Performance Optimizations and Bounds for Sparse Matrix Kernels

Richard Vuduc James W. Demmel* Katherine A. Yelick
Shoaib Kamil Rajesh Nishtala
Benjamin Lee

Computer Science Division
University of California, Berkeley
Berkeley, California, USA
{richie,demmel,yelick,skamil,rajeshn,blee20}@cs.berkeley.edu

Abstract

Building high-performance implementations of sparse matrix-vector multiply (SpM×V), an important and ubiquitous computational kernel, is fundamentally limited by a variety of factors: the increasing performance gap between processors and memory, the storage and instruction overhead of manipulating sparse data structures, and the irregular memory access due to sparse storage. Moreover, the complexity of modeling execution of modern microprocessors makes selecting the best data structure and SpM×V implementation for a given sparse matrix a difficult task.

In this paper, we consider a range of performance bounds and models for SpM×V, both practical and hypothetical, which quantify these limits. The models vary both in cost and by what information is assumed—from the purely static to those that assume perfect knowledge of run-time information available via processor hardware counters. We evaluate these models and bounds on a variety of hardware platforms and matrices, and show that the task of selecting the best implementation and data structure for a particular matrix will require a combination of modeling and run-time searching.

Furthermore, we use our performance bounds to show that our previously developed optimization technique, *register blocking*, which improves register-level reuse by exploiting naturally occurring dense subblocks, is unlikely to be improved upon significantly by additional low-level instruction scheduling efforts. Instead, we examine our recent efforts to overcome the fundamental limits through the use other kernels: multiplication with symmetric matrices, by multiple vectors, and higher-level kernels like multiplication of a vector by $A^T A$.

*CS Division and Department of Mathematics

1 Introduction

general intro goes here

architectural trends affecting performance

We are interested in using performance bounds and models to answer the following three questions:

- How close are our register blocking implementations to the best possible? An answer would indicate how much improvement can be gained by low-level scheduling (*i.e.*, rearranging the innermost, unrolled loops).
- What are the limits of what models can do to select transformations? REFINE THIS QUESTION

notational stuff: operation is $y = y + Ax$, where A is an $m \times n$ sparse matrix, and x and y are vectors. We refer to x as the *source* vector, and y as the *destination* vector.

related work:

Cache and memory behavior have been well-studied for dense matrix kernels. A variety of sophisticated static models have been developed, each with the goal of providing a compiler with sufficiently precise models for selecting memory hierarchy transformations and parameters such as tile sizes [6, 9, 19, 5, 28].

Analysis tools for the sparse case are less well-developed, though there have been a number of notable efforts. Temam and Jalby [26], Heras, *et al.*[13], and Fraguela, *et al.*[8] have developed Temam and Jalby have formulated analytic models of the cache miss behavior of SpM×V for single and multiple vectors. These models vary in their ability to account for self- and cross-interference misses. One weakness of all the models is that only

prior modeling efforts for the irregular case: [26], [13], [8], [20]

prior use of bounds in performance tuning: [10]

Bik's work on non-zero data structure selection: [2]

multilevel blocking with multiple vectors: [21]

other related: [25], [17], [11], [22].

2 Experimental Setup

The following is a brief summary of our experimental setup and methodology, which we assume throughout the remainder of the paper.

Platforms

All of our experimental evaluations are performed on machines based on the microprocessors shown in Table 1. The table summarizes their hardware and compiler configurations, and our measurements of key dense kernels for reference. Latency estimates shown were obtained from published sources and confirmed experimentally using the memory system microbenchmark due to Saavedra-Barrera [24].

Property	Sun Ultra 2i	Intel Pentium III	IBM Power3	Intel Itanium
Clock rate	333 MHz	500 MHz	375 MHz	800 MHz
Peak Main Memory Bandwidth	500 MB/s	680 MB/s	1.6 GB/s	2.1 GB/s
Peak Flop Rate	667 Mflop/s	500 Mflop/s	1.5 Gflop/s	3.2 Gflop/s
DGEMM ($n = 1000$)	425 Mflop/s	331 Mflop/s	1.3 Gflop/s	2.2 Gflop/s
DGEMV ($n = 1000$)	58 Mflop/s	96 Mflop/s	260 Mflop/s	345 Mflop/s
STREAM Triad Bandwidth	250 MB/s	350 MB/s	715 MB/s	1.1 GB/s
L1 data cache size	16 KB	16 KB	64 KB	16 KB
L1 line size	16 B	32 B	128 B	32 B
L1 latency	2 cy	1 cy	1 cy	2 cy (int)
L2 cache size	2 MB	512 KB	8 MB	96 KB
L2 line size	64 B	32 B	128 B	64 B
L2 latency	7 cy	18 cy	9 cy	6 cy (int) 9 cy (double)
L3 cache size	N/A	N/A	N/A	2 MB
L3 line size				64 B
L3 latency				21 cy (int) 24 cy (double)
TLB entries	64	64	256	32 (L1 TLB) 96 (L2 TLB)
Page size	8 KB	4 KB	4 KB	4 KB
Largest Cache + TLB latency (≈)	36 cy	26 cy	35 cy	36 cy (int) 47 cy (double)
Maximum memory latency (≈)	66 cy	60 cy	139 cy	85 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	IBM C v5.0	Intel C v5.0.1
Flags	`-dalign -xtarget=native -xO5 -xarch=v8plusa -xrestrict=all`	`-O3 -tpp6 -xK -unroll`	`-O3 -qalias=allp -qarch=pwr3 -qtune=pwr3`	`-O3`

Table 1: Basic data for the machines used in our experiments. Performance figures for the BLAS on the Sun Ultra 2i platform are the best of Sun's performance library v6.0 and ATLAS 3.2.0 [27]; on the Pentium III (Katmai) platform: figures reported are the best of Intel's MKL v5.2, ATLAS 3.3.5 [27], and ITXGEMM 1.1 [12]; on the Power3 platform: IBM ESSL 3.1.2; on the Itanium platform: Intel MKL v5.2.

Matrices

We evaluate the SpM×V implementations on the matrix benchmark suite used by Im [14]. Table 2 summarizes the size and source of each matrix. Most of the matrices are available in the collections at NIST (MatrixMarket [3]) and the University of Florida [7].

The matrices in Table 2 are arranged in roughly four groups. Matrix 1 is a dense matrix stored in sparse format; matrices 2–17 arise in finite element method (FEM) applications; 18–39 come from assorted applications; 40–44 are linear programming examples.

Timing

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

The largest cache on some machines (notably, the Power3) is large enough to contain some of the matrices. To avoid inflated findings, within a platform we report performance results only on the subset of out-of-cache matrices. Figures will still always use the numbering scheme shown in Table 2.

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

3 Improving Register Reuse

This section provides a brief overview of SPARSITY's *register blocking* optimization, a technique for improving register reuse over that of a conventional implementation. For concreteness, we assume a baseline that stores the matrix in compressed sparse row (CSR) format.[2]

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 those blocks which contain at least one non-zero are stored. The computation of SpM×V proceeds by iteration over blocks. 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 is available.

In SPARSITY, the 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.[3] A 2×2 example of BCSR is shown in Figure

[1]The standard deviation of these trials is typically less than 1% of the median.
[2]See Barrett, *et al.*, [1] for a list of common formats.
[3]Row-major is SPARSITY's convention; column-major or other layouts are possible.

	Name	Application Area	Dimension	Nonzeros
1	dense1000	Dense Matrix	1000x 1000	1000000
2	raefsky3	Fluid structure interaction	21200x21200	1488768
3	olafu	Accuracy problem	16146x16146	1015156
4	bcsstk35	Stiff matrix automobile frame	30237x30237	1450163
5	venkat01	Flow simulation	62424x62424	1717792
6	crystk02	FEM Crystal free vibration	13965x13965	968583
7	crystk03	FEM Crystal free vibration	24696x24696	1751178
8	nasasrb	Shuttle rocket booster	54870x54870	2677324
9	3dtube	3-D pressure tube	45330x45330	3213332
10	ct20stif	CT20 Engine block	52329x52329	2698463
11	bai	Airfoil eigenvalue calculation	23560x23560	484256
12	raefsky4	buckling problem	19779x19779	1328611
13	ex11	3D steady flow caculation	16614x16614	1096948
14	rdist1	Chemical process separation	4134x 4134	94408
15	vavasis3	2D PDE problem	41092x41092	1683902
16	orani678	Economic modeling	2529x 2529	90185
17	rim	FEM fluid mechanics problem	22560x22560	1014951
18	memplus	Circuit Simulation	17758x17758	126150
19	gemat11	Power flow	4929x 4929	33185
20	lhr10	Light hydrocarbon recovery	10672x10672	232633
21	goodwin	Fluid mechanics problem	7320x 7320	324784
22	bayer02	Chemical process simulation	13935x13935	63679
23	bayer10	Chemical process simulation	13436x13436	94926
24	coater2	Simulation of coating flows	9540x 9540	207308
25	finan512	Financial portfolio optimization	74752x74752	596992
26	onetone2	Harmonic balance method	36057x36057	227628
27	pwt	Structural engineering problem	36519x36519	326107
28	vibrobox	Structure of vibroacoustic problem	12328x12328	342828
29	wang4	Semiconductor device simulation	26068x26068	177196
30	lnsp3937	Fluid flow modeling	3937x 3937	25407
31	lns3937	Fluid flow modeling	3937x 3937	25407
32	sherman5	Oil reservoir modeling	3312x 3312	20793
33	sherman3	Oil reservoir modeling	5005x 5005	20033
34	orsreg1	Oil reservoir simulation	2205x 2205	14133
35	saylr4	Oil reservoir modeling	3564x 3564	22316
36	shyy161	Viscous flow calculation	76480x76480	329762
37	wang3	Semiconductor device simulation	26064x26064	177168
38	mcfe	astrophysics	765x 765	24382
39	jpwh991	Circuit physics modeling	991x 991	6027
40	gupta1	Linear programming matrix	31802x31802	2164210
41	lpcreb	Linear Programming problem	9648x77137	260785
42	lpcred	Linear Programming problem	8926x73948	246614
43	lpfit2p	Linear Programming problem	3000x13525	50284
44	lpnug20	Linear Programming problem	15240x72600	304800

Table 2: Matrix benchmark suite. These matrices were chosen to present consist results with previous work on Sparsity. Matrices are categorized roughly as follows: 1 is a dense matrix stored in sparse format; 2–17 arise in finite element applications; 18–39 come from assorted applications; 40–44 are linear programming examples.

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

b_row_start $= \begin{pmatrix} 0 & 2 & 4 \end{pmatrix}$
b_col_idx $= \begin{pmatrix} 0 & 4 & 2 & 4 \end{pmatrix}$
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 1: **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].)

1. When $r = c = 1$, BCSR reduces to CSR.[4]

Note that BCSR potentially stores fewer column indices than CSR implementation (one per block instead of one per non-zero), reducing both storage and data structure manipulation overhead. Furthermore, SPARSITY implementations fully unroll the $r \times c$ submatrix computation, reducing loop overheads and exposing scheduling opportunities to the compiler. An example of a 2×2 implementation is given in Appendix A.

However, the figure 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. By analogy to tiling in the dense case, the most difficult aspect of applying register blocking is knowing when (*i.e.*, on which matrices) to apply it and how to select the block size.

This difficulty is striking when we examine register blocking performance for various values of r and c. In Figure 2, we show, for our four hardware platforms, the performance (Mflop/s) of block sizes up to 12×12 on a very regular "sparse" problem: a dense 1000×1000 matrix stored in sparse (BCSR) format.[5] Performance is a strong function of the architecture, compiler, and block size. Moreover, the irregularity of the spaces suggests that performance will in general be difficult to model; however, the profiles shown clearly contain a lot of information, which we exploit in our modeling efforts (Section 5).

[4]The performance of this code is comparable to that of the CSR implementation from the NIST Sparse BLAS [23].

[5]Note that for the performance profiles shown, the matrix size is actually $\lceil \frac{1000}{r} \rceil r \times \lceil \frac{1000}{c} \rceil c$. For the block sizes considered, the true matrix size differs from the 1000×1000 case by no more than 2%.

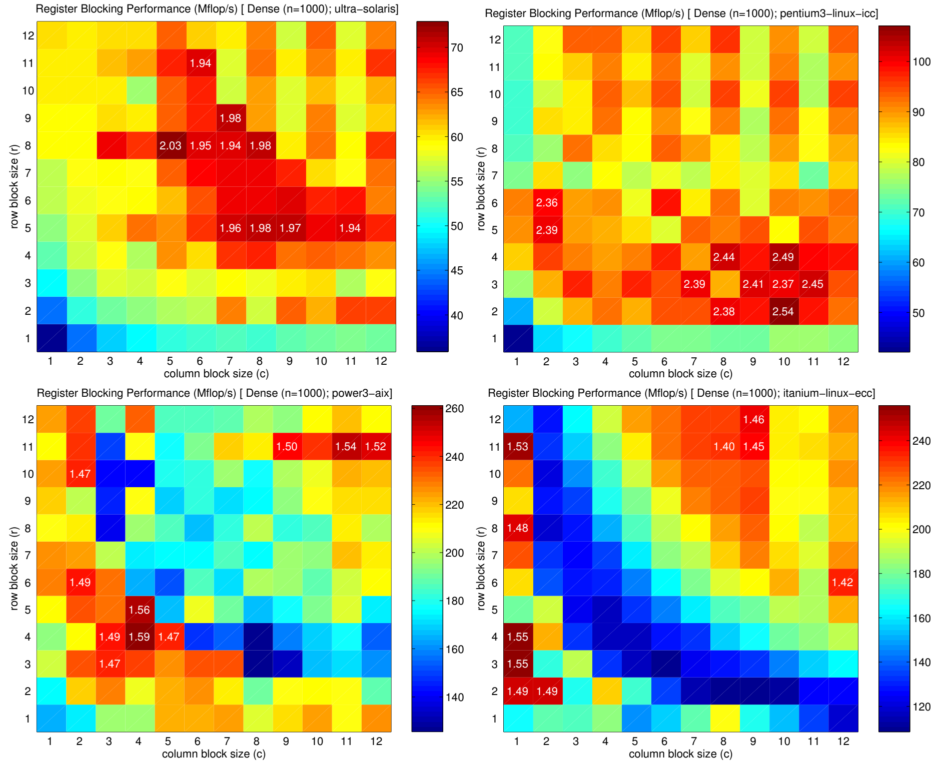

Figure 2: The performance of $r \times c$ register blocked implementations on a dense 1000×1000 matrix stored in BCSR format, on block sizes up to 12×12. These performance profiles are shown for the four platforms listed in Table 1. On each platform, each square is an $r \times c$ implementation shaded by its performance, in Mflop/s. The top 10 implementations are labeled by their speedup relative to the 1×1 implementation. Though the performance has irregular structure and therefore appears difficult to model, the best implementations differ in performance by little more than 10%; furthermore, they appear "semi-clustered." Platforms (clockwise from upper-left): Sun Ultra 2i, Intel Pentium III, IBM Power3, Intel Itanium.

4 Bounds on Register Blocking Performance

Below, develop a sequence of performance bounds of varying complexity to try to understand performance plots like those shown in Figures 2. At the end of this section, we evaluate the SPARSITY implementations relative to these bounds.

4.1 Preliminaries

To be concrete, we fix the data structure to be the BCSR format outlined in Section 3. We can count the number of loads and stores required for SpM×V using this format as follows. Let A be an $m \times n$ matrix with k non-zeros. Let K_{rc} be the number of $r \times c$ non-zero blocks required to store the matrix in $r \times c$ BCSR format; note that $K_{1,1} = k$. The matrix requires storage of $K_{rc}rc$ double precision values, K_{rc} integers for the column indices, and $\lceil \frac{m}{r} \rceil + 1$ integers for the row pointers. The fill ratio is $f_{rc} = \frac{K_{rc}rc}{k}$, and is always at least 1.

Every matrix entry must be loaded once. We assume that SpM×V iterates over block rows, and that all r entries of the destination vector can be kept in registers for the duration of a block row multiply. Thus, we only need to load each element of the destination vector once, and store each element once. Finally, we assume that all c source vector elements can be kept in registers during the multiplication of each block, thus requiring a total of $K_{rc}c = \frac{kf_{rc}}{r}$ loads of the source vector. In terms of the number of non-zeros and the fill ratio, the total number of loads is

$$\begin{aligned} \text{Loads} &= \underbrace{kf_{rc} + \frac{kf_{rc}}{rc} + \left\lceil \frac{m}{r} \right\rceil + 1}_{\text{matrix}} + \underbrace{\frac{kf_{rc}}{r}}_{\text{source vec}} + \underbrace{m}_{\text{dest vec}} \\ &= kf_{rc}\left(1 + \frac{1}{rc} + \frac{1}{r}\right) + m + \left\lceil \frac{m}{r} \right\rceil + 1 \end{aligned} \qquad (1)$$

and the total number of stores is m.

Observe that if there were little or no fill (*e.g.*, for a dense matrix stored in sparse format), then increasing the block size would reduce the overhead for storing the column indices by $\frac{1}{rc}$. Also note that the source vector load term depends only on r, introducing a slight asymmetry in the number of loads as a function of block size.

Finally, in the bounds we derive below, we denote the ratio of a double-precision word (`double` in C) to an integer index (`int`) by γ.

4.2 Memory bandwidth bounds and estimates

Since SpM×V is memory-bandwidth limited, we can use the machine's peak memory bandwidth to compute the minimum time to move all of the matrix and vector data from memory to the processor, and to write out the vector data. If the bandwidth is β (in units of double-precision words per second), then the

time to move the matrix and vector data is

$$T_{\mathrm{bw}}(r,c) = \frac{kf_{rc}\left(1+\frac{1}{\gamma rc}\right) + \frac{\lceil \frac{m}{r} \rceil + 1}{\gamma} + 2m + n}{\beta} \tag{2}$$

Note that we count only the time to read the source vector once ($\frac{n}{\beta}$). Assuming the latencies due to computation are completely hidden, our coarse performance bound is simply the ideal flop count ($2k$) divided by this time. We refer to this bound as the *memory bandwidth bound*.

An even more optimistic, though not entirely unreasonable, upper-bound on performance is a modification equation 2 that excludes the indices. This bound is not entirely unreasonable because some sparsity patterns (*e.g.*, band matrices) would not require any index storage.[6]

We can obtain a third estimate of performance using the bandwidth β_{STREAM} reported by the STREAM microbenchmark [18], shown in Table 1. Note that β_{STREAM} is not a bound but an estimate of sustained memory bandwidth, measured with respect to the vector scale operation $z = y + \alpha x$ for double-precision vectors x, y, z and scalar α.

4.3 Bounds based on modeling cache misses

We can estimate a tighter, analytic upper-bound on performance by specifying a lower bound on cache misses.

We start with the L1 cache. Let l_1 be the L1-cache line size, in double-precision words. One compulsory L1 read miss is incurred for every matrix element (value and index) and destination vector element. The source vector miss count is more complicated to predict. If the source vector size is less than the L1 cache size, in the best case we would incur only n cold-start misses for the source vector. Thus, a lower bound $M^{(1)}_{\mathrm{lower}}$ on L1 misses is

$$M^{(1)}_{\mathrm{lower}}(r,c) = \frac{1}{l_1}\left[kf_{rc}\left(1+\frac{1}{\gamma rc}\right) + n + \frac{1}{\gamma}\left(\left\lceil\frac{m}{r}\right\rceil + 1\right) + m\right]. \tag{3}$$

The factor of $\frac{1}{l_1}$ accounts for the L1 line size. An analogous expression applies at the other cache levels by simply substituting the right line size.

In the worst case, we will miss on every access to a line of the source vector in each block due to capacity and conflict (both self- and cross-interference) misses; thus, an upper bound on misses is

$$M^{(1)}_{\mathrm{upper}}(r,c) = \frac{1}{l_1}\left[kf_{rc}\left(1+\frac{1}{\gamma rc}+\frac{1}{r}\right) + \frac{1}{\gamma}\left(\left\lceil\frac{m}{r}\right\rceil + 1\right) + m\right]. \tag{4}$$

A qualitative consequence of this simple model is that cache line size is an important architectural parameter.[7] This may help partially explain why,

[6] One could also think of this bound as a hypothetical bound on what the ultimate reordering algorithm might achieve, *i.e.*, if it could rearrange the matrix into a form sufficiently structured so as to not require index storage.

[7] Though cache size is implicit in the miss lower bound.

for instance, both the Power3 and Itanium attain similar peak performance in Figure 2: even though both the memory bandwidth and peak flop rate on Itanium are higher, the L2 line size on the Power 3 is twice as large.

Armed with bounds on the number of cache misses, we can construct the following simple model of execution time. As with the memory bandwidth bound, we assume that we can overlap the latencies due to computation with memory access. Let h_i be the number of hits at cache level i, m_i be the number of misses.

$$T_{\text{lower}} = \sum_{i=1}^{\kappa-1} h_i \alpha_i + m_\kappa \alpha_{\text{mem}}, \tag{5}$$

where α_i is the access time (in cycles or seconds) at cache level i, κ is the lowest level of cache, and α_{mem} be the memory access time. Assuming a perfect nesting of the caches, so that a miss at level i is an access at level $i+1$, then $h_{i+1} = m_i - m_{i+1}$ for $i \geq 2$, and h_1 is the total number of load operations given by equation (1).

To get an estimate of the upper bound on performance, we can substitute the lower bounds on cache misses given by equation (3) to get a lower bound on execution time, and convert to Mflop/s. Similarly, we can get a lower bound on performance by substituting $M^{(i)}_{\text{upper}}$ for $M^{(i)}_{\text{lower}}$.

Interaction with the TLB complicates our estimate of the average memory access latency. We incorporate the TLB into our performance upper bound by replacing the memory access latency α_{mem} by the cache hit + TLB hit time shown in Table . For the lower bound, we assume α_{mem} is the full memory access time. This assumption will tend to make the upper bound appear more optimistic.

When appropriate, we apply slight refinements to this model to incorporate features of our evaluation platforms. For instance, both the Power3 and the Itanium can commit two loads per cycle if they hit in the L1 cache. Thus, we reduce the L1 latency α_1 by two to obtain a performance upper bound. Also, we take into account the fact that on Itanium, the cache hit times depend on whether the data is tied to integer or double-precision registers [16].

4.4 Evaluating the bounds

We performed an exhaustive search over all register block sizes up to 12×12 for all matrices and platforms. Figures 3–6 show where the performance of the best SPARSITY implementations appear relative to our estimated bounds. Note that the upper and lower bounds are functions of r, c; for both bounds, we show the bound for r, c with the *best* performance.

On the Ultra 2i, Pentium III, and Itanium platforms, SPARSITY implementations achieve 70–80% or more of the upper-bound on most of matrices in the FEM set. Such matrices have natural dense structure which register blocking is able to exploit; the proximity to the bound suggests that additional low-level tuning of the register block implementations is unlikely to lead to significant additional gains.

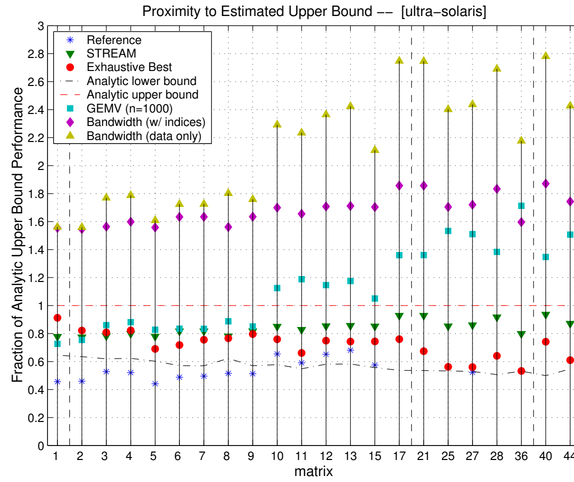

Figure 3: Proximity to the upper-bound estimate: Fraction of upper-bound performance achieved for the various bounds in Section 4 and the best blocked implementation ("exhaustive best"). The analytic upper bound is the best performance according to the execution time model given by equation (5) for all r, c. Similarly, the analytic lower bound is the best lower-bound performance for all r, c. Where the reference implementation data point appears to be missing, it actually coincides with the exhaustive best implementation. On the FEM matrices, SPARSITY performance is 70–80% of our estimated upper bound.

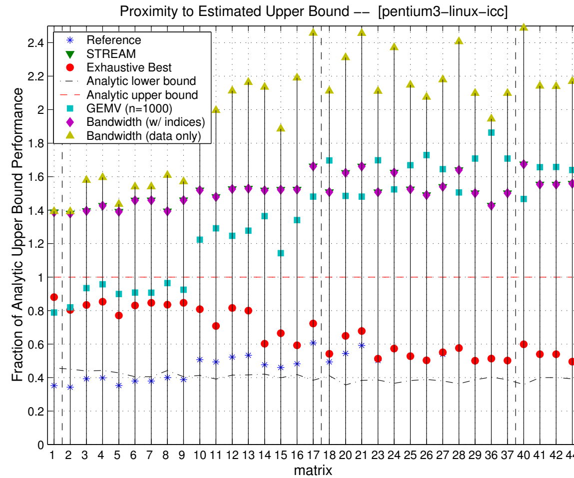

Figure 4: Same as Figure 3 for the Pentium III. SPARSITY implementations achieve 80% or more of the upper-bound on many of the FEM matrices.

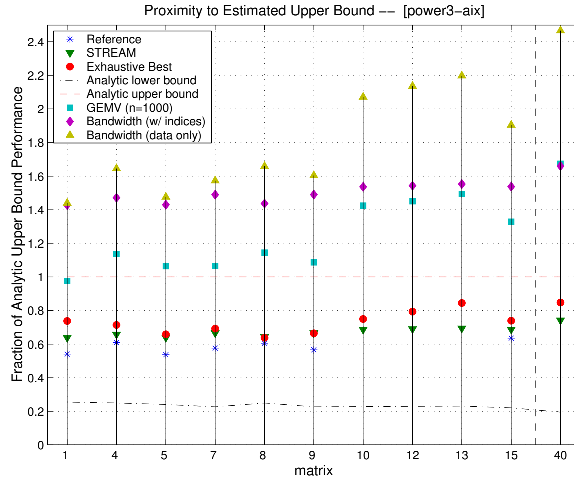

Figure 5: Same as Figure 3 for the IBM Power3. SPARSITY implementations achieve anywhere between 60–85% of our upper-bound estimate.

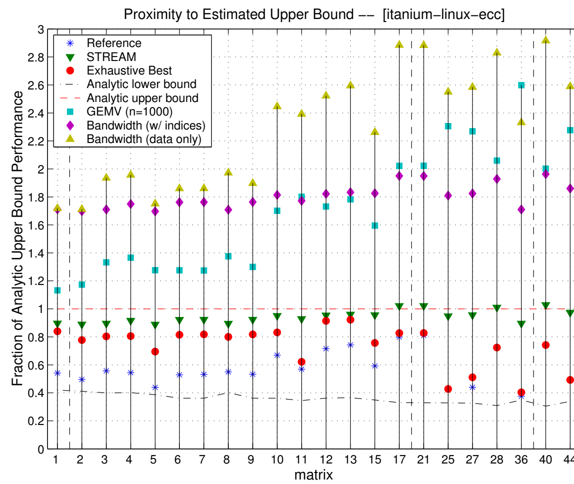

Figure 6: Same as Figure 3 for the Intel Itanium. SPARSITY implementations achieve 80% or more of the upper-bound on most of the FEM matrices.

The matrices from other applications display more varied behavior, though they tend to perform closer to the lower performance bound than the upper, typically ranging from 40%–60% of the upper bound performance. Recall that the lower bound assumes that accesses to the source vector cache lines will always miss, due either to capacity or conflict misses. This suggests that the non-FEM matrices, possibly due to their particular sparsity patterns, exhibit more conflicts or reduced spatial locality. Some form of reordering is likely to be the most effective way to address this performance issue.

On the Power3, the implementations all fall between 60%–85% of the estimated upper bound. Although this fraction is low compared to the other platforms, observe that the performance is comparable to the performance estimate using the STREAM benchmark.

Though these results are encouraging, they are also limited. The upper bounds are based on expected upper-bounds with respect to our particular register blocking data structure. It is possible that other data structures (for instance, those that might remove the uniform block size assumption and therefore change the dependence of f_{rc} on r, c) could do better.

Finally, note that on the Ultra 2i and Pentium III, it appears that the Sparsity implementation is running faster than DGEMV on the dense matrix stored in sparse format (matrix #1). It is likely that the vendor-supplied routine in this case was not optimally tuned. The primary intention of showing DGEMV is as a useful scale reference, not to advocate conversion from dense to sparse formats.

5 Block Size Selection

Though we report the performance of the best implementation by exhaustive search in the previous section, such searches can be costly: on Itanium, reorganizing the matrix *once* (*i.e.*, for one value of $r\times c$) is 10–30 times the cost of running the reference SpM$\times$V once.

Since we are assuming selection will in general occur only when the final matrix is known at run-time, the cost of an exhaustive search is prohibitive.[8] Sparsity uses the hybrid off-line, on-line heuristic described below. The goal of this section is to evaluate the cost and accuracy of the current heuristic and a recently developed improvement. We use idealized static models as a way of comparing indirectly to the best that might be expected of a purely static model.

5.1 The Sparsity heuristic, and an improvement

The Sparsity block size selection heuristic is based on the following, simple model of performance. Our goal is to choose an r, c that maximizes $P_A(r, c)$,

[8] Though we have reported results for searches of up to 12×12, in practice we have observed block size selection as high as 8×8, and there is not obvious bound on application submatrix density. Thus, it is not obvious where to prune.

the performance of an $r{\times}c$ blocked version of a matrix A. We estimate the unknown function $P_A(r,c)$ as follows.

1. Determine the performance $P_{\text{dense}}(r,c)$ (in Mflop/s) of an $r{\times}c$ register blocked sparse matrix-vector multiply (SMVM) on a dense matrix stored in sparse format. Examples of $P_{\text{dense}}(r,c)$ are shown in Figure 2 and discussed in Section 3. This performance is independent of any A, so this step need be prformed only once per platform. We refer to the set $P = \{\forall r, c : P_{\text{dense}}(r,c)\}$ as the *register profile*.[9]

2. When the specific matrix A is known, compute an estimate $\hat{f}_A(r,c)$ of the fill ratio. The idea is that estimating the fill ratio should be cheaper than calculating it exactly, or reorganizing A into $r{\times}c$ blocks.

3. Choose $r{\times}c$ that maximizes

$$\hat{P}_A(r,c) = \frac{P_{\text{dense}}(r,c)}{\hat{f}_A(r,c)} \tag{6}$$

   The assumption is that $\hat{P}_A(r,c) \approx P_A(r,c)$.

Equation (6) is the SPARSITY performance model, and the three steps together comprise a heuristic register block size selection procedure.

Current Sparsity heuristic

To keep the cost of the fill estimation step low, the current SPARSITY system does not compute $\hat{f}_A(r,c)$ for all r,c. Instead, it first computes $\hat{f}_A(1,c)$ for all c by computing the fill for some fraction of the rows.[10] Then, since the matrix is stored internally in CSR format, SPARSITY converts the matrix into CSC format and computes $\hat{f}_A(r,1)$ on a fraction of the columns. Finally, SPARSITY chooses the row block size by maximizing

$$\frac{P_{\text{dense}}(r,r)}{\hat{f}_A(r,1)}. \tag{7}$$

The system maximizes the analogous ratio for the columns, substituting c for r. Note that this scheme uses only the diagonal entries of the profile to select the block sizes.

The advantage of this scheme is that $\hat{f}_A(r,1)$ and $\hat{f}_A(1,c)$ can be estimated accurately and efficiently. However, for certain register profiles—notably, on the Itanium, as shown in Figure 2 (*bottom-right*)—this scheme is problematic since it assumes diagonals characterize off-diagonal performance. We refer to this scheme as the *current* SPARSITY *heuristic*.

[9] Though we have used a dense matrix, in principle, any matrix could be used to characterize performance for classes or families of matrices.

[10] Currently, SPARSITY samples every 100th row, i.e., 1% of all matrix rows.

A new heuristic

A natural improvement is to estimate $\hat{f}_A(r, c)$ directly for all r, c. We can perform this estimate by scanning a fraction of block rows and counting the number of non-zero blocks that will be created. (Due to space constraints, we omit a more detailed discussion of implementation.) We refer to this scheme as the *new heuristic*.

5.2 Idealized static cache miss models

The current and new SPARSITY heuristics rely on a partial one-time, off-line computation and a run-time estimation. It is interesting to ask whether a purely static model select the optimal (or near-optimal) block size more often. Though we cannot answer this question definitively (for instance, by showing no such model could exist), we use the following two idealized static models as an estimate of the limits of static modeling.

Static Cache-Miss Model

The execution time model given by equation (5), which we use as a bound in Section 4, can also be viewed as a *nearly* static model for selecting a register block size, assuming we can accurate count cache misses via equations (3)–(4), or more sophisticated variants [26, 13, 8]. The model is not truly static because accurate estimates of the miss counts will in turn rely on accurate fill estimation, as with the SPARSITY heuristics, in general requiring access to the matrix which may not be available until at run-time.

Nevertheless, for evaluation purposes in this paper, we idealize the execution time model and *assume* we know the fill ratio exactly.

Data-based Miss Model

Instead of using estimates of cache misses, we assume an oracle can provide perfect predictions at compile-time. To instantiate this model for evaluation on a given matrix, we use post-mortem miss data collected for each matrix and all block sizes using the PAPI hardware counters.

We emphasize that we cannot rule out the existence of better, truly static models, but for evaluation purposes, the two models above are reasonable estimates of the limits of static modeling.

5.3 Evaluation

We evaluate the heuristics along two dimensions: accuracy (quality of the selected implementation) and time to evaluate the heuristic. (Due to space limitations, we show representative results for the Ultra 2i and Itanium platforms. Results for the Pentium III are similar to those of the Ultra 2i, and results for the Power3 are similar to those on Itanium.)

Accuracy

Figure 7 summarizes the quality of the implementation chosen by each heuristic. Comparing the current and new SPARSITY heuristics, the new heuristic makes fewer and less severe mispredictions. On the Ultra 2i, the new heuristic correctly selects the exhaustively best implementation on matrix #15; the current heuristic chose an implementation that achieved 82% of the best performance. On the Itanium, in 3 of the 22 cases, the current heuristic chose an implementation achieving only about 75% of the best performance (#6, #7, #9) which were not mispredicted by the new heuristic.

Comparing the new heuristic to the idealized static models, we see that the models, independent of whether they use analytic or true miss counts, make poor selections (20% or more away from optimal) in 6 out of the 22 cases shown. The execution time model appears to work well on the Ultra 2i but fares poorly on the Itanium. Clearly, this observation does not rule out the existence of better execution time models, but does suggest the level of modeling sophistication that will be required to make accurate selections.

Time to select

Although the new heuristic appears more accurate and robust to the machine to the machine profile than the current heuristic, the trade-off is the time to make more accurate fill estimates. We show a preliminary evaluation of cost on the Itanium in Figure 8. The new heuristic is about 2–5 times the cost of the current heuristic, and on the order of one matrix reorganization. The cost of the new heuristic is comparable to the cost of one matrix reorganization. Thus, if the optimal block size turns out to be 1×1, then we must pay the price of the analysis.

These costs are somewhat preliminary because the settings in the implementation of the new heuristic estimate the fill ratio to within 5–10%. This bound could be much looser when the fill appears to be large. We intend to report on these costs in more detail in the final paper, but the preliminary results serve as a useful upper-bound.

6 Conclusions and Future Directions

The high-level themes of this paper are (1) the development and use of realistic bounds to guide evaluation, and (2) the use of hybrid off-line/run-time techniques to address the apparent limitations of static modeling.

As an example of the first theme, our performance upper-bound led us to the conclusion that more effort spent on low-level scheduling of the inner-most loops of the register blocked code might yield additional improvements, but the expected pay-off appears small. This suggests that we pursue opportunities for tuning to exploit reuse in other kernels. An example of our recent work in this direction is the application of register blocking in sparse matrix-dense matrix multiply (SpM$\times$M) [21, 8, 15]. This kernel can be exploited in iterative solvers

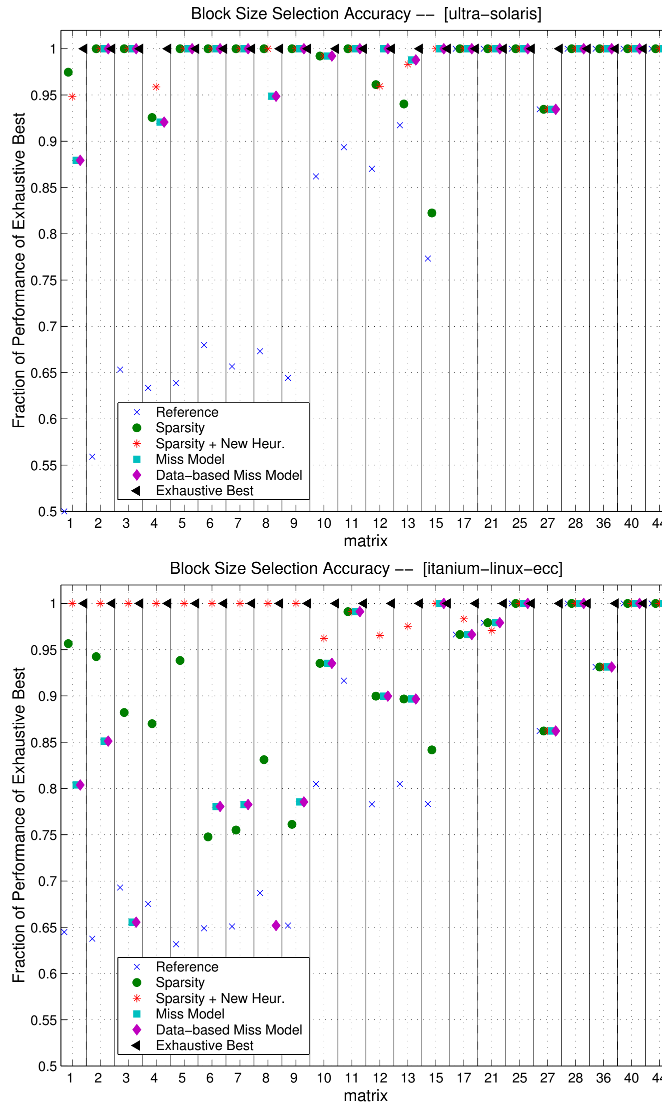

Figure 7: Accuracy of block size selection on the Ultra 2i (*top*) and Itanium (*bottom*): We evaluated the four block size selection techniques—a static miss model, SPARSITY's hybrid off-line/on-line selection heuristic, a new variant of the SPARSITY heuristic, and exhaustive search—on the matrix benchmark suite. The performance of the selected implementation is shown as a fraction of the performance of the best implementation found by exhaustive search. The 1×1 implementation is shown for reference. On the Ultra, all of the heuristics get within approximately 10% of the best most of the time. Matrix #15 (`vavasis3`) is particularly troublesome for the SPARSITY heuristic, resulting in a penalty of 15–20% on both platforms. On the Itanium, both of the static models make more selection errors than the new heuristic. Evidently, the exact miss counts are not enough to characterize the processor performance.

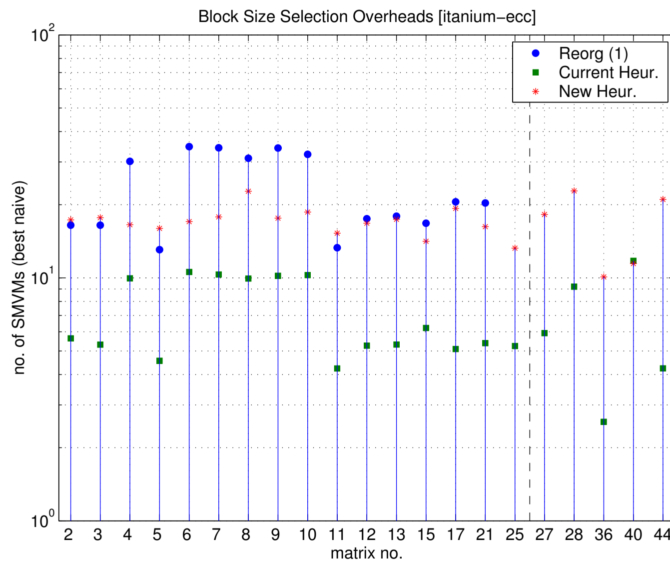

Figure 8: The costs of block size selection. We compare the time to perform *one* matrix reorganization, and the time to evaluate both the current and proposed SPARSITY selection heuristics. Note that the selection time for the new heuristic includes the time to perform all register block sizes estimates up to 12×12. Also note the logarithmic scale on the y-axis. Missing reorganization points indicate that the optimal implementation was unblocked.

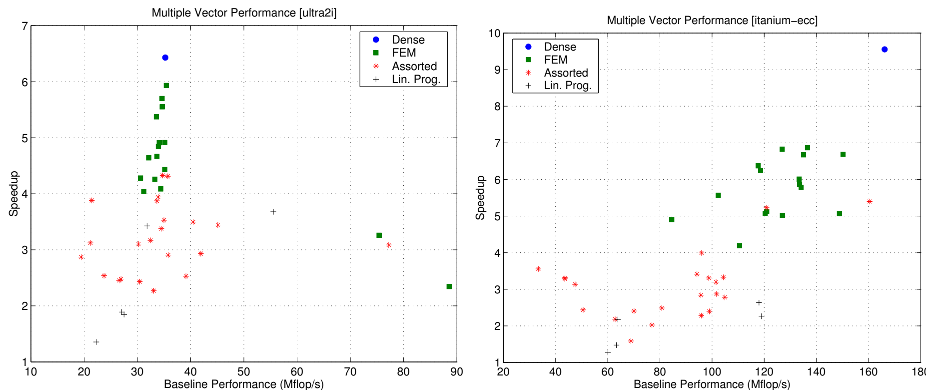

Figure 9: Speedup of the multiple vector optimization over the single vector implementations on two platforms: Sun Ultra 2i (*left*) and Intel Itanium (*right*). Each point corresponds to a matrix whose reference implementation performance is given by the x-axis, and whose multiple vector speedup is shown on the y-axis. Speedups of up to 6.5 and 9 are achievable on the Ultra and Itanium platforms, respectively.

with multiple right-hand sides and also in block eigensolvers. On the Itanium and Ultra 2i, as shown in Figure 9, we have observed speedups of up to 6.5 and 9 times that of SpM×V with a single right-hand side. Exploiting symmetry, both numerical and structural, and higher-level kernels such as multiplication of dense vectors by sparse matrix powers A^k or $A^T A$, all provide ample directions for future work.

On the theme of models, observe that with each additional kernel will come new tuning spaces and the need to select from among many possible tuning transformations. For register blocked SpM×V, we evaluated a heuristic for selecting register blocking sizes which is robust to platform and matrix-specific features. In this case, the heuristic is really a model which combines both off-line computation (register profiles) and run-time computation (fill estimation). We expect techniques of this sort to play a central role in transformation selection.

References

[1] 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.

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

[3] 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`.

[4] 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.

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

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

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

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

[9] 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.

[10] W. D. Gropp, D. K. Kasushik, D. E. Keyes, and B. F. Smith. Towards realistic bounds for implicit CFD codes. In *Proceedings of Parallel Computational Fluid Dynamics*, pages 241–248, 1999.

[11] G. Heber, A. J. Dolgert, M. Alt, K. A. Mazurkiewicz, and L. Stringer. Fracture mechanics on the intel itanium architecture: A case study. In *Workshop on EPIC Architectures and Compiler Technology (ACM MICRO 34)*, Austin, TX, December 2001.

[12] G. M. Henry. Flexible, high-performance matrix multiply via a self-modifying runtime code. Technical Report TR-2001-46, University of Texas at Austin, December 2001.

[13] 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.

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

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

[16] Intel. Intel itanium processor reference manual for software optimization, November 2001.

[17] P. Knijnenburg, T. Kisuki, K. Gallivan, and M. O. Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. In *3rd ACM Workshop on Feedback-Directed Dynamic Optimization*, December 2000.

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

[19] 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.

[20] N. Mitchell, L. Carter, and J. Ferrante. A modal model of memory. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 81–96. Springer, May 2001.

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

[22] A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing*, 1999.

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

[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] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 137–146. Springer, May 2001.

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

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

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

A Register Blocking 2×2 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 *row_start,
                const int *col_idx, const double *value,
                const double *x, double *y )
   {
       int i, jj;

       /* loop over block rows */
1      for( i = 0; i < bm; i++, y += 2 )
       {
2          register double d0 = y[0];
3          register double d1 = y[1];
4          for( jj = row_start[i]; jj < row_start[i+1];
                  jj++, col_idx++, value += 2*2 )
           {
5              d0 += value[0] * x[*col_idx+0];
6              d1 += value[2] * x[*col_idx+0];
7              d0 += value[1] * x[*col_idx+1];
```

```
 8             d1 += value[3] * x[*col_idx+1];
          }
 9        y[0] = d0;
10        y[1] = d1;
      }
   }
```