When Cache Blocking of Sparse Matrix Vector Multiply Works and Why

Rajesh Nishtala, Richard Vuduc, James W. Demmel, and Katherine A. Yelick

University of California at Berkeley, Computer Science Division
Berkeley, California, USA

Abstract. We present new performance models and a new, more compact data structure for *cache blocking* when applied to the sparse matrix-vector multiply (SpM×V) operation, $y \leftarrow y + A \cdot x$. Prior work indicates that cache blocked SpM×V performs very well for some matrix and machine combinations, yielding speedups as high as 3x. We look at the general question of when and why performance improves, finding that cache blocking is most effective when simultaneously 1) x does not fit in cache, 2) y fits in cache, 3) the non-zeros are distributed throughout the matrix, and 4) the non-zero density is sufficiently high. We extend our prior performance models, which bounded performance by assuming x and y fit in cache, to consider these classes of matrices. Unlike our prior model, the updated models are accurate enough to use as a heuristic for predicting the optimum block sizes. We conclude with architectural suggestions that would make processor and memory systems more amenable to SpM×V.

1 Introduction and Overview

We consider the problem of building high-performance implementations of sparse matrix-vector multiply (SpM×V), or $y \leftarrow y + A \cdot x$. 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. Moreover, in contrast to optimizing dense matrix kernels (dense BLAS), performance depends on the non zero structure of the matrix which may not be known until run-time.

In prior work on the Sparsity system (version 1.0) [6], 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*. Cache blocking differs from register blocking in that cache blocking reorders memory accesses to increase temporal locality, whereas register blocking compresses the data structure to reduce memory traffic. This paper focuses on cache blocking (Section 2) and asks the fundamental questions of what limits exist on such performance tuning, and how close tuned code gets to these limits. The models presented in this paper (Section 3) extend our prior models

[13] by accounting for the TLB, enabling accurate selection of optimal *cache block* sizes. It increases the complexity of the data structures used to represent the matrix by adding an extra set of row pointers for each block. The fundamental trade off we need to make is whether the benefit of the added temporal locality outweighs the costs associated with accessing the added overhead.

We classify the set of matrices on which we see benefits from cache blocking, concluding that cache blocking is most effective when simultaneously 1) x does not fit in cache 2) y fits in cache, 3) the non zeros are distributed throughout the matrix and 4) the non-zero density is sufficiently high. In particular all the test matrices in Table 1 (except Matrix 1) are sparse enough so that register blocking [6, 13] has no significant effect.

Traditional static models of cache behavior used to select tile sizes for dense kernels cannot be applied to sparse kernels due to the presence of indirect and irregular memory accesses known only at run-time. Nevertheless, there have been a number of notable attempts. Temam and Jalby [11], Heras, *et al.* [5], and Fraguela, *et al.* [2] 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. Our model in Section 3 differs from the prior work in that 1) we consider multi-level memory hierarchies including the TLB, and 2) explicitly model the execution time in addition to cache misses.

Gropp, *et al.*, use bounds similar to the ones we develop to analyze and tune a computational fluid dynamics code [3]; Heber, *et al.*, present a detailed performance study of a fracture mechanics code on Itanium [4]. This paper considers tuning for matrices that come a variety of other domains, and is furthermore concerned with performance modeling for cache block size selection.

Due to space limitations we only present the high level intuitions and summary data. We refer the reader to the full report [7] for a detailed investigation.

2 Summary of the Cache Blocking Optimization

We assume a reference implementation which stores the matrix in a compressed sparse row (CSR) format [8]. Cache blocking breaks the CSR matrix into multiple smaller r_{cache} x c_{cache} CSR matrices and then stores these sequentially in memory. Below, we discuss how 1)we compress the size of each block using the *row start/end* (RSE) optimization, and 2) further exploit the fact that each cache block is a smaller matrix. The latter technique also allows easy recursion with multiple levels of cache blocking.

Row Start / End (RSE) When matrices (especially band matrices) are blocked it is possible that within a cache block non-zeros do not exist on all the rows. The first cache block, for example, might have only non zero elements in the first tenth of the rows and have the rest of the cache block be empty. However the basic cache blocked data structure would loop over all zero rows without doing any useful work. In order to avoid the unnecessary accesses, a new vector that contains row start(RS) and row end (RE) information for each cache block is also created to point to the first and last nonzero rows in the cache block. This new indexing information makes the performance less sensitive to the size of the cache block. Performance results have shown that this optimization can only help improve performance [7].

Exploiting Cache Block Structure As described above, the cache blocked matrix can be thought of as many smaller sparse matrices stored sequentially in memory. We can exploit this fact by calling our prior sparse matrix vector multiplication routines on each smaller matrix, passing the appropriate part of the source and destination vectors as arguments. The advantage of handling the multiplication in this fashion is that the inner loops can be generated independently of the code for cache blocking and code previously written for non-cache blocked implementations can be reused. This optimization also allows easy recursion with multiple levels of cache blocking. Tests indicate that the function call overhead is negligible since the number of cache blocks for a matrix is usually small compared to the total memory operations.

3 Analytic Models of Memory System Performance

We create analytic upper and lower bounds on performance by modeling various levels of the memory hierarchy. The load and cache models are identical to our prior models [12]. Due to space limitations we do not present those models here. The lower bound model assumes only compulsory misses while the upper bound assumes that every access to x, y, and matrix miss.

Overall Performance Model We extend our prior model [12] by adding a term to account for TLB misses. The new execution time model is as follows:

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

We assume perfect nesting of the caches, thus $h_{i+1} = m_i - m_{i+1}$, where h_i and m_i are the hits and misses at the i^{th} level of cache respectively. To estimate the *upper bound on performance*, we set the m_i terms to count only the compulsory misses at the i^{th} level. In addition we set the m_{TLB} term to be $M^{(TLB)}_{\text{model}}(r, c)$ which is described below. For the *lower bound on performance* we set all the cache miss terms to be the upper bound on cache misses at each level and set the TLB misses to be the upper bound on TLB misses.

TLB Miss Model According to our simple load and cache miss models, cache blocking has no benefit since the blocking adds overhead to the data storage. To factor this in, we need to be able to model at least the most important level of the memory system more accurately to expose the advantages of locality. Empirical evidence suggests that the largest performance gains using cache blocking come from minimizing TLB misses. Below, we present intuition behind our TLB miss modeling, m_{TLB}, and refer the reader elsewhere [7] for full expressions.

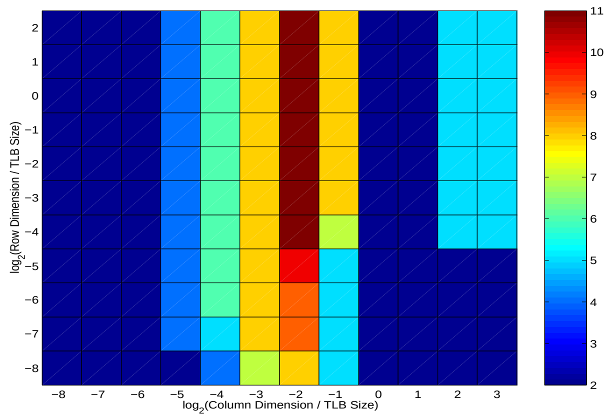

Fig. 1. Histogram of Block sizes for Itanium 2. For each row and column block size shown above, the value in the cell contains the number of matrices whose performance was within 90% of peak if that block size was chosen. We define *TLB Size* to be the number of entries in the TLB multipliled by the page size. On the Itanium 2 this was 2MB or 256 doubles.

As shown in Figure 1, measurements indicate two distinct categories of good block sizes for our matrix suite for the Itanium 2. Matrices 2–11 showed the best performance when c_{cache} equaled $\frac{1}{4}$th the TLB size (in words). Matrices 12–14 did not benefit at all from blocking, *i.e.*, c_{cache} equals the column dimension. This dichotomy existed on other platforms as well. Furthermore, performance was relatively insensitive to the row block size, suggesting no row blocking is needed. Our TLB model reflects these observations by switching between expressions for lower and upper bounds on TLB misses as the block size varies [7].

4 Verification of the Analytic Model

We evaluate SpM×V on a set of matrices that are large enough and sparse enough for cache blocking to have a significant effect. The properties of the 14 matrices that were chosen are referenced in Table 1. We evaluate the performance model in which we use true hardware counters through PAPI [1] to predict the performance (henceforth called the PAPI model) and compare it to the model in which we use estimates of lower and upper bound of cache and TLB misses (henceforth termed the analytic model). The cache and memory latencies were derived from published processor manuals, curve fitting, and experimental work using the Saavedra-Barrera memory system microbenchmark [9] and MAPS benchmarks [10]. Due to space limitations we only present a summary of the data for the Itanium 2.

Table 1. Matrix Benchmark Suite. Matrices are listed in alphabetical order. Note that matrices 6, 7, and 8 are just modified versions of matrix 5.

	Application Area	Dimension	Nonzeros	Density
1	Dense Matrix	2000 x 2000	4000000	1.00
2	Statistical Experimental Design	231 x 319770	8953560	1.21e-1
3	Linear programming (LP)	52260 x 379350	1567800	7.91e-5
4	LP	10280 x 243246	1408073	5.63e-4
5	Latent Semmantic Indexing	10000 x 255943	3712489	1.45e-3
6	column wise expansion of LSI	10000 x 2559430	3712489	1.45e-4
7	row wise expansion of LSI	100000 x 255943	3712489	1.45e-4
8	row wise stamping of LSI	100000 x 255943	37124890	1.45e-3
9	Queuing model of mutual exclusion	65535 x 65535	1114079	2.59e-4
10	Italian Railways scheduling (LP)	4284 x 1092610	11279748	2.41e-3
11	Italian Railways scheduling (LP)	4284 x 546305	5661231	2.42e-3
12	Web connectivity graph (WG)	1000005 x 1000005	3105536	3.11e-6
13	WG after MMD reordering	1000005 x 1000005	3105536	3.11e-6
14	WG after RCM reordering	1000005 x 1000005	3105536	3.11e-6

The model of Section 3 over predicts absolute performance by up to a factor of 2 on the Itanium 2, implying time still unaccounted for. Moreover, the relative performance as a function of block size is well predicted, meaning we can use the model as a heuristic for choosing a good block size. Indeed, performance at the optimal block sizes in the PAPI model are all within 90% of the best on Itanium 2, implying the model is a good heuristic if the miss models are accurate. Furthermore, except in the case of Matrix 3, the analytic model makes similarly good predictions on the Itanium 2, yielding 90% of the best performance.

5 Evaluation Across Matrices and Platforms

Matrix Structure The speedups for each matrix varied across machines, but the best speedups (Table 2) were observed for the same matrices. The best speedups occurred with Matrices 5–8, 2, and 10–11. Except for Matrices 7 and 8, these matrices have small row dimension and very large column dimension, with non-zeros scattered throughout the matrix. Furthermore, the largest increases in cache misses as c_{cache} increased occurred on the matrices the largest speedups, implying that cache blocking had the intended effect of increasing locality.

Matrices 12–14 are so sparse that there is effectively no reuse when accessing the source vector and thus blocking does not help, even though their source vector is large. Matrices with densities higher than 10^{-5} were helped with cache blocking, provided that their column block size is large enough (greater than

Table 2. Speedups across Matrices and Across Platforms. This table shows the performance of the optimum cache block divided by the performance of the non-blocked implementation on that platform for that matrix.

	Matrix No.													
Platform	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Itanium 2	1.00	1.27	1.28	1.14	2.00	2.84	1.72	1.94	1.00	1.40	1.34	1.00	1.00	1.00
Pentium 3	1.01	1.61	1.02	1.15	1.40	1.33	1.10	N/A	1.00	1.21	1.21	1.00	1.00	1.00
Power 4	1.01	1.77	1.24	1.37	1.97	2.93	1.68	N/A	1.00	1.75	1.73	1.01	1.09	1.01

200,000 elements). There was enough reuse in x for the blocking to payoff. We also find that in general matrices in which the row dimension is much less than the column dimension benefit the most from cache blocking. The smaller row dimension implies the overhead added by cache blocking is small since the number of rows themselves are limited. The larger column dimension implies that the unblocked implementations lack locality. Even though Matrix 3 has a large column dimension, blocking did not yield much performance improvement. We performed additional experiments on random but banded matrices confirming theoretical work by Temam and Jalby [11]. As expected, cache blocking does not help when the band is relatively narrow because the natural access pattern to x is optimal, but pays off as the band grows. In this latter case, the RSE optimization smooths out differences in performance across block sizes [7].

Platform Evaluation Certain matrices such as Matrix 5 experienced significant performance gains through cache blocking on the Itanium 2 and the Power 4, but the speedup was less drastic on the Pentium 3. We expect that as the average number of cycles to access the memory grows, cache blocking will provide a good improvement in performance since cache blocking allows us to reduce expensive accesses to the main memory. The behavior of cache blocked SpM$\times$V has a number of implications for architecture and systems. First, the TLB misses reduced by cache blocking can also be avoided by creating large page sizes. Second, two paths to memory would be ideal since only access to x are helped by caches, and not accesses to the matrix itself. Separate paths would prevent cache conflicts between matrix data and source vector data. In contrast, increased associativity only partially addresses this issue since it still allows premature eviction of “old” source vector elements by matrix elements. Future work might verify the impact of separate memory paths on the hybrid scalar-vector architecture of the Cray X1.

6 Conclusions and Future Work

The empirical findings discussed in this paper and our full report indicate that TLB misses have the largest impact on performance. Cache blocking significantly reduces these misses particularly when x is large, y is small, the distribution of non-zeros is nearly random, and the non-zero density is sufficiently high. Our new performance bounds models incorporate the effect of TLB by implicitly modeling capacity and conflict misses ignored by our prior models [12, 13]. Moreover, these new models predict optimal (or near-optimal) cache block sizes.

Future work should focus on improving the accuracy of the miss models at all the levels in the memory hierarchy and obtain more accurate memory latencies. More accurate models should lead to even more accurate heuristics that decide when and how to cache block a sparse matrix, given the platform and matrix structure. Future work would also analyze the problem on novel architectures.

References

1. 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.
2. B. B. Fraguela, R. Doallo, and E. L. Zapata. Memory hierarchy performance prediction for sparse blocked algorithms. *Parallel Processing Letters*, 9(3), 1999.
3. 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.
4. 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, 2001.
5. 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.
6. E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication.* PhD thesis, University of California, Berkeley, May 2000.
7. R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Technical report, University of California, Berkeley, EECS Dept., 2004. (to appear).
8. Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.
9. R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking.* PhD thesis, University of California, Berkeley, February 1992.
10. A. Snavely, L. Carrington, and N. Wolter. Modeling application performance by convolving machine signatures with application profiles. 2001.
11. O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing '92*, 1992.
12. R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.
13. R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels.* PhD thesis, University of California, Berkeley, 2003.