Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply*†

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

We consider performance tuning, by code and data structure reorganization, of sparse matrix-vector multiply (SpM×V), one of the most important computational kernels in scientific applications. This paper addresses the fundamental questions of what limits exist on such performance tuning, and how closely tuned code approaches these limits.

Specifically, we develop upper and lower bounds on the performance (Mflop/s) of SpM×V when tuned using our previously proposed register blocking optimization. These bounds are based on the non-zero pattern in the matrix and the cost of basic memory operations, such as cache hits and misses. We evaluate our tuned implementations with respect to these bounds using hardware counter data on 4 different platforms and on a test set of 44 sparse matrices. We find that we can often get within 20% of the upper bound, particularly on a class of matrices from finite element modeling (FEM) problems; on non-FEM matrices, performance improvements of 2× are still possible. Lastly, we present a new heuristic that selects optimal or near-optimal register block sizes (the key tuning parameters) more accurately than our previous heuristic. Using the new heuristic, we show improvements in SpM×V performance (Mflop/s) by as much as 2.5× over an untuned implementation.

Collectively, our results suggest that future performance improvements, beyond those that we have already demonstrated for SpM×V, will come from two sources: (1) consideration of higher-level matrix structures (*e.g.*, exploiting symmetry, matrix reordering, multiple register block sizes), and (2) optimizing kernels with more opportunity for data reuse (*e.g.*, sparse matrix-multiple vector multiply, multiplication of A^TA by a vector).

*This research was supported in part by LLNL Memorandum Agreement No. B504962 under the Department of Energy Contract No. W-7405-ENG-48 and DOE Grant No. DE-FG03-94ER25219, the National Science Foundation under Grant No. ASC-9813362, NSF Cooperative Agreement No. ACI-9619020, and NSF Infrastructure Grant No. EIA-9802069, and by a gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

We consider the problem of building high-performance implementations of sparse matrix-vector multiply (SpM×V), or $y = y + A \cdot x$, which is an important and ubiquitous computational kernel. We call x the *source vector* and y the *destination vector*. Making SpM×V fast is complicated both by modern hardware architectures and by the overhead of manipulating sparse data structures. It is not unusual to see SpM×V run at under 10% of the peak floating point performance of a single processor.

In hardware, the oft-cited performance gap between processor and memory drives the need to exploit locality and the memory hierarchy. Designing locality-aware data structures and algorithms can be a daunting and time-consuming task, because the best implementation will vary from processor to processor, from compiler to compiler, and from matrix to matrix. This need to have a different data structure for each sparse matrix is a major distinction from the problem of tuning dense matrix kernels (dense BLAS), since the information about matrix structure is not available until run-time. Our approach here, as with prior work in tuning the dense BLAS [3, 31], is to (1) generate a set of candidate algorithms, any one of which might well be the best, and (2) search that set by a combination of running the algorithms and performance modeling. Since the set of candidate algorithms is not known until the matrix structure is given at run-time, we must be careful not to spend too much time either generating the set of candidate algorithms or searching. By contrast, all the algorithm generation and search can be done off-line for the dense BLAS.

In prior work on the Sparsity system (Version 1.0), [17, 16], Im developed an algorithm generator and search strategy for SpM×V that was quite effective in practice. The Sparsity generators employed a variety of performance optimization techniques, including *register blocking*, *cache blocking*, and multiplication by *multiple vectors*. In this paper, we focus on register blocking (Section 3) and ask the fundamental questions of what limits exist on such performance tuning, and how close tuned code gets to these limits (Section 4).

First, we develop upper and lower bounds on the execution rate (in Mflop/s) of SpM×V, based on the nonzero pattern in the matrix and the cost of basic memory operations, such as cache hits and misses. The two bounds differ only in their assumption about whether conflict misses occur: in the upper bound any value that has been used before is modeled as a cache hit (no conflict misses), whereas the lower bound assumes that all data must be reloaded. We then use detailed hardware counter data collected on 4 different computing platforms (Table 1) over a test set of 44 sparse matrices (Table 2) to show that our upper bound is in fact a quite accurate approximation of reality, *i.e.*, that conflict misses are rare. We then show the following further results:

- We present a new register block selection heuristic, which significantly outperforms our previous heuristics on machines that favor non-square block sizes. Our new heuristic is part of Sparsity Version 2. The new heuristic can improve SpM×V by as much as 2.5× over an unblocked

implementation, and takes a modest amount of time to execute. Tuning works best on matrices with a natural block structure, such as matrices from finite element modeling (FEM).

- For FEM matrices on Itanium, Pentium III, and Ultra 2i platforms, our performance tuning heuristic is usually within 20% of the performance upper bound, indicating that further compiler-level tuning effort is not warranted for this class of matrices. This is especially surprising on the Itanium, which has the worst performance of the four machines relative to hardware peak. For non-FEM matrices performance improvements of $2\times$ or more are still possible.
- For the Power3 architecture, tuning does not significantly improve performance, which is farther below the upper bound than on the other 3 platforms. Our analyses suggest that additional effort toward low-level tuning (*e.g.*, instruction scheduling) could be beneficial.

Taken together, these results show that while performance tuning of SpM×V is beneficial, any additional improvements will likely come from considering higher level matrix structures (*e.g.*, exploiting symmetry, reordering the matrix, or cache blocking) and from optimizing kernels with more opportunity for data reuse (*e.g.*, multiplying A by multiple vectors, multiplying a vector by A^TA).

In related work on dense matrices, cache and memory behavior have been well-studied. 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 [8, 11, 22, 7, 32]. However, it is difficult to apply these analyses directly to sparse matrix kernels due to the presence of indirect and irregular memory access patterns.

Despite the difficulty of analysis in the sparse case, there have been a number of notable attempts. Temam and Jalby [29], Heras, *et al.* [15], and Fraguela, *et al.* [10] have developed sophisticated probabilistic cache miss models, but assume uniform distribution of non-zero entries. These models are primarily distinguished from one another by their ability to account for self- and cross-interference misses. In this study, we will see that on current and future machines, whose cache sizes continue to grow, conflict misses are less important to accurate miss modeling (Section 4).

Gropp, *et al.*, use bounds similar to the ones we develop to analyze and tune a computational fluid dynamics code [12]; Heber, *et al.*, present a detailed performance study of a fracture mechanics code on Itanium [13]. However, we are interested in tuning for matrices that come from a variety of domains. Furthermore, we explicitly model execution time (instead of just modeling misses) in order to evaluate the extent to which our tuned implementations achieve optimal performance.

Finally, we mention examples of work in the sparse compiler literature by Bik [2], Pugh and Shpeisman [25], and the Bernoulli compiler [28]. The first analyzes matrices for high-level structure using techniques complementary those

that we consider; the latter two consider sparse code specification and generation issues, but do not specialize for specific matrix structures. The tools and technology developed in these research projects could serve as the code generation infrastructure of an automatic tuning system such as SPARSITY. We distinguish our work by the use of a hybrid off-line, on-line model for selecting transformations (register blocking sizes, as described in Section 3).

2 Experimental Setup

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 published sources and confirmed experimentally using the memory system microbenchmark due to Saavedra-Barrera [27].

Matrices

We evaluate the SpM×V implementations on the matrix benchmark suite used by Im [16]. Table 2 summarizes the size and source of each matrix. Most of the matrices are available from either of the collections at NIST (MatrixMarket [5]) and the University of Florida [9].

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 (including chemical process engineering, oil reservoir modeling, circuits, and finance); 40–44 are linear programming examples.

Timing

We use the PAPI v2.1 library for access to hardware counters on all platforms [6]; 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 L2 cache on the Power3) is large enough to contain some of the matrices. To avoid inflated findings, for each 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), arithmetic with these extra zeros are *not* counted as flops when determining performance.

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

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	664 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 ($n = 2000$)	345 Mflop/s
STREAM Triad Bandwidth [21]	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	16 KB
Minimum memory latency ($\approx$)	36 cy	26 cy	35 cy	36 cy
Maximum memory latency ($\approx$)	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: **Evaluation platforms**. We list the basic configuration data for the machines and compilers 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 [31]; on the Pentium III (Katmai) platform: figures reported are the best of Intel's MKL v5.2, ATLAS 3.3.5 [31], and ITXGEMM 1.1 [14]; on the Power3 platform: IBM ESSL 3.1.2; on the Itanium platform: Intel MKL v5.2.

	Name	Application Area	Dimension	Nonzeros
1	dense1000	Dense Matrix	1000	1000000
2	raefsky3	Fluid structure interaction	21200	1488768
3	olafu	Accuracy problem	16146	1015156
4	bcsstk35	Stiff matrix automobile frame	30237	1450163
5	venkat01	Flow simulation	62424	1717792
6	crystk02	FEM Crystal free vibration	13965	968583
7	crystk03	FEM Crystal free vibration	24696	1751178
8	nasasrb	Shuttle rocket booster	54870	2677324
9	3dtube	3-D pressure tube	45330	3213332
10	ct20stif	CT20 Engine block	52329	2698463
11	bai	Airfoil eigenvalue calculation	23560	484256
12	raefsky4	buckling problem	19779	1328611
13	ex11	3D steady flow caculation	16614	1096948
14	rdist1	Chemical process separation	4134	94408
15	vavasis3	2D PDE problem	41092	1683902
16	orani678	Economic modeling	2529	90185
17	rim	FEM fluid mechanics problem	22560	1014951
18	memplus	Circuit Simulation	17758	126150
19	gemat11	Power flow	4929	33185
20	lhr10	Light hydrocarbon recovery	10672	232633
21	goodwin	Fluid mechanics problem	7320	324784
22	bayer02	Chemical process simulation	13935	63679
23	bayer10	Chemical process simulation	13436	94926
24	coater2	Simulation of coating flows	9540	207308
25	finan512	Financial portfolio optimization	74752	596992
26	onetone2	Harmonic balance method	36057	227628
27	pwt	Structural engineering problem	36519	326107
28	vibrobox	Structure of vibroacoustic problem	12328	342828
29	wang4	Semiconductor device simulation	26068	177196
30	lnsp3937	Fluid flow modeling	3937	25407
31	lns3937	Fluid flow modeling	3937	25407
32	sherman5	Oil reservoir modeling	3312	20793
33	sherman3	Oil reservoir modeling	5005	20033
34	orsreg1	Oil reservoir simulation	2205	14133
35	saylr4	Oil reservoir modeling	3564	22316
36	shyy161	Viscous flow calculation	76480	329762
37	wang3	Semiconductor device simulation	26064	177168
38	mcfe	astrophysics	765	24382
39	jpwh991	Circuit physics modeling	991	6027
40	gupta1	Linear programming matrix	31802	2164210
41	lpcreb	Linear Programming problem	9648×77137	260785
42	lpcred	Linear Programming problem	8926×73948	246614
43	lpfit2p	Linear Programming problem	3000×13525	50284
44	lpnug20	Linear Programming problem	15240×72600	304800

Table 2: **Matrix benchmark suite**. 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.

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 [17]. Register blocking, as we describe below, is designed to exploit naturally occuring dense blocks by reorganizing the matrix data structure into a sequence of small (enough to fit in register) dense blocks. We close this section with a description of our new heuristic for selecting the register block size, which overcomes a short-coming of the previously published SPARSITY heuristic [17]. As presented in this section, the SPARSITY system including the new heuristic is SPARSITY version 2.0.

For concreteness, we assume a baseline that stores the matrix in compressed sparse row (CSR) format.[2]

3.1 The register blocking optimization

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 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 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. 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). 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. An example of a 2×2 implementation appears in Appendix A.

However, Figure 1 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 on

[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.

[4] The performance of this code is comparable to that of the CSR implementation from the NIST Sparse BLAS [26].

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

`b_row_start` = (0 2 4)
`b_col_idx` = (0 4 2 4)
`b_value` =
(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})

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 [16].)

which matrices to apply it and how to select the block size. (We assume a single block size of $r \times c$ is suitable for the whole matrix.)

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 $n \times n$ matrix stored in sparse (BCSR) format.[5] Performance is a strong function of the architecture, compiler, and block size. Figure 2 is also an estimate of the potential performance gains from performance tuning: maximum speedups range from 1.5x on the Itanium and the Power3 to 2.5x on the Pentium III.

Furthermore, the irregularity of the spaces in Figure 2 suggests that performance will in general be difficult to model. Nevertheless, the profiles shown clearly contain a lot of information, which we exploit in our heuristic for selecting a block size (below).

3.2 Selecting the $r \times c$ register block size

The best block size $r \times c$ depends both on the machine and the matrix. In general, the best block size may not be square. For instance, even if the matrix is naturally expressed in 6×6 blocks, 3×1 blocks may be considerably faster, as suggested by the Itanium register profile shown in Figure 2.

We assume that in the general case, we do not know the matrix until runtime.[6] Depending on the application, the cost of exhaustively searching for the

[5] Note that for the performance profiles shown, the matrix size is actually $\lceil \frac{n}{r} \rceil r \times \lceil \frac{n}{c} \rceil c$. On the Power3, which has a very large 8 MB L2 cache, $n = 2000$; on the other platforms, $n = 1000$. For the block sizes considered, the true matrix size differs from the $n \times n$ case by no more than 2%.

[6] This mode of usage is not unreasonable; in fact, it is the implied mode of operation in the new Sparse BLAS standard [4].

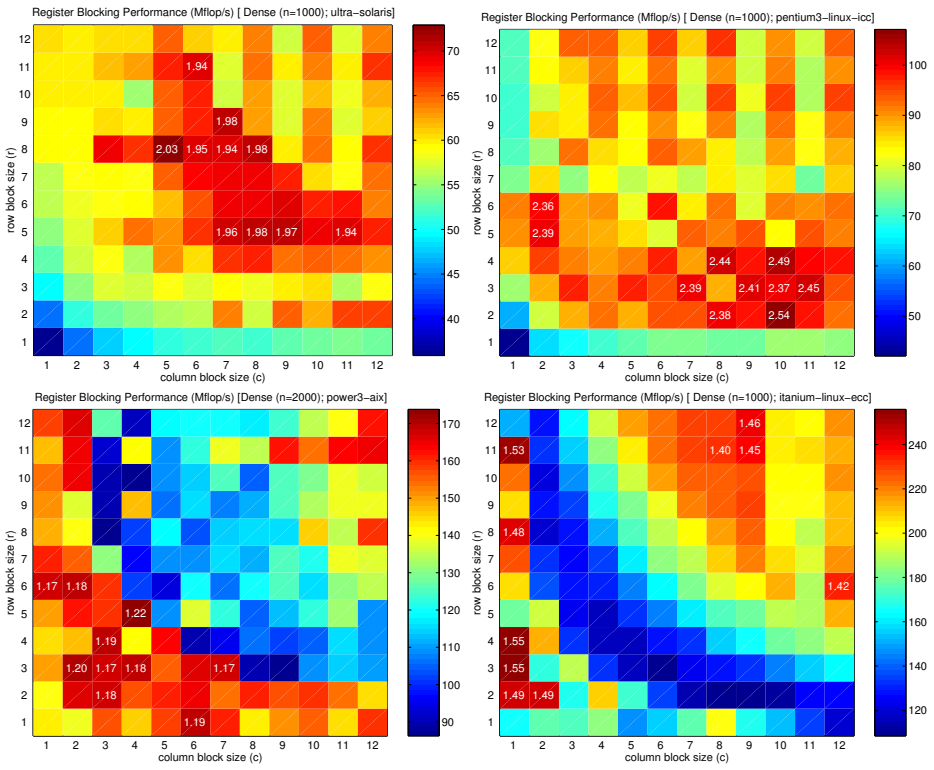

Figure 2: **Register profiles**. The performance (Mflop/s) of $r\times c$ register blocked implementations on a dense $n\times n$ matrix stored in BCSR format, on block sizes up to 12×12. Results on four platforms are shown, clockwise from the upper-left: Sun Ultra 2i, Intel Pentium III, Intel Itanium, and IBM Power3. On the Power3, $n = 2000$; otherwise, $n = 1000$. 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; they range from a 1.2x speedup (Power 3) to 2.5x (Pentium III). 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."

optimal block size could be prohibitive: on Itanium, we have observed that the cost of reorganizing the matrix *once* (*i.e.*, for one value of $r\times c$) is 10–30 times the cost of running the reference SpM×V once.

SPARSITY uses the register profiles shown in Figure 2 as an estimate of the performance of the $r\times c$ implementations assuming no fill. When the matrix is known, SPARSITY randomly samples a fraction of the matrix rows to compute an estimate of the fill for all $r\times c$ (see below). Then, SPARSITY chooses the block size by maximizing the following ratio over all r, c:

$$\frac{\text{Dense performance at } r\times c}{\text{Fill ratio estimate at } r\times c} \quad . \tag{1}$$

Since the register profile does not depend on any particular sparse matrix, it is collected off-line only once per machine.

We now describe the new SPARSITY version 2.0 heuristic for selecting r and c, and contrast it briefly with the previous heuristic. To estimate the fill, for each r we select 1% of the block rows uniformly at random and count the number of zeros which would be filled in for all c simultaneously. Currently, we limit our estimate to sizes up to 12×12, though on the matrix benchmark suite we have not observed optimal sizes greater than 8×8. Also, we perform the 1% scan independently for each r, though this could obviously be improved by simultaneously scanning r and its factors (*e.g.*, while scanning $r = 12$, simultaneously search $r = 1$, 2, 3, 6, and 12). As implemented, we scan up to 12% of the matrix. The cost of this procedure for all r, c is usually less than the cost of one reorganization; even so, we are able to estimate the fill ratio to within 1% on FEM matrices, and to within 5–10% on average on the other matrices in our benchmark suite.[7]

The previously published SPARSITY heuristic [17] estimated the fill ratio not for all $r\times c$ sizes, but only for $r\times 1$ and $1\times c$ block sizes. Then, r and c were selected independently by maximizing separately the following two ratios:

$$\frac{\text{Dense performance at } r\times r}{\text{Fill ratio estimate at } r\times 1} \qquad \frac{\text{Dense performance at } c\times c}{\text{Fill ratio estimate at } 1\times c} \tag{2}$$

For most matrices, the previous heuristic tended to selected square block sizes, which led to performance that was as much as 30–40% below that of the best block sizes on the Itanium platform.

4 Bounds on Register Blocking Performance

Below, we develop upper and lower bounds on performance to understand and evaluate SPARSITY's register blocking optimization with the new heuristic.

[7] This level of accuracy is probably more than adequate, though a detailed study is currently the subject of investigation.

4.1 Preliminaries

We use 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; for 1×1 blocks, $K_{1,1} = k$. 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 of floating point and integer data is

$$\begin{aligned} \text{Loads}(r,c) &= \underbrace{kf_{rc} + \frac{kf_{rc}}{rc} + \left\lceil \frac{m}{r} \right\rceil + 1}_{\text{matrix}} + \underbrace{\frac{kf_{rc}}{r}}_{\text{source vec}} + \underbrace{m}_{\text{dest vec}} \\ &= kf_{rc}\left(1 + \frac{1}{rc} + \frac{1}{r}\right) + m + \left\lceil \frac{m}{r} \right\rceil + 1 \end{aligned} \tag{3}$$

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.

4.2 Bounds based on modeling cache misses

We can estimate an 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 per cache line 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 1 compulsory miss per cache line for each of the n source vector elements. Thus, a lower bound $M_{\text{lower}}^{(1)}$ on L1 misses is

$$M_{\text{lower}}^{(1)}(r,c) = \frac{1}{l_1}\left[kf_{rc}\left(1 + \frac{1}{\gamma rc}\right) + \frac{1}{\gamma}\left(\left\lceil \frac{m}{r} \right\rceil + 1\right) + m\right] + \frac{n}{l_1}. \tag{4}$$

where the size of one floating point value equals γ integers. In this paper, we use double-precision (64-bit) 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 the other cache levels by simply substituting the right line size.

In the worst case, we will miss on every access to a source vector element due to capacity and conflict (both self- and cross-interference) misses; thus, an upper bound on misses is[8]

$$M^{(1)}_{\text{upper}}(r,c) = \frac{1}{l_1}\left[kf_{rc}\left(1+\frac{1}{\gamma rc}\right)+\frac{1}{\gamma}\left(\left\lceil\frac{m}{r}\right\rceil+1\right)+m\right]+\frac{kf_{rc}}{r}. \quad (5)$$

We model execution time as follows. First, since we want an upper bound on performance (lower bound on time), we assume 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. Then the execution time T is

$$T = \sum_{i=1}^{\kappa-1} h_i\alpha_i + m_\kappa\alpha_{\text{mem}}, \quad (6)$$

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} is the memory access time. The L1 hits h_1 are given by $h_1 = \text{Loads}(r,c) - m_1$. 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 $i \geq 1$. The performance in Mflop/s is $\frac{2k}{T}\cdot 10^{-6}$.

To get an estimate of the *upper bound on performance*, let $m_i = M^{(i)}_{\text{lower}}$ in Equation (6), and convert to Mflop/s. Similarly, we can get a lower bound on performance by letting $m_i = M^{(i)}_{\text{upper}}$.

Interaction with the TLB complicates our estimate of α_{mem}. We incorporate the TLB into our performance upper bound by letting α_{mem} be the *minimum* memory access latency shown in Table 1. This latency assumes a memory access but also a TLB hit. For the lower bound, we assume α_{mem} is the maximum memory access latency shown in Table 1, which corresponds to a memory access and a TLB miss.

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 both 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 [18].

Finally, note that our model of execution time, Equation (6), charges the full latency for each memory access. We address this assumption with respect to main memory bandwidth in Section 5.

[8] Equation (5) 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 nonzero pattern to identify when spatial locality is present, but do not do so here for simplicity.

4.3 Validating the cache miss bounds

We collected data over all register block sizes up to 12×12 for all matrices and platforms, measuring execution time and cache misses using PAPI. We validate our cache miss bounds in Figures 3–6. For each matrix, we compared the cache misses as measured by PAPI to the cache miss lower and upper bounds given by Equation (4) and Equation (5), respectively. Figures 3–6 show data for the best block size, chosen by exhaustive search. We see that the miss bounds are a good match to the true misses. 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 accurately in the off-chip caches (*i.e.*, the L2 cache on the Ultra 2i, Pentium III, and Power3 platforms, and the L3 cache on the Itanium platform).

4.4 Evaluating register blocking performance

We now evaluate the register blocking optimization with respect to the upper and lower bounds on performance derived above. Figures 7–10 summarize our evaluation on the four hardware platforms in Table 1 and the matrix benchmark suite in Table 2, with respect to the upper and lower performance bounds. We compare the following implementations:

- **Reference**: The unblocked (1×1) implementation is represented by asterisks.
- **Sparsity (heuristic)**: The implementation in which r and c are chosen by the Sparsity v2.0 heuristic, as described in Section 3.2, is represented by circles.
- **Sparsity (exhaustive)**: The implementation in which r and c are chosen by exhaustive search is represented by squares. We refer to this block size as $r_{\text{opt}} \times c_{\text{opt}}$.
- **Analytic upper and lower bounds**: The analytic upper and lower bounds on performance (Mflop/s) for the $r_{\text{opt}} \times c_{\text{opt}}$ implementation, as computed in Section 4.2, are shown as dash-dot and solid lines, respectively.
- **PAPI upper bound**: An upper bound on performance for the $r_{\text{opt}} \times c_{\text{opt}}$ implementation, obtained by substituting measured cache miss data from PAPI into Equation (6) and using the minimum memory latency for α_{mem}, is represented by triangles.

We are particularly interested in making observations on the following topics:

- *Performance of* Sparsity *vs. the reference implementation*: Figures 7–10 show that Sparsity implementations, whether chosen by our heuristic or exhaustively, achieve speedups of up to 2.5 over the reference implementation. Register blocking is particularly effective on the matrices arising in FEM applications. However, on the Power3 architecture, we observe relatively small speedups even on FEM matrices.

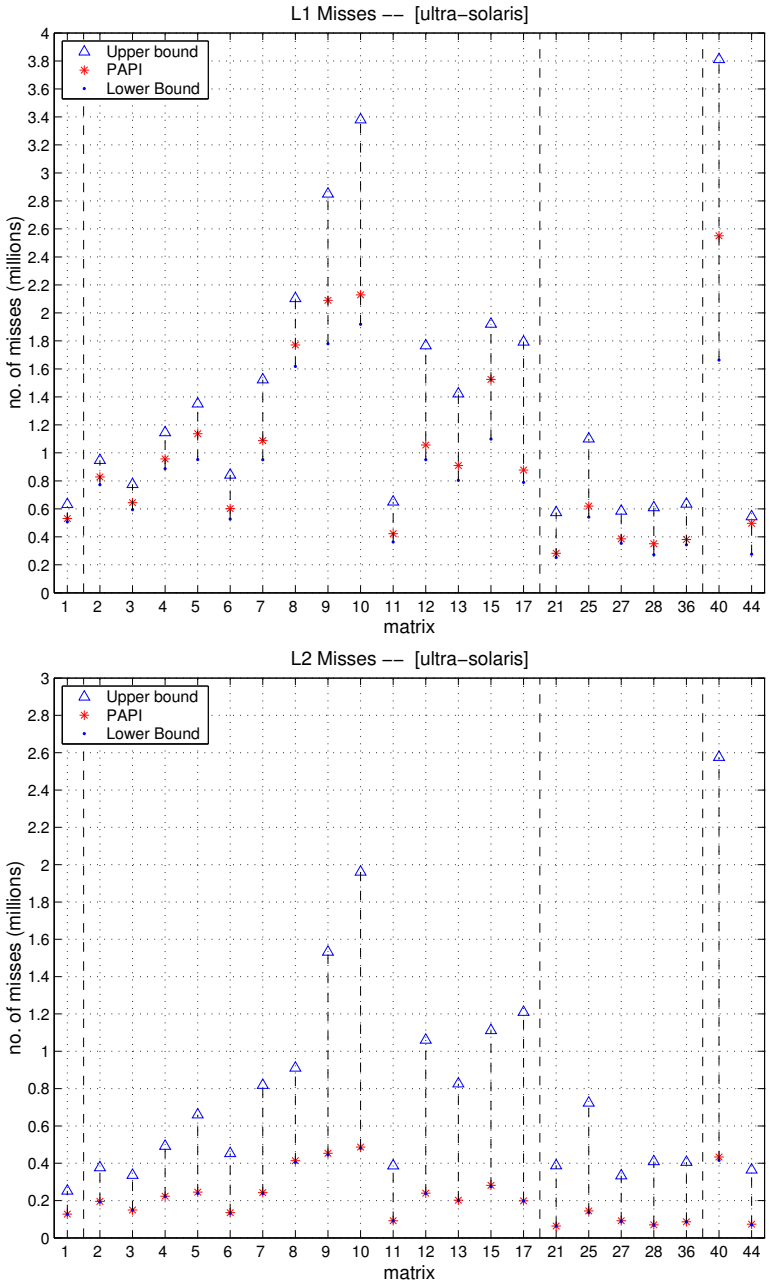

Figure 3: **Cache miss model validation, Sun Ultra 2i**. Our upper and lower bounds on L1 and L2 cache misses compared to PAPI measurements for the $r \times c$ implementation with the best performance. The bounds match the data well. The true L2 misses match the lower bound well in the larger (L2) cache, suggesting the vector sizes are small enough that conflict misses play a relatively small role.

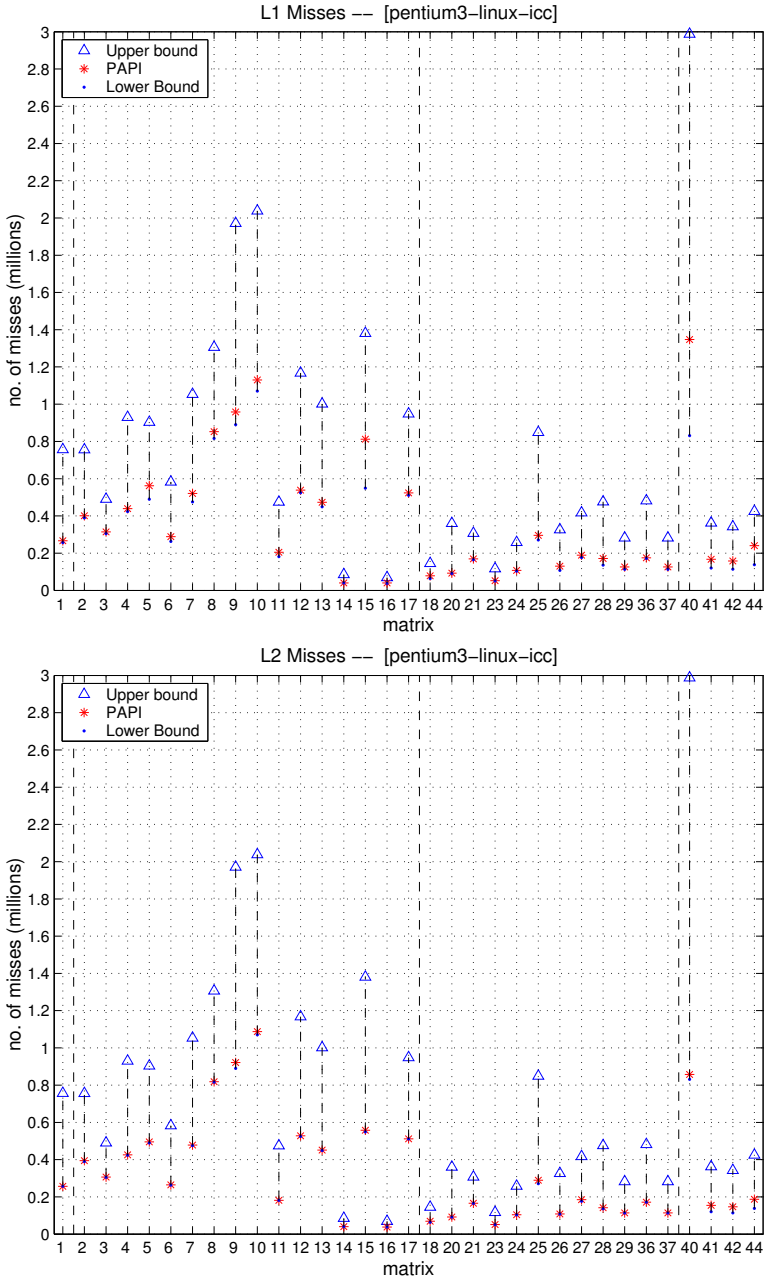

Figure 4: **Cache miss model validation, Intel Pentium III**. Our upper and lower bounds on L1 and L2 cache misses on the Intel Pentium III compared to PAPI measurements. As with Figure 3, the lower bounds are a good match in the largest (L2) cache.

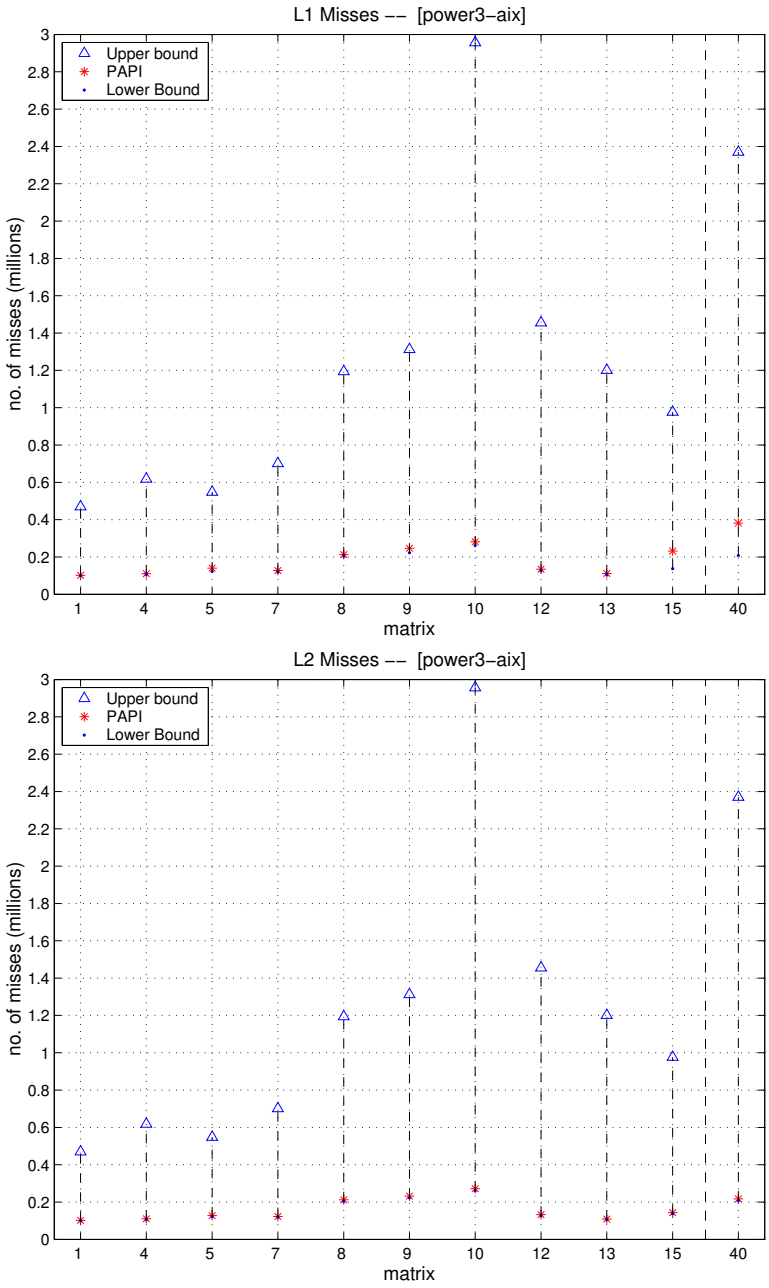

Figure 5: **Cache miss model validation, IBM Power3**. Our upper and lower bounds on L1 and L2 cache misses on the IBM Power3 compared to PAPI measurements. As with Figure 3, the lower bounds are a good match in the largest (L2) cache.

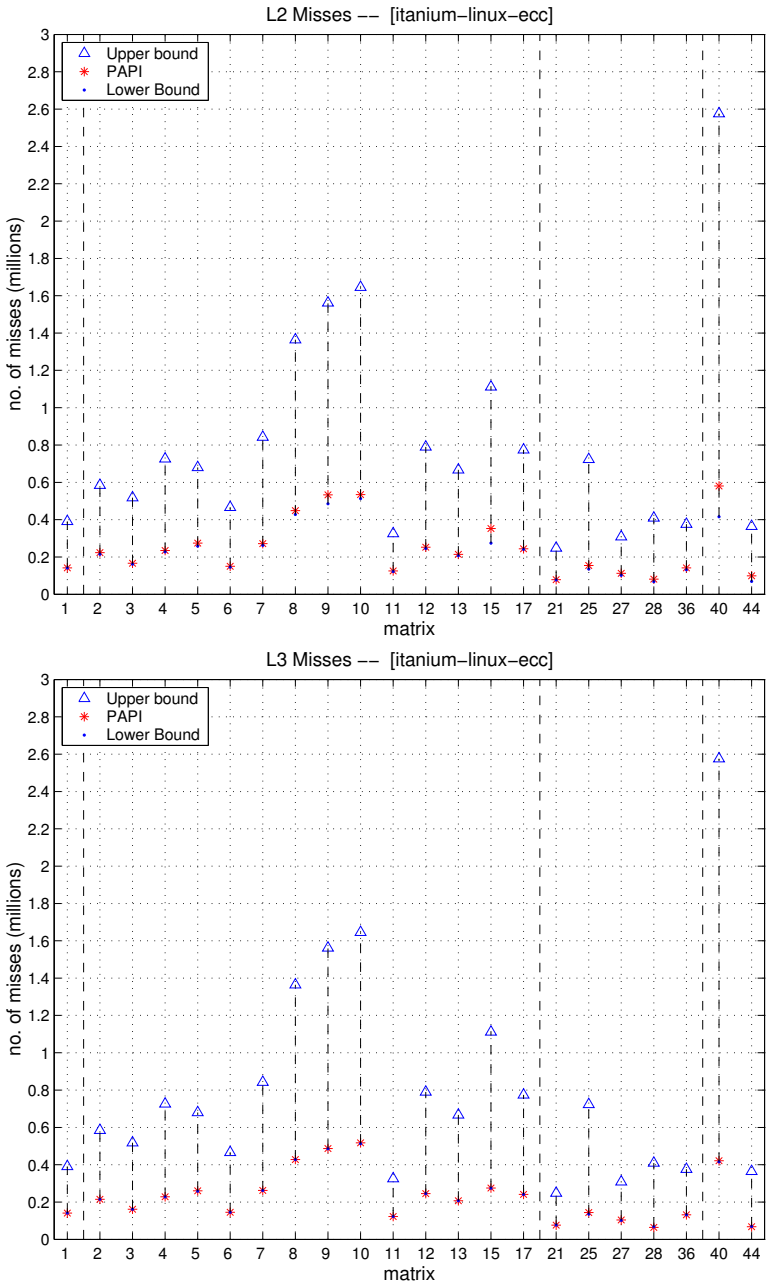

Figure 6: **Cache miss model validation, Intel Itanium**. Our upper and lower bounds on L2 and L3 cache misses on the Intel Itanium compared to PAPI measurements. As with Figure 3, the lower bounds are a good match in the largest (L3) cache.

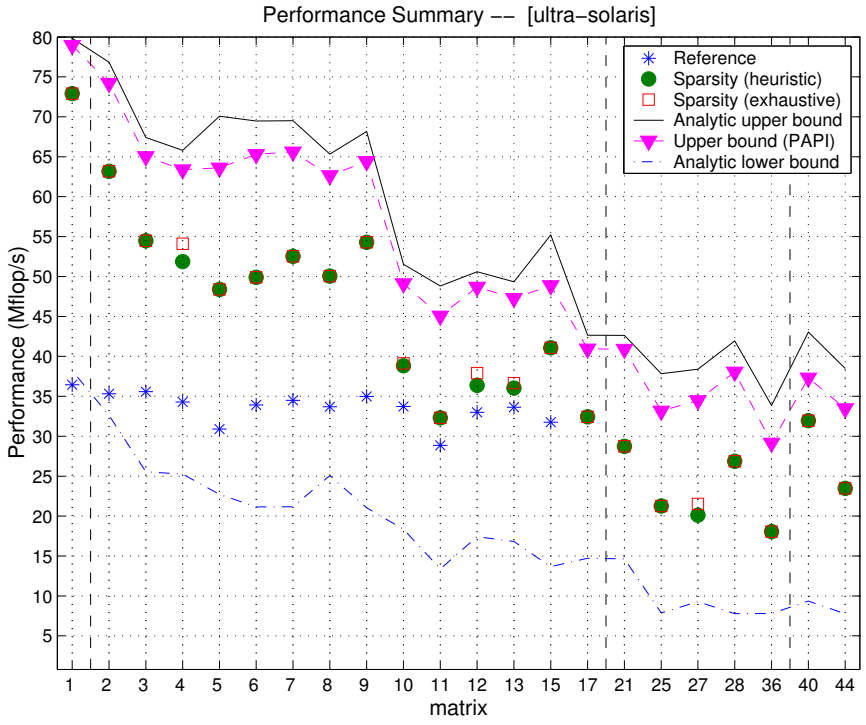

Figure 7: **Performance summary, Ultra 2i**. Performance (Mflop/s) of the implementation chosen by SPARSITY compared to the best implementation chosen by exhaustive search and our upper and lower bounds. Here, we show data for the Ultra 2i. On the FEM matrices, SPARSITY performance is 70–80% of our estimated upper bound.

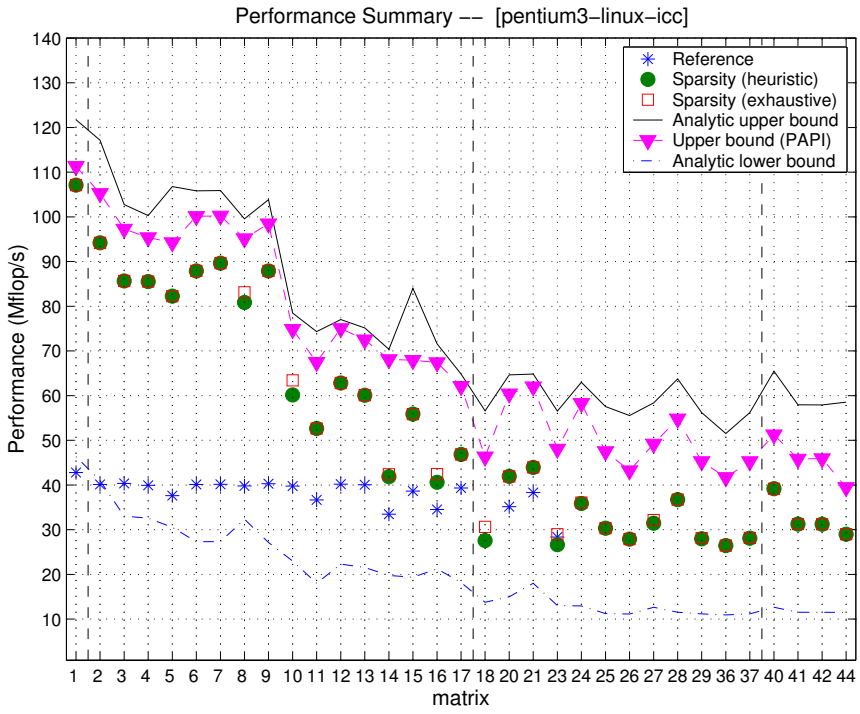

Figure 8: **Performance summary, Pentium III**. Same as Figure 7 for the Pentium III. SPARSITY implementations achieve 80% or more of the upper-bound on many of the FEM matrices.

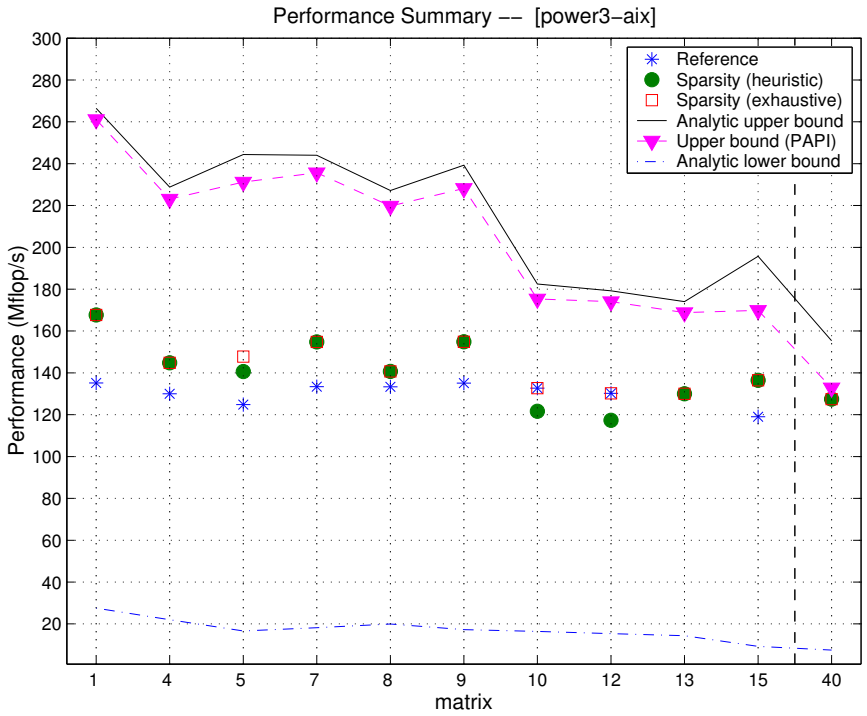

Figure 9: **Performance summary, Power3**. Same as Figure 7 for the IBM Power3. SPARSITY implementations achieve anywhere between 60–85% of our upper-bound estimate.

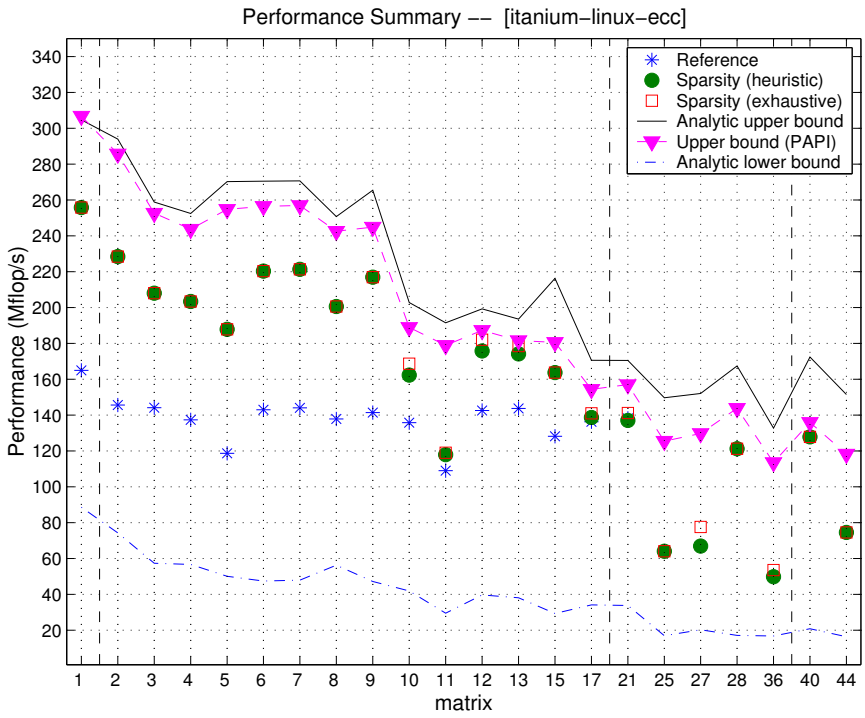

Figure 10: **Performance summary, Itanium**. Same as Figure 7 for the Intel Itanium. SPARSITY implementations achieve 80% or more of the upper-bound on most of the FEM matrices.

- *Quality of the* SPARSITY *v2.0 block size selection heuristic*: The SPARSITY v2.0 heuristic for selecting the block size generally chooses either the optimal block size, or a block size whose performance is within 5–10% of optimal. (A detailed comparison of block sizes and performance appears in Tables 4–7 of Appendix B.)
- *Proximity of* SPARSITY *performance to the upper and lower bounds*: It is valid to compare SPARSITY (exhaustive), shown as squares, and the upper and lower bounds, since they are based on the same block size, $r_{\text{opt}} \times c_{\text{opt}}$. The gap between SPARSITY and the upper bound indicates how much room is left for improvement, which in most cases is 20% or less.

Below, we elaborate on these high-level conclusions for specific matrices and platforms.

On the Ultra 2i, Pentium III, and Itanium platforms, SPARSITY implementations achieve 70–80% or more of the upper-bound on most of the matrices in the FEM set (matrices 2–17). 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.

Performance on the non-FEM matrices is more varied on the Ultra 2i, Pentium III, and Itanium platforms, though it tends to be 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 matrix reordering [30, 19, 24, 15], or the use of multiple $r \times c$ block sizes are likely to be the most effective way to address this performance issue.

On the Power3, the performance of all implementations falls between 60%–70% of the estimated upper bound, a smaller fraction than on the other machines. One factor differentiating the Power3 architecture from the others in this study is its high-performance memory system, resulting in comparable performance for all implementations.

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. We report DGEMV as a useful scale reference, not to advocate conversion from dense to sparse formats.

5 Model Justification: Latency vs. Bandwidth

The primary assumption of our execution time model, Equation (6), is that we must charge the full latency (α_{mem} or α_i) for each memory or cache access. However, the combination of high peak memory bus bandwidth and the ability in most modern processors to tolerate multiple outstanding misses could in principle allow much higher data stream rates. Indeed, if we compute an upper bound based only on the total memory transfer requirements of the algorithm at memory bus speeds, we obtain a much higher upper bound. In this section, we explore whether these bandwidth numbers can be obtained by examining several simpler benchmarks, known as the STREAM benchmarks [21, 20], and show that they are not. Our conclusion is that our model, which charges the full memory latency to fill the first element of each cache line, is a better match to actual performance than anything based on the peak memory bus bandwidth.

As an example, consider the time to load an uncached double-precision word from memory to the processor on the Ultra 2i, both from the perspective of peak memory bandwidth and our latency model. Assuming the word moves at the peak memory bus bandwidth of 664 MB/s, the time to load a double-precision word from main memory is (8 B) / (664 MB/s) $\approx$ 12 ns, or approximately 4 cycles. By contrast, if we apply our full-latency model, we find that streaming through a double-precision array costs 8.125 cycles per word;[9] the effective bandwidth in the full-latency model is (8 B) / (8.125 cy) * (333 MHz) = 327.9 MB/s, or just under half the theoretical peak. Thus, it is natural to ask whether the full-latency model is justified for streaming applications in practice, given the gap between the effective bandwidth in the full-latency model and peak memory bus bandwidth.

The standard benchmark for assessing sustainable memory bandwidth is the STREAM benchmark [21, 20]. STREAM consists of four vector kernels operating on long (out-of-cache) vector operands. We ran the STREAM benchmark on our four evaluation platforms, and wrote several additional kernels intended to mimic the access patterns characteristic of sparse matrix-vector multiply. All of the kernels are summarized in Table 3, where the standard STREAM benchmark comprises the first four kernels. For each kernel, STREAM calculates the memory bandwidth by dividing the volume of data moved by the time to execute the kernel.[10] A few of our kernels deserve additional explanation:

- **Sum** and **Dot**: For each of these kernels, we manually unrolled the loops by hand, summing to separate scalars in order to reduce the dependence in successive additions to the same scalar. We explored all unrolling depths up to 8, and report bandwidth results for the best case in this section.

[9] The cost in the full-latency model for the Ultra 2i is determined as follows. For every 8 words (or 64 B, the L2 line size), the first word requires an access to main memory, incurring $\alpha_{\text{mem}} = 36$ cy; the second word hits in the L1 cache, incurring $\alpha_1 = 2$ cycles; the final 3 pairs of words cost $\alpha_2 + \alpha_1 = 2 + 7 = 9$ cy. Thus, the total time to execute 8 double precision loads is $36 + 2 + 3 \times 9 = 65$ cy, or 8.125 cy per word.

[10] Note that STREAM does not give credit for the additional cache-line load required for store operations in write-back caches. For more details, see the STREAM home page [21].

Kernel	Vector Operation	Volume of Data Transferred (in doubles)
Copy	a[i] = b[i]	2N
Scale	a[i] = α*b[i]	2N
Add	a[i] = b[i] + c[i]	3N
Triad	a[i] = b[i] + α*c[i]	3N
Sum	s += a[i]	N
Dot	s += a[i]*b[i]	2N
Load Sparse Matrix	s += a[i], k += ind[i]	$(1 + \frac{1}{\gamma})$N
SpM×V (on-chip cache)	s += a[i]*x[ind[i]]	$(1 + \frac{1}{\gamma})$N + n1
SpM×V (external cache)	s += a[i]*x[ind[i]]	$(1 + \frac{1}{\gamma})$N + n2

Table 3: **Memory bandwidth microkernels**. The standard STREAM benchmark [21] comprises the first four kernels; for this paper, we consider 5 additional kernels that mimic selected aspects of SpM×V. The letters a, b, c, and x to denote double-precision arrays, ind to denote an array of integer indices. We use α and s to denote a double precision scalars, and i and k to denote integer scalars. All scalars are assumed to be in CPU registers during execution of the kernel. We use i as a vector index which ranges over all elements of the vector. All vectors, except x, have N elements, where N is chosen on each platform so that the double-precision vectors are at least 10 times larger than the largest cache. For SpM×V (on-chip), the length of x is chosen to be $\frac{1}{3}$ the size of the largest on-chip cache; for SpM×V (external), x is $\frac{1}{3}$ the size of the largest off-chip cache.

- **Load Sparse Matrix**: s += a[i], k += ind[i]. This kernel measures the peak rate at which the matrix—non-zero values and integer indices—can be brought through memory. Note that we model access in CSR storage, *i.e.*, one integer index per non-zero, but ignore the row pointers.
- **SpM×V (on-chip cache)**: Performs the following SpM×V-like operation: s += a[i]*x[ind[i]]. This kernel simulates the indirect access in true SpM×V. However, it contains no stores to memory, and the source vector x is sized to be $\frac{1}{3}$ the size of the largest on-chip cache (L2 on the Itanium, L1 on all other platforms) to reduce the effect of conflicts. As with the load sparse matrix kernel (above), this kernel models access in CSR storage.
- **SpM×V (external cache)**: Similar to SpM×V (on-chip), except that the size of the source vector x is $\frac{1}{3}$ the size of the largest external ("off-chip") cache (L3 on Itanium, L2 otherwise).

We summarize the results of executing all of the kernels on the four evaluation platforms in Figure 11. Specifically, we compare three bandwidths:

- **Peak bandwidth**: The y-axis measures bandwidth relative to the peak

memory bus bandwidth (hereafter, *peak bandwidth*) given in Table 1, so peak bandwidth is shown implicitly.

- **Measured bandwidth**: For each kernel, the *measured bandwidth* is computed as the volume of data shown in Table 3 (column 3) divided by the measured execution time.
- **Model bandwidth**: We refer to the bandwidth predicted by our full-latency model, Equation (6), as the *model bandwidth*. We compute the model bandwidth by applying our latency model to each kernel, computing Equation (6), and dividing the volume of data shown in Table 3 by Equation (6).

Figure 11 shows a set of bars for each platform, where each bar indicates the measured bandwidth (normalized by the peak bandwidth) of one of the kernels.

In addition to the kernels, three bars are shown for reference on each platform: the measured bandwidths of DGEMV, the 1×1 SpM×V routine for a dense matrix in CSR format, and the best $r \times c$ register blocked routine for a dense matrix in BCSR format. For these reference bars, we measured the execution time on an $n \times n$ matrix ($n = 2000$), and computed the measured bandwidth by dividing the volume of data by the measured time. The volume of data is the volume of vector data plus the volume of matrix data. For the volume of vector data, we count one load each of the source and destination vector elements plus a store of the destination vector, for a total of $3n$ doubles. The matrix data volume is n^2 double-precision words for DGEMV, and $(1 + \frac{1}{\gamma rc})n^2 + \left(\lceil \frac{m}{r} \rceil + 1\right)/\gamma$ doubles for the dense matrix in sparse format.

Finally, the model bandwidth is shown for several kernels by asterisks (*) vertically aligned with the corresponding bar. For the synthetic kernels, we use formulas for h_i and m_κ in Equation (6). For DGEMV, and the blocked and unblocked SpM×V, we use PAPI data for h_i and m_κ. Thus, the model bandwidth for the best $r \times c$ SpM×V corresponds to the PAPI upper-bounds on matrix 1 in Figures 7–10.

We make the following observations about the results of Figure 11. Note that we report bandwidth, and since the volume of data is generally different for each kernel, care should be taken when attempting to compare or infer execution time among the kernels.

- The measured bandwidth is never more than 80% of peak bandwidth, and typically under 65%, across all platforms. On the Ultra 2i, the bandwidth is always below 45% of peak.
- The model bandwidth is either an upper-bound on the measured bandwidth (all kernels on the Ultra 2i and Power3), or within 5–15% of the measured bandwidth (on the Pentium III and Itanium platforms). When the model bandwidth is below the measured bandwidth, this indicates that it may be possible to achieve some additional performance on top of that predicted by the model (Figures 7–10). However, the proximity of the model bandwidth to the measured bandwidth suggests that such gains will be small. The gap between the model and measured bandwidth

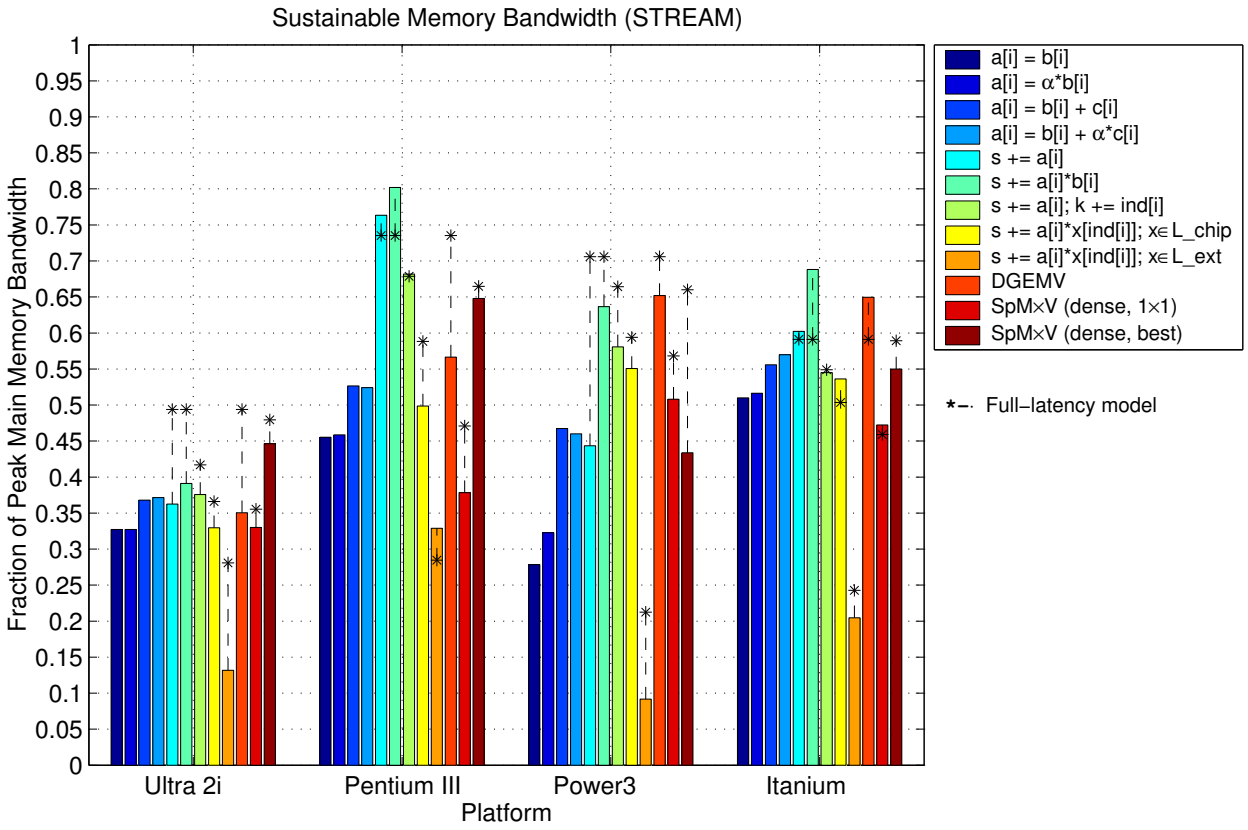

Figure 11: **Memory bandwidth benchmark results**. We show the measured main memory bandwidth (MB/s), as a fraction of the theoretical peak shown in Table 1, for each of the kernels shown in Table 3, on all four evaluation platforms. In each set of bars, the first 9 bars shows the bandwidth for one of the kernels. For reference, the last 3 bars show the measured bandwidth of DGEMV, the 1×1 SpM×V routine for a dense matrix stored in sparse format, and the exhaustive best $r\times c$ register blocked SpM×V routine for the dense matrix. Finally, we also show, using asterisks (*), the model bandwidth computed for each of the last 8 bars (non-standard STREAM kernels).

is largest for the dot product kernel on Itanium, though it is still a very good approximation for the SpM×V (on-chip), SpM×V (external), and true SpM×V kernels.

- On all platforms, the dot product kernel achieves the highest bandwidth among the synthetic kernels. Furthermore, the bandwidth of loading two double-precision streams (dot product) has a higher bandwidth than loading a single stream (sum). Nevertheless, they are comparable on the Ultra 2i, Pentium III, and Itanium, and are treated as the same in the latency model. On the Power3, the sum kernel runs at a much lower bandwidth than the dot product, though we do not yet know why.[11]
- We would expect the bandwidth of the true 1×1 (unblocked) SpM×V to fall between the two synthetic variants, SpM×V (on-chip) and SpM×V (external). Indeed, this is the case. In fact, the unblocked bandwidth tends to be close (within 10%) to the SpM×V (on-chip) bandwidth on all platforms except the Pentium III; on the Pentium III, the unblocked bandwidth is close to SpM×V (external). This phenomenon can be explained by the size of the on-chip cache on the Pentium III, which is the smallest of all the on-chip caches.
- Consider the unblocked and blocked SpM×V bandwidths shown in Figure 11. On all platforms except the Power3, the improvement in model bandwidth is similar to the observed improvement in measured bandwidth, showing that tuning has realized the positive effect expected by the model.
- Section 4 shows that on the Power3, there still exists a significant gap between the performance of our register blocked implementation and our upper-bound. Figure 11 suggests that additional low-level tuning could prove beneficial on this platform. In particular, the gap between measured bandwidth and our model bandwidth for the dot product kernel and DGEMV is much smaller than the gap for SpM×V.

These results confirm that the latency model offers a reasonable upper-bound for SpM×V. In certain settings, namely, the dot product, it may be possible to do better than the limit suggested by the model through additional low-level tuning. However, based on the observed bandwidth results, it is not clear whether such improvements could be realized for SpM×V, and even if they could, whether they would yield performance benefits beyond an additional 5–10%. The exception is the Power3, on which additional effort at scheduling could lead to substantial performance gains.

6 Conclusions and Future Directions

Our results show that for sparse matrices with natural block structures, and on several computer architectures, we are close to the best possible performance for SpM×V. This leads us to ask where further performance improvements lie.

[11] Changing the sum kernel to include a scalar multiply with each add, *i.e.*, `s` += α*`a`[i], did not change the measured bandwidth.

First, matrices without natural block structures remain difficult. Techniques such as reordering the rows and columns to create more blocks, using multiple block sizes, or cache blocking (storing large rectangular submatrices as separate sparse matrices) show promise and we are pursuing them [16, 30, 19, 24].

Second, our register blocking techniques have not been very effective on the Power3 architecture. We need to determine whether we can do better.

Third, we need to exploit more structure in the sparse matrices that will let us improve data reuse. For example, if the matrix is symmetric, then each matrix entry can be reused twice. Our preliminary results indicate that we can go significantly faster, up to a factor of 2.

Fourth, we can identify and tune "higher level" sparse kernels that also permit more matrix reuse. An example is applying register blocking to sparse matrix-multiple-vector multiplication (SpM×M) [23, 10, 17]. This kernel can be exploited in iterative solvers with multiple right-hand sides and also in block eigensolvers. On the Itanium and Ultra 2i we have observed speedups of up to 6.5 and 9 times that of SpM×M with a single right-hand side. Another example is computing $A^T Ax$, which is used in information retrieval and computing the singular value decomposition; each entry of A can be used twice.

Acknowledgements

We would like to thank Gautam Doshi at Intel for comments on memory bandwidth issues.

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] J. Bilmes, K. Asanović, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC. see `http://www.icsi.berkeley.edu/~bilmes/phipac`.

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

[5] 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`.

[6] 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.

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

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] Intel. Intel Itanium processor reference manual for software optimization, November 2001.

[19] I. James B. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings of the International Conference on High-Performance Computing*, 1997.

[20] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *Newsletter of the IEEE Technical Committee on Computer Architecture*, December 1995. `http://tab.computer.org/tcca/NEWS/DEC95/DEC95.HTM`.

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

[22] 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.

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

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

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

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

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

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

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

[30] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

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

[32] 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 *b_row_start,
                  const int *b_col_idx, const double *b_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 = b_row_start[i]; jj < b_row_start[i+1];
                    jj++, b_col_idx++, b_value += 2*2 )
            {
 5              d0 += b_value[0] * x[*b_col_idx+0];
 6              d1 += b_value[2] * x[*b_col_idx+0];
 7              d0 += b_value[1] * x[*b_col_idx+1];
 8              d1 += b_value[3] * x[*b_col_idx+1];
            }
 9          y[0] = d0;
10          y[1] = d1;
        }
    }
```

B Register Block Sizes

In Tables 4–7, we show for each platform and matrix, the performance (Mflop/s) of the following register block sizes:

- SPARSITY exhaustive best: The value of $r \times c$ yielding the best performance, and its performance.
- SPARSITY v2.0 heuristic: The value of $r \times c$ chosen by the new heuristic, and its performance.
- SPARSITY v1.0 heuristic: The value of $r \times c$ chosen by the heuristic in SPARSITY v1.0, and its performance.

In addition, the performance data has been annotated to faciliate pairwise comparisons. In particular, an asterisk ($*$) next to the best performance indicates that the performance of the block size chosen by the v2.0 heuristic was less than 90% of the best performance. A circle ($\circ$) indicates that the performance of the block size chosen by the v1.0 heuristic was less than 90% of the best performance. Finally, a dagger ($\dagger$) shows when the performance of the implementation selected by the v1.0 heuristic was less that chosen by the v2.0 heuristic.

	Sparsity exhaustive best			Sparsity v2.0 heuristic			Sparsity v1.0 heuristic		
Matrix	Block size	Perf. (Mflop/s)	Fill ratio	Block size	Perf. (Mflop/s)	Fill ratio	Block size	Perf. (Mflop/s)	Fill ratio
1	8×5	72.9	1.00	8×5	72.9	1.00	8×8	71.0	1.00
2	8×8	63.2	1.00	8×8	63.2	1.00	8×8	63.2	1.00
3	6×6	54.5	1.12	6×6	54.5	1.12	6×6	54.5	1.12
4	6×2	54.1	1.13	3×3	51.9	1.06	6×6	50.1	1.19
5	4×4	48.4	1.00	4×4	48.4	1.00	4×4	48.4	1.00
6	3×3	49.9	1.00	3×3	49.9	1.00	3×3	49.9	1.00
7	3×3	52.5	1.00	3×3	52.5	1.00	3×3	52.5	1.00
8	6×6	50.1	1.15	6×6	50.1	1.15	6×6	50.1	1.15
9	3×3	54.3	1.02	3×3	54.3	1.02	3×3	54.3	1.02
10	2×1	39.1	1.10	2×2	38.8	1.21	2×2	38.8	1.21
11	2×2	32.3	1.23	2×2	32.3	1.23	2×2	32.3	1.23
12	2×2	37.9	1.24	2×3	36.4	1.36	3×3	36.4	1.46
13	2×1	36.7	1.14	2×2	36.0	1.28	3×3	34.5	1.52
15	2×1	41.1 ∘	1.00	2×1	41.1 †	1.00	2×2	33.8	1.35
17	1×1	32.4	1.00	1×1	32.4	1.00	1×1	32.4	1.00
21	1×1	28.7	1.00	1×1	28.7	1.00	1×1	28.7	1.00
25	1×1	21.3	1.00	1×1	21.3	1.00	1×1	21.3	1.00
27	2×1	21.5	1.53	1×1	20.1	1.00	1×1	20.1	1.00
28	1×1	26.9	1.00	1×1	26.9	1.00	1×1	26.9	1.00
36	1×1	18.1	1.00	1×1	18.1	1.00	1×1	18.1	1.00
40	1×1	31.9	1.00	1×1	31.9	1.00	1×1	31.9	1.00
44	1×1	23.5	1.00	1×1	23.5	1.00	1×1	23.5	1.00

Table 4: **Register block sizes, Sun Ultra 2i**. To aid pairwise comparisons, the data has been annotated as follows. An asterisk (∗) indicates that the v2.0 performance (Mflop/s) is less than 90% of the best performance (Mflop/s); a circle (∘) indicates that the v1.0 performance is less than 90% of the best; a dagger (†) indicates that the v1.0 performance is less than 90% of the v2.0 performance. On the Ultra 2i, the v2.0 heuristic never selected a block size below 90% of the best.

Matrix	SPARSITY exhaustive best			SPARSITY v2.0 heuristic			SPARSITY v1.0 heuristic		
	Block size	Perf. (Mflop/s)	Fill ratio	Block size	Perf. (Mflop/s)	Fill ratio	Block size	Perf. (Mflop/s)	Fill ratio
1	2×10	107.1	1.00	2×10	107.1	1.00	6×6	98.0	1.00
2	4×8	94.2	1.00	4×8	94.2	1.00	4×4	85.0	1.00
3	6×2	85.7	1.12	6×2	85.7	1.12	3×3	83.0	1.12
4	3×3	85.5	1.06	3×3	85.5	1.06	3×3	85.5	1.06
5	4×2	82.3	1.00	4×2	82.3	1.00	4×4	74.3	1.00
6	3×3	87.9	1.00	3×3	87.9	1.00	3×3	87.9	1.00
7	3×3	89.6	1.00	3×3	89.6	1.00	3×3	89.6	1.00
8	6×2	83.1	1.13	3×3	80.8	1.11	3×3	80.8	1.11
9	3×3	87.9	1.02	3×3	87.9	1.02	3×3	87.9	1.02
10	4×2	63.4	1.45	2×2	60.2	1.21	3×3	58.5	1.57
11	2×2	52.6	1.23	2×2	52.6	1.23	2×2	52.6	1.23
12	3×3	62.8	1.46	3×3	62.8	1.46	3×3	62.8	1.46
13	3×3	60.1	1.52	3×3	60.1	1.52	3×3	60.1	1.52
14	3×2	42.4	1.47	2×2	41.9	1.33	3×3	42.2	1.60
15	2×1	55.9 ∘	1.00	2×1	55.9 †	1.00	1×1	38.6	1.00
16	4×1	42.4	1.43	3×3	40.6	1.69	3×3	40.6	1.69
17	4×1	46.9 ∘	1.75	4×1	46.9 †	1.75	3×3	39.5	2.36
18	2×1	30.6	1.36	1×2	27.6	1.36	2×2	28.9	1.79
20	1×2	41.9 ∘	1.17	1×2	41.9 †	1.17	3×3	35.5	2.35
21	4×1	43.9 ∘	1.77	4×1	43.9 †	1.77	1×1	38.3	1.00
23	2×1	29.0	1.46	1×2	26.6	1.47	1×1	28.4	1.00
24	2×1	36.1	1.52	1×1	35.9	1.00	1×1	35.9	1.00
25	1×1	30.4	1.00	1×1	30.4	1.00	1×1	30.4	1.00
26	1×1	27.9	1.00	1×1	27.9	1.00	1×1	27.9	1.00
27	2×1	32.1	1.53	1×1	31.4	1.00	1×1	31.4	1.00
28	1×1	36.7	1.00	1×1	36.7	1.00	1×1	36.7	1.00
29	2×2	28.1	1.98	1×1	28.0	1.00	2×2	28.1	1.98
36	1×1	26.4	1.00	1×1	26.4	1.00	2×2	24.2	2.31
37	2×2	28.2	1.98	1×1	28.1	1.00	2×2	28.2	1.98
40	1×1	39.2	1.00	1×1	39.2	1.00	1×1	39.2	1.00
41	1×1	31.3	1.00	1×1	31.3	1.00	1×1	31.3	1.00
42	1×1	31.2	1.00	1×1	31.2	1.00	1×1	31.2	1.00
44	1×1	29.0	1.00	1×1	29.0	1.00	1×1	29.0	1.00

Table 5: **Register block sizes, Intel Pentium III**. Data has been annotated as in Table 4. On the Pentium III, the v2.0 heuristic never selected a block size below 90% of the best.

Matrix	Sparsity exhaustive best Block size	Perf. (Mflop/s)	Fill ratio	Sparsity v2.0 heuristic Block size	Perf. (Mflop/s)	Fill ratio	Sparsity v1.0 heuristic Block size	Perf. (Mflop/s)	Fill ratio
2	4×4	167.7	1.00	4×4	167.7	1.00	4×4	167.7	1.00
4	3×3	144.8	1.06	3×3	144.8	1.06	3×3	144.8	1.06
5	4×2	147.8	1.00	4×4	140.6	1.00	4×4	140.6	1.00
7	3×3	154.8	1.00	3×3	154.8	1.00	3×3	154.8	1.00
8	3×3	140.7	1.11	3×3	140.7	1.11	3×3	140.7	1.11
9	3×3	154.9	1.02	3×3	154.9	1.02	3×3	154.9	1.02
10	1×1	132.7 ∘	1.00	2×2	121.6 †	1.21	4×4	99.9	1.73
12	1×1	130.2 ∘	1.00	2×2	117.3	1.24	3×3	107.3	1.46
13	1×1	130.0 ∘	1.00	1×1	130.0 †	1.00	3×3	102.9	1.52
15	2×1	136.4	1.00	2×1	136.4	1.00	2×1	136.4	1.00
40	1×1	127.4	1.00	1×1	127.4	1.00	1×1	127.4	1.00

Table 6: **Register block sizes, IBM Power3**. Data has been annotated as in Table 4. On the Power3, the v2.0 heuristic never selected a block size below 90% of the best.

Matrix	Sparsity exhaustive best Block size	Perf. (Mflop/s)	Fill ratio	Sparsity v2.0 heuristic Block size	Perf. (Mflop/s)	Fill ratio	Sparsity v1.0 heuristic Block size	Perf. (Mflop/s)	Fill ratio
1	4×1	255.8	1.00	4×1	255.8	1.00	2×2	244.7	1.00
2	4×1	228.4	1.00	4×1	228.4	1.00	2×2	215.3	1.00
3	3×1	208.0 ∘	1.06	3×1	208.0 †	1.06	2×2	183.5	1.12
4	3×1	203.4 ∘	1.04	3×1	203.4 †	1.04	2×2	177.0	1.07
5	4×1	187.9	1.00	4×1	187.9	1.00	2×2	176.3	1.00
6	3×1	220.4 ∘	1.00	3×1	220.4 †	1.00	2×2	164.8	1.23
7	3×1	221.4 ∘	1.00	3×1	221.4 †	1.00	2×2	167.2	1.22
8	3×1	200.6 ∘	1.06	3×1	200.6 †	1.06	2×2	166.8	1.10
9	3×1	216.9 ∘	1.01	3×1	216.9 †	1.01	2×2	165.1	1.25
10	3×1	168.7	1.27	2×1	162.3	1.10	2×2	157.8	1.21
11	4×1	119.0	1.70	2×2	117.9	1.23	2×2	117.9	1.23
12	3×1	182.1 ∘	1.24	2×1	175.8	1.13	2×2	163.9	1.24
13	3×1	178.5 ∘	1.26	2×1	174.1	1.14	2×2	160.1	1.28
15	2×1	163.7 ∘	1.00	2×1	163.7 †	1.00	2×2	137.8	1.35
17	3×1	141.0	1.59	2×1	138.7	1.36	1×1	136.3	1.00
21	3×1	141.2	1.59	2×1	137.0	1.38	1×1	138.2	1.00
25	1×1	64.1	1.00	1×1	64.1	1.00	1×1	64.1	1.00
27	3×1	77.6 ∗∘	1.94	1×1	66.9	1.00	1×1	66.9	1.00
28	1×1	121.3	1.00	1×1	121.3	1.00	1×1	121.3	1.00
36	3×1	53.6	2.31	1×1	49.9	1.00	1×1	49.9	1.00
40	1×1	127.8	1.00	1×1	127.8	1.00	1×1	127.8	1.00
44	1×1	74.6	1.00	1×1	74.6	1.00	1×1	74.6	1.00

Table 7: **Register block sizes, Intel Itanium**. Data has been annotated as in Table 4. For Matrix #27, both the v1.0 and v2.0 heuristics chose an implementation which achieved 86% of the best. Note that the fill ratio at the best block size is nearly 2.