A Library for Pattern-based Sparse Matrix Vector Multiply

Mehmet Belgin Godmar Back Calvin J. Ribbens

December 30, 2009

Abstract

Pattern-based Representation (PBR) is a novel approach to improving the performance of Sparse Matrix-Vector Multiply (SMVM) numerical kernels. Motivated by our observation that many matrices can be divided into blocks that share a small number of distinct patterns, we generate custom multiplication kernels for frequently recurring block patterns. The resulting reduction in index overhead significantly reduces memory bandwidth requirements and improves performance. Unlike existing methods, PBR requires neither detection of dense blocks nor zero filling, making it particularly advantageous for matrices that lack dense nonzero concentrations. SMVM kernels for PBR can benefit from explicit prefetching and vectorization, and are amenable to parallelization. The analysis and format conversion to PBR is implemented as a library, making it suitable for applications that generate matrices dynamically at runtime. We present sequential and parallel performance results for PBR on two current multicore architectures, which show that PBR outperforms available alternatives for the matrices to which it is applicable, and that the analysis and conversion overhead is amortized in realistic application scenarios.

1 Introduction

Sparse Matrix Vector Multiply (SMVM) is a numerical kernel that dominates the runtime of iterative solvers for systems of linear equations, which form the core of many scientific codes. Compressed representations of sparse matrices lead to a low ratio of floating point operations to memory accesses. For instance, if a matrix is represented in the commonly used compressed sparse row (CSR) format, each floating point multiply operation is accompanied by at least two memory accesses that trigger compulsory cache misses: one to retrieve the matrix element and a second one to retrieve its column index. Since accesses to main memory proceed at only a fraction of CPU speed [39], sparse solvers achieve only a small fraction of the processor's peak performance.

Blocking [18, 33, 17, 25, 26, 35] reduces this memory overhead if a matrix contains dense substructures. Instead of recording one index per matrix element, blocked representations record one index per block. However, blocking may require zero-filling, which introduces unnecessary memory and floating point operations, and it cannot be applied if a matrix does not contain dense substructures or if those structures cannot be identified.

This paper introduces a novel representation called PBR, or *Pattern-based Representation*, which reduces the index overhead for many matrices without zero-filling and without requiring the existence or identification of dense substructures. Instead, PBR exploits a simple analysis that identifies recurring block structures that share the same pattern of nonzeros within a matrix. For any pattern that covers more than a threshold number of nonzeros, PBR represents the submatrix formed by this pattern in block coordinate (BCOO) format, along with a "block code" bitmask that describes the repeated pattern. A code generator generates optimized custom kernels for each block code. Thus, PBR expresses matrix structure in terms of specialized inner loops, thereby creating locality for repeating structure via the processor's instruction cache, and reducing the amount of index data that must be fetched from memory.

We have implemented PBR and applied it to a number of matrices from different application areas. For a majority of matrices, we found that a large proportion of nonzeros is covered by PBR, with coverage equal or close to 100% in some cases. When applicable, PBR can shorten time to solution by up to $3.4\times$, with $1.4\times$ on average, when compared to CSR in a sequential implementation, and it can also improve time to solution when compared to the widely used OSKI [33] library. SMVM using PBR is amenable to three optimizations, which we have implemented: explicit prefetching, vectorization, and parallelization. First, explicit prefetching ensures that data is brought into the L1 cache and tagged with the correct temporal locality. We exhaustively search for the optimal prefetch distance for each architecture. Second, we have adapted our code generator to generate vectorized code using SSE-intrinsics, which is possible since the substructure for each block code is known at code generation time. Third, PBR is amenable to parallelization for use on multi-core architectures. PBR can be applied in situations where the repeated use of a matrix justifies the tuning effort, or where multiple matrices with identical structure are used, such as in so-called ensemble computations, e.g., model reduction [15, 4], weather modeling [12], and drug design applications [23].

We presented the idea of PBR and an initial performance evaluation in [5]. This evaluation assumed off-line matrix analysis, structure conversion, and code generation, which precluded a quantitative overhead vs. benefit analysis. In this paper, we add several significant contributions: First, we implemented a library that performs all required steps at run time. This library can be used as a drop-in replacement SMVM kernel, minimizing the number of changes users have to make to benefit from our method. Second, we validated the actual implementation costs of the analysis and structure conversion by comparing them to the expected costs based on the asymptotic complexity of our algorithms. We verified that PBR can be implemented with a linear time cost of *O(N + NNZ)* for a matrix of dimension *N* with *NNZ* nonzeros. Third, we modeled the performance of PBR for large matrices to derive a predictor for choosing a block size. This predictor yields optimal or near-optimal performance for almost all matrices in the training set we considered. Finally, we

```
// SMVM using CSR
for (i = 0; i < n; i++) {
  double y = 0.0;
  for (j = ia[i]; j < ia[i+1]; j++)
    y += aa[j] * x[ja[j]];

  y[i] = y;
}
```

Figure 1: Inner loop of SMVM kernel for CSR format.

present a comprehensive evaluation of PBR's runtime costs relative to benefits it provides. For memory bound matrices, our analysis shows that the cost of PBR are, on average, amortized after 325 steps if codes are available and 1100 steps if codes are generated.

The remainder of this paper is structured as follows: Section 2 describes our method in detail. Section 3 describes our implementation, including matrix structure analysis, blocksize detection and code generation. Section 4 provides an extensive performance analysis and discusses limitations. Section 5 compares PBR to related approaches and Section 6 concludes.

2 Pattern Based Representation

2.1 Background

SMVM computes a dense vector y that is the product of a sparse matrix A and a dense vector x: $y = Ax$. Figure 1 shows the basic form of an SMVM kernel's inner loop if a matrix is represented in CSR format. The matrix nonzeros are stored in a continuous array `aa`. Two indices record the structure of the matrix: an index `ja[j]` records the column index of the j^{th} nonzero element, and a row pointer `ia[i]` records at which matrix element row i begins. Each pair of multiply-add floating point operations is thus accompanied by two memory accesses: one to load the matrix element `aa[j]`, and another one to retrieve the column index `ja[j]` for each matrix element. Neither of these values is reused within the same invocation of the kernel. Therefore, even if there is maximum reuse of x, the performance is limited by the speed with which aa, ja, and ia can be fetched from memory. Except when SMVM is repeatedly called for small matrices that fit in the cache, each of these accesses will encounter compulsory or capacity misses.

2.2 Exploiting Recurring Patterns

Because we cannot reduce the number of nonzero values that must be read from main memory, our approach focuses on reducing the size of the index data structure. We identify blocks of recurring patterns and generate custom code for those patterns. As a result, fewer indices are needed since a pair of coordinates expresses the coordinates of each block, rather than needing indices for each nonzero within a block. The microstructure of each block is expressed in the machine code of the inner loop that iterates over all blocks of identical structure within a matrix.

We use a simple analysis to identify repeating patterns. Given a block size $R \times C$, we divide a $m \times n$ matrix into a grid of $\lceil \frac{m}{R} \rceil \times \lceil \frac{n}{C} \rceil$ rectangular blocks and count how often each of the possible $2^{R \times C}$ patterns occurs. We represent the aggregate submatrix for each block pattern by recording block coordinates in COO format, along with a "block code," which is a bit vector of size $R \times C$ that encodes the nonzero micro-pattern.

We exclude two types of blocks. First, we exclude blocks with patterns that include less than three nonzeros, because such patterns would yield little or no reduction in terms of index overhead. Second, we exclude blocks whose patterns do not occur frequently enough to cover a significant number of nonzero elements, because the overhead of dispatching to the kernel specific to their block code may not be amortized. We empirically set this threshold as one thousand. Our analysis aggregates nonzeros that belong to excluded blocks in a remainder matrix, which is stored using conventional CSR representation.

Figure 2 shows an example 12×12 matrix in which three recurring patterns occur when using a block size of 4×4. 7 of the matrix's 9 blocks exhibit recurring patterns with more than 2 nonzeros. 3 remainder elements (shown in red) are located within blocks that have 2 or fewer nonzeros. For this example, PBR reduces index overhead by 35% when compared to CSR, as demonstrated in Figure 3. Figure 4 shows how the original matrix is split into a sum of submatrices and a remainder.

2.3 Block Size Selection and Nonzero Coverage

To benefit from PBR, we must choose a blocksize that yields sufficient nonzero coverage relative to the number of nonzeros assigned to the remainder matrix. For a given matrix, the nonzero coverage depends on the choice of the number of rows R and columns C within each block and the described cutoff criteria. Depending on the structure of a matrix, different values of R and C will yield different degrees of coverage. For our experiments, we chose square block sizes $R = C = 2, 3, 4, \ldots, 8$ and performed the analysis described in Section 2.2 for each size. We found that the block size that yields the largest coverage does not always yield the best performance, because different block sizes lead to different decompositions of the

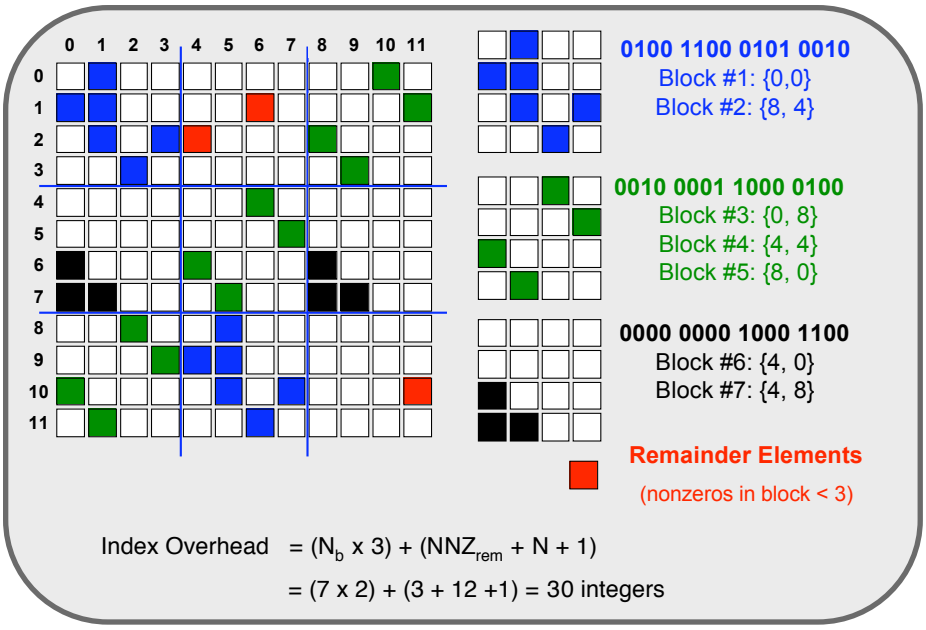

Figure 2: PBR representation of a square matrix of dimension $N = 12$ with $NNZ = 33$ nonzero elements. This matrix is composed of 3 recurring 4x4 block patterns and two remainder blocks with less than three nonzeros. $N_b = 7$ is the number of blocks with a shared pattern, $NNZ_{rem} = 3$ is the number of remainder nonzeros.

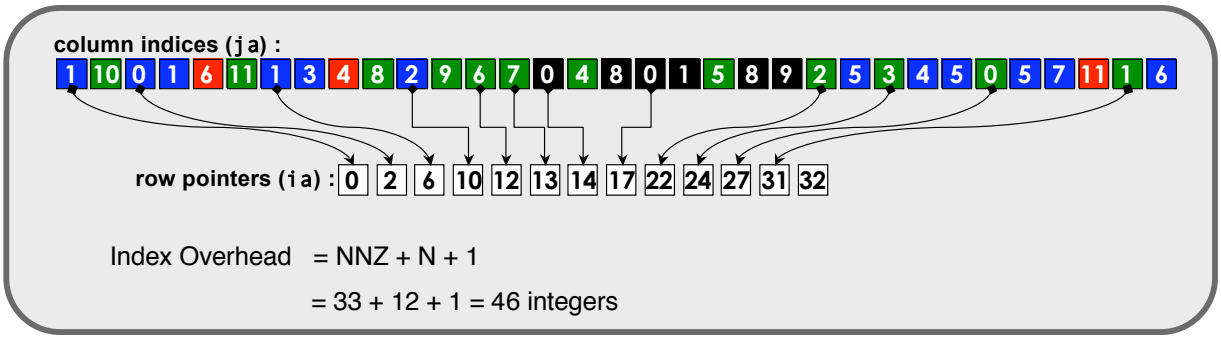

Figure 3: CSR representation of the matrix shown in Figure 2. Compared to CSR, PBR reduces the index overhead for this matrix by 35%.

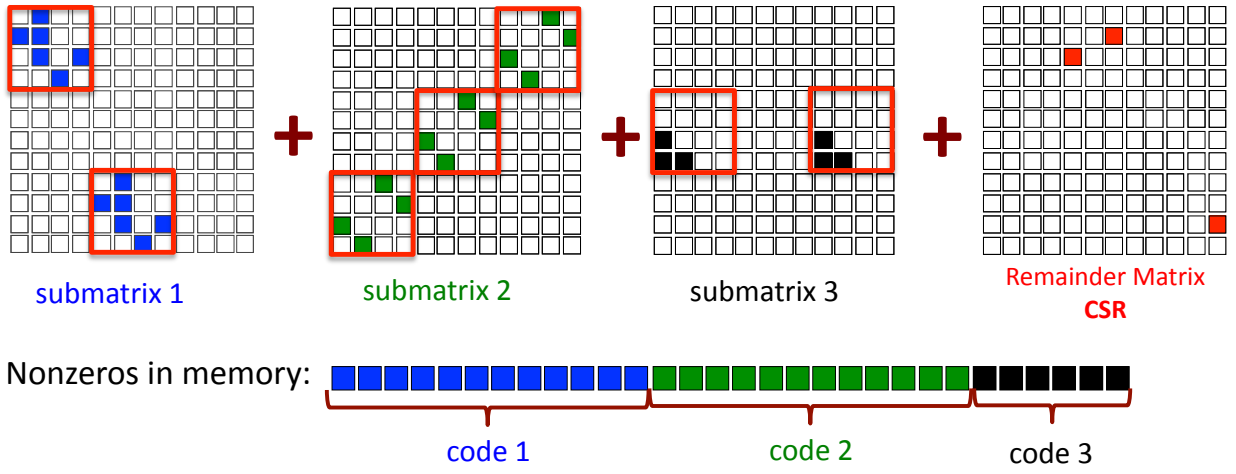

Figure 4: Splitting of the 12x12 matrix illustrated in Figure 2.

matrix, which results in different memory access patterns for the x and y vectors. We describe a model and a predictor that consider these factors in Section 3.2.

We selected a set of 53 matrices to evaluate the applicablity and potential of PBR. We included all obtainable matrices from two sets that were previously used for sequential and parallel evaluation of SMVM optimizations by Im et al. [18] and by Williams et al. [37]. We also added a number of larger matrices from the University of Florida repository [8] to include matrices of widely varying patterns, stemming from a variety of application areas. The properties of the matrices we considered can be found in Table 1. The "Opt. Blocksize" column list the block size that yielded the best performance for the Intel Harpertown (HPT) and AMD Opteron (OPT) architectures we benchmarked. The columns "PBR Coverage," "Index Savings," and "# of Patterns" list the coverage, index overhead reduction, and the number of patterns for the optimal block size for the respective architecture. This table excludes the *jpwh_991* matrix, which is a small matrix that includes no shared patterns that satisfy PBR criteria for any of the blocksizes.

Figure 5 provides a histogram that summarizes the achieved nonzero coverage, based on the block size that yielded the best performance on both architectures. For 56% (HPT) and 58% (OPT) of matrices, PBR encodes over 80% of nonzeros; coverage is 40% or less for only 19% (HPT) and 26% (OPT) of matrices. Table 2 displays how often each block size yielded the best performance. We do not consider block sizes larger than 8 because good coverage can be obtained from block sizes less than or equal to 8, and because larger block sizes tend to increase the number of patterns, making it less likely that individual patterns meet the cut-off criteria.

Our analysis appears to capture the underlying regularity in the nonzero structure of many types of ma-

			PBR Coverage (%)		Index Savings (%)		Opt. Blocksize		# of Patterns	
Matrix Name	**N**	**NNZ**	**HPT**	**OPT**	**HPT**	**OPT**	**HPT**	**OPT**	**HPT**	**OPT**
orsreg_1	2205	14133	97.03	97.03	66.6	66.6	7	7	2	2
sherman3	5005	20033	77.34	77.34	42.9	42.9	5	5	2	2
sherman5	3312	20793	100	100	54.8	54.8	3	3	4	4
saylr4	3564	22316	81.88	85.87	43.8	34.1	6	3	5	2
mcfe	765	24382	37.8	37.8	32.1	32.1	4	4	1	1
lns_3937	3937	25407	40.86	38.38	29	14	8	2	5	3
lnsp3937	3937	25407	74.81	74.81	43.7	43.7	5	5	5	5
gemat11	4929	33185	13.38	13.38	5.82	5.82	2	2	1	1
bayer02	13935	63679	5.53	5.53	2.27	2.27	2	2	1	1
orani678	2529	90158	70.68	70.68	56.6	56.6	4	4	12	12
rdist1	4134	94408	71.02	71.02	63.4	63.4	6	6	8	8
bayer10	13436	94926	39.47	39.47	17.3	17.3	2	2	1	1
memplus	17758	126150	55.16	55.16	41.1	41.1	8	8	18	18
wang3	26064	177168	96.08	96.08	63	63	6	6	2	2
wang4	26068	177196	94.72	94.72	62.1	62.1	6	6	2	2
coater2	9540	207308	0.61	0.61	0.29	0.29	5	5	1	1
onetone2	36057	227628	45.72	3.5	20.1	2.15	3	8	18	3
lhr10	10672	232633	76.62	91.02	67.9	63	8	4	50	30
raefsky1	3242	294276	91.5	97.38	82	75.7	6	4	31	18
goodwin	7320	324784	75.42	75.42	52.6	52.6	4	4	44	44
pwt0	36519	326107	53.6	44.93	41.8	19.5	8	2	57	5
shyy161	76480	329762	86.33	88.89	45.1	52.2	4	5	6	6
vibrobox	12328	342828	6.42	6.42	3.1	3.1	2	2	1	1
af23560	23560	484256	100	100	87.4	87.4	8	8	7	7
finan512	74752	596992	79.66	31.39	44.9	10.8	6	2	74	4
scircuit	170998	958936	46.16	28.44	24	11.7	3	2	24	5
crystk02	13965	968583	99.91	99.64	76.6	88.8	3	6	1	10
rim	22560	1014951	88.95	87.62	61.2	72.9	4	6	111	220
ex11	16614	1096948	93.56	93.56	65	65	3	3	36	36
mac_econ_fwd500	206500	1273389	5.18	5.18	1.59	1.59	2	2	5	5
raefsky4	19779	1328611	95.53	95.53	66.9	66.9	3	3	28	28
bcsstk35	30237	1450163	98.56	98.56	74.2	74.2	3	3	17	17
raefsky3	21200	1488768	100	100	95.5	49.3	8	2	1	1
av41092	41092	1683902	84.81	82.61	55.6	60.1	4	6	59	236
venkat01	62424	1717792	100	100	85.8	48.2	8	2	15	1
crystk03	24696	1751178	99.95	99.95	76.7	76.7	3	3	1	1
qcd5_4	49152	1916928	100	100	75.8	75.8	3	3	1	1
mc2depi	525825	2100225	37.28	37.28	9.94	9.94	2	2	2	2
rma10	46835	2374001	95.84	95.84	63.7	63.7	3	3	33	33
nasasrb	54870	2677324	99.27	99.27	73.6	73.6	3	3	22	22
ct20stif	52329	2698463	84.1	93.39	72.7	63.8	6	3	227	47
webbase-1M	1000005	3105536	10.31	10.31	3.76	3.76	2	2	5	5
3dtube	45330	3213618	99.78	99.78	76.2	76.2	3	3	14	14
laminar_duct3D	67173	3833077	100	100	72.4	72.4	3	3	12	12
dense2	2000	4000000	100	100	94.4	77.7	6	3	3	3
cant	62451	4007383	99.97	99.97	74.9	74.9	3	3	19	19
pkustk04	55590	4218660	100	100	76.8	76.8	3	3	1	1
pdb1HYS	36417	4344765	99.87	99.98	90.7	77.1	6	3	15	1
Si34H36	97569	5156379	71.11	71.11	43	43	3	3	68	68
consph	83334	6010480	99.85	99.85	75.4	75.4	3	3	13	13
shipsec1	140874	7813404	100	100	92.8	76.4	6	3	1	1
pwtk	217918	11634424	97.65	97.65	71.6	71.6	3	3	34	34

Table 1: Matrices used in the evaluation of PBR. The coverage, index overhead savings, and number of patterns are based on the block size that yielded the best performance on the Intel Harpertown (HPT) and AMD Opteron (OPT) architectures.

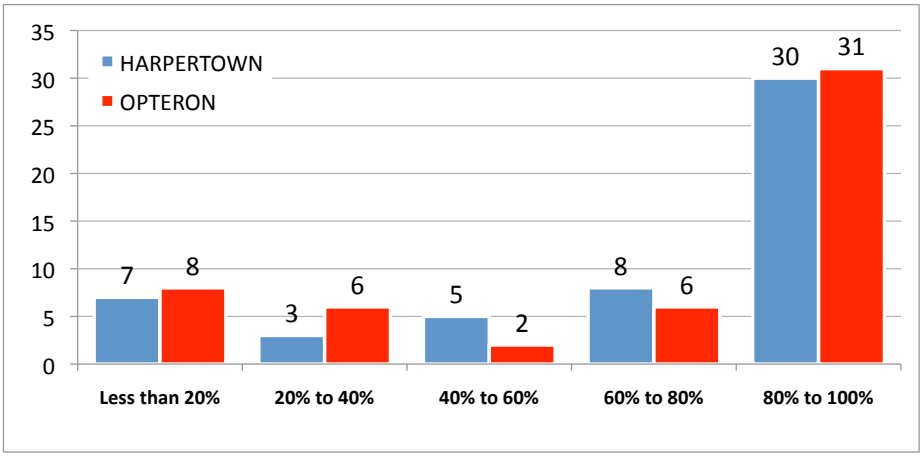

Figure 5: PBR nonzero coverage for matrix set shown in Table 1. Values are tabulated for the block size that yielded highest performance on each of the architectures shown.

	2×2	3×3	4×4	5×5	6×6	7×7	8×8
Harpertown	7	18	6	3	10	1	7
Opteron	13	20	5	4	6	1	3

Table 2: Frequency with which each block size yielded maximum performance for matrix set shown in Table 1.

	< 10	< 25	< 50	< 100	> 100
Harpertown	27	12	7	4	2
Opteron	33	9	7	1	2

Table 3: Number of distinct block codes for matrix set used in Table 1, aggregated for block sizes that yielded highest performance.

trices. Only matrices that are nearly random, such as those that model the relationship between hyperlinked web pages, tend to yield lower coverage.

Table 3 shows the distribution of the number of distinct patterns that satisfied our inclusion criteria. This table shows that the high degree of nonzero coverage we observe requires only a small subset of all theoretically possible $2^{R \times C}$ patterns, thus making it feasible to generate code for all qualifying patterns.

3 PBR Library

The PBR library provides a conversion routine that implements the matrix analysis, structure conversion, and any necessary code generation. Users provide the input matrix in CSR format, thus making PBR drop-in compatible for any codes exploiting this widely used format. The conversion routine returns an opaque handle that is used in subsequent SMVM operations. The handle refers to an internal data structure that encapsulates matrix-specific information (such as dimension and sparsity), PBR-specific information (such as blocksize, number of recurring patters, nonzero coverage and names and paths of generated code and compiled object files), the data structures to hold the converted matrix itself (including nonzero values, block indices, list of patterns and their occurrence) and the remainder matrix in CSR format. An analyzed matrix structure can be saved to and restored from disk, allowing re-use of the analysis results for structurally identical matrices.

The library is callable from C code, but it is implemented in portable C++ using several high-performance container and utility classes provided by the STL [2] and Boost [1] libraries. The custom SMVM kernels that implement the multiplication step are generated as C code and compiled with an optimizing C compiler.

PBR conversion involves the following steps: analyzing the matrix structure to find recurring patterns, selection of a block size, structural conversion, code generation, compilation (if necessary), and loading. The following sections describe these steps.

3.1 Matrix Structure Analysis

The structure analysis determines which block patterns occur in the matrix and how often. To avoid reading the matrix index structure multiple times, we analyze all block sizes from 2×2 to 8×8 in a single pass, as illustrated in Figure 6. We divide the matrix into stripes, each stripe comprising $lcm(2, 3, ..., 8) = 840$ rows. For each block size, we use an instance of Boost's unordered hash map to store the indices of those blocks

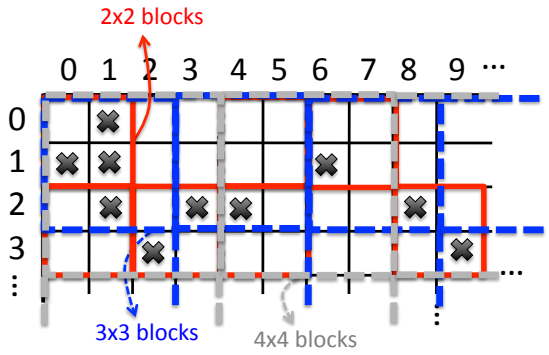

Figure 6: Each nonzero's contribution to the block patterns for each considered block size choice is recorded in a single pass.

that contain at least one nonzero element in the stripe. We cumulatively update these blocks as we process the input matrix index array. To provide for efficient iteration over the blocks we additionally store their column indices in a linear STL vector. We optimized this step further by using a custom memory allocator based on GNU's obstack implementation, which allows for fast allocation and deallocation of these temporary data structures. After all nonzeros within a stripe are accounted for, all encountered patterns and their frequency are tabulated in a separate hash map that is keyed by bitsets encoding each pattern. Since all lookups are performed with hash maps that provide $O(1)$ average lookup time, the asymptotic time complexity of these analysis steps is linear in the number of nonzeros contained in the matrix. Some operations, such as the initialization and teardown of the hash maps containing the indices of the nonzeros within each stripe, are performed once for each stripe. These operations incur an asymptotic complexity that is linear in the number of matrix rows N.

Our actual implementation collects pattern statistics only for the 8×8, 7×7, 6×6, and 5×5 block sizes. The recurring patterns and their frequencies for blocks of size 4×4 and 2×2 are derived from the recurring 8×8 patterns by examining subsets of the bit patterns of each 8×8 pattern, as shown in Figure 7. Similarly, 3×3 patterns and frequency are derived from their encapsulating 6×6 blocks. This derivation reduces the cost of our analysis substantially, because it is done only once for each distinct 8×8 and 6×6 pattern.

The result of the analysis step provides, for each of the considered block sizes, the number and kind of distinct recurring patterns. We exclude patterns for which the product of pattern frequency and number of bits contained in the pattern does not meet our threshold, and we exclude patterns with less than three nonzeros. The analysis step does not record the block indices of the blocks for each recurring pattern.

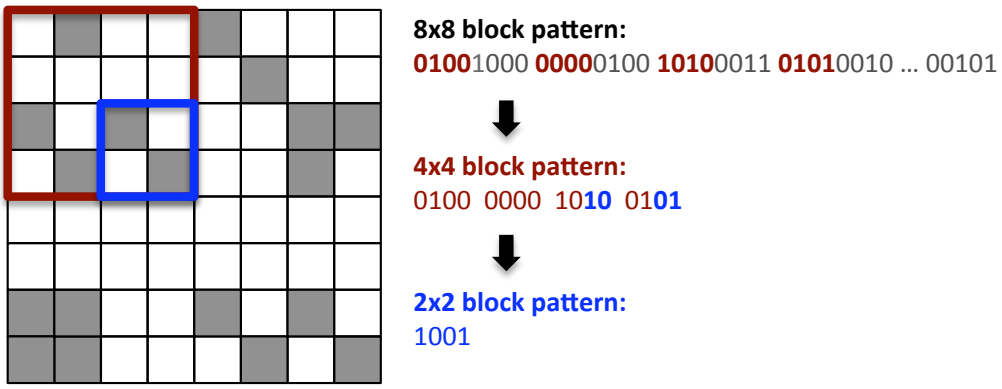

Figure 7: Derivation of 4×4 and 2×2 block patterns using the patterns of their parent 8×8 block.

3.2 Block Size Selection

Choosing a block size requires predicting which block size will yield the best performance. We developed a simple multiple linear regression model to predict the performance of PBR for a decomposition resulting from each potential choice of block size. For sufficiently large matrices, we expect that PBR's execution time is dominated by memory accesses, therefore our model includes three variables that capture the memory accesses performed during the multiplication. The values of these input variables can be derived from statistics collected during the analysis step.

- The number of bytes fetched from memory while accessing the covered nonzero values, computed as the product of $coverage \times NNZ \times$ `sizeof(double)`, plus the number of bytes comprising the block index array, which is $\sum_{i \in P} 2 \times freq(P_i) \times$ `sizeof(int)` for the set of qualifying patterns P.
- An approximation of the number of accesses to the x and y vectors while multiplying the submatrix for each included block pattern. Since each submatrix may cover all rows and columns, we approximate this number as the product of the matrix dimension and the number of patterns: $N \times |P|$.
- The third model variable is the number of writes to y, which can be derived for each block pattern from its structure. This variable is needed because unlike CSR, which writes each element of y exactly once, PBR may read and write each y element multiple times.

Our model must also predict the performance of the remainder matrix, which is kept in CSR format. We modeled CSR's performance with a separate multiple regression model that is based on matrix dimension

and number of nonzeros. Section 4 discusses the computation of parameter estimates and the fit of these models for our training set and the architectures we considered.

3.3 Structure Conversion

The structure conversion step creates the block indices and arranges the nonzero values in PBR format for the selected blocksize. We keep the (row, col) block indices and the matrix nonzero values in two one-dimensional contiguous arrays. As depicted in Figure 4, the nonzeros and indices of all blocks that belong to the same pattern are stored in contiguous slices of these arrays, which guarantees spatial locality in the inner loop of each kernel. Because the number of blocks for each recurring pattern has been determined during the analysis step, the start and end locations of these slices can be precomputed in a single pass over all patterns. Thus, nonzeros and block indices can be copied directly to their target destinations by maintaining pointers to the current nonzero and block index offsets within each pattern's slices.

3.4 Code Generation

The code generation step generates custom C functions for all qualifying patterns, stores them to disk, and invokes the C compiler to create shared .so object files. Each .so file is dynamically linked into the process's address space. A pointer to the C function it contains is added to an array of function pointers. The outer loop of the SMVM routine iterates over this array and invokes the generated block multiplication functions.

The shared object modules are created on demand and stored in a repository on disk. Since object modules are specific to only the block pattern and size, they can be reused both across multiple uses of the PBR library on the same matrix as well as across multiple matrices that contain the same pattern. We expect caching to be particularly beneficial for application domains that repeatedly generate similarly structured matrices, e.g., k-stencil FEM methods. Optionally, the code cache can be primed by generating and compiling all possible patterns for blocksizes 2×2, 3×3 and 4×4, which would pay for these code generation costs upfront and avoid any cost for individual matrices when these block sizes are chosen.

Our code cache is designed to hold modules for different target architectures and with different optimization options (e.g., SSE) simultaneously, thus allowing it to be shared by multiple machines on a network. If needed, the size of the code cache repository could be managed by a periodically scheduled cronjob that deletes files that have not been accessed for a certain time period, which is similar to the strategy used by many Unix installations to purge stale files from the `/tmp` directory.

3.5 Explicit Prefetching

We optimized our baseline implementation by applying explicit prefetching for the matrix elements. Although the streaming pattern of these accesses would ordinarily prevent gains from prefetching, prior work on SMVM optimizations has shown [37] that explicit prefetching is beneficial on SSE-enabled x86-based architectures, which are used by 87.6% of the machines in the November 2008 Top-500 list [24]. Prefetching can place data directly into the L1 cache and labels cache entries with the correct temporal locality, thus allowing eviction in preference to other data such as elements of the x and y vectors, which may be reused. Explicit prefetching is implemented via the GCC `__builtin_prefetch()` compiler intrinsic, which results in the emission of `prefetch*` SSE instructions. Our code allows varying the prefetch distance at runtime, which enables dynamic tuning of the prefetch distance on a per matrix basis.

3.6 Vectorization

We adapted our code generator to emit vectorized code that exploits SSE2/3 SIMD intrinsics. This instruction set allows simultaneous operations on a vector of two double values via the `__mm128d` data type, which is mapped to a 128 bit SSE register. For simplicity, we vectorize only blocks with even sizes. Each $R \times C$ block is divided into $R/2 \times C/2$ 2×2 subblocks. We allocate $C/2$ and $R/2$ vector variables for each (x_i, x_{i+1}) and (y_j, y_{j+1}) pair. We allocate 1 vector variable for aa elements if a subblock has 1 or 2 nonzeros, and 2 variables if it has 3 or 4 nonzero elements, independent of where those nonzeros are located within the subblock. To minimize the number of shuffle operations that are required if aa elements are not located in the order in which they are needed, we store the aa elements in zigzag order in memory. Figure 8 shows the code our generator produces for the pattern $\begin{smallmatrix}\circ & \bullet \\ \bullet & \circ\end{smallmatrix}$.

We introduce padding to maximize the use of 128bit aligned load instructions provided by the SSE2 instruction set. The first nonzero element in each block is aligned at a 16 byte boundary. In addition, we implemented two different alignment strategies within a block. Our first strategy uses aligned loads only when it is known that the elements to be loaded have the correct alignment, based on the alignment of the initial nonzero within the block. If the number of nonzero elements is odd (1 or 3), we use 64-bit double load instructions and set the unused element within the vector to zero. Our second strategy stores an additional zero in memory whenever the number of nonzeros within a 2×2 subblock is odd, thus allowing the use of aligned load instructions throughout. Our performance evaluation reports the results for the strategy that

```
/*
 * SSE custom kernel for 2x2 block code
[ ][x]
[x][ ]
*/
__attribute__ ((noinline)) void
pbr_multiply_6(
    real_t *y, real_t *x, real_t *Abase,
    int nblocks, struct _nnzblock *nnzblock)
{
  int i;
  for (i = 0; i < nblocks; i++) {
    int topleftrow = nnzblock[i].topR;
    int topleftcol = nnzblock[i].topC;
    // _r76 = [_a1, _a0]
    __m128d _r76 = _mm_load_pd(Abase);
    // _r75 = [_y1, _y0]
    __m128d _r75 = _mm_load_pd(y + topleftrow);
    // _r73 = [_x1, _x0]
    __m128d _r73 = _mm_load_pd(x + topleftcol);
    // _r77 = [_x0, _x1]
    __m128d _r77 = _mm_shuffle_pd(_r73, _r73, 1);
    // _r78 = [_a1 * _x0, _a0 * _x1]
    __m128d _r78 = _mm_mul_pd(_r76, _r77);
    // _r79 = [_y1 + _a1 * _x0, _y0 + _a0 * _x1]
    __m128d _r79 = _mm_add_pd(_r75, _r78);
    _mm_store_pd(y + topleftrow + 0, _r79);
    Abase += 2;
  }
}
```

Figure 8: SSE3 custom kernel for PBR format.

yielded the best performance.

3.7 Parallelization

SMVM using a PBR representation contains inherent data parallelism since all floating point multiply operations are independent from each other. Moreover, since the distribution of repeating blocks, as well as the number of nonzeros within each block, is known post analysis, the workload can be distributed across threads easily. However, unlike in parallel CSR-based SMVM implementations, we cannot row-partition accesses to the y vector across threads. To avoid expensive synchronization, each thread maintains a private y_i vector that represents the partial product $A_i x$ of the submatrices for which it is responsible. Subsequently, a reduction step sums up the partial sums to obtain the final result. This reduction step is itself easily parallelized.

4 Evaluation

Our evaluation contains multiple parts: first, we demonstrate how PBR-based SMVM shortens time to solution, or overall runtime, in both sequential and parallel environments, relative to readily available alternatives. Second, we demonstrate the relative impact of the prefetching and vectorization optimizations we applied to PBR. Finally, we evaluate the runtime costs associated with the PBR implementation in comparison to benefits by PBR. For all experiments, we used the matrix set in Table 1, which was introduced in Section 2.3.

4.1 Methodology

We used two x86-based architectures for our evaluation: First architecture is a 2.8Ghz 2-socket quadcore (8-core total) Intel Xeon Harpertown 5400 with 12 MB L2 cache per socket (each core pair sharing a 6MB cache,) and 8GB of RAM. Second architecture is a 2.5 GHz 2-socket quadcode (8-core total) AMD Opteron 2380 with 512KB L2 and 128KB L1 caches per core, a 6MB shared L3 cache per socket, and 16GB of RAM. We use the GCC compiler version 4.1.2 on the Harpertown and version 4.3.2 on the Opteron with optimization flags: `-O2, -funroll-loops, -mfpmath=sse, -msse, -mtune` and `-march` using proper values for each architecture.

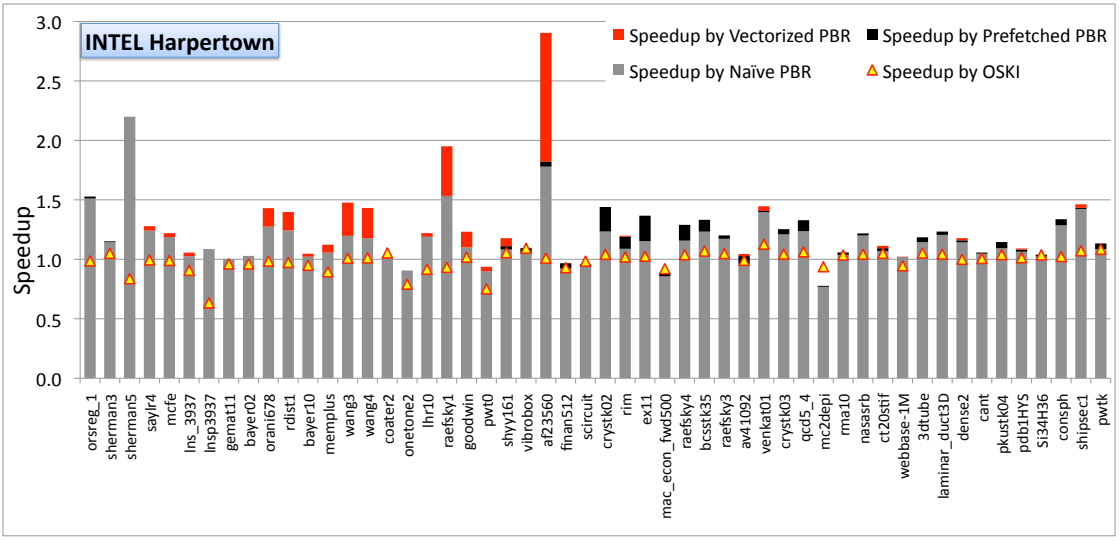

Figure 9: Sequential speedup relative to naïve CSR by (1) plain, (2) prefetched, (3) vectorized PBR and (4) OSKI on Intel Harpertown.

4.2 Sequential Performance

Figures 9 and 10 compare the performance of unoptimized (plain) sequential PBR to the performance of PBR with prefetching and PBR with vectorization on Harpertown and Opteron, respectively. Speedup results are shown relative to the naïve CSR performance.

For matrices with 80% or more nonzero coverage, which account for at least 30 of the 53 matrices on both architectures, PBR provides a maximum speedup of 3.41 with an average of 1.53 on the Harpertown and a maximum speedup of 2.32 with an average of 1.64 on the Opteron.

The relative contribution of prefetching and vectorization is shown as zero if prefetching or vectorization did not improve performance. For a few matrices, these optimizations yielded slight slowdown. On Harpertown, small matrices benefit particularly from vectorization, whereas the benefit of prefetching is more pronounced on the Opteron.

These charts also contain the results we obtained using OSKI 1.0.1h [33], a widely used optimization library. We used OSKI's aggressive tuning option without providing structural hints. OSKI is consistently slower than PBR, and even provides little or no speedup compared to naïve CSR on the architectures considered, which is consistent with earlier results obtained by others [37].

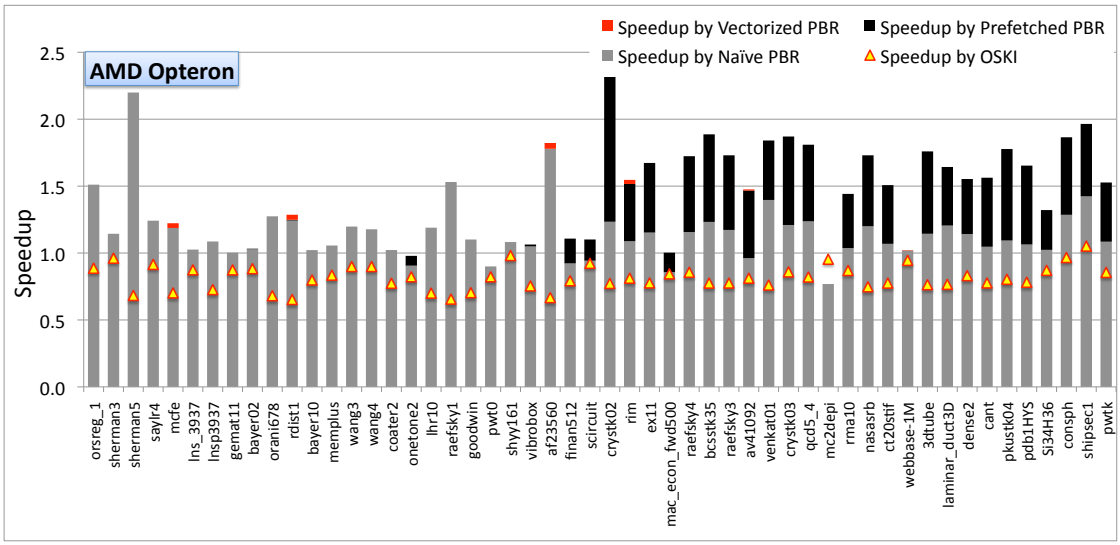

Figure 10: Sequential speedup relative to naïve CSR by (1) plain, (2) prefetched, (3) vectorized PBR and (4) OSKI on AMD Opteron.

4.3 Parallel Performance

We built a basic thread pool implementation on top of Linux's PThreads API. To avoid the potentially random placement of threads onto cores by the OS scheduler, we used the *Portable Linux Processor Affinity (PLPA)* interface by the OpenMPI project [14] to explicitly set CPU affinity for each thread, which forces a fixed mapping of threads to cores. When using 2 threads, we placed threads on neighboring cores that share the same L2 cache on Harpertown and the same L3 cache on Opteron. When using 4 threads, we placed them onto the same socket, thus sharing a single front-side memory bus.

We compared our parallel implementation of PBR to a parallel implementation of naïve CSR. We parallelized CSR in a manner that partitions the matrices' rows such that each thread operates on roughly the same number of nonzero elements. We used the same thread pool implementation as for PBR.

Figure 11 shows the results for 2, 4, and 8 threads on Intel Harpertown (top) and AMD Opteron (bottom). We expressed overall runtime of parallelized PBR relative to parallelized CSR. For this discussion, we consider only the subset of matrices that provided more than 80% coverage and that yielded at least $1.1\times$ sequential speedup. A value greater than 1.0 indicates that the use of PBR is beneficial for a given number of threads. These results indicate that PBR retains its performance advantage over CSR for 2 and 4 threads for most matrices. PBR loses its performance advantage when using 8 threads for small and medium-sized

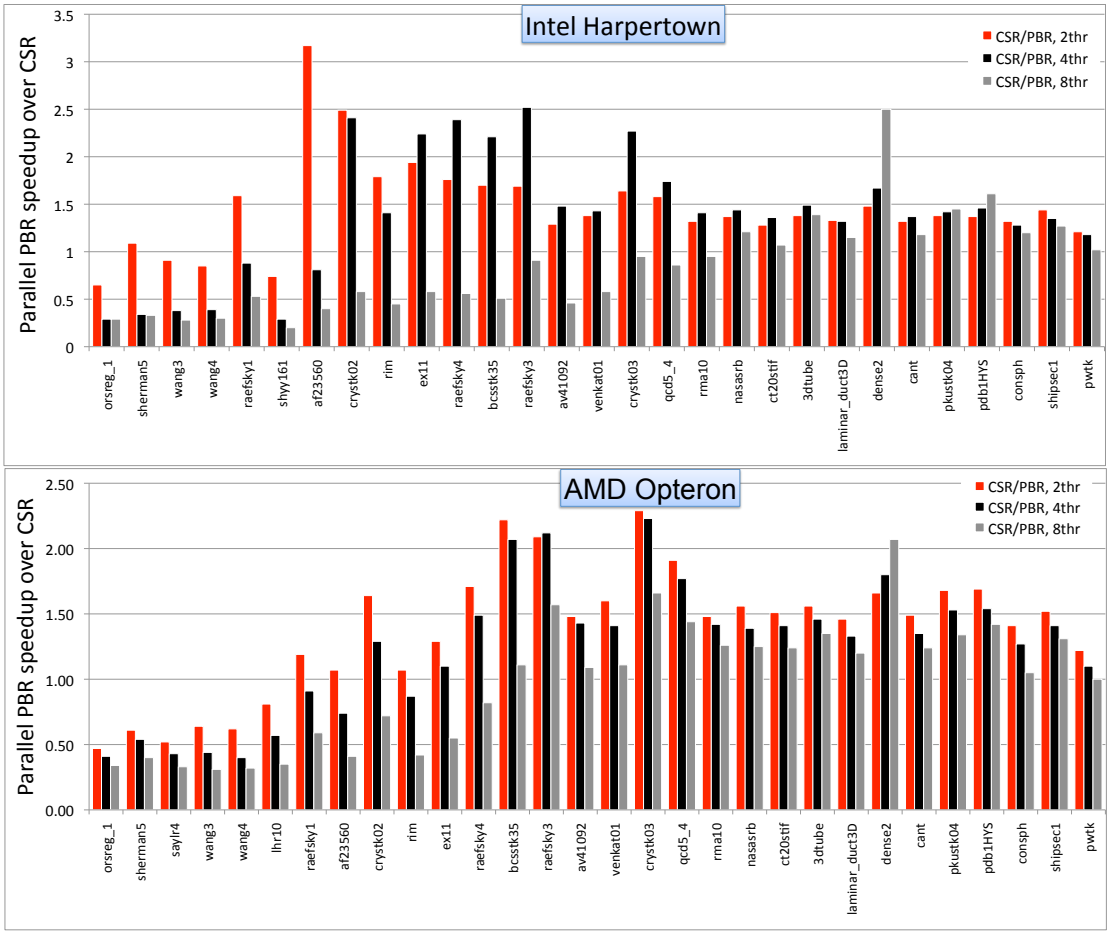

Figure 11: Parallel performance of PBR relative to parallelized CSR using 2, 4, and 8 threads on Harpertown (top) and Opteron (bottom). Only matrices with more than 80% coverage and sequential PBR speedup greater than 1.1 are included. Matrices are sorted by their size, starting from the smallest on the left.

Harpertown	2 core	4 core	8 core
PBR	1.05	1.17	2.35
CSR	1.10	1.17	2.70

Opteron	2 core	4 core	8 core
PBR	1.58	2.17	2.23
CSR	1.63	2.39	3.02

Table 4: Average of parallel speedups given in Figure 12.

matrices. As has been observed by others [37], such matrices benefit from superlinear speedup under CSR.

Figure 12 depicts parallel speedup by CSR and PBR relative to their respective sequential implementation, focusing on large matrices whose performance is memory bound. The averages of the speedups in this figure are summarized in Table 4. The limited speedup on these multicore architectures reflects the contention for memory bandwidth among cores. For instance, on Harpertown, significant speedup is realized only when switching from 4 cores to 8, which uses two sockets instead of one and doubles the available memory bandwidth.

4.4 Overhead Analysis

As with any optimization method, PBR's overhead must be evaluated relative to the benefits it provides. We relate PBR's overhead to its performance benefits over the CSR method in the context of iterative solvers. The break-even column (*BEP*) in Table 5 shows how many iterations would be needed to compensate for PBR's overhead, which includes matrix analysis and structure conversion, but excludes the code generation/compilation. Table 5 is sorted by matrix size, with smaller matrices in the top half. Since small matrices benefit less from PBR, their break-even point is generally higher. The break-even point reduces significantly with increasing matrix size.

Since the break-even point depends on the achieved speedup for a given matrix, we also report as a second point of comparison how many CSR iterations could be performed in the time it takes to analyze and convert the matrix to PBR. These numbers are shown in column *CSR itr*; they provide a cutoff value below which PBR cannot be amortized.

To ensure that the overhead measured by our evaluation accurately reflects the inherent complexity of our method, we validated that our implementation in fact exhibits the asymptotic complexity we predicted. During this validation process, we eliminated several lurking quadratic complexities in our implementation by applying the techniques described in Section 3.1. We fit a multiple linear regression model using the

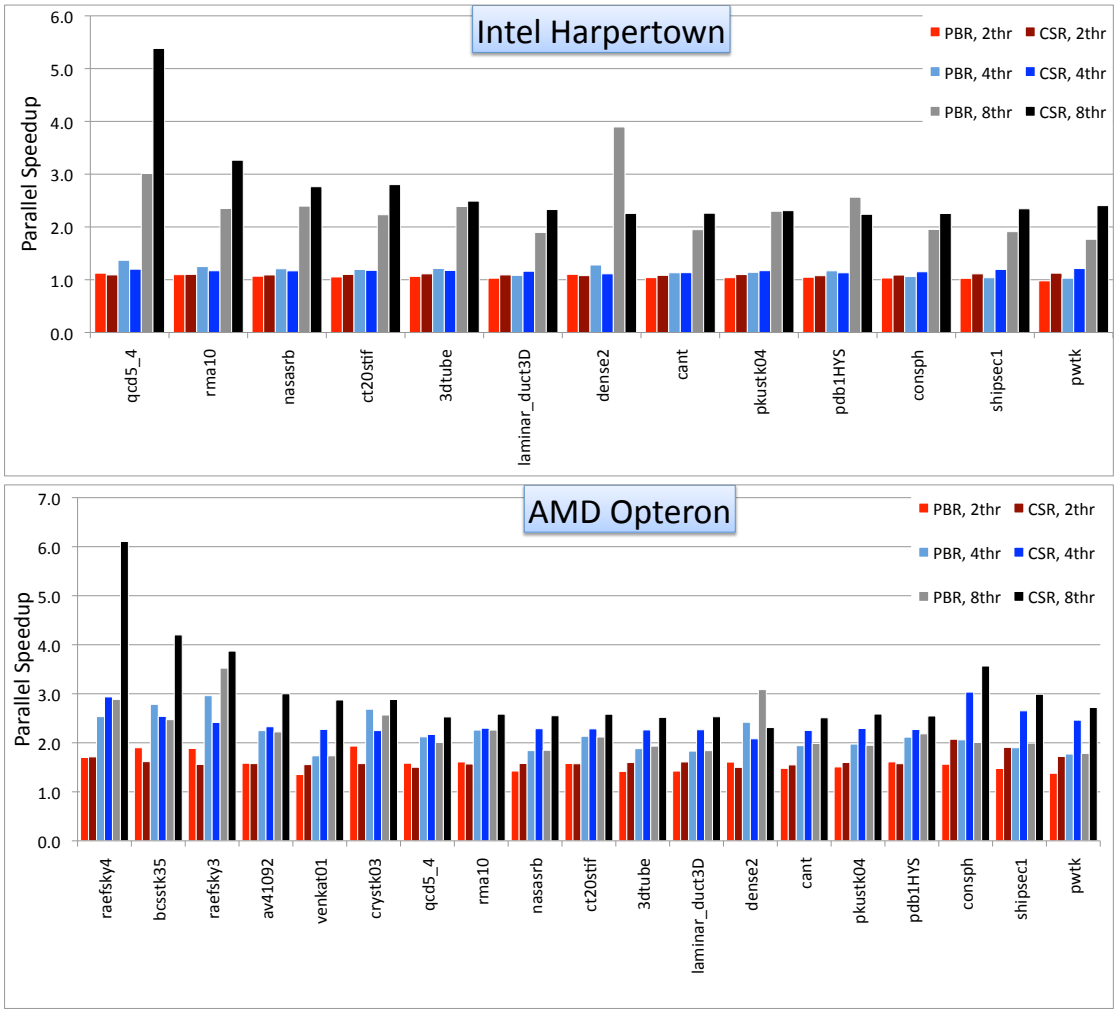

Figure 12: Parallel speedup of CSR and PBR using 2, 4, and 8 threads on Harpertown (top) and Opteron (bottom). Performance data is shown for large matrices with more than 80% coverage and sequential PBR speedup greater than 1.1.

Intel Harpertown						AMD Opteron					
Matrix	**BS**	**Analysis**	**Structure**	**BEP**	**CSR itr**	**Matrix**	**BS**	**Analysis**	**Structure**	**BEP**	**CSR itr**
orsreg_1	7	2.99E-03	1.69E-03	1046	160	orsreg_1	7	4.60E-03	2.33E-03	558	189
sherman3	5	4.42E-03	3.15E-03	1318	128	sherman3	5	7.04E-03	4.42E-03	1359	170
sherman5	3	3.44E-03	2.73E-03	295	135	sherman5	3	5.58E-03	3.72E-03	311	169
mcfe	4	6.48E-03	4.79E-03	1762	272	saylr4	3	7.18E-03	5.16E-03	968	188
orani678	4	1.39E-02	1.28E-02	697	205	mcfe	4	9.10E-03	6.10E-03	1596	290
rdist1	6	1.09E-02	1.11E-02	438	152	orani678	4	2.09E-02	1.68E-02	1013	217
memplus	8	3.07E-02	2.18E-02	1089	181	rdist1	6	1.81E-02	1.50E-02	799	177
wang3	6	2.86E-02	2.15E-02	541	143	wang3	6	4.85E-02	3.08E-02	1019	168
wang4	6	2.86E-02	2.12E-02	571	143	wang4	6	4.89E-02	3.11E-02	1126	169
lhr10	8	2.89E-02	2.59E-02	498	143	lhr10	4	4.81E-02	3.98E-02	1135	180
raefsky1	6	3.07E-02	2.88E-02	321	154	raefsky1	4	5.14E-02	4.26E-02	494	171
goodwin	4	4.12E-02	4.76E-02	604	167	af23560	8	9.93E-02	6.23E-02	320	144
shyy161	4	5.34E-02	5.04E-02	512	125	finan512	2	2.02E-01	2.37E-01	2070	201
vibrobox	2	6.88E-02	1.22E-01	2004	282	scircuit	2	4.96E-01	4.49E-01	1867	171
af23560	8	5.69E-02	4.04E-02	131	92	crystk02	6	1.72E-01	1.21E-01	172	97
finan512	6	1.21E-01	1.06E-01	675	107	rim	6	2.10E-01	1.66E-01	324	114
crystk02	3	9.78E-02	9.57E-02	119	55	ex11	3	2.15E-01	1.92E-01	294	118
rim	4	1.20E-01	1.34E-01	213	68	raefsky4	3	2.44E-01	2.26E-01	271	114
ex11	3	1.27E-01	1.37E-01	184	66	bcsstk35	3	2.63E-01	2.23E-01	223	105
raefsky4	3	1.39E-01	1.59E-01	179	61	raefsky3	2	2.49E-01	2.80E-01	279	117
bcsstk35	3	1.47E-01	1.49E-01	151	53	av41092	6	4.55E-01	3.72E-01	456	147
raefsky3	8	1.37E-01	1.12E-01	129	45	venkat01	2	3.74E-01	3.38E-01	274	125
av41092	4	2.80E-01	2.88E-01	430	92	crystk03	3	3.15E-01	2.59E-01	227	105
venkat01	8	2.12E-01	1.50E-01	127	50	qcd5_4	3	3.79E-01	2.90E-01	249	111
crystk03	3	1.79E-01	1.77E-01	166	55	rma10	3	4.83E-01	4.19E-01	385	117
qcd5_4	3	2.13E-01	1.91E-01	159	55	nasasrb	3	4.98E-01	4.31E-01	255	108
rma10	3	2.77E-01	2.91E-01	262	63	ct20stif	3	5.60E-01	4.86E-01	359	120
nasasrb	3	2.72E-01	2.82E-01	195	55	3dtube	3	5.99E-01	6.01E-01	277	120
ct20stif	6	3.25E-01	2.95E-01	239	60	laminar_duct3D	3	7.30E-01	6.24E-01	284	111
3dtube	3	3.31E-01	4.12E-01	201	61	dense2	3	5.97E-01	5.91E-01	296	105
laminar_duct3D	3	4.03E-01	4.28E-01	199	57	cant	3	7.48E-01	6.23E-01	304	109
dense2	6	3.40E-01	3.00E-01	148	46	pkustk04	3	8.16E-01	7.16E-01	259	113
cant	3	4.13E-01	4.19E-01	213	57	pdb1HYS	3	7.85E-01	6.44E-01	266	105
pkustk04	3	4.55E-01	4.75E-01	190	59	Si34H36	3	1.36E+00	1.41E+00	606	147
pdb1HYS	6	4.36E-01	3.65E-01	178	51	consph	3	1.13E+00	9.32E-01	192	89
Si34H36	3	8.22E-01	1.06E+00	650	96	shipsec1	3	1.48E+00	1.20E+00	178	87
consph	3	6.19E-01	6.16E-01	198	55	pwtk	3	2.14E+00	1.93E+00	303	104
shipsec1	6	7.93E-01	6.31E-01	132	47						
pwtk	3	1.11E+00	1.24E+00	185	51						

Table 5: PBR overhead for matrices that achieved at least $1.1\times$ speedup, sorted by NNZ. This table includes cost of block size detection (Analysis) and conversion of the PBR structure (Structure), but excludes the code generation/compilation step, which is explored separately in Table 7. BEP denotes the break even point, which is the number of iterations needed to compensate PBR overhead. CSRitr is the total PBR overhead expressed in number of CSR iterations.

	Model Parameters		
	Intercept	**N**	**NNZ**
Parameters:	0.021	5.6E-07	9.79E-08
Std Error:	0.013	6.81E-08	4.69E-09
	Adjusted $R^2 = 0.929$		

Table 6: Multi-regression parameters for the matrix analysis costs on the Harpertown architecture, which models the 'Analysis' column of Table 5 for this architecture.

variables N and NNZ to the benchmarked analysis times. This model yielded an adjusted R^2 value of 0.929. Parameter estimates and standard errors for the model are shown in Table 6. We performed the same validation for the structure conversion step. We fit separate models for each block size; the adjusted R^2 values for these models ranged from 0.77 to 0.88. These results confirm that PBR can be implemented with a cost that is linear in matrix size and number of nonzeros.

Intel Harpertown					AMD Opteron				
Matrix	**Blocksize**	**# Patterns**	**Δ BEP**	**Δ CSR itr**	**Matrix**	**Blocksize**	**# Patterns**	**Δ BEP**	**Δ CSR itr**
orsreg_1	7	2	25128	3860	orsreg_1	7	2	10167	3444
sherman3	5	2	17845	1745	sherman3	5	2	13534	1702
rdist1	6	8	8964	3125	rdist1	6	8	12606	2806
memplus	8	18	20695	3439	wang3	6	2	1551	256
wang3	6	2	1156	308	wang4	6	2	1704	257
wang4	6	2	1220	305	af23560	8	7	979	441
lhr10	8	50	27774	7998	crystk02	6	10	379	216
raefsky1	6	31	9408	4509	rim	6	220	12434	4398
af23560	8	7	561	397	av41092	6	236	8660	2794
finan512	6	74	11438	1812					
raefsky3	8	1	42	14					
venkat01	8	15	362	143					
ct20stif	6	227	5398	1371					
dense2	6	3	44	13					
pdb1HYS	6	15	208	60					
shipsec1	6	1	8	3					

Table 7: Cost of code generation/compilation for matrices that yielded the best performance with a block size $> 4 \times 4$, triggering code generation/compilation. ΔBEP and ΔCSR itr columns show number of extra iterations incurred by code generation/compilation, to be added on values given in BEP and CSR itr columns in Table 5.

Table 5 assumed that no code generation is necessary, which is true if the code cache contains already compiled modules for all encountered patterns. Table 7 shows the additional cost incurred by the code generation/compilation step for the subset of matrices that yielded the best performance for a blocksize greater than 4×4 in columns Δ *BEP* and Δ *CSR itr*. We report numbers only for these matrices because we assume that the code cache contains precompiled modules for all smaller patterns.

Detection of the optimal prefetch distance on systems that benefit from prefetching adds to these overheads. This tuning step can be performed during the first few SMVM operations in which the converted matrix is used. We vary the distance in 64-byte increments within a range between 256 and 1024 bytes for the first 12 iterations, then retain the distance that yielded the best performance in all subsequent iterations.

4.5 Predictor Model for Blocksize Detection

In Section 3.2, we described a method for estimating the PBR execution time in order to choose a block size that yields optimal or near-optimal performance. For each candidate block size, we first estimate the PBR and remainder CSR execution time using separate models; the overall predicted execution time is the sum of these two components. The model for the PBR part uses three estimation factors as described in Section 3.2 (the memory reads due to the nonzero and index arrays, the approximation to the number of accesses to the

2 × 2	**3 × 3**	**4 × 4**	**5 × 5**
0.9948	0.9887	0.9754	0.9612
6 × 6	**7 × 7**	**8 × 8**	**Rem. CSR**
0.9524	0.9562	0.9707	0.9806

Table 8: Predictor model R^2 values for each block size and the remainder CSR.

x and y vectors, and the number of writes to y). The model for the remainder CSR uses matrix size and number of remainder nonzeros. Then, we pick the block size that leads to the smallest predicted execution time.

Since our model is based on the assumption that PBR performance is dominated by memory accesses, we chose the 29 largest matrices of our set to train the model. The resulting adjusted R^2 values for these models on the Intel Harpertown architecture are given in Table 8. We verified that the overall memory consumption of these matrices exceeds the available L2 cache capacity, making PBR memory bound. For these experiments, we do not consider the impact of vectorization and we assume a constant prefetching distance.

Our model correctly selected the best-performing block size for 12 matrices. For the remaining 17 matrices, the performance loss due to mispredicted block size was less than 3%, with a maximum slowdown of 12% for the finan512 matrix.

Because our prediction includes two components, the PBR and the remainder CSR parts, prediction errors with opposite signs may cancel each other out. To validate that the high accuracy of our model was not due to chance, we computed the weighted sum of the absolute values of the relative error η for both the PBR and the CSR remainder parts:

$$\eta_{sum} = (cov) \times |\eta_{PBR}| + (1 - cov) \times |\eta_{Rem.CSR}|$$

where cov denotes the nonzero coverage by PBR. We saw that for 25 matrices, the total error was less than 25% for both the predicted block size as well as the block size that performed best. Therefore, the cancelation of model errors is not significant for most matrices, and the high model accuracy is due to the good fit of the model.

Overall, these results show that the block sizes selected by our predictor model lead to optimal or near-optimal speedup. We note that block size prediction is an optimization; if the optimal block size is desired and a matrix structure is sufficiently often reused, it is possible to perform a trial structure conversion using each block size and choose the optimal block size via benchmarking.

4.6 Limitations

PBR faces a number of limitations that affect its applicability and performance. First, there is the inherent requirement that a block size exists for which there is significant nonzero coverage. Second, PBR may decrease the locality of the right-hand side x vector. Third, whereas SMVM using CSR writes each element of the left-hand side vector y only once, PBR may require multiple reads and writes. PBR's parallel implementation also requires an additional reduction step for y, which is a cost parallel CSR can avoid. If the x or y vectors do not fit into the cache, such as for very sparse problems, the performance impact of these limitations may reduce PBR's effectiveness.

5 Related Work

The optimization and tuning of numerical kernel for sparse matrices has received a substantial amount of attention in the literature. An overview of techniques for performance tuning of sparse kernels is provided in [32].

Register blocking [31, 17, 18, 35, 37] identifies dense substructures and fills in zeros to obtain dense subblocks, which can then be stored using more efficient indices. By contrast, PBR achieves the same index reduction without storing (or computing with) zeros. As such, it will include blocks that register blocking heuristics would reject because they would require too much zero-filling. Others have proposed pre-processing [3] and matrix permutation to find or create dense substructures [19].

Cache blocking [17, 25, 26] improves performance by improving the locality of accesses to the x vector, which is also the goal of classical bandwidth reduction techniques [29]. Cache blocking could be applied to the individual submatrices on which PBR operates as well, although likely with smaller benefits.

Most related to our work are a number of techniques that attempt to reduce index overhead. For instance, if a row contains many contiguous columns of nonzeros, run-length encoding can be used to reduce the index overhead [27]. The benefit of run-length encoding is amplified if the matrix is first reordered to create contiguous nonzero columns [28]. The use of delta coding and row pattern compression was proposed in [36]. These two methods compress the index overhead relative to CSR, but may require hand-tuned assembly language decompression routines to be effective. An alternative to delta coding that can take advantage of near dense, but not necessarily contiguous nonzero concentrations was suggested in [20]. By contrast, PBR does not perform decompression at runtime, but may not be applicable for some matrices that benefit from compression.

Capturing sparse matrix structure in generated code was previously proposed for specialized sparse problems in [10] and [16]. In comparison, PBR does not make any assumptions about the structure of a matrix when identifying repeating patterns. Compilation techniques that optimize for specific sparse structures have been considered in [6].

Vectorized versions of CSR were proposed in [7] and [9]. Similarly, length-sorted CSR, which processes pairs of rows with equal lengths, can use unroll-and-jam to increase instruction-level parallelism [22]. Neither technique reduces index overhead, however.

Other tuning techniques include diagonal cache blocking [30], the detection of diagonal substructures [11], the exploitation of symmetries [21], and optimizations for specific higher-level kernels, such as sparse triangular solve [34]. The impact of prefetching on SMVM performance was previously explored in [31] and [37].

The results of a comparative evaluation study of techniques for improving SMVM performance [13] concur with our emphasis on reducing memory bandwidth usage. Another recent study [37, 38] considered the impact of several known optimization techniques for SMVM on emerging multicore architectures. These techniques included thread blocking, cache and register blocking, prefetching, SSE-based SIMDization of dense substructures, and others. We included the matrices used in this study in ours and found that many significantly benefit from PBR. We did not compare PBR to the optimizations implemented in that study because the implementation is not readily available.

6 Conclusion

Sparse matrix vector multiply (SMVM) has long been recognized as an extremely challenging numerical kernel. Its speed is limited by available memory bandwidth, particularly on modern architectures. We proposed a novel method to reduce its memory bandwidth requirements by exploiting a memory-efficient index structure that identifies and exploits repeating patterns, which are captured in generated code. Unlike previous techniques, our method is agnostic with respect to structure. Perhaps counter-intuitively, the lion's share of the structure of many matrices that arise in scientific problems can be captured using a relatively small number of frequently recurring patterns. Our technique can benefit from features that are available on dominant modern architectures, such as prefetching and vectorization, and it is easily parallelizable.

We presented a library that performs matrix conversions on the fly, allowing our method to be used as a drop-in replacement for existing methods. We evaluated PBR by demonstrating its performance advantage for sequential and parallel implementations and by quantifying its overheads relative to its benefits. We

described a performance predictor for choosing a blocksize that achieves an optimal or near-optimal performance. For many practical scenarios in which matrix structures are reused sufficiently often, pattern-based representation can augment the arsenal of tuning methods for sparse matrix-vector multiplication.

References

[1] BOOST C++ Libraries. URL http://www.boost.org. http://www.boost.org

[2] Standard Template Library Programmer's Guide. URL http://www.sgi.com/tech/stl/. http://www.sgi.com/tech/stl/

[3] Agarwal, R.C., Gustavson, F.G., Zubair, M.: A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In: Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing, pp. 32–41. IEEE Computer Society Press, Los Alamitos, CA, USA (1992)

[4] Antoulas, A., Sorensen, D., Gugercin, S.: A survey of model reduction methods for large-scale systems. In: Structured Matrices in Operator Theory, Numerical Analysis, Control, Signal and Image Processing, Contemporary Mathematics, 280, pp. 193–219. AMS Publications (2001)

[5] Belgin, M., Back, G., Ribbens, C.J.: Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In: M. Gschwind, A. Nicolau, V. Salapura, J. Moreira (eds.) ICS'09: Proceedings of the 23rd International Conference on Supercomputing, Yorktown Heights, NY, USA, June 8-12, 2009, pp. 100–109. ACM (2009). URL http://doi.acm.org/10.1145/1542275.1542294

[6] Bik, A.J.C., Wijshoff, H.A.G.: Compilation techniques for sparse matrix computations. In: Proceedings of the 1993 International Conference on Supercomputing, pp. 416–424. ACM Press (1993)

[7] Blelloch, G.E., Heroux, M.A., Zagha, M.: Segmented operations for sparse matrix computation on vector multiprocessors. Tech. rep., Carnegie-Mellon University (1993)

[8] Davis, T.: The University of Florida sparse matrix collection. Http://www.cise.ufl.edu/research/sparse/matrices

[9] D'Azevedo, E.F., Fahey, M.R., Mills, R.T.: Vectorized sparse matrix multiply for compressed row storage format. In: International Conference on Computational Science (2005)

[10] Fukui, Y., Yoshida, H., Higono, S.: Supercomputing of circuits simulation. In: Supercomputing '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 81–85. ACM, New York, NY, USA (1989). DOI http://doi.acm.org/10.1145/76263.76272

[11] Geus, R., Röllin, S.: Towards a fast parallel sparse symmetric matrix-vector multiplication. Parallel Computing **27**(7), 883 – 896 (2001). DOI DOI: 10.1016/S0167-8191(01)00073-4. URL http://www.sciencedirect.com/science/article/B6V12-430G3NG-3/2/f5bd2afe4dc99c1e7dca40af04255712

[12] Gneiting, T., Raftery, A.E.: Weather forecasting with ensemble methods. Science (Washington, D.C.) **310**(5746), 248 (2005)

[13] Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Understanding the performance of sparse matrix-vector multiplication. In: PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), pp. 283–292. IEEE Computer Society, Washington, DC, USA (2008). DOI http://dx.doi.org/10.1109/PDP.2008.41

[14] Graham, R.L., Shipman, G.M., Barrett, B.W., Castain, R.H., Bosilca, G., Lumsdaine, A.: Open MPI: A high-performance, heterogeneous MPI. In: Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks. Barcelona, Spain (2006)

[15] Gugercin, S., Antoulas, A.C., Beattie, C.: $\mathcal{H}_2$ model reduction for large-scale linear dynamical systems. SIAM Journal on Matrix Analysis and Applications **30**(2), 609–638 (2008). DOI 10.1137/060666123. URL http://link.aip.org/link/?SML/30/609/1

[16] Gustavson, F.G., Liniger, W., Willoughby, R.: Symbolic generation of an optimal crout algorithm for sparse systems of linear equations. J. ACM **17**(1), 87–109 (1970). DOI http://doi.acm.org/10.1145/321556.321565

[17] Im, E.J., Yelick, K.A.: Optimizing sparse matrix computations for register reuse in sparsity. In: ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I, pp. 127–136. Springer-Verlag, London, UK (2001)

[18] Im, E.J., Yelick, K.A., Vuduc, R.: SPARSITY: Framework for optimizing sparse matrix-vectormultiply. International Journal of High Performance Computing Applications **18**(1), 135–158 (2004)

[19] James B. White, I., Sadayappan, P.: On improving the performance of sparse matrix-vector multiplication. In: HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing, p. 66. IEEE Computer Society, Washington, DC, USA (1997)

[20] Kourtis, K., Goumas, G., Koziris, N.: Optimizing sparse matrix-vector multiplication using index and value compression. In: CF '08: Proceedings of the 2008 conference on computing frontiers, pp. 87–96. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1366230.1366244

[21] Lee, B.C., Vuduc, R.W., Demmel, J.W., Yelick, K.A.: Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In: ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing, pp. 169–176. IEEE Computer Society, Washington, DC, USA (2004). DOI http://dx.doi.org/10.1109/ICPP.2004.58

[22] Mellor-Crummey, J., Garvin, J.: Optimizing sparse matrix-vector product computations using unroll and jam. International Journal of High Performance Computing Applications **18**(2), 225–236 (2004). DOI http://dx.doi.org/10.1177/1094342004038951

[23] Merlitz, H., Herges, T., Wenzel, W.: Fluctuation analysis and accuracy of a large-scale in silico screen. Journal of Computational Chemistry **25**(13), 1568 (2004)

[24] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Top500 supercomputing sites (2008). Http://www.top500.org/

[25] Nishtala, R., Vuduc, R., Demmel, J., Yelick, K.: Performance modeling and analysis of cache blocking in sparse matrix vector multiply. Tech. rep., Berkeley, EECS Dept. (2004). URL citeseer.ist.psu.edu/article/nishtala04performance.html

[26] Nishtala, R., Vuduc, R., Demmel, J.W., Yelick, K.A.: 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 (2004)

[27] Park, S.C., Draayer, J.P., Zheng, S.Q.: An efficient algorithm for sparse matrix computations. In: SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing, pp. 919–926. ACM, New York, NY, USA (1992). DOI http://doi.acm.org/10.1145/130069.130108

[28] Pinar, A., Heath, M.T.: Improving performance of sparse matrix-vector multiplication. In: Proceedings of Supercomputing'99 (CD-ROM). ACM SIGARCH and IEEE, Portland, OR (1999)

[29] Rose, D.J.: A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. Graph Theory and Computing pp. 183–217 (1973)

[30] Temam, O., Jalby, W.: Characterizing the behavior of sparse algorithms on caches. In: In Proceedings of Supercomputing 92, pp. 578–587 (1992)

[31] Toledo, S.: Improving the memory-system performance of sparse-matrix vector multiplication. IBM Journal of Research and Development **41**(6), 711–725 (1997). URL http://citeseer.ist.psu.edu/toledo97improving.html

[32] Vuduc, R.: Automatic performance tuning of sparse matrix kernels. Ph.D. thesis, Univerity of California, Berkeley (2003). URL citeseer.ist.psu.edu/vuduc03automatic.html

[33] Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005, Journal of Physics: Conference Series. Institute of Physics Publishing, San Francisco, CA, USA (2005)

[34] Vuduc, R., Kamil, S., Hsu, J., Nishtala, R., Demmel, J.W., Yelick, K.A.: Automatic performance tuning and analysis of sparse triangular solve. In: ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries (2002)

[35] Vuduc, R.W., Moon, H.J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In: High Performance Computing and Communcations, *Lecture Notes in Computer Science*, vol. 3726, pp. 807–816. Springer (2005). URL http://dx.doi.org/10.1007/11557654_91. Editors: Laurence Tianruo Yang and Omer F. Rana and Beniamino Di Martino and Jack Dongarra

[36] Willcock, J., Lumsdaine, A.: Accelerating sparse matrix computations via data compression. In: ICS '06: Proceedings of the 20th annual international conference on Supercomputing, pp. 307–316. ACM Press, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1183401.1183444

[37] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pp. 1–12. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1362622.1362674

[38] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Computing **35**(3), 178 – 194 (2009). DOI DOI: 10.1016/j.parco.2008.12.006. URL http://www.sciencedirect.com/science/article/B6V12-4V7643Y-1/2/1abb45bf37ef1e2c25765dc0fc6b45a3. Revolutionary Technologies for Acceleration of Emerging Petascale Applications

[39] Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. SIGARCH Computer Architecture News **23**(1), 20–24 (1995). DOI http://doi.acm.org/10.1145/216585.216588