Fast sparse matrix-vector multiplication by exploiting variable block structure

Richard W. Vuduc[1] and Hyun-Jin Moon[2]

[1] Lawrence Livermore National Laboratory (richie@llnl.gov)
[2] University of California, Los Angeles (hjmoon@cs.ucla.edu)

Abstract. We improve the performance of sparse matrix-vector multiplication (SpMV) on modern cache-based superscalar machines when the matrix structure consists of multiple, irregularly aligned rectangular blocks. Matrices from finite element modeling applications often have this structure. We split the matrix, A, into a sum, $A_1 + A_2 + \ldots + A_s$, where each term is stored in a new data structure we refer to as *unaligned block compressed sparse row (UBCSR) format* . A classical approach which stores A in a block compressed sparse row (BCSR) format can also reduce execution time, but the improvements may be limited because BCSR imposes an alignment of the matrix non-zeros that leads to extra work from filled-in zeros. Combining splitting with UBCSR reduces this extra work while retaining the generally lower memory bandwidth requirements and register-level tiling opportunities of BCSR. We show speedups can be as high as 2.1× over no blocking, and as high as 1.8× over BCSR as used in prior work on a set of application matrices. Even when performance does not improve significantly, split UBCSR usually reduces matrix storage.

1 Introduction

The performance of diverse applications in scientific computing, economic modeling, and information retrieval, among others, is dominated by sparse matrix-vector multiplication (SpMV), $y \leftarrow y + A \cdot x$, where A is a sparse matrix and x, y are dense vectors. Conventional implementations using compressed sparse row (CSR) format storage usually run at 10% of machine peak or less on uniprocessors [16]. Higher performance requires a compact data structure and appropriate code transformations that best exploit properties of both the sparse matrix—which may be known only at run-time—*and* the underlying machine architecture. Compared to dense kernels, sparse kernels incur more overhead per non-zero matrix entry due to extra instructions and indirect, irregular memory accesses. We and others have studied techniques for selecting good data structures and for automatically tuning the resulting implementations (Section 6). Tuned SpMV can in the best cases achieve 31% of peak and 4× speedups over CSR [6, 16, 18].

The best performance occurs for the class of applications based on finite element method (FEM) modeling, but within this class there is a performance gap between matrices consisting primarily of dense blocks of a single size, uniformly

aligned, and matrices whose structure consists of multiple block sizes with irregular alignment. For the former class, users often rely on so-called block compressed sparse row (BCSR) format, BCSR, which stores A as a sequence of fixed-size $r \times c$ dense blocks. BCSR uses one integer index of storage per block instead of one per non-zero as in CSR, reducing the index storage by $\frac{1}{rc}$. Moreover, fixed-sized blocks enable unrolling and register-level tiling of each block-multiply.

However, two difficulties arise in practice. First, *the best $r \times c$ varies both by matrix and by machine* [16], motivating automatic tuning. Secondly, *speedup is mitigated by fill-in of explicit zeros.* We observed cases in which BCSR reduces the execution time of SpMV to $\frac{2}{3}$ that of CSR (1.5× speedup) while also requiring storage of 50% additional explicit zero entries [16]. To reduce this work and achieve still better speedups, this paper considers simultaneously (a) *splitting* A into the sum $A = A_1 + \cdots + A_s$, where each A_i may be stored with a different block size, and (b) storing each A_i in a *unaligned block compressed sparse row (UBCSR) format* that relaxes *both* row and column alignments of BCSR, at the cost of indirection to x and y instead of just x as in BCSR and CSR.

We recently compared BCSR-based SpMV to CSR on 8 platforms and 44 matrices, and identified 3 classes of matrices [16, Chap. 4]: FEM matrices 2–9, whose structure consists essentially of a single block size uniformly aligned, FEM matrices 10–17, whose structure contains mixed block structure, and matrices from other applications (*e.g.*, economic modeling, linear programming).[3] The median speedups for FEM 2–9 were between 1.1× and 1.54× higher than the median speedups for FEM 10–17. Split UBCSR reduces this gap, running in as little as half the time of CSR (2.1× speedup) and $\frac{5}{9}$ the time of BCSR (1.8× speedup). Moreover, splitting can significantly reduce matrix storage. We are making our techniques available in OSKI [17], a library of automatically tuned sparse matrix kernels that builds on the Sparsity framework [6, 5].

This paper summarizes our recent technical report [19]. We will refer the reader there for more detail when appropriate.

2 Characterizing Variable Block Structure in Practice

We use variable block row (VBR) format [12, 11], which logically partitions rows and columns into block rows and columns, to distinguish the dense block substructure of FEM Matrices 10–17 from 2–9 in two ways:[4]

- **Unaligned blocks**: Consider any $r \times c$ dense block starting at position (i, j) in an $m \times n$ matrix, where $0 \leq i < m$ and $0 \leq j < n$. BCSR typically assumes a *uniform alignment constraint*, $i \bmod r = j \bmod c = 0$. Relaxing the column constraint so that $j \bmod c$ is any value less than c has yielded some improvements in practice [2]. However, most non-zeros of Matrices 12 and 13 lie in blocks of the same size, with $i \bmod r$ and $j \bmod c$ distributed

[3] We omit test matrix 1, a dense synthetic matrix stored in sparse format.

[4] We treat Matrix 11, which contains a mix of blocks and diagonals, using other techniques [16, Chap. 5]; Matrices 14 and 16 are eliminated due to their small size.

#	Matrix	Dimension	No. of Non-zeros	Dominant block sizes (% of non-zeros)
10	`ct20stif` Engine block	52329	2698463	6×6 (39%) 3×3 (15%)
12	`raefsky4` Buckling problem	19779	1328611	3×3 (96%)
13	`ex11` 3D flow	16614	1096948	1×1 (38%) 3×3 (23%)
15	`vavasis3` 2D partial differential equation	41092	1683902	2×1 (81%) 2×2 (19%)
17	`rim` Fluid mechanics problem	22560	1014951	1×1 (75%) 3×1 (12%)
A	`bmw7st_1` Car body analysis	141347	7339667	6×6 (82%)
B	`cop20k_m` Accelerator cavity design	121192	4826864	2×1 (26%), 1×2 (26%) 1×1 (26%), 2×2 (22%)
C	`pwtk` Pressurized wind tunnel	217918	11634424	6×6 (94%)
D	`rma10` Charleston Harbor model	46835	2374001	2×2 (17%) 3×2 (15%), 2×3 (15%) 4×2 (9%), 2×4 (9%)
E	`s3dkq4m2` Cylindrical shell	90449	4820891	6×6 (99%)

Table 1. Variable block test matrices. Problem sources are summarized elsewhere [16, Appendix B].

uniformly over values up to $r-1$ and $c-1$, respectively. One goal of our UBCSR is to relax the row alignment as well.

– **Mixtures of "natural" block sizes**: Matrices 10, 15, and 17 possess a mix of block sizes when viewed in VBR format. This motivates the decomposition, $A = A_1 + A_2 + \cdots + A_s$, where each term A_i consists of the subset of blocks of a particular size. Each term can then be tuned separately.

These observations apply to the matrices listed in Table 1, which include a subset of the matrices referred to previously as Matrices 10–17, and 5 additional matrices (labeled Matrices A–E) from other FEM applications.

VBR can reveal dense block substructure. Consider storage of Matrix 12 in VBR format, using a CSR-to-VBR conversion routine available in the SPARSKIT library [12]. This routine partitions the rows by looping over rows in order, starting at the first row, and placing rows with identical non-zero structure in the same block. The same procedure is used to partition the columns, with the result shown in Figure 1 (*top*). The maximum block size in VBR turns out to be 3×3, with 96% of non-zeros stored in such blocks (Figure 1 (*bottom-left*), where a label of '0' indicates that the fraction is zero when rounded to two digits but there is at least 1 block at the given size). Moreover, these blocks are not uniformly aligned on row boundaries as assumed by BCSR. Figure 1 (*bottom-right*) shows

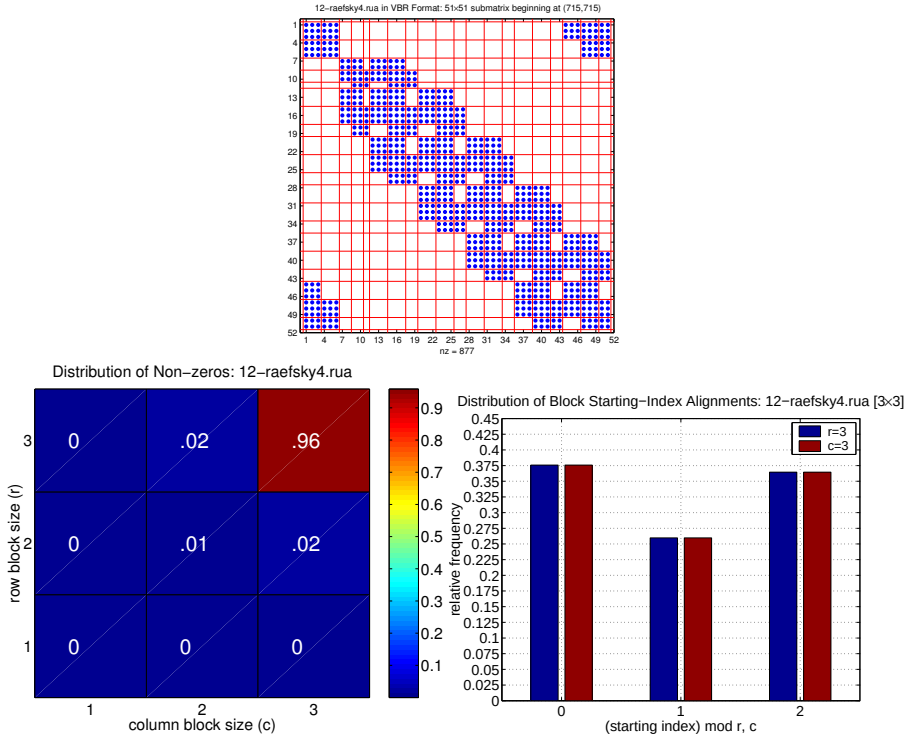

Fig. 1. Logical grid (block partitioning) after greedy conversion to variable block row (VBR) format: Matrix 12-`raefsky4`. (*Top*) Partitioning after conversion to VBR format. (*Bottom-left*) 96% of the non-zero blocks are in 3×3 blocks. (*Bottom-right*) The 3×3 blocks have varying alignments.

the distributions of $i \bmod r$ and $j \bmod c$, where (i, j) is the starting position in A of each 3×3 block, and the top-leftmost entry of A is $A(0, 0)$. The first row index of a given block row can start on any alignment, with 26% of block rows having $i \bmod r = 1$, and the remainder split equally between 0 and 2. This observation motivates UBCSR which allows flexible alignments.

We summarize the variable block structure of the matrix test set used in this paper in the rightmost column of Table 1. This table includes a short list of dominant block sizes after conversion to VBR format, along with the fraction of non-zeros for which those block sizes account. The reader may assume that the dominant block size is also irregularly aligned except in the case of Matrix 15. More information on the distribution of non-zeros and block size alignments appears elsewhere [16, Appendix F].

3 A Split Unaligned Block Matrix Representation

We consider an SpMV implementation which

1. Converts (or stores) the matrix A first in VBR format, allowing for padding of explicit zeros to affect the block size distribution.
2. Splits A into a sum of s terms, $A = A_1 + \cdots + A_s$, according to the distribution of block sizes observed when A is in VBR.
3. Stores each term A_i in UBCSR format, a modification of BCSR which relaxes row and column alignment.

The use of VBR is a *heuristic* for identifying block structure. Finding the maximum number of non-overlapping dense blocks in a matrix is NP-Complete [15], so there is considerable additional scope for analyzing dense block structure.

Though useful for characterizing the structure (Section 2), VBR yields poor SpMV performance. The innermost loops, which carry out multiplication by an $r \times c$ block, cannot be unrolled in the same way as BCSR because c may change from block to block within a block row. VBR performance falls well below that of alternative formats on uniprocessors [16, Chap. 2].

This section very briefly summarizes a recent technical report [19, Sec. 3].

3.1 Converting to variable block row format with fill

The default SPARSKIT CSR-to-VBR conversion routine only groups rows (or columns) when the non-zero patterns between rows (columns) match exactly. However, this convention can be too strict on some matrices in which it would be profitable to fill in zeros, just as with BCSR. We instead use the following measure of similarity between columns (or equivalently, rows). Let u^T and v^T be two row (or u, v for column) vector patterns, *i.e.*, whose non-zero elements are equal to 1. Let k_u and k_v be the number of non-zeros in u and v, respectively. Then we use $S(u^T, v^T)$ to measure the similarity between u^T and v^T:

$$S(u^T, v^T) = \frac{u^T \cdot v}{\max(k_u, k_v)} \tag{1}$$

This function is symmetric, equals 0 when u^T, v^T have no non-zeros in common, and equals 1 when u^T and v^T have identical patterns. Our partitioning procedure greedily examines rows sequentially, starting at row 0, computing S between the first row of the current block row and the candidate row. If the similarity exceeds a specified threshold, θ, the row is added to the current block row. Otherwise, the procedure starts a new block row. We partition columns similarly if the matrix pattern is non-symmetric, and otherwise use the same partition for rows and columns. We fill in explicit zeros to make the blocks conform to the partitions.

At $\theta = 0.7$, Matrix 13 has 81% of non-zeros in 3×3 blocks, instead of just 23% when $\theta = 1$ (Table 1). Moreover, the fill ratio is just 1.01: a large blocks become available at the cost of only a 1% increase in flops.

3.2 Splitting the non-zero pattern

Given the distribution of work (*i.e.*, non-zero elements) over block sizes for a matrix in VBR at a given threshold θ, given the desired number of splittings s,

and given a list of block sizes $\{r_1 \times c_1, \ldots, r_{s-1} \times c_{s-1}\}$, we greedily extract blocks from the VBR representation of A to form a splitting $A = A_1 + \cdots + A_s$ where the terms are structurally disjoint and each A_i is stored in $r_i \times c_i$ UBCSR format (see Section 3.3) and A_s is stored in CSR format. We use a greedy splitting procedure (see the full report [19] in which the order of the block sizes specified matters.

3.3 An unaligned block compressed sparse row format

We handle unaligned block rows in UBCSR by simply augmenting the usual BCSR data structure with an additional array of row indices `Arowind` such that `Arowind`$[I]$ contains the starting index of block row I. Each block-multiply is fully unrolled as it would be for BCSR.

4 Experimental Methods

The split UBCSR implementation of SpMV has the following parameters: the similarity threshold θ which controls fill, the number of splitting terms s, and the block sizes for all terms, $r_1 \times c_1, \ldots, r_s \times c_s$. Given a matrix and machine, we select these parameters by a constrained empirical search procedure, described precisely in the technical report [19]. This procedure is not completely exhaustive, but could examine up to roughly 250,000 implementations for a given matrix depending on the block size distribution. However, fewer than 10,000 were examined in all the cases in Table 1.

Nevertheless, any such an exhaustive search is generally not practical at run-time, owing to the cost of conversion (between 5–40 SpMVs [16]). While automated methods exist for selecting a block size in the BCSR case [2, 6, 16, 18], none exist for the split UBCSR case. Thus, our results (Section 5) are an empirical upper-bound on how well we expect to be able to do.

5 Results and Discussion

The split UBCSR implementation of Section 3 often improves performance relative to a traditional register-tiled BCSR implementation, as we show for the matrices in Table 1 and on four hardware platforms. When performance does not improve significantly, we may still reduce the overall storage.

We are mainly interested in comparing the best split UBCSR implementation (see Section 4) against the best BCSR implementation chsen by by exhaustively searching all block sizes up to 12×12, as done in prior work [6, 16, 18]. We also sometimes refer to the BCSR implementation as the *register blocking* implementation following the convention of earlier work. We summarize the 3 main conclusions of our technical report as follows. "Speedups" compare actual execution time, and summary statistics (minimum, maximum, and median speedups) are taken with respect to the matrices and shown in figures by end- and mid-points of an arrow. The full report shows data for each matrix and platform [19].

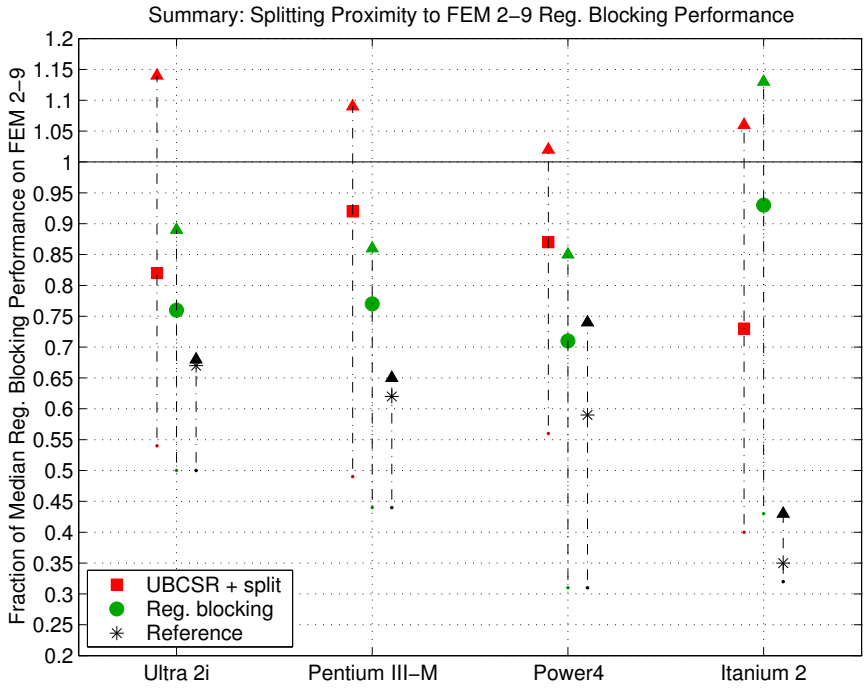

Fig. 2. Fraction of median BCSR performance over Matrices 2–9.

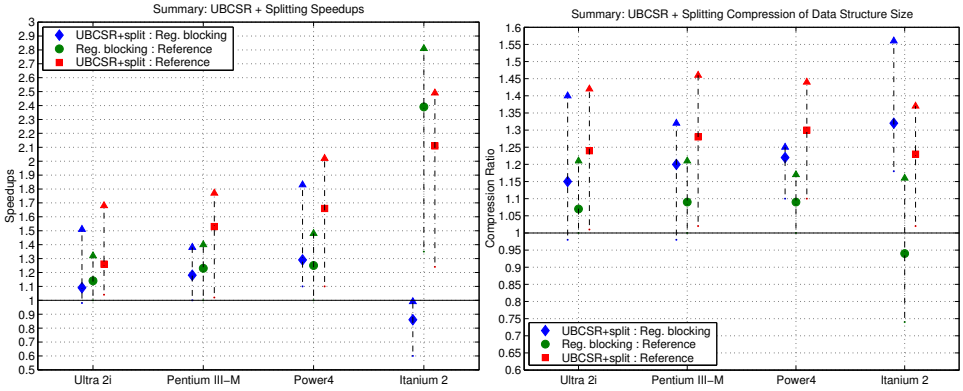

Fig. 3. Speedups and compression ratios after splitting + UBCSR storage, compared to BCSR.

Finding 1: *By relaxing block row alignment using UBCSR storage, it is possible to approach the performance seen on Matrices 2–9.* Figure 2 shows split UBCSR performance as a fraction of median BCSR performance taken over Matrices 2–9. We also show statistics for BCSR only and reference implementations. The median fraction achieved by splitting exceeds the median fraction achieved by BCSR on all but the Itanium 2. On the Pentium III-M and Power4, the median fraction of splitting exceeds the maximum of BCSR only, demonstrating the potential utility of splitting and the UBCSR format. The maximum fraction due to splitting slightly exceeds 1 on all platforms.

Finding 2: *Median speedups relative to the reference performance range from* 1.26× *(Ultra 2i) up to* 2.1× *(Itanium 2).* Figure 3 (*left*) compares the speedups of (a) splitting over BCSR (blue solid diamonds), (b) BCSR over the reference (green solid circles), and (c) splitting over the reference (red solid squares). Splitting is at least as fast as BCSR on all but the the Itanium 2 platform. Median speedups, taken over the matrices in Table 1 and measured relative to the reference performance, range from 1.26× (Ultra 2i) up to 2.1× (Itanium 2). Relative to BCSR, median speedups are relatively modest, ranging from 1.1–1.3×. However, these speedups can be as much as 1.8× faster.

Finding 3: *Splitting can lead to a significant reduction in total matrix storage.* The compression ratio of splitting over the reference is the size of the reference (CSR) data structure divided by the size of the split+UBCSR data structure. We summarize the compression ratios in Figure 3 (*right*). The median compression ratios compared to the reference are between 1.26–1.3×. Compared to BCSR, the compression ratios of splitting can be as high as 1.56×.

The asymptotic storage for CSR is roughly 1.5 doubles per non-zero when the number of integers per double is 2 [16, Chap. 3]. When abundant dense blocks exist, the storage decreases toward a lower limit of 1 double per non-zero. Relative to the reference, the median compression ratio for splitting ranges from 1.24 to 1.3, but can be as high as 1.45, which is close to this limit.

6 Related Work

The inspiration for this study is recent work on splitting by Geus and Röllin [4], Pinar and Heath [10], and Toledo [14], and the performance gap observed in prior work [6, 18, 5]. Geus and Röllin explore up to 3-way splittings based on row-aligned BCSR format. (The last splitting term in their implementations is also fixed to be CSR, as in our work.) Pinar and Heath examine 2-way splittings where the first term is $1\times c$ format and the second in 1×1. Toledo also considers 2-way splittings and block sizes up to 2×2, as well as low-level tuning techniques (*e.g.*, prefetching) to improve memory bandwidth. The main distinction of this paper is the relaxed row-alignment of UBCSR.

Split UBCSR can be combined with other techniques that improve register-level reuse and reuse of the matrix entries, including multiplication by multiple vectors where 7× speedups over CSR are possible [5, 1]. Exploiting numer-

ical symmetry with BCSR storage yields speedups as high as 2.8× over non-symmetric CSR, 2.1× over non-symmetric BCSR, and reduces storage [7].

Matrices 18–44 of the SPARSITY benchmark suite largely remain difficult, though they should be amenable to cache-level blocking in which the matrix is stored as a collection of smaller, disjoint rectangular [9, 5] (or even diagonal [13]) blocks to improve temporal access to elements of x.

Better low-level tuning of the CSR SpMV implementation may also be possible. Recent work on low-level tuning of SpMV by unroll-and-jam (Mellor-Crummey, *et al.* [8]), software pipelining (Geus and Röllin [4]), and prefetching (Toledo [14]) are promising starting points. On the vector Cray X1, just one additional permutation of rows in CSR, with no other data reorganization, yields order of magnitude improvements [3].

For additional related references, see our full report [19].

7 Conclusions and Future Work

This paper shows that it is possible to extend the classical BCSR format to handle matrices with irregularly aligned and mixed dense block substructure, thereby reducing the gap between various classes of FEM matrices. We are making this new split UBCSR data structure available in the Optimized Sparse Kernel Interface (OSKI), a library of automatically tuned sparse matrix kernels that builds on SPARSITY, an earlier prototype [17, 6].

However, our results are really only empirical bounds on what may be possible since they are based on exhaustive search over split UBCSR's tuning parameters (Section 4). We are pursuing effective and cheap heuristics for selecting these parameters. Our data already suggest the form this heuristic might take. One key component is a cheap estimator of the non-zero distributions over block sizes, which we measured in this paper exactly using VBR. This estimator would be similar to those proposed in prior work for estimating fill in the BCSR case [6, 16], and would suggest the number of splittings and candidate block sizes. Earlier heuristic models for the BCSR case, which use benchmarking data to characterize the machine-specific performance at each block size, could be extended to the UBCSR case and combined with the estimation data [2, 6, 16].

We are also pursuing combining split UBCSR with other SpMV optimizations surveyed in Section 6, including symmetry, multiple vectors, and cache blocking. In the case of cache blocking, CSR is often used as an auxiliary data structure; replacing the use of CSR with the 1×1 UBCSR data structure itself could reduce some of the row pointer overhead when the matrix is very sparse.

References

1. A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted GMRES. Technical Report CU-CS-045-03, University of Colorado, Dept. of Computer Science, January 2003.

2. A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. Technical Report ICL-UT-04-05, Innovative Computing Laboratory, University of Tennessee, Knoxville, 2005.
3. E. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed sparse row storage. In *Proceedings of the International Conference on Computational Science*, LNCS 3514, pages 99–106, Atlanta, GA, USA, May 2005. Springer.
4. R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Proceedings of the International Conference on Parallel Computing (ParCo)*, pages 308–315. Imperial College Press, 1999.
5. E.-J. Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, May 2000.
6. E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.
7. B. C. Lee, R. Vuduc, J. Demmel, and K. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *Proceedings of the International Conference on Parallel Processing*, Montreal, Canada, August 2004.
8. J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix vector multiply using unroll-and-jam. In *Proceedings of the Los Alamos Computer Science Institute Third Annual Symposium*, Santa Fe, NM, USA, October 2002.
9. R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When cache blocking sparse matrix vector multiply works and why. In *Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing*, Copenhagen, Denmark, June 2004.
10. A. Pinar and M. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of Supercomputing*, 1999.
11. K. Remington and R. Pozo. NIST Sparse BLAS: User's Guide. Technical report, NIST, 1996. `gams.nist.gov/spblas`.
12. Y. Saad. SPARSKIT: A basic toolkit for sparse matrix computations, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.
13. O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing '92*, 1992.
14. 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.
15. V. Vassilevska and A. Pinar. Finding nonoverlapping dense blocks of a sparse matrix. Technical Report LBNL-54498, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, 2004.
16. R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.
17. R. Vuduc, J. Demmel, and K. Yelick. OSKI: An interface for a self-optimizing library of sparse matrix kernels, 2005. `bebop.cs.berkeley.edu/oski`.
18. 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.
19. R. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable blocks structure. Technical Report UCRL-TR-213454, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA, July 2005.