less accurate, at 85% of the best or greater. Nevertheless, in nearly all cases the selected implementation yields at least some speedup over the symmetric register blocked single vector case (7 versus 4).

The heuristic does not select near-optimal implementations in the case of Matrix 12 (a linear programming matrix). We discuss this case in detail in the full report [3].

7.3 Proximity of Performance to Upper Bounds

To evaluate our performance models, we consider the proximity of the upper bounds to the measured performance of the symmetric, register and vector blocked code. The finite element matrices (FEM Matrices 2–8) achieve 72% to 90% of the PAPI bound, but only 53% to 73% of the analytic bound on the Ultra 2i and Itanium 1 . This difference suggests that further performance improvements on these platforms will require reducing the gap between the number of predicted and measured cache misses. The measured performance of these matrices on the Itanium 2 and Power 4 were 38% to 63% of the analytic bound. [4]

The non-FEM matrices realize relatively lower measured performance, achieving 65% to 120% of the PAPI bound and 44% to 62% of the analytic bound on the Ultra 2i and Itanium 1. Cases in which the measured performance exceeds the PAPI upper bound (Matrix 12 on Ultra 2i and Matrix 1 on Itanium 1) may be caused by limitations in the PAPI counters. The realized performance of these matrices on the Itanium 2 is 29% to 38% and 23% to 32% of the PAPI and analytic bounds, respectively, and 38% to 54% of the analytic bound on the Power 4.

[4] Note that no PAPI data was available for a bound on the Power 4.

7.4 Effects of Symmetry on Storage

Symmetry can significantly save storage . We show maximum and median savings of 64.79% and 56.52%, respectively, for symmetric register blocking (Table 3, I). Symmetry may also use almost 10% more memory in the case of matrix 12 on the Itanium 1 and 2. We show maximum and median savings of 64.79% and 53.70%, respectively, when adding multiple vectors (Table 3, II).

A large register block size in a symmetric code can save more than 50% of storage because the memory for matrix indices decreases by up to a factor of $r \times c$, augmenting the savings from storing half the matrix. An increase in storage from symmetric storage is also possible, however, if the chosen register block size results in significant fill.

8 Related Work

Temam and Jalby [18] and Fraguela, *et al.*, [19] developed probabilistic cache miss models for SpMV, but assumed a uniform distribution of non-zero entries. In contrast, our lower bounds account only for compulsory misses. Gropp, *et al.*, use similar bounds to analyze and tune a computational fluid dynamics code [17] on Itanium 1. However, we tune for a variety of architectures and matrix domains. Work in sparse compilers (*e.g.* Bik *et al.* [20], Pugh and Spheisman [21], and the Bernoulli compiler [22]) complements our own work. These projects consider expressing sparse kernels and data structures for code generation. In contrast, we use a hybrid off-line, on-line model for selecting transformations.

9 Conclusions and Future Directions

Symmetry significantly reduces storage requirements, saving as much as 64.79% when combined with register blocking. Symmetry, register and vector blocking, improves performance by as much as $7.3\times$ (median $4.15\times$) over a naïve code, $2.08\times$ (median $1.34\times$) over non-symmetric register blocked SpMV, and $2.6\times$ (median $1.1\times$) over non-symmetric register and vector blocked SpMM. Moreover, the performance effects of these optimizations appear to be cumulative, making the case to combine these techniques.

Our heuristic, based on an empirical performance modeling and search procedure, is reasonably accurate, particularly for matrices arising from FEM applications. Heuristic chosen tuning parameters yield performance within 85% of the performance achieved from exhaustive search. Additional refinements may improve the accuracy on matrices with little or no block structure.

The performance of our optimized implementations are, on average, within 68% of the performance bounds, smaller

than previously observed for non-symmetric SpMV. Additional refinements to explictly model low-level code generation and employing automated low-level tuning techniques (*e.g.*, ATLAS/PHiPAC [8, 9]) may close the gap.

Sparse kernels may be optimized for other forms of *symmetry* (*e.g.* structural, skew, hermitian, skew hermitian). Symmetric *cache blocking* may mitigate the effects of increasing matrix dimensions and vectors that do not fit in cache, grouping the matrix into large blocks whose sizes are determined by cache size. Optimizations for sparse kernels may also be implemented and evaluated for *parallel systems*, such as SMPs and MPPs [6]. Lastly, performance optimized kernels will be distributed to *application end-users*.

References

[1] E.-J. Im., K. Yelick, R. Vuduc. SPARSITY: Framework for Optimizing Sparse Matrix-Vector Multiply. *International Journal of High Performance Computing Applications*, 18(1), 2004.

[2] R. Vuduc. Automatic Performance Tuning of Sparse Matrix Kernels. PhD thesis, U.C. Berkeley, Dec. 2003.

[3] B. Lee, R. Vuduc, J. Demmel, K. Yelick, M. de Lorimier, L. Zhong. *Performance Optimizations and Bounds for Sparse Symmetric Matrix-Multiple Vector Multiply*. Technical Report UCB/CSD-03-1297, University of California, Berkeley, November 25, 2003.

[4] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, B. Lee. *Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply*. In *Supercomputing*,Baltimore, MD, November 2002.

[5] E.-J. Im. *Optimization the Performance of Sparse Matrix–Vector Multiplication*. PhD thesis, U.C. Berkeley, May 2000.

[6] 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.

[7] 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.

[8] 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.

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

[10] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. PhD thesis, U.C. Berkeley, February 1992.

[11] 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.

[12] 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`.

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

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

[15] 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.

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

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

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

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

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

[21] 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.

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