Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication

Aydın Buluç [*] Samuel Williams[*] Leonid Oliker[*] James Demmel[*†]
abuluc@lbl.gov swwilliams@lbl.gov loliker@lbl.gov demmel@cs.berkeley.edu
[*]Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
[†]Mathematics Department and CS Division
University of California
Berkeley, CA 94720

***Abstract*—On multicore architectures, the ratio of peak memory bandwidth to peak floating-point performance (byte:flop ratio) is decreasing as core counts increase, further limiting the performance of bandwidth limited applications. Multiplying a sparse matrix (as well as its transpose in the unsymmetric case) with a dense vector is the core of sparse iterative methods. In this paper, we present a new multithreaded algorithm for the symmetric case which potentially cuts the bandwidth requirements in half while exposing lots of parallelism in practice. We also give a new data structure transformation, called *bitmasked register blocks*, which promises significant reductions on bandwidth requirements by reducing the number of indexing elements without introducing additional fill-in zeros. Our work shows how to incorporate this transformation into existing parallel algorithms (both symmetric and unsymmetric) without limiting their parallel scalability. Experimental results indicate that the combined benefits of bitmasked register blocks and the new symmetric algorithm can be as high as a factor of 3.5x in multicore performance over an already scalable parallel approach. We also provide a model that accurately predicts the performance of the new methods, showing that even larger performance gains are expected in future multicore systems as current trends (decreasing byte:flop ratio and larger sparse matrices) continue.**

I. Introduction

Sparse-matrix vector multiplication (SpMV) is a frequent bottleneck in scientific computing and is notorious for sustaining a low fraction of peak performance on modern microprocessors. For multicore optimization, one key performance aspect is exposing sufficient parallelism to avoid idle processors and attain scalability. Another critical performance component is minimizing bandwidth requirements by reducing the indexing overhead — as the SPMV kernel does not exhibit temporal locality. Significant bandwidth savings are also possible by reading only half of a symmetric matrix, and storing only a single copy of the matrix while performing both $y \leftarrow Ax$ (SpMV) and $u \leftarrow A^{\mathsf{T}}v$ (SpMV_T) for the unsymmetric case.

Decades of research proposed varying methods to reduce the index overhead of sparse matrix storage schemes. Among those techniques, register blocking has proved especially useful, with block compressed sparse row (BCSR) and its variants becoming standard in high-performance auto-tuned SpMV kernels [29] (see related work in Section VIII). One limitation of BCSR is the matrix fill-in zeros, which often occur when a register block is not completely dense, creating a performance bottleneck on matrices that do not have small dense block structures. Because SpMV is a bandwidth limited calculation, this impediment is primarily due to the added bandwidth pressure of storing and streaming through explicit zeros; not the overhead of additional flops on zero elements.

Recently, the compressed sparse blocks (CSB) [6] format was proposed as a new parallel sparse matrix data structure that allows efficient computation of SpMV and SpMV_T in a multithreaded environment. CSB proves scalability with increasing number of cores while being serially competitive with standard sparse matrix storage schemes such as compressed sparse row (CSR) and compressed sparse column (CSC). However, CSB does not provide any bandwidth reduction schemes. Although it supports symmetric matrices by storing only the upper (or lower) triangle, the multiplication algorithm requires streaming the triangular matrix twice into main memory, due to potential race conditions that occur while performing parallel symmetric updates. Serial algorithms do not suffer from this doubling of the bandwidth pressure.

In this work, we propose a methodology to accelerate SpMV computations on multicore processors by exposing parallelism while minimizing bandwidth requirements. Our study builds on the CSB work and present four major contributions:

1) For symmetric matrices, we present a new algorithm that cuts the bandwidth requirements in half by using floating-point atomics and matrix reordering.
2) We introduce the notion of a *bitmasked register block* used to reduce index overhead without introducing fill-in zeros, and we describe an efficient method to implement them for SpMV.
3) We show how to integrate bitmasked register blocks to unsymmetric CSB algorithm and the newly introduced symmetric algorithm. The unsymmetric algorithm retains its ability to perform both SpMV

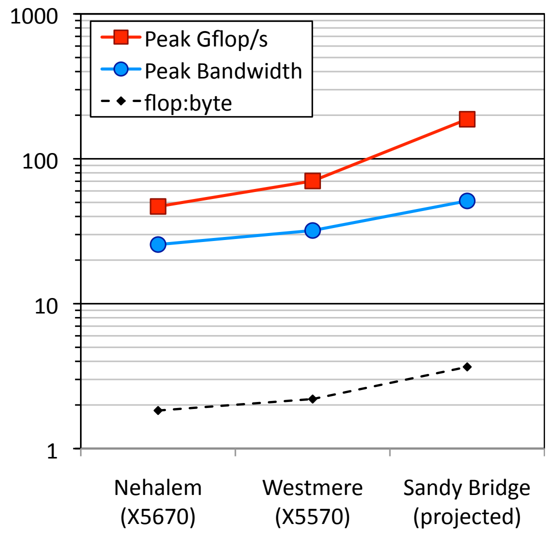


Fig. 1: **Scaling trends for three generations of Intel processors. As the flop:byte ratio increases, more and more of the compute capability goes unused.**

and SpMV_T and we can still prove plenty of parallelism.

4) We present a new performance model that gives insight about the matrix dependent analysis of parallel SpMV for a given architecture.

Impact on Future Multicores: Since our work addresses bandwidth limitations, the advantages of our approach will continue to grow as next-generation multicore systems continue to suffer from increased flop:byte ratios due to technology constraints. As an example, Figure 1 presents the trends in peak performance, bandwidth, and machine balance for three generations of Intel processors. The Sandy Bridge node is projected based on a 8-core server part running at the same frequenecy using Intel Advanced Vector Extensions (AVX) and memory bandwidth corresponding to DDR3-1600. We observe that peak performance has grown quickly via a doubling of both the core count and SIMD (single instruction multiple data) width. As peak bandwidth has not kept pace, the machine balance (as measured in flops/byte) has roughly doubled. Such trends have profound implications for memory bound computations like SpMV as the memory subsystem cannot keep the cores occupied, resulting in wasted compute capability.

Examining the SpMV algorithm, each nonzero generates about 12 bytes of memory traffic and will perform 2 floating-point operations. The trend in Figure 1 shows that on processors like Nehalem, 22 floating-point operations can be performed in the time required to transfer a nonzero; wasting 20 possible operations. With the advent of Sandy Bridge, we expect this waste to increase to nearly 42, and likely even larger degradations as we look forward towards projected technology trends. It is therefore critical to find a solution that regularizes SpMV computation while reducing the average memory traffic per nonzero — which is the focus of this work.

Our algorithms work on a work-stealing environment. Work stealing helps mitigate any potential load imbalances that arise due to the inherent challenges of the SpMV kernel and the parallelism in modern multicore processors. For example, if certain blocks of the sparse matrix are more dense than others, work stealing tasks other threads with a larger number of sparser blocks and thus makes our algorithms dynamically scheduled and more resilient to non-uniform nonzero distributions among blocks.

To validate our methodology, we examine single-socket performance results on two of the most sophisticated multicore processors available: the octa-core Intel Nehalem-EX and the hexa-core AMD Istanbul. We also present a performance model for bitmasked register blocks within the CSB matrix structure that predicts the performance gains of register blocking. Overall results show the potential impact of our algorithmic contribution on existing and emerging multicore platforms.

II. Terminology

We analyze parallelism in terms of ***work*** and ***span*** [9, Ch. 27]:

- The ***work***, denoted by T_1, is the running time on a single processor.
- The ***span***, denoted by T_∞, is running time on an infinite number of processors.

The ***parallelism*** of the algorithm is T_1/T_∞, which corresponds to the maximum possible speedup on any number of processors.

Additionally, we analyze the asymptotic cost of transfering data to/from the main memory. The ***bwind*** captures the bandwidth costs due to indexing overhead. Bandwidth costs due to streaming actual nonzero values always sum up to nnz, since our algorithms do not store explicit zeros.

In sparse matrix computations, the word "block" is used in multiple contexts. In order to avoid confusion, we use the full phrase "register block" to denote $r \times r$, small ($r \approx 2 \dots 8$), typically dense blocks. While referring to a "compressed sparse block", which is the $\beta \times \beta$, large ($\beta \approx \sqrt{n}$), typically hypersparse [7] (the ratio of nonzeros to block dimension is asymptotically zero) block within the CSB format, we will sometimes abbreviate and simply use "block".

III. An Overview of the CSB format

In order to make the paper as self contained as possible, this section provides an overview of the CSB sparse-matrix storage format [6] and the high-level description of parallel SpMV and SpMV_T algorithms that use CSB.

CSB partitions an $n \times n$ matrix into n^2/β^2 blocks of size $\beta \times \beta$ each. A_{ij} denotes the $\beta \times \beta$ submatrix containing nonzero elements falling in rows $i\beta, \dots, (i+1)\beta - 1$ and columns $j\beta, \dots, (j+1)\beta - 1$ of A. The nonzero elements within each sparse block are ordered recursively using the Z-morton layout [21].

In practice, CSB stores its matrices in three arrays. The *blk_ptr* array is a dense 2D array that points to the first nonzero in each compressed sparse block. The *low_ind* array holds the lower order bits of the row and column index of each nonzero, relative to the beginning of the sparse block (hence making them small enough to be concatenated and stored in a single integer). The *val* array stores the actual numerical values. Both *val* and *low_ind* arrays are of size nnz.

The algorithms for SpMV and SpMV_T are almost identical except that the roles of rows and columns are switched. Hence, only the SpMV case is described for conciseness. The performance bounds on the parallel algorithms assume the existence of a work-stealing runtime scheduler [5], such as the one used in Cilk [4].

The parallel SpMV algorithm employs three levels of parallelism. First, each blockrow (the row of blocks $A_{i0}, \dots, A_{i,n/\beta-1}$) is multiplied in parallel without any data races. If the nonzeros among blockrows are distributed unevenly, then a second level of parallelism (the CSB_BLOCKROWV subroutine) is used within densely populated blockrows. This involves recursively dividing into two subblockrows that has roughly equal number of nonzeros. Each invocation of CSB_BLOCKROWV uses temporary vectors of size $O(\beta)$ to avoid data races when combining updates from different subblockrows. The overall space required by the temporaries for an execution on P threads is $O(P\beta \lg n)$ [6, Cor. 10], using a space-efficient scheduler.

To facilitate the recursive subdivision into subblockrows, CSB_BLOCKROWV uses the notion of a "chunk", which is a set of consecutive blocks, either consisting of a single block containing $\Omega(\beta)$ nonzeros, or multiple blocks containing $O(\beta)$ nonzeros in total. A chunk is terminated if adding the next block to the chunk increases its total number of nonzeros to more than $\Theta(\beta)$. At the compressed sparse block level, if it is dense, a (sub)block M is parallelized via the CSB_BLOCKV subroutine. It recursively divides M into four quadrants $M_{00}, M_{01}, M_{10}, M_{11}$, using binary searches to find dividing points (that is, it searches for the first nonzero whose row index is larger than $\beta/2$ to divide M through the middle row, then performs parallel binary searches in those two pieces to find their column mid points). The CSB_BLOCKV subroutine first recursively calls itself on M_{00} and M_{11} in parallel, synchronizes, then recursively calls itself on M_{01} and M_{10} in parallel. The recursion continues until a subblock of size $dim \times dim$ has only $\Theta(dim)$ nonzeros, after which a serial multiplication is performed.

IV. SYMMETRIC ALGORITHM

A typical serial symmetric SpMV code can save half of the bandwidth performing both $y_i \leftarrow y_i + a_{ij}x_j$ and $y_j \leftarrow y_j + a_{ij}x_i$ with a single fetch of $a_{ij} = a_{ji}$. These 50% savings in memory bandwidth might potentially give close to a factor of 2 improvement for the bandwidth limited SpMV kernel. In parallel, this approach creates potential race conditions on the output array y. Unfortunately, using locks or atomic updates for all nnz updates is slower than reading the matrix twice due to the higher overheads. A completely algorithmic solution, on the other hand, seems hard to achieve without matrix reordering. We thus present an algorithm that performs atomic updates for a relatively small portion of the matrix. Our symmetric algorithm combines matrix reordering, atomic updates and the lock-free BLOCKV_SQ and BLOCKV_TRI kernels that operate on individual compressed sparse blocks.

$$A = \begin{pmatrix} A_{00} & A_{01} & A_{02} & * & * \\ & A_{11} & A_{12} & A_{13} & * \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & \ddots \\ & & & & A_{kk} \end{pmatrix}$$

Fig. 2: **A sparse matrix in CSB format (where $k = n/\beta - 1$) after Cuthill-Mckee ordering. The locations marked with $*$ contain $w\%$ of the nonzeros, depending on the matrix.**

Matrix bandwidth is defined as the maximum distance of any nonzero to the main diagonal. Similarly, block bandwidth is the maximum block-distance of a nonempty block. For instance, the main block diagonal has a block bandwidth of zero, the next block diagonal has a block bandwidth of 1, and so on.

Our algorithm works on a matrix that has been reordered using the reverse Cuthill-McKee (RCM) algorithm [10] to reduce its bandwidth. The matrix is then stored in CSB format as shown in Figure 2. Many of the nonzeros are clustered within the three block diagonals:

1) The triangular blocks on the main diagonal.
2) The rectangular blocks $A_{i,i+1}$.
3) The rectangular blocks $A_{i-1,i+1}$.

These block diagonals are treated purely algorithmically, without using atomic updates. The seemingly arbitrary choice of three is not an inherent property of our algorithm, and can easily be increased or decreased depending on the matrix. The optimal choice for the number of block diagonals that gets executed without atomic is a complex function of the matrix, architecture and run-time characteristics. We give a sketch of our future autotuning strategy in Section IV-C.

After reordering, three levels of $\sqrt{n} \times \sqrt{n}$ block diagonals (for $\beta \approx \sqrt{n}$) cover a significant portion of the nonzeros for matrices that can be permuted to block diagonal structure. For such matrices, three block diagonals were sufficient to cover most of the nonzeros, whereas for harder-to-permute matrices, increasing the block bandwidth by a constant amount increased the

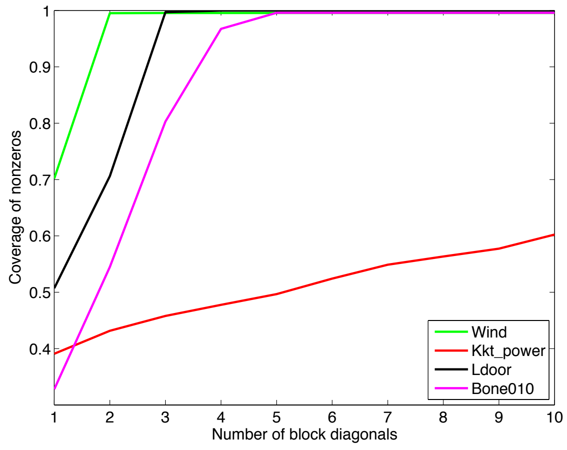


Fig. 3: **The nonzero coverage of the first ten block diagonals for the symmetric matrices in our test suite. The sparse block dimension β is automatically selected by our implementation, as described in the original CSB paper [6].**

$$U = \begin{pmatrix} U_0 & S \\ & U_1 \end{pmatrix}$$

Fig. 4: **Recursive decomposition of a triangular compressed sparse block. Submatrices U_0 and U_1 are upper triangular like U, while S is a square block.**

nonzero coverage only by a modest amount. Figure 3 shows the nonzero coverage of the first ten block diagonals for the symmetric matrices in our test suite.

The algorithm then splits the output array y into four temporary arrays, three for block diagonals and one for the remaining scattered nonzeros. Finally, after multiplications of all submatrices, it accumulates the temporaries back to y. The pseudocode for the algorithm is given in Figure 5. The **in parallel do** . . . **do** . . . construct indicates that all of the **do** code blocks may execute in parallel. The "**for** . . . **in parallel do**" construct means that each iteration of the **for** loop may be executed in parallel with the others.

The BLOCKV_SQ subroutine is similar to the corresponding CSB routine CSB_BLOCKV [6] except that it performs symmetric multiplication. BLOCKV_TRI is a its generalization for the triangular case. We recursively divide an upper-triangular block into three pieces U_0, U_1, and S, as shown in Figure 4. The pseudocode for BLOCKV_TRI is shown in Figure 6.

The algorithm exposes plenty of parallelism, shown in the execution diagram of Figure 7. The only synchronizations are within the rectangular block diagonals where the algorithm properly alternates blocks in lines 6–8 and similarly in lines 11–13, guaranteeing a race-free algorithm. The two step process avoids race conditions that would happen when multiplying $A_{i,i-1}$ and $A_{i+1,i}$ at the same time because the former updates both y_{i-1} and y_i where the latter updates both y_i and

```
SYM_SPMV($A, x, y$)
1   Initialize temporary arrays $y_1, y_2, y_3$
2   in parallel
3     do for $i \leftarrow 0$ to $n/\beta - 1$ in parallel
4           do BLOCKV_TR($A_{i,i}, \beta, y_1, x$)
5   ($list\_a, list\_b$) ← SEPARATEINDICES($n/\beta, 2$)
6     do for all $i \in list\_a$ in parallel
7           do BLOCKV_SQ($A_{i,i+1}, \beta, y_2, x$)
8         for all $i \in list\_b$ in parallel
9           do BLOCKV_SQ($A_{i,i+1}, \beta, y_2, x$)
10  ($list\_a, list\_b$) ← SEPARATEINDICES($n/\beta, 3$)
11    do for all $i \in list\_a$ in parallel
12          do BLOCKV_SQ($A_{i-1,i+1}, \beta, y_3, x$)
13        for all $i \in list\_b$ in parallel
14          do BLOCKV_SQ($A_{i-1,i+1}, \beta, y_3, x$)
15    do for all other compressed blocks $A_{i,j}$
              where $j > i + 1$ in parallel
16          do MULTADDATOMICS($A_{i,j}, y, x$)
17  in parallel
18    do $y \leftarrow y + y_1 + y_2 + y_3$
```

Fig. 5: **Pseudocode for the symmetric matrix-vector multiplication $y \leftarrow Ax$ (assuming n/β is an even number, the odd case just changes the loop indices). The procedure SEPARATEINDICES($size, blkdiag$) divides the whole iteration space of the $blkdiag$th block diagonal into two independent pieces that will not conflict with each other. In particular, alternating indices is sufficient for the second block diagonal, where all blocks $A_{2i,2i+1}$ are performed in parallel, followed by all blocks $A_{2i+1,2i+2}$ in parallel. For the third block diagonal, we should alternate in groups of two instead.**

y_{i+1}, due to symmetry. Here, we use a loose notation where y_i denotes the ith subvector when we divide the vector to equally sized portions of β. In other words, $y_i = y(i\beta + 1 : (i + 1)\beta)$ in Matlab notation.

Since each of the four main execution branches are independent from each other, the run time system can steal from other branches in case load is imbalanced within a given branch. It is possible to trade-off some parallelism in order to avoid temporaries altogether, by using a single vector and synchronizing in between the four parallel paths that are spawned from line 17. However, we use the original description of the algorithm in our experiments.

A. Analysis of the Symmetric Algorithm

In this section, we analyze the work and span of our symmetric algorithm for a matrix that has only ω% of its nonzeros outside the first three β-by-β block diagonals. Since atomic updates are more expensive than normal additions, an architecture dependent constant $c > 1$ is used to quantify the extra cost of atomic operations.

The SYM_SPMV algorithm performs only a constant number of synchronizations between calls to BLOCKV_SQ and BLOCKV_TR. The subprocedure BLOCKV_SQ runs with work $\Theta(nnz)$ and span $O(\beta)$,

```
BLOCKV_TRI($U, dim, x, y$)
    // $U$ is a $dim \times dim$ upper-triangular matrix.
 1  if $nnz(U) \leq \Theta(dim)$
 2      then // Perform $y \leftarrow y + Ux$ serially.
 3          for all $i, j$ where $U(i,j) \neq 0$
 4              do $y(i) \leftarrow y(i) + U(i,j) \cdot x(j)$
 5                  if $i \neq j$
 6                      then// Perform symmetric update
 7                          $y(j) \leftarrow y(j) + U(i,j) \cdot x(i)$
 8          return
 9  // Recurse. Find the indices of the quadrants.
10  binary search $1, \ldots, nnz(U)$ for the smallest $s_1$
        such that $(U.row\_ind(s_1) \geq dim/2)$
11  binary search $1, \ldots, s1$ for the smallest $s_2$
        such that $(U.col\_ind(s_2) \geq dim/2)$
12  in parallel
13    do BLOCKV_TRI($U, dim/2, x, y$) // $U_0$.
14    do BLOCKV_TRI($U + s_1, dim/2, x, y$) // $U_1$.
15  BLOCKV_SQ($U + s_2, dim/2, x, y$) // $S$.
```

Fig. 6: **Pseudocode for the triangular subblock-vector product $y \leftarrow Ux$, where U is stored in CSB format with recursive Z-Morton order according. The notation $U + s$ gives a submatrix of U that is formed by skipping the first s nonzeros.**

when called on a $\beta \times \beta$ submatrix that has nnz nonzeros. The span recurrence for BLOCKV_TR is $S_{TR}(\beta) = S_{TR}(\beta/2) + S_{SQ}(\beta/2) + O(\lg \beta)$ with S_{SQ} being the span for BLOCKV_SQ. It solves to $O(\beta)$ as well. It is straightforward to show that work is also linear in the number of nonzeros, by constructing a proof similar to [6, Lemma 4].

The nonzeros that are handled with atomic updates are embarrassingly parallel, although with much lower efficiency. As a result the SYM_SPMV algorithm runs with work $\Theta(nnz \cdot (c \cdot w + (1-w)))$ and span $O(\beta + \lg n)$ where the $\lg n$ term in the span is due to the parallel reduction on temporary vectors.

B. Implementation of the Symmetric Algorithm

For the implementation of the atomic updates, we use Madduri et al.'s method [18] of exploiting the `cmpxchg` instruction on x86 architectures to perform floating point increments. To assess the feasibility of this approach and to find out the ratio of atomic updates that are computationally viable, we ran a microbenchmark that performs atomic additions to vectors of varying lengths. In serial, the performance is approximately 7x slower than non-atomic additions, varying slightly with different vector sizes and locality of updates. Using 16 OpenMP workers, however, the performance of streaming atomic updates are 1.6 to 2.2x slower than the non-atomic case, and the random atomic updates are only 1.2 to 1.6x slower. Here streaming updates refers to updates of the form $y_i \leftarrow y_i + inc$ for $i = 0, ..., n-1$ in order. In the case of random updates, we shuffle the indices i. In fact, this microbenchmark gave the motivation for the current form of our symmetric algorithm.

C. Discussion and autotuning opportunities

Instead of relying on atomic operations, we could have modified our algorithm to execute every block diagonal until all nonzeros are finished. Think about the solution to the 3D heat problem by using finite differencing on a regular grid. The discretized matrix can be permuted to have $O(n^{2/3})$ bandwidth, yielding $O(n^{2/3}/\beta)$ block diagonals. The modified SYM_SPMV algorithm would then have a span of $O(n^{2/3})$ and a parallelism of $O(nnz/n^{2/3})$, which is ample for large matrices. However, some matrices can not be permuted to have $o(n)$ bandwidth (asymptotically less than n). In this case, our parallelism would be limited to merely $O(nnz/n)$.

In the future, we plan to make our symmetric algorithm more resilient to different inputs by using an autotuning strategy. The final performance is a complex function of the architectural parameters, matrix nonzero structure, and run-time parameters such as the number of available threads. Below is a sketch of our strategy:

(i) The first step is architecture dependent. The autotuner calculates the maximum nonzero percentage $w\%$ that falls outside the block diagonals, which can be tolerated without a significant slowdown in the execution.

(ii) The second step is matrix dependent. The autotuner calculates the number of block diagonals that needs to be treated without atomics according to the value of w from step (i). We call this number $L < n/\beta$.

(iii) The third step, which determines the number of temporary vectors y_i, depends on the run-time characteristics such as the number of execution threads and the available memory per thread. The autotuner calculates a lower bound on $|y_i|$ that exposes enough parallelism to keep all the cores busy (t_1), and an upper bound that is limited by the available memory (t_2).

(iv) The autotuner picks a number t in the range $t_1 \leq t \leq t_2$. It runs a loop with increments of L/t, each processing t diagonal blocks. The rest of the nonzeros are executed with atomics. The autotuner times the SpMV operation with different t values within the permitted range and picks one with the best performance.

V. BITMASKED REGISTER BLOCKS

We can reduce the memory bandwidth costs of our symmetric algorithm as well as the original SpMV and SpMV_T algorithms for unsymmetric matrices, using bitmasked register blocks. Normally, register blocking creates small dense matrices by filling in the block with explicit zeros. This is wasteful on today's bandwidth-constrained processors. In our approach, proposed by

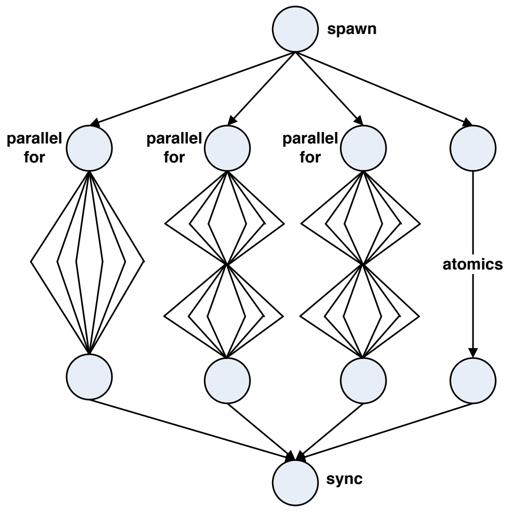


Fig. 7: **Parallel execution diagram of Sym_SpMV.**

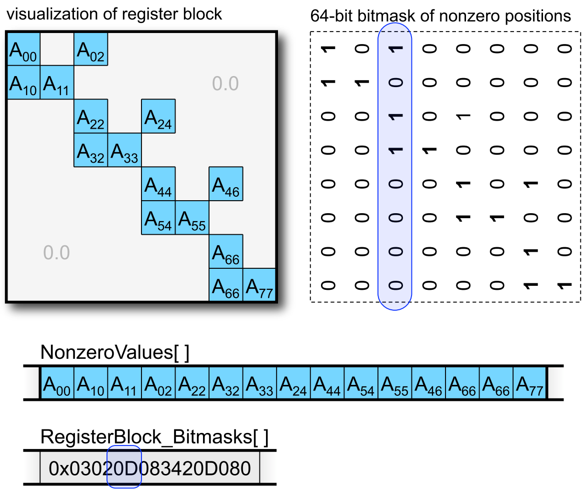

visualization of register block
64-bit bitmask of nonzero positions
0.0
0.0
NonzeroValues[ ]
RegisterBlock_Bitmasks[ ]
0x03020D083420D080

Fig. 8: **Visualization of a bitmasked register block. Top left: register block with nonzeros called out in blue. Top right: resultant column-major bitmask with 1's denoting nonzeros. Bottom, visualization of the resultant data in memory. Note, explicit zeros are never stored in memory.**

Williams [36] and visualized in Figure 8, we pack the nonzeros of a register block contiguously, but use a bitmask to mark their positions in the dense block. As the block is transfered from memory to registers, the bitmask is used to dynamically fill in zeros. As such, we perform flops on explicit zeros, but never transfer them from memory. In summary, the latency of accessing vectors stays constant, the number of arithmetic operations increases, and the bandwidth requirements decrease. To retain the parallelism benefits and the ability to perform SpMV_T, we modify CSB to store bitmasked register blocks of size $r \times r$ for $r = 2$, 4, or 8. The conversion from standard CSB to register blocked version is linear, hence making our approach practical to situations even where there are not many SpMV operations to offset the conversion.

In this work, we focus on square register blocks only because one of the distinguishing advantages of CSB is its symmetry: accessing columns is as easy as accessing rows. If we were to use rectangular $r \times c$ register blocks, then the performance of SpMV and SpMV_T would differ. Rectangular blocks might outperform square blocks, if for some (r, c) pair, $(P(r, c)+P(c, r))/2 > P(s, s)$ for any s, where $P(r, c)$ denotes the row SpMV performance using $r \times c$ register blocks. We will explore rectangular register blocks in future work .

A. Bitmasked CSB and algorithms

In this section, we describe how to integrate bitmasked register blocks to the original CSB algorithm. A similar modification to Sym_SpMV algorithm is straightforward, and we omit the details for brevity.

In the proposed modification to the CSB data structure, *low_ind* holds only one entry per register block, irrespective of the number of actual nonzeros it contains. An additional array with same length, named *msk_arr*, holds the bitmask structure of the register block. The register blocks never cross compressed sparse block boundaries, hence making the lower order bits stored in *low_ind* sufficient for indexing x and y arrays.

The original parallel SpMV and SpMV_T algorithms need only two modifications to operate on the enhanced data structure. First, the inner-most loop uses the bitmap associated with the register block to index the *val* array. Second, the smallest addressible unit of the matrix is now a register block and all the dynamic parallelization decisions are now made according to the number of register blocks, instead of nonzeros. For example, the CSB_BlockRowV routine used to split a given block row into two smaller ones with approximately equal number of nonzeros. With the current bitmask enhancements, the splitting is done in a way so that both sides get approximately same number of register blocks. This modification is practically necessary since we cannot calculate the number of nonzeros in the compressed sparse blocks without counting the number of bits set in the *msk_arr*. It is also intuitive since to a first degree approximation (by bandwidth and flops costs), our computational complexity now depends on the number of register blocks, as opposed to nnz (which only affects the latency costs associated with accessing x and y vectors). Concretely, this modification means that a chunk is terminated only if adding the next block to the chunk would increase the number of register blocks to more than $\Theta(\beta)$.

B. Analysis of the bitmasked SpMV algorithm

In order to qualify the performance gains with register blocking, we asymptotically analyze the unsymmetric CSB algorithm using bitmasked register blocks.

Let A be an n-by-n sparse matrix with nnz nonzeros. There are n^2/β^2 compressed sparse blocks, each of which is β-by-β. The smallest directly addressible entity inside a compressed sparse block is a $r \times r$ register block that consists of at least one nonzero. Hence, A contains nrb register blocks where $nnz\,/r^2 \leq nrb \leq nnz$.

We expect the performance to be mostly memory bound until work-to-bwind ratio hits a certain threshold. We also provide insights on this threshold by using the performance model [35] in Section V-D.

Lemma 1. *The work of multiplying a $r \times r$ register block having nnz nonzeros with a dense vector is $O(r^2)$. Similarly, its bwind is also $O(r^2)$, but with a large reduction in the constant factor.*

Lemma 2. *On a β-by-β block containing nnz nonzeros and nrb register blocks,* CSB_BLOCKV *runs with work $\Theta(nrb \cdot r^2)$ and span $O(r\beta)$.*

The proof follows Lemma 4 of the original CSB paper [6]. The span recurrence is still $S(\beta) = 2S(\beta/2) + O(\lg \beta)$, but this time with a different base case $S(r) = r^2$, which solves to $\beta/r \cdot S(r) = O(r\beta)$ even when the block is completely dense. This looseness in the analysis is particularly helpful in the case of applying permutations that might lead to dense blocks, as we do in the symmetric algorithm. The sum of the multiplication costs on the leaves of the recursion tree is $O(nrb \cdot r^2)$ from Lemma 1. The costs due to the binary searches on the internal nodes can be accounted as follows. For an internal node at height h, which corresponds to a $(r \cdot 2^h) \times (r \cdot 2^h)$ subblock, we have at least 2^h (otherwise we would not recurse further) and at most 2^{2h} (completely dense case) register blocks. Hence, both the bandwidth and the floating-point costs on a internal node of height h are $O(\lg 2^h) = O(h)$ from the binary searches. Summing over all internal nodes gives $O(nrb)$ extra work due to binary searches, which is subsumed by the work on leaves.

Lemma 3. *On a blockrow containing n/β blocks and nrb register blocks,* CSB_BLOCKROWV *runs with work $\Theta(nrb \cdot r^2)$ and span $O(r^2\beta \lg(n/\beta))$.*

Each blockrow contains at most $O(nrb\,/\beta)$ chunks since any two consecutive chunks contain at least $\Omega(\beta)$ register blocks by construction. The work recurrence can be written as $f(C) \leq 2f(\lceil C/2 \rceil) + \Theta(\beta)$, where C is the number of chunks. The base case $C = 1$ has work $\Theta(nrb \cdot r^2)$. The recursion solves to $\Theta(C\beta + nrb \cdot r^2)$. The first term is due to initializing/adding to the temporary vector and the second term is the cost of actual floating point operations. Substituting $C = O(nrb\,/\beta)$ gives the conjectured bound.

When calculating span, notice that for the base cases, the algorithm either multiplies a single chunk containing at most $O(\beta)$ register blocks, or a single compressed sparse block in parallel. The former has $O(r^2\beta)$ span as it is done serially. The latter has $O(r\beta)$ span because CSB_BLOCKV is called. The recurrence $S(C) = S(\lceil C/2 \rceil) + O(\beta) = O(\beta \lg C) + S(1)$ is unchanged. Since each chunk contains at least one compressed sparse block, we have $C \leq n/\beta$. Consolidating the two terms, we get $O(\beta \lg(n/\beta)) + O(r^2\beta) = O(r^2\beta \lg(n/\beta))$

Theorem 4. *On an $n \times n$ matrix containing nnz nonzeros and nrb register blocks,* CSB_SPMV *runs with work $\Theta(n^2/\beta^2 + nrb \cdot r^2)$ and span $O(r^2\beta \lg n/\beta + n/\beta)$.*

The additional n^2/β^2 term in work and the n/β term in span is due to the overhead of computing the chunks that are required for parallelism, as in the original algorithm.

Corollary 5. *On an $n \times n$ matrix containing nnz nonzeros and $nrb \geq n/r^2$ register blocks, by choosing $\beta = \sqrt{n}$,* CSB_SPMV *runs with work $\Theta(nrb \cdot r^2)$ and span $O(r^2\sqrt{n} \lg n)$, achieving a parallelism of $\Omega(nrb\,/\sqrt{n} \lg n)$.*

A similar analysis for CSB_SPMV_T gives exactly the same bounds as both CSB and our register blocking are symmetric. We showed that the parallelism guarantees of the CSB algorithm are retained, after the incorporation of bitmasked register blocks. We managed to reduce bwind by up to a factor of 32 (assuming $r = 8$), and provided a graceful increase in work as the register blocks get sparser.

C. Implementation of the bitmasked SpMV kernel

Efficient implementation of this bitmasked approach can be challenging. We leveraged SSE instructions in order to produce a highly-tuned inner kernel. Appropriately indexing into the *val* array is accomplished by an inner kernel that performs $r^2/2$ times the following bundle of 128-bit SSE operations: `blendvpd`, `mulpd`, `addpd`, `popcnt`, `psslq`, for an $r \times r$ register block. This approach allowed us to free the inner loop of conditionals at the cost of performing a multiply-add operation for every potential nonzero in the register block. If a location is indeed zero, as indicated by the bitmask, the algorithm multiply-adds with zero, hence not changing the result. On AMD architectures, this kernel executes about 10% slower due to the absence of `blendvpd` operation, which we simulate using multiple SSE2 instructions. For the symmetric kernel, the number of `mulpd` and `addpd` operations are doubled while the rest of the operations are performed the same number of times.

As the register block arrays have effectively been compressed, there is no simple algorithm that allows us to index into the middle of the array at runtime. A thread that starts operating on the ith register block needs to know how many nonzeros exist in register blocks 0 to $i-1$ so that it can determine the address from which it should stream data. Computationally, this requires checking how many bits are set in the range $msk\_arr[0...(i-1)]$ — clearly a computationally intensive task that should be performed offline. To that end, when the matrix is created, we popcount the whole *msk_arr* array and run prefix sums on the output. The construction has $O(nrb)$ work and $O(\lg nrb)$ span, and is required only once for multiple SpMV operations.

The overall cost of accessing the *prescan* array is the same as the number of strands on the execution DAG of the parallel program. In other words, it is equal to the number of serial SpMV invocations. Since each invocation has a certain number of nonzeros to multiply, the extra costs are subsumed by useful work.

D. Performance Model

In the past, simple Roofline models have been used to bound BCSR SpMV performance using bound and bottleneck analysis based on STREAM bandwidth, in-core compute bounds based on Little's Law, and a fixed arithmetic intensity [35]. Unfortunately, such an approach cannot be easily mapped to our SpMV kernel as both the memory traffic and the useful computation per register block varies across blocks within the matrix. To that end, we developed a similar themed model also premised on bound and bottleneck analysis. However, we use empirical data for register block performance to bound in-core performance. Moreover, we use average nonzero density per register block instead of arithmetic intensity as our horizontal axis. Finally, our model predicts useful GFlop/s rather than raw GFlop/s.

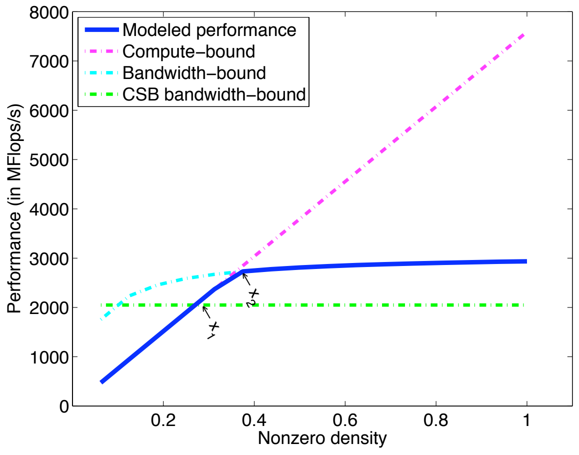
Bandwidth Bound: Given an $r \times r$ bitmasked register block, the time required to transfer the block from DRAM is:

$$\frac{bandwidth}{4 + bitmask\_size + 8 \cdot density \cdot r^2}$$

where $bandwidth$ is the measured STREAM bandwidth, $bitmask\_size$ is the size of the bitmask in bytes, and $density$ is the average density of nonzeros using $r \times r$ register blocks. We can thus bound useful performance by multiplying this quantity by the number of useful floating-point operations per register block: $2 \cdot density \cdot r^2$.

Compute Bound: For each $r \times r$ bitmasked register block size, we construct a simple benchmark that would repeatedly access the same block and perform the requisite computation. Doing so allows us to gauge an upper limit to the achievable performance on each architecture as a function of register block size, irrespective of bandwidth or sparsity. Within the performance model we scale this number by the average density to determine the number of useful floating-point operations per second.

Performance Bound: Ultimately, we may simply bound performance as the minimum of either the bound



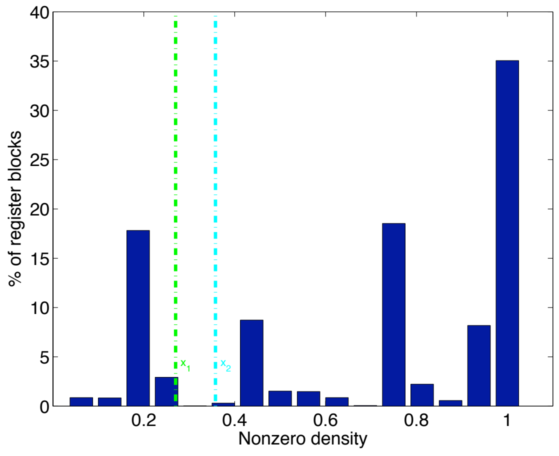


Fig. 9: (*top*) **Performance model construction for the $r = 4$ case when running 16 threads on Nehalem.** (*bottom*) **Histogram for Wind Tunnel matrix on 4×4 register blocks.**

calculated using bandwidth or the bound calculated using the compute rate for our rather complex inner loop.

The model construction is illustrated in Figure 9(top) where the actual bound on the performance is the minimum of compute bound (purple dashed line) and bandwidth bound (cyan dashed curve) numbers for a given nonzero density. Additionally, we present the bandwidth bound (green dashed line) derived from the the original CSB algorithm.

Our scheme provides performance gains over CSB for register blocks that are denser than the x_1 point on Figure 9(top). Any register block sparser than x_1 runs slower than the case without register blocking. For register blocks that are denser than x_2, the performance is compute bound with maximum possible performance gain. For register blocks whose nonzero density is between x_1 and x_2, the performance gain is proportional to how far away we are from x_1.

Figure 9(bottom) shows the histogram of nonzero densities across blocks of the Wind Tunnel matrix when using 4×4 register blocks, with x_1 and x_2 once again representing register block densities above which CSB is bandwidth bound and performance is compute bound

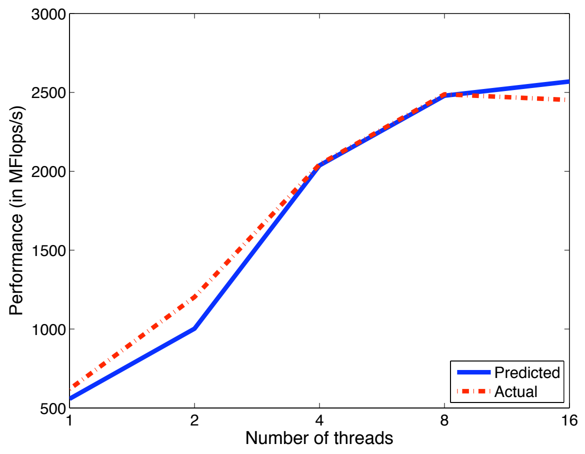


Fig. 10: **Correlation of the actual performance with the predicted performance when running on Nehalem. (Wind Tunnel matrix using 4×4 register blocks)**

(maximum possible gain), respectively. The pointwise multiplication of the histogram counts with the modeled performance from Figure 9(top) provides a relatively accurate performance prediction. That is,

$$\text{predicted}(p) = \sum_{i \in densities} \text{count}(i) \cdot \text{modeled}(i, p) \quad (1)$$

Figure 10 uses Equation 1 to present the predicted (solid blue line) as well as the actual performance (red dashed line) of the algorithm running the Wind matrix using register blocks of size 4×4. We see that performance can be highly correlated with the model.

We believe that auto-tuning based on empirical measurements is a better approach to optimizing performance than a performance model that requires the totality of the matrix in order to select an optimized implementation. Our performance model, although strongly correlated with actual measurements, merely provides insight into the forces that constrain SpMV performance. For example, densities vary significantly among register blocks; the non-uniform fill ratio of register blocks may make certain portions of the execution compute bound while the rest stays bandwidth bound. In high concurrencies, these compute-bound and bandwidth-bound regions might execute simultaneously, resulting in a faster than predicted performance.

VI. Experimental Setup

A. Hardware

In this paper, we explore single-socket multicore performance. To that end, we use two of the most advanced commodity multicore processors available today — Intel's eight-core Xeon 7550 (Nehalem-EX) and AMD's six-core Opteron 8431 (Istanbul).

Xeon 7550 (Nehalem-EX) is the latest enhancement to the Intel "Core" architecture, and represents a dramatic departure from Intel's previous large multisocket designs. Each of the eight cores runs at 2 GHz, supports two-way simultaneous multithreading (16 thread contexts per socket), can simultaneously execute one SIMD floating-point multiply and one SIMD floating-point add, and has private 32 KB and 256 KB L1 and L2 caches. Unlike previous Nehalem processors, the cores are connected to a very large 18 MB L3 cache via a ring architecture. The ring ensures much higher bandwidth to the L3. Interestingly, this design is premised on memory capacity, not memory bandwidth. As such, the two memory controllers emulate an evolved FBDIMM standard and provide a sustained bandwidth of about 14 GB/s per socket.

Opteron 8431 (Istanbul) is a simple evolution of AMD's Barcelona processor. Like Nehalem, each core is a superscalar out-of-order core capable of simultaneously executing one SIMD floating-point multiply and one SIMD floating-point add. The private L1 and L2 caches are 64 KB and 512 KB respectively. The six cores of this processor are serviced by a 6 MB L3 cache. To mitigate snoop effects on multisocket SMPs and maximize the effective memory bandwidth, Istanbul reserves 1MB of each 6MB cache for HT assist (a snoop filter). The snoop filter enables higher bandwidth by caching knowledge as to whether the data is cached on other chips. The cores at 2.4 GHz and the effective memory bandwidth is a comparable 11 GB/s per socket.

B. Programming Model

Cilk++, based on the earlier MIT Cilk system, is a faithful extension of C++ that for multithreaded programming. It enables dynamic parallelism via work stealing instead of the static parallelism in pthreads. In that sense, it falls into the same category with Intel TBB [23] and OpenMP [1] that supports tasks from Version 3.0. Cilk++ employs a provable optimal task scheduler with minimal parallelization overheads.

We used the CilkArts build 8503 that is based on `gcc 4.2`. In order to take full advantage of the SSE instructions, we compiled our innermost kernel separately with the Intel C++ Compiler 11.1 and linked it with the rest of the application.

C. Experimental Methodology

We ran our code using different number of threads, up to the maximum that is supported by a single socket processor (16 for Nehalem-EX, 6 for Istanbul). We only performed single-socket experiments to decouple the NUMA effects. Optimizing performance for NUMA is an orthogonal issue that is beyond the scope of this paper.

Our baseline is the original CSB code, which has been shown to scale linearly with cores until limited by off-chip memory bandwidth [6], on a variety of difficult matrices. For unsymmetric matrices, we only report the performance of the SpMV operation, but we note that

SpMV_T performance is comparable, both using original CSB and our bitmask enchanced version. This is in stark constrast to parallel algorithms that use row-based (such as CSR and BCSR) formats, where the performance degrades significantly for the SpMV_T case.

The reported numbers are for double precision performance, and the matrix indices are represented as 32-bit unsigned integers. We did not employ any low-level optimizations such as prefetching or TLB blocking, which will be the subject of future work.

D. Matrix Suite

The sparse matrix test suite used in our experiments are shown in Figure 11. These matrices come from a set of real applications such as web-connectivity analysis, nonlinear optimization, and finite element computations. They are chosen to represent a variety of different structures and nonzero densities. Four out of seven matrices are symmetric and we run both unsymmetric and symmetric algorithms on them. For the remaining three matrices, we only run the unsymmetric algorithm.

In the last column of Figure 11, average nonzero density within register blocks, for three different register blocking sizes, are shown in the second row for each matrix. The nonzero density value is also the expected floating-point efficiency (ratio of useful flops to actual flops performed) of the algorithm.

VII. Experimental Results

The performance of our algorithms are shown in Figures 12, using as many threads as available on that architecture. The red baseline stacks represent the performance of the original CSB code, which is publicly available. The circles mark the performance achieved with bitmasked register blocking (best of $r = 2, 4, 8$). For symmetric matrices, the squares mark the performance of the symmetric algorithm without bitmasked register blocking, while the triangles mark the symmetric algorithm with bitmasked register blocking. Finally, the green stack on top of the red one shows the performance improvement over original CSB performance.

We see that bitmasked register blocking improves performance in five out of seven matrices, sometimes by large margins. For example, the performance of Tsopf2, which has a nonzero density of 84% for $r = 8$ case, increased by a factor of 2.2 on Nehalem and 3.5 on Istanbul. Performance boost beyond what is possible with bandwidth reductions are due to SIMDization and smaller parallelization overheads (e.g. during CSB_BlockV parallelization binary searches are performed on much smaller data as the smallest addressible entity is now a register block). At the other end of the spectrum, Wiki2007 matrix has the sparsest register blocks, with the nonzero density for $r = 2$ case being only 26% (the smallest possible value is 25% when all

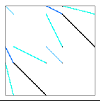
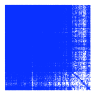
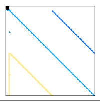
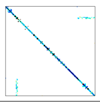
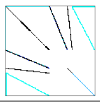
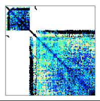
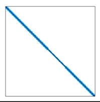
Name Description	Spy Plot	Dimensions Nonzeros Symmetric	nnz/row nonzero density (r = 2,4,8)
Largebasis Optimization problem		440K×440K 5.24M	11.9 (92,45,28)%
Wiki2007 Wikipedia pages		3.56M×3.56M 45.04M	12.6 (26,7,2)%
Tsopf Optimal power flow		35.7K×35.7K 8.78M	246 (99,93,84)%
Wind Wind Tunnel Stiffness Matrix		218K×218K 11.52M ✓	73.3 (81,69,50)%
Kkt_power KKT matrix		2.06M×2.06M 12.77M ✓	6.2 (43,9,3)%
Ldoor structural prob.		952K×952K 42.49M ✓	44.6 (78, 56, 33)%
Bone010 3D trabecular bone		986K×986K 47.85M ✓	48.5 (56,41,25)%

Fig. 11: **Structural information on the sparse matrices used in our experiments. The first three are unsymmetric while the last four are symmetric. All matrices are from the University of Florida sparse matrix collection [11]. Spy plot of Wiki2007 shows only a representative fraction for visibility. The colors in the spy plots only stress the value of the nonzeros (darker colors mean larger values).**

register blocks contain only one nonzero). Consequently, any register blocking merely hurts the performance.

Among symmetric matrices, Wind and LDoor benefit from both symmetry and bitmasked register blocking. On the other hand, KKT_Power benefitted from neither. The reason for the poor performance of the symmetric algorithm on this matrix is due to the large fraction of nonzeros outside the main block diagonals, as shown in Figure 3.

The highest performance gains for the unblocked symmetric algorithm over the unsymmetric is realized for the Wind matrix with 36%. The 50% reduction in memory bandwidth does not readily translate to a 2x improvement, possibly because the number of cache misses due to vector accesses is increased due to the use of temporary output vectors. The symmetric algorithm's scalability is comparable to the unsymmetric version. However, applying bitmasked register blocking increased the performance improvement to over 80% for the Wind matrix.

To decouple the effects of RCM ordering on the performance, we ran our unsymmetric algorithm on the

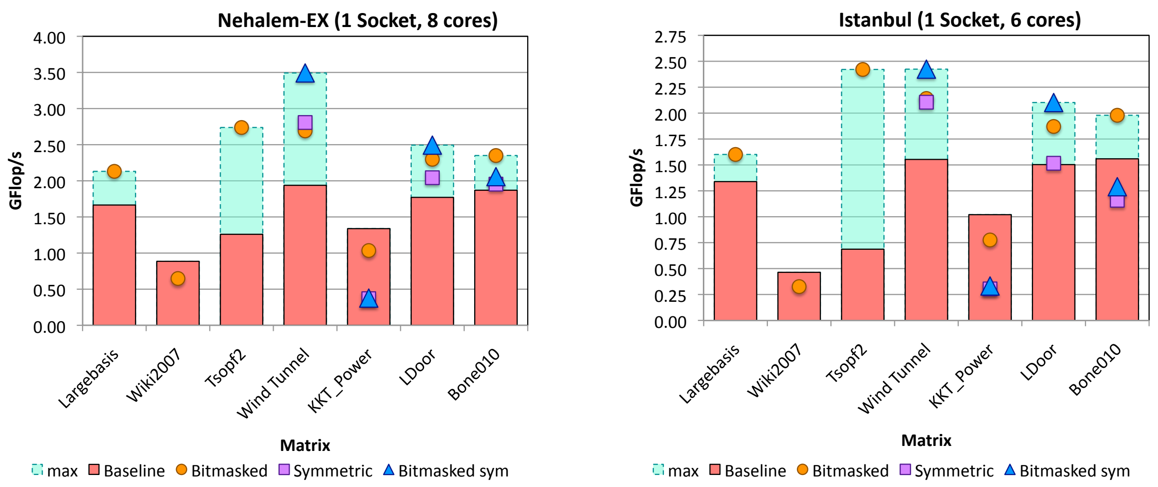


Fig. 12: **Parallel performance of SpMV kernels on Nehalem-EX (*left*) and Istanbul (*right*). Baseline original CSB code is in red; circles show bitmasked register blocks; triangles and squares show symmetric algorithm with and without bitmasked register blocking, respectively; green shows overall performance gain of our approach. The first three matrices are unsymmetric, followed by four symmetric matrices. The unsymmetric algorithms ('Baseline' and 'Bitmasked') run on original matrices while the symmetric algorithms ('Symmetric' and 'Bitmasked sym') run on RCM ordered matrices.**

RCM ordered data as well. For Ldoor and KKT_power matrices, this resulted in denser register blocks, but for Bone010 and Wind Tunnel matrices, it actually decreased the density of register blocks. This is because RCM is designed to reduce bandwidth, not to produce dense blocks. We plan to experiment with different ordering methods, such as the traveling-salesman based ordering [22], as future work.

The parallel scaling results for the symmetric algorithm (with and without register blocking) is shown in Figure 13(middle,right). We achieve $6.3x$ speedup on 8 threads for the Wind Tunnel matrix that completely avoids atomic updates. Note that the scalability of the algorithm is only slightly effected with the integration of bitmasked register blocks. A similar scaling plot for the unsymmetric algorithm is also shown in Figure 13(left), which demonstrates even better scaling. The speedup values for all three algorithms are relative to the single threaded performance of the respective algorithm.

VIII. Related Work

Previous algorithmic work on sparse matrix-vector multiplication has two main directions. First one focuses on reducing communication volume in a distributed-memory setting via hypergraph partitioning [8, 27]. The performance of the partitioning approach mostly depends on the structure of the input matrix. Our work, by contrast, targets multicore and manycore architectures and gives reduced bandwidth algorithms with parallelism guarantees for any nonzero distribution.

The second algorithmic direction strives to achieve optimal theoretical I/O complexity by using cache-oblivious algorithms [3]. From a high-level view, Bender's algorithm first generates all the intermediate triples of the output vector y, possibly with repeating indices. Then, it sorts them with respect to their row indices, summing up triples with same row index on the fly. Although theoretically optimal, sorting based approaches are not competitive in practice yet [14]. A more practical algorithm uses sparse matrix partitioning methods [37]. Both approaches, however, target serial SpMV.

The literature on optimization and tuning of SpMV is extensive. The OSKI [29] framework provides an extensive collection of low-level primitives that provide auto-tuned computational kernels on sparse matrices.

A number consider techniques that compress the data structure by recognizing patterns in order to eliminate the integer index overhead. These patterns include blocks [13], variable or mixtures of differently-sized blocks [12] diagonals, which may be especially well-suited to machines with SIMD and vector units [32, 28], general pattern compression [33], value compression [15], and combinations.

Others have considered improving spatial and temporal locality by rectangular cache blocking [13], diagonal cache blocking [25], and reordering the rows and columns of the matrix. Researchers have also examined low-level tuning of SpMV by unroll-and-jam [20], and software pipelining [12], and prefetching [26]. A completely recursive layout for SpMV, motivated by CSB, is recently examined by Martone et al.[19].

Higher-level kernels and solvers provide opportunities to reuse the matrix itself, in contrast to non-symmetric SpMV. Such kernels include block kernels and solvers that multiply the matrix by multiple dense vectors [13], $A^T Ax$ [30], and matrix powers [24, 28].

A massively parallel implementation [16], mostly targeting to discretized 3D meshes, focuses on scaling

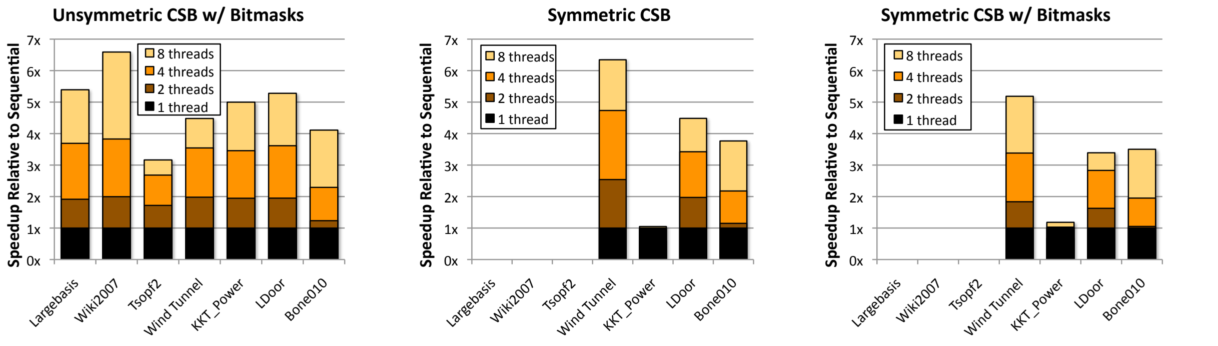


Fig. 13: **Parallel scalability of** Sym_SpMV **as a function of optimization on Nehalem-EX. (left) Bitmasked register blocks, (middle) symmetry, (right) symmetry and register blocks. For the register blocked versions, the scaling of the best register block size (in terms of raw performance on that matrix) is reported. As the first three matrices unsymmetric, the symmetric algorithm is not applicable to them.**

symmetric SpMV to thousands of cores by applying a combination of known methods. Methodologies for exploring symmetry in serial are also examined [17].

A recent study [2] utilized pattern-based accelerated SPMV to reduce indexing overhead by representing repeated sparsity patterns in the matrix with a single index, using specialized kernels to perform the operations. This method avoids filling in zeros by using bit vectors to concisely represent frequently recurring block patterns. Our bitmasking approach is different in the sense that it performs the same multiplication kernel for each register block, regardless of its nonzero pattern. We also provide complete algorithms with asymptotic analysis.

Our previous studies examined auto-tuned optimizations in the context of multicore architectures [34] and demonstrated its significant potential. Given that the focus of this study is primarily algorithmic, these low-level optimization techniques will further enhance the performance of the methods presented in this paper, and will be the subject of future work.

IX. Conclusions

The last decade has seen advances in multicore and SIMD that have dramatically increased the computational capabilities of modern processors. However, the bandwidth these machines may exploit has grown much more slowly. The resultant bandwidth-induced memory wall is an impediment to performance and squanders floating-point capability and will only exacerbate in the future. There is a need for parallel algorithms and methods that reduce bandwidth requirements without compromising scalability. In this paper, we explored two techniques (bitmasked register blocks and symmetry) that tap into the unused computational capability and use it to reduce the memory bandwidth requirements for sparse-matrix vector multiplication. In many respects, these methods are forward looking as they require a high flop:byte to be effective. We evaluated their effectiveness on two state-of-the-art multicore processors, and demonstrated substantial speedups on both. However, as both architectures require an increase in flops, it is quite possible that the evaluated machines lacked the compute capability to perfectly exploit both techniques simultaneously. This effect is transitory as the flop:byte ratio on future machines will continue to increase.

At the low-level, future work will investigate rectangular register blocks as mentioned in Section V, and using variable sized blocks via splitting [31]. A practical approach to address variable blocking that exploits the recursive structure of CSB is as follows. Using the model in Section V-D, first determine the minimum density d that is required to achieve the desired performance for different r values. Then, during the CSB to bitmasked CSB construction, start with the maximum r considered and if its density is less than d, recursively explore its $r/2 \times r/2$ quadrants, else encode it as a register block. This is reminiscent of a quadtree approach except that the leaves do not have to be completely dense.

At the higher-level we will investigate the best approach to extend parallelism to multiple sockets and to multiple nodes. Whether hybrid programming will be necessary in the future depends on many factors such as the availability of a NUMA-aware task scheduler and the unstructured nature of the target matrices.

Finally, in the context of hardware-software co-design, we will explore how ISA changes can facilitate the implementation and performance of bitmasked register blocks as well as whether changes to the memory model or cache hierarchy can simplify the data synchronization challenges we currently resolve via algorithms and atomics.

Acknowledgments

We thank Grey Ballard of UC Berkeley for his constructive comments on the paper, especially the algorithms analysis. All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. Additional support comes from Microsoft (Award #024263) and Intel (Award #024894) funding, by matching funding by U.C.

Discovery (Award #DIG07-10227), and from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

References

[1] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[2] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *ICS*, 2009.

[3] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *Proc SPAA '07*, pages 61–70. ACM, 2007.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, Santa Barbara, CA, July 1995.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, September 1999.

[6] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*, pages 233–244, 2009.

[7] Aydın Buluç and John R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IPDPS*, pages 1–11. IEEE, 2008.

[8] Umit V. Catalyurek and Cevdet Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Computing*, 10:673–693.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[10] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference*, pages 157–172, New York, NY, USA, 1969. ACM.

[11] Timothy A. Davis. University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[12] R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. *Parallel Computing*, 27(1):883–896, 2001.

[13] 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.

[14] R. Jacob and R. Schnupp. Experimental performance of I/O-optimal sparse matrix dense vector multiplication algorithms within main memory. In *PARA*, 2010. extended abstract no. 104.

[15] K. Kourtis, G. I. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Conf. Computing Frontiers*, pages 87–96, 2008.

[16] M. Krotkiewski and M. Dabrowski. Parallel symmetric sparse matrix-vector product on scalar multi-core cpus. *Parallel Comput.*, 36(4), 2010.

[17] B. C. Lee, R. Vuduc, J. Demmel, and K. Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *ICPP*, Montreal, Canada, August 2004.

[18] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, R. Strohmaier, and K. A. Yelick. Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In *SC*. ACM, 2009.

[19] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci. On BLAS operations with recursively stored sparse matrices. In *SYNASC*, 2010.

[20] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix vector multiply using unroll-and-jam. In *Proc. LACSI Symposium*, Santa Fe, NM, USA, October 2002.

[21] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, March 1966.

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

[23] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[24] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–114, February 2004.

[25] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proc. Supercomputing*, 1992.

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

[27] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

[28] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.

[29] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, June 2005.

[30] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick. Memory hierarchy optimizations and bounds for sparse $A^T Ax$. In *Proceedings of the ICCS Workshop on Parallel Linear Algebra*, volume LNCS, Melbourne, Australia, June 2003. Springer.

[31] R. Vuduc and H. J. Moon. Fast sparse matrix vector multiplication by exploiting variable block structure. In *Proceedings of the International Conference on High-Performance Computing and Communications*, LNCS 3726, pages 807–816, Sorrento, Italy, September 2005.

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

[33] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS*, Cairns, Australia, June 2006.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[35] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[36] Samuel Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[37] A. N. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. *SIAM J. Sci. Comput.*, 31(4):3128–3154, 2009.