SAND2016-2968C

Optimization of Block Sparse Matrix-Vector Multiplication on Shared-Memory Parallel Architectures

Ryan Eberhardt
William Rainey Harper College
Palatine, IL
r_eberhardt4@mail.harpercollege.edu

Mark Hoemmen
Sandia National Laboratories
Albuquerque, NM
mhoemme@sandia.gov

***Abstract*—We examine the implementation of block compressed row storage (BCSR) sparse matrix-vector multiplication (SpMV) for sparse matrices with dense block substructure, optimized for blocks with sizes from 2x2 to 32x32, on CPU, Intel many-integrated-core, and GPU architectures. Previous research on SpMV for matrices with dense block substructure has largely focused on the design of novel data structures to optimize performance for specific architectures or to store variable-sized, variably-aligned blocks, but depending on alternate storage formats breaks compatibility with existing preconditioners and solvers or imposes significant runtime costs when converting between matrix formats. This paper instead focuses on the optimization of SpMV using the standard block compressed row storage (BCSR) format. We give a set of algorithms that performs SpMV up to 4x faster than the NVIDIA cuSPARSE cusparseDbsrmv routine, up to 147x faster than the Intel Math Kernel Library (MKL) mkl_dbsrmv routine (a single-threaded BCSR SpMV kernel), and up to 3x faster than the MKL mkl_dcsrmv routine (a multi-threaded CSR SpMV kernel).**

I. Introduction

Sparse matrix-vector multiplication (SpMV) computes the operation $\boldsymbol{y} \leftarrow \alpha \boldsymbol{y} + \beta \boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{A}$ is a sparse matrix and $\boldsymbol{x}$ and $\boldsymbol{y}$ are dense vectors. It dominates the running time in many scientific and engineering applications and is notorious for sustaining low percentages of machine peak performance due to its typically poor cache usage and memory-bound nature. Improving performance of SpMV generally requires selecting appropriate data structure transformations and memory access patterns for the matrices being used and the underlying hardware architecture. Sparse matrices arising from finite-element analysis often exhibit a dense block substructure, as do matrices from discretizations in which each node in a graph has multiple degrees of freedom. This block substructure can be exploited to represent a sparse matrix with less space. Block Compressed Sparse Row (BCSR) is the most popular format for representing general sparse matrices with constant-sized blocks; it achieves high performance in the general case without being overly dependent on matrix structure.

This paper contributes algorithms for the efficient execution of SpMV using BCSR on shared-memory parallel architectures, including traditional CPU architectures (e.g., Sandy Bridge), the Intel Knights Corner (KNC) many-integrated-core architecture, and NVIDIA GPU architectures.

II. Related Work

A great amount of literature exists regarding the optimization of SpMV for various parallel architectures using standard non-block formats. Williams et al. investigate the tuning of CSR SpMV on AMD and Intel CPU architectures (among others) [7], and Bell and Garland compare the performance of SpMV using a variety of sparse matrix formats on NVIDIA GPUs [1]. Some have developed new sparse matrix representations to optimize performance on specific architectures [3]. Other researchers have developed alternate matrix formats to optimize SpMV when matrices exhibit block substructure. Vuduc and Moon offer the Unaligned Block Compressed Sparse Row (UBCSR) format [6] and Shahnaz and Usman present the Sparse Block Compressed Row Storage (SBCRS) format for matrices with variable-sized and variably-aligned blocks [4].

However, algorithms depending on alternate matrix formats are often impractical, as they require rewriting other algorithms such as preconditioners and solvers to use the new format or impose significant runtime overhead in converting between formats. Algorithms optimized for specific architectures may also be impractical in heterogeneous computing environments where data structures are passed between different architecutres. Little research has been published regarding the optimization of SpMV using the standard Block Compressed Sparse Row (BCSR) format on parallel architectures. In this paper, we explore general-purpose algorithms compatible with existing solvers and preconditioners for the BCSR format and compare them against the proprietary algorithms in the NVIDIA cuSPARSE and Intel MKL libraries.

III. BCSR SpMV on Shared-Memory Parallel Architectures

A. Storage Format – Block CSR

The algorithms presented in this paper use sparse matrices stored in the Block Compressed Sparse Row (BCSR/BSR) format with column-major blocks. BCSR is the most popular blocked sparse matrix storage format, as it performs well on nearly any matrix, is not highly dependent on matrix structure, and is a simple data structure to construct.

BCSR stores nonzero blocks contiguously, row by row, in an array val of length $nnzb \cdot bs^2$; for each nonzero block, an entry is added to an array col_idx of length $nnzb$ with its column index. An array row_ptr stores $m+1$ values, where each value stores the index of a row's first block in val and col_idx. For a more in-depth description of the Block CSR format, please refer to Richard Vuduc's Ph.D. thesis [5].

We have elected to use column-major blocks in order to improve temporal cache locality for $\boldsymbol{x}$. By streaming through columns of val, we access $\boldsymbol{x}$ consecutively; a row-major layout would access segments of val multiple times, causing L1 evictions for larger blocks. Additionally, using column-major blocks simplifies the reductions for the GPU algorithms presented in Sections III-D1 and III-D2.

B. Primary Algorithm

In our algorithm, we assign a parallel worker (a thread or core for CPU and CPU-like architectures, or a warp of threads for GPU architectures) to one or more block rows. In each of its block rows, a parallel worker iterates down each column, across the block row, in order to achieve streaming access to the val array (which has column-major blocks stored row-wise). The manner in which a parallel worker does this is tuned for different hardware architectures (e.g., for GPU architectures, each warp has threads enabling it to iterate through multiple columns at a time) and is described in Sections III-C and III-D.

C. CPU and Many-Integrated-Core Architectures

For these architectures, we delegate the block rows of the sparse matrix across n threads, such that each thread is responsible for approximately mb/n block rows.

Each thread iterates over its block rows serially; for each block row, a thread retrieves pointers to the start and end of the row in the val and col_idx arrays by retrieving $row\_ptr[thread\_idx]$ and $row\_ptr[thread\_idx+1]$. It then iterates from this first block pointer to the last block pointer. In each block iteration, the algorithm finds the column index of the block in $col\_idx[block\_ptr]$ and iterates over the columns within the block. In each column iteration, the algorithm retrieves $\boldsymbol{x}[block][col]$, then iterates through each element in the column, multiplying by $\boldsymbol{x}[block][col]$ and updating a temporary output array in registers. After processing the block row, the thread updates the output array $\boldsymbol{y}$. No interactions between threads are present.

A pseudocode implementation of this algorithm is shown in Algorithm 1 and a diagram of the memory access pattern is shown in Figure 1.

Memory access patterns are highly optimal for this algorithm. Each thread achieves streaming access to val; the access pattern is also predictable by a hardware prefetcher,

```
1: $target\_block\_row \leftarrow thread\_idx$
2: $first\_block \leftarrow row\_ptr[target\_block_row]$
3: $last\_block \leftarrow row\_ptr[target\_block_row + 1]$
4: $local\_out[bs] \leftarrow$ zero-initialized array of $bs$ elements
5: **for** $block \leftarrow first\_block$; $block < last\_block$; $block$++ **do**
6:     $target\_block\_col \leftarrow col\_idx[block]$
7:     **for** $c \leftarrow 0$; $c < bs$; $c$++ **do**
8:         $vec\_this\_col \leftarrow \boldsymbol{x}[target\_block\_col][c]$
9:         **for** $r \leftarrow 0$; $r < bs$; $r$++ **do**
10:            $local\_out[r] \mathrel{+}= \boldsymbol{A}[block][c][r] \cdot vec\_this\_col$
11:        **end for**
12:    **end for**
13: **end for**
14: **for** $r \leftarrow 0$; $r < bs$; $r$++ **do**
15:     $\boldsymbol{y}[target\_block\_row][r] \mathrel{+}= local\_out[r]$
16: **end for**
```

Algorithm 1. BCSR SpMV kernel for CPU architectures. r and c denote a thread's position (row, column) within a block.

so access latencies are reduced. If the architecture offers a large cache and bs is small, the algorithm also achieves streaming access to $\boldsymbol{x}$ and col_idx. row_ptr is accessed only twice per thread, so its memory access cost is generally insignificant.

This algorithm is also friendly to SIMD vectorization. If bs is known at compile time, the innermost loop over the elements in a column has compile-time bounds and can be vectorized by the compiler. In our tests, vectorization provided up to 4.7x the performance of non-vectorized code on KNC.

Reuse of $\boldsymbol{x}$ is limited; though better than naive non-block CSR, $\boldsymbol{x}$ is reused a maximum of bs times. Performance could be improved by tiling blocks to improve temporal locality; this possibility is discussed in Section V, Conclusions and Future Work.

D. GPU Architectures

In this section, we seek to adapt the previous algorithm for highly multi-threaded GPU architectures. The CUDA programming model operates with a large number of threads operating in SIMT (single instruction, multiple thread) style, where a group of 32 threads (known as a warp) concurrently execute the same instruction. The threads in a warp must work cooperatively; if threads diverge and issue different sets of instructions, the warp scheduler will mask off parts of the warp, executing different branch paths separately and reducing the rate at which instructions can be issued.

For optimal performance, the threads in a warp must also access contiguous segments of memory. NVIDIA devices have a global memory bus width of 128 bytes, which can hold 16 8-byte double-precision values. If 16 threads access a sequential range of doubles that lie in a 128-byte row of DRAM in the same SIMD instruction issue, the memory accesses is “coalesced” into a single memory access instruction.

In the following three sections, we propose algorithms that each assign a warp to a single block row, but assign

$$\begin{bmatrix} \begin{bmatrix} \text{T0, I0} & \text{T0, I2} \\ \text{T0, I1} & \text{T0, I3} \end{bmatrix} & \begin{bmatrix} \text{T0, I4} & \text{T0, I6} \\ \text{T0, I5} & \text{T0, I7} \end{bmatrix} & \begin{bmatrix} \text{T0, I8} & \text{T0, I10} \\ \text{T0, I9} & \text{T0, I11} \end{bmatrix} \\ \begin{bmatrix} \text{T1, I0} & \text{T1, I2} \\ \text{T1, I1} & \text{T1, I3} \end{bmatrix} & \begin{bmatrix} \text{T1, I4} & \text{T1, I6} \\ \text{T1, I5} & \text{T1, I7} \end{bmatrix} & \begin{bmatrix} \text{T1, I8} & \text{T1, I10} \\ \text{T1, I9} & \text{T1, I11} \end{bmatrix} \\ \begin{bmatrix} \text{T2, I0} & \text{T2, I2} \\ \text{T2, I1} & \text{T2, I3} \end{bmatrix} & \begin{bmatrix} \text{T2, I4} & \text{T2, I6} \\ \text{T2, I5} & \text{T2, I7} \end{bmatrix} & \begin{bmatrix} \text{T2, I8} & \text{T2, I10} \\ \text{T2, I9} & \text{T2, I11} \end{bmatrix} \end{bmatrix}$$

Figure 1. Matrix access pattern for CPU architectures. T_a, I_b indicates thread a, iteration b. Each thread is responsible for one block row.

the threads within a warp to the elements in a block row in different ways based on the block size of the matrix.

1) Block row per warp, operating by block: In this algorithm, a warp is assigned to a single block row, and threads in the warp are assigned to elements in the block row column-wise (see Figure 2 for an illustration of thread assignment). To cover the entire block row, a warp iterates over the row's blocks, handling $32/bs^2$ blocks at a time, where bs is the block size. This is the number of blocks that a warp can cover completely if each thread is assigned to a single element. For example, with a block size of 3, the warp can cover 3 complete blocks (27 elements) at a time. The warp will only cover complete blocks; in the case of 3x3 blocks, five threads will be inactive in each iteration $(32 - (3 \cdot 3) \cdot 3)$.

Because a warp only covers complete blocks, a thread's assigned position within a block will never change between iterations. On initialization, a thread calculates the index of the block row it is targeting, finds pointers to the start and end of its row from row_ptr, and finds a pointer to the block it should begin its work on using the row start pointer and its lane number. It also calculates its assigned position within a block (r and c). It then iterates through the block row, beginning at its assigned block and advancing by $32/bs^2$ blocks at a time (the number of complete blocks that the warp can cover), multiplying its target element in $\boldsymbol{A}$ by the corresponding value from $\boldsymbol{x}$ and adding the product to a register that it uses to maintain a running total. Once the threads in a warp have iterated through a block row, threads that covered the same vertical (r) position in a block reduce the values stored in their local registers and write the reduced output to global memory (Figure 3).

This algorithm is described in pseudocode in Algorithm 2, and its memory access pattern is illustrated in Figure 2. The reduction step could be implemented using a warp shuffle but is shown using shared memory in Algorithm 2.

This algorithm exhibits high-performing memory access patterns for val and $\boldsymbol{x}$. Accesses to val will be fully coalesced. Accesses to $\boldsymbol{x}$ are fully coalesced when the block size is 2 and partially coalesced for larger block sizes. Accesses to row_ptr and writes to $global\_out$ are generally not coalesced, but these transactions represent an

```
$bs \leftarrow$ block size
$target\_block\_row \leftarrow (t\_block\_idx \cdot t\_block\_dim + t\_idx)$ / 32
$lane \leftarrow t\_idx$ % 32
$first\_block \leftarrow row\_ptr[target\_block\_row]$
$last\_block \leftarrow row\_ptr[target\_block\_row + 1]$
$target\_block \leftarrow first\_block + lane$ / $(bs \cdot bs)$
$c \leftarrow (lane$ / $bs)$ % $bs$
$r \leftarrow lane$ % $bs$
▷ Shared memory for reduction step:
$s\_out \leftarrow t\_block\_size \cdot sizeof(double)$ bytes shared mem
$s\_out[t\_idx] \leftarrow 0$
▷ Only process whole blocks (disable threads that can only cover partial blocks):
if $lane < (32$ / $bs^2) \cdot bs^2$ then
    ▷ Iterate through block row:
    $local\_out \leftarrow 0$
    for ; $target\_block < last\_block$; $target\_block$ += 32 / $bs^2$ do
        $x\_elem \leftarrow \boldsymbol{x}[col\_ind[target\_block]][c]$
        $A\_elem \leftarrow \boldsymbol{A}[target\_block][c][r]$
        $local\_out$ += $x\_elem \cdot A\_elem$
    end for
    ▷ Reduction:
    $s\_out \leftarrow local\_out$
    $stride \leftarrow round\_up\_to\_power\_of\_two((32$ / $bs)$ / 2)
    for ; $stride \geq 1$; $stride$ /= 2 do
        if $lane < stride \cdot bs$ and $lane + stride \cdot bs < 32$ then
            $s\_out[t\_idx]$ += $s\_out[t\_idx + stride \cdot bs]$
        end if
    end for
    ▷ Write reduced value to global memory:
    if $lane < bs$ then
        $\boldsymbol{y}[target\_block\_row][lane]$ += $s\_out[t\_idx]$
    end if
end if
```

Algorithm 2. Block-by-block BCSR SpMV kernel for GPU architectures. r and c denote a thread's position (row, column) within a block. $thread\_index$, $thread\_block\_index$, $thread\_block\_dim$, and $shared\_out$ have been shortened to t_idx, t_block_idx, t_block_dim, and s_out, respectively.

insignificant portion of all memory operations.

Unfortunately, accesses to col_idx are generally not coalesced, and since the values in col_idx are required to access $\boldsymbol{x}$, this adds latency to those memory transactions. However, as the algorithm uses few registers, many warps may fit into the streaming multiprocessors, so the GPU can generally hide the latency and keep the memory bus saturated by switching between warps.

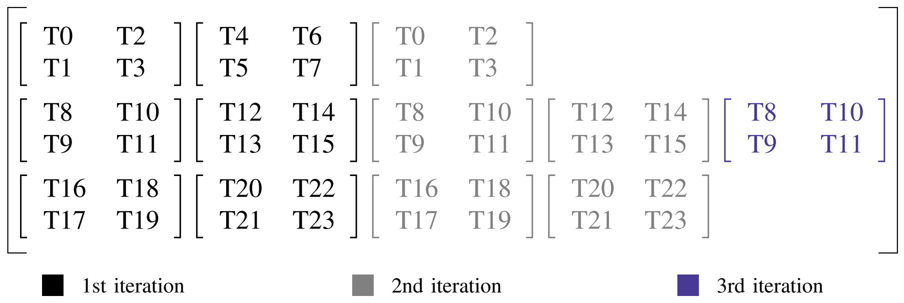

T0 T2 T4 T6 T0 T2
T1 T3 T5 T7 T1 T3
T8 T10 T12 T14 T8 T10 T12 T14 T8 T10
T9 T11 T13 T15 T9 T11 T13 T15 T9 T11
T16 T18 T20 T22 T16 T18 T20 T22
T17 T19 T21 T23 T17 T19 T21 T23
1st iteration
2nd iteration
3rd iteration

Figure 2. Matrix access pattern of the by-block algorithm for a matrix with 2x2 blocks. The warp size has been reduced to 8 for the purposes of visualization (though in practice, the warp size will always be 8). Warp 0 (threads 0-7) handles block row 0, warp 1 (threads 8-15) handles block row 1, and warp 2 (threads 16-23) handles block row 2.

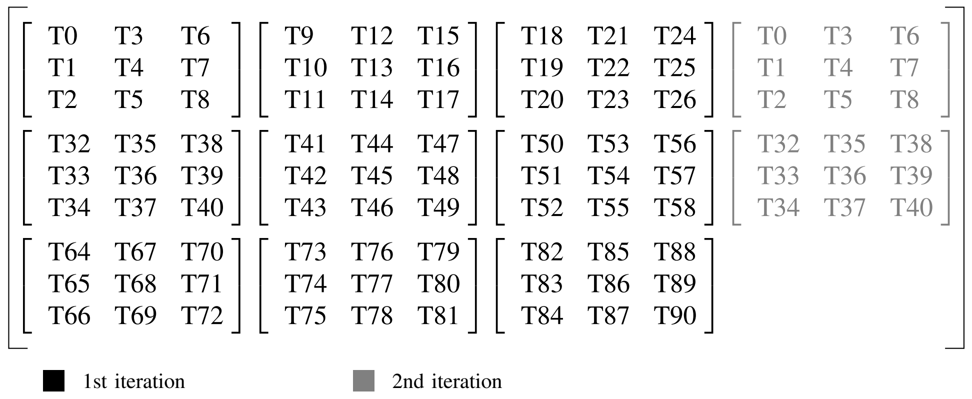

T0 T3 T6 T9 T12 T15 T18 T21 T24 T0 T3 T6
T1 T4 T7 T10 T13 T16 T19 T22 T25 T1 T4 T7
T2 T5 T8 T11 T14 T17 T20 T23 T26 T2 T5 T8
T32 T35 T38 T41 T44 T47 T50 T53 T56 T32 T35 T38
T33 T36 T39 T42 T45 T48 T51 T54 T57 T33 T36 T39
T34 T37 T40 T43 T46 T49 T52 T55 T58 T34 T37 T40
T64 T67 T70 T73 T76 T79 T82 T85 T88
T65 T68 T71 T74 T77 T80 T83 T86 T89
T66 T69 T72 T75 T78 T81 T84 T87 T90
1st iteration
2nd iteration

Figure 4. Matrix access pattern of the by-block algorithm for a matrix with 3x3 blocks. Note that five threads per warp (e.g., warp 0, threads 27-31) are left inactive, as they do not cover a complete block.

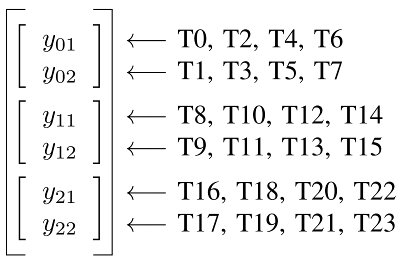

y_{01} ⟵ T0, T2, T4, T6
y_{02} ⟵ T1, T3, T5, T7
y_{11} ⟵ T8, T10, T12, T14
y_{12} ⟵ T9, T11, T13, T15
y_{21} ⟵ T16, T18, T20, T22
y_{22} ⟵ T17, T19, T21, T23

Figure 3. Warp reduction for the matrix-vector multiplication example shown in Figure 2. Each thread in a warp reduces with other threads that also covered the same row within a block; the output is then written to global memory.

Each thread accumulates a result in a register and there is no interaction between threads until the final reduction, so latency is low. The reduction always occurs within a warp, so no synchronization primitives are required.

While this algorithm theoretically achieves 100% theoretical thread utilization for block sizes that are powers of two (because warps can cover blocks evenly, with no threads leftover), it potentially produces a large number of inactive threads for other block sizes. Consider the case of 3x3 blocks; a warp can cover only 3 full blocks (27 elements), leaving 5 threads in each warp (15.6%) inactive. (See Figure 4.) The algorithm uses few registers and achieves high occupancy, so it is generally able to issue enough instructions from a large number of warps to keep the memory bus saturated despite this decrease in active threads. However, this is an important consideration, and performance does decrease relative to the algorithm described in the next section (which is able to handle these cases more efficiently) for 3x3 and 5x5 blocks (see Figure 9).

A more significant problem is the block size limitation that the whole-block constraint imposes. Because a warp of only 32 threads must cover an entire block, this algorithm cannot multiply matrices that have block sizes larger than 5. This motivates the design of a similar algorithm that better handles large block sizes.

2) Block row per warp, operating by column: This algorithm is similar to the previous one, but relaxes the requirement that warps cover whole blocks. Instead, warps cover whole columns, enabling larger block sizes (up to 32x32, where a warp would cover a single column) and a more efficient handling of matrices with block sizes that are not powers of two. Unlike the previous algorithm, a warp iterates through its block row's *columns*, covering $32/bs$ columns at a time. With the whole-block requirement replaced with a whole-column requirement, the assigned vertical position of a thread within a block will never change, but a thread must compute its target block and target column in every iteration. The algorithm is shown in pseudocode in Algorithm 3 and its memory access pattern for 3x3 blocks – improved over that of the previous algorithm – is shown in Figure 5.

```
$bs \leftarrow$ block size
$target\_block\_row \leftarrow (t\_block\_idx \cdot t\_block\_size + t\_idx)$ / 32
$lane \leftarrow t\_idx \ \% \ 32$
$first\_block \leftarrow row\_ptr[target\_block\_row]$
$last\_block \leftarrow row\_ptr[target\_block\_row + 1]$
$target\_col \leftarrow first\_block \cdot bs + lane \ / \ bs$
$r \leftarrow lane \ \% \ bs$
▷ Shared memory for reduction step:
$s\_out \leftarrow t\_block\_size \cdot sizeof(double)$ bytes shared mem
$s\_out[t\_idx] \leftarrow 0$
▷ Only process whole columns:
if $lane < (32 \ / \ bs) \cdot bs$ then
    ▷ Iterate through columns:
    $local\_out \leftarrow 0$
    for ; $target\_col < last\_block \cdot bs$; $target\_col$ += 32 / $bs$ do
        $block \leftarrow target\_col \ / \ bs$
        $c \leftarrow target\_col \ \% \ bs$
        $A\_elem \leftarrow \boldsymbol{A}[block][c][r]$
        $x\_elem \leftarrow \boldsymbol{x}[col\_ind[block]][c]$
        $local\_out$ += $x\_elem \cdot A\_elem$
    end for
    ▷ Reduction:
    $s\_out \leftarrow local\_out$
    $stride \leftarrow round\_up\_to\_power\_of\_two((32 \ / \ bs) \ / \ 2)$
    for ; $stride \geq 1$; $stride$ /= 2 do
        if $lane < stride \cdot bs$ and $lane + stride \cdot bs < 32$ then
            $s\_out[t\_idx]$ += $s\_out[t\_idx + stride \cdot bs]$
        end if
    end for
    ▷ Write reduced value to global memory:
    if $lane < bs$ then
        $\boldsymbol{y}[target\_block\_row][lane]$ += $s\_out[t\_idx]$
    end if
end if
```

Algorithm 3. Column-by-column BCSR SpMV kernel for GPU architectures.

Like the previous block-by-block algorithm, this algorithm has minimal interaction between threads and achieves coalesced or partially coalesced accesses to val and $\boldsymbol{x}$, but it is now able to handle large block sizes more efficiently. For the case of a matrix with 3x3 blocks, this algorithm has only 2 inactive threads (an improvement from the 5 inactive threads of the previous algorithm for this scenario). However, this comes at the cost of increased latency from integer operations. The algorithm must compute a block index and a horizontal position within that block, and the memory requests stall on these operations. By maximizing occupancy, we are usually able to minimize this additional latency, but the previous algorithm tends to outperform this one for block sizes up to 5x5.

3) Row-per-thread: When block sizes are sufficiently large, a simpler, more efficient algorithm may be introduced. For these cases, a CUDA thread block is assigned to a block row, and each thread within a thread block is responsible for a single row within the block row. A thread iterates through its assigned (non-block) row, multiplying each element by the appropriate value from $\boldsymbol{x}$ and accumulating the results in a register. A simple version of this algorithm is shown in Algorithm 4.

```
$bs \leftarrow$ block size
$target\_block\_row \leftarrow thread\_block\_idx$
$r \leftarrow lane \leftarrow thread\_idx$
$first\_block \leftarrow row\_ptr[target\_block\_row]$
$last\_block \leftarrow row\_ptr[target\_block\_row + 1]$
if $r < bs$ then
    $local\_out \leftarrow 0$
    for $block \leftarrow first\_block$; $block < last\_block$; $block$++ do
        for $c \leftarrow 0$; $c < bs$; $c$++ do
            $A\_elem \leftarrow \boldsymbol{A}.val[block][c][r]$
            $x\_elem \leftarrow \boldsymbol{x}[col\_ind[block]][c]$
            $local\_out$ += $x\_elem \cdot A\_elem$
        end for
    end for
    $global\_out[target\_block\_row][r] \leftarrow local\_out$
end if
```

Algorithm 4. Simple row-per-thread BCSR SpMV kernel.

This algorithm achieves fully-coalesced accesses to val when the block size is a multiple of 16 and partially-coalesced accesses when this is not the case. As threads use few registers, depend on little arithmetic for memory requests, and do not interact with other threads, occupancy is high and latency is low. Additionally, the inner loop can be unrolled for decreased instruction traffic and increased instruction-level parallelism.

In the basic implementation shown in Algorithm 4, access to $\boldsymbol{x}$ is not coalesced. This problem can be remedied by loading a portion of $\boldsymbol{x}$ into shared memory in a fully-coalesced fashion, where it can be reused by all threads in a thread block. An implementation of this is shown in Algorithm 5. Using shared memory in this way requires two barrier synchronizations for every block iteration, but we found this cost not to be significant.

We find that this algorithm performs best for block sizes $\geq$ 16, as access to val and $\boldsymbol{x}$ have the opportunity to be fully-coalesced. See Figure 9 in Section IV-B.

The performance of this algorithm would seem to depend heavily on how well the matrix block size is matched to the thread block size; since the thread block size must be a multiple of 32, it would seem that the algorithm would perform poorly for a block size of 33, where 31 of 64 threads would be inactive. While performance certainly drops in this case, we did not notice as great of a performance impact as expected. The algorithm appears to have sufficient occupancy and instruction-level parallelism to hide latency

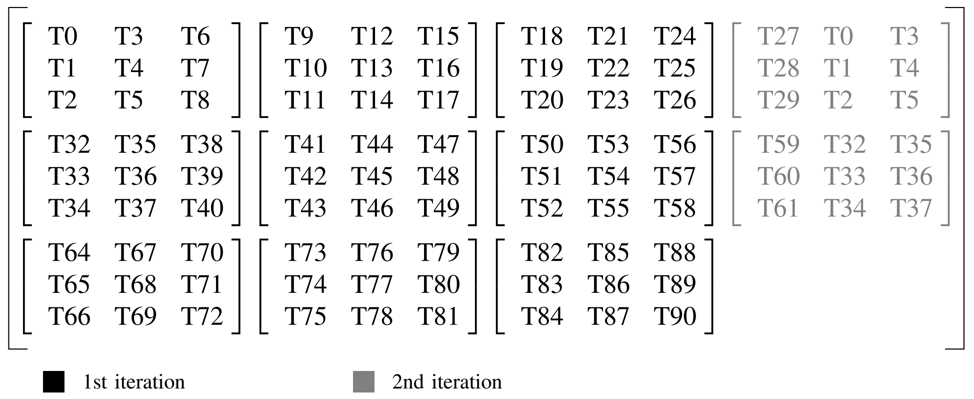


Figure 5. Matrix access pattern of the by-column algorithm for a matrix with 3x3 blocks. Note that for this case, only two two threads per warp (e.g., warp 0, threads 30-31) are left inactive, an improvement over the by-block algorithm.

```
1: $bs \leftarrow$ block size
2: $target\_block\_row \leftarrow t\_block\_idx$
3: $r \leftarrow lane \leftarrow t\_idx$
4: $first\_block \leftarrow row\_ptr[target\_block\_row]$
5: $last\_block \leftarrow row\_ptr[target\_block\_row + 1]$
6: $shared\_x \leftarrow t\_block\_size \cdot sizeof(double)$ bytes shared mem
7: if $r < bs$ then
8:     $local\_out \leftarrow 0$
9:     for $block \leftarrow first\_block$; $block < last\_block$; $block{+}{+}$ do
10:        Barrier synchronization
11:        $shared\_x[t\_idx] \leftarrow \boldsymbol{x}[col\_ind[block]][t\_idx]$
12:        Barrier synchronization
13:        for $c \leftarrow 0$; $c < bs$; $c{+}{+}$ do
14:            $local\_out \mathrel{+}= shared\_x[c] \cdot \boldsymbol{A}.val[block][c][r]$
15:        end for
16:    end for
17:    $global\_out[target\_block\_row][r] \leftarrow local\_out$
18: end if
```

Algorithm 5. Improved row-per-thread BCSR SpMV kernel with shared memory for $\boldsymbol{x}$.

despite the drop in active threads.

IV. Experimental Results

In this section, we examine the performance of our algorithms and compare them against vendor implementations (Intel Math Kernel Library and NVIDIA cuSPARSE) on Intel Sandy Bridge, Intel Knights Corner, and NVIDIA Kepler architectures.

Experiments with Intel hardware were run on the Sandia National Laboratories Compton test bed with two 8-core Sandy Bridge Xeon E5-2670 processors running at 2.6GHz (see Table I) and a KNC Xeon Phi 3120A card (see Table II). Intel ICC 15.0.2 and MKL 11.2.2.164 were used. We compared our algorithm against MKL's mkl_dcsrmv and mkl_dbsrmv functions (CSR and BCSR, respectively). The mkl_dbsrmv routine is not multithreaded,[1] so we have also compared our algorithm against mkl_dcsrmv as a multithreaded reference.

Experiments with NVIDIA hardware were run on the Shannon test bed with an NVIDIA K80S dual-GPU card (see Table III). Only one GPU on the card was used to run the tests. ECC was on and Boost Clock was off, which reduced the theoretical maximum bandwidth. GCC 4.9.0 and CUDA 7.0.18 were used. Implementations of our GPU algorithms used a texture cache to optimize accesses to $\boldsymbol{x}$. We compared our algorithms against cuSPARSE's cusparseDbsrmv function.

Test matrices were obtained from the University of Florida Sparse Matrix Collection [2] and are shown in Table IV. Matrices were selected from a variety of real-world applications.

To measure the performance of algorithms, we measured the time it took to execute them 3000 times and then divided that time by 3000 to obtain an average execution time. We divided the execution time by the amount of unique data read (i.e. the total size of $\boldsymbol{x}$, $\boldsymbol{A}.val$, $\boldsymbol{A}.col\_idx$, and $\boldsymbol{A}.row\_ptr$) to determine the achieved throughput. This throughput represents the performance of the algorithm only and does not include the population of the input matrices or the zero-filling of the output matrices.

All computations were performed using double-precision values.

A. Algorithm Performance

We partitioned our test matrices into their dominant block sizes (see Table IV) and compared our algorithms to vendor implementations. On Sandy Bridge, all 16 cores were used and hyperthreading was enabled; on KNC, all 57 cores were

[1]The recent MKL 11.3 introduced a multithreaded BCSR algorithm for iterative solvers in the inspector-executor API, involving a separate setup/inspection phase in which MKL analyzes the sparsity pattern and applies matrix transformations. We did not compare against this algorithm.

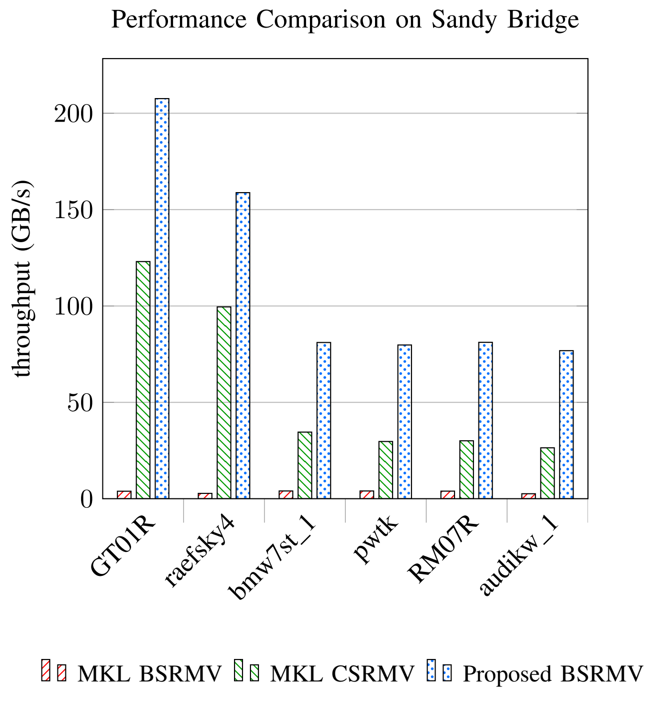


Figure 6. Performance comparison on the Sandy Bridge CPU architecture for various test matrices using each matrix's dominant block size (see Table IV).

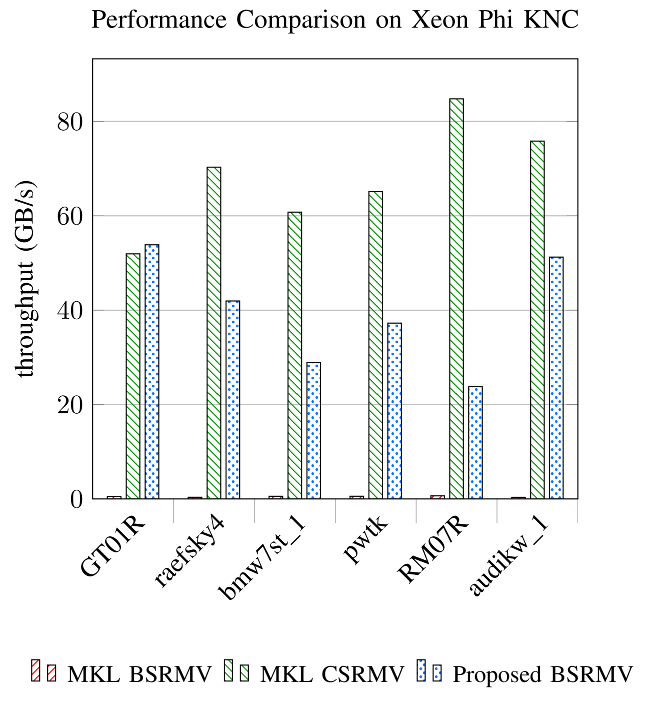


Figure 7. Performance comparison on the Knights Corner (KNC) Xeon Phi architecture for various test matrices using each matrix's dominant block size (see Table IV).

Clock speed	2.60 GHz
Cores	8
Threads	16 (2 hyperthreads/core)
L2 cache per core	256 KB
L3 cache	20 MB
Max memory bandwidth	51.2 GB/s

Table I. Overview of the Intel Xeon E5-2670 processor

Clock speed	1.10 GHz
Cores	57
Threads	228 (4 hardware threads/core)
L2 cache	28.5 MB
Max memory bandwidth	240 GB/s

Table II. Overview of the Intel Xeon Phi 3120A accelerator

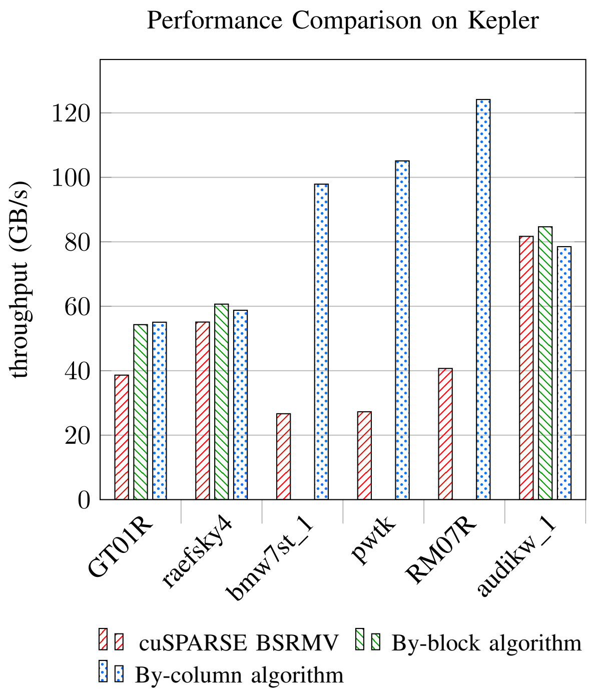


Figure 8. Performance comparison on the Kepler GPU architecture for various test matrices using each matrix's dominant block size (see Table IV). The blocks of bmw7st_1, pwtk, and RM07R are larger than 5x5, so they cannot be handled by the by-block algorithm. No matrices in our test set had blocks large enough to justify the use of the row-per-thread algorithm, so we have omitted it from this comparison.

used with all 4 hardware threads active on each core. Results are shown in Figures 6, 7, and 8.

Performance on Sandy Bridge is the most consistent of all three architectures. Our CPU algorithm exhibits L3 cache effects for the smallest two matrices (GT01R and raefsky4) but achieves a consistent throughput of approximately 80GB/s for the rest of the matrices (78% of the maximum bandwidth of two Xeon E5-2670 processors).

Our algorithm did not perform well on the Xeon Phi KNC architecture, sustaining approximately 40GB/s throughput (16% of the theoretical maximum bandwidth). We suspect this is due to the poor performance of gather/scatter instructions on the KNC architecture – this is discussed further in Section IV-C. MKL's BSRMV algorithm also performs

Core clock speed	562 MHz
Memory clock speed	2505 MHz
Max memory bandwidth (with ECC off)	240 GB/s

Table III. Overview of the NVIDIA Tesla K80S GPU

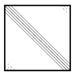
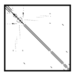
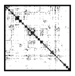
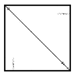
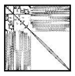
Plot	Name	bs	Dimensions (in blocks)	nnzb (nnzb/row)	Description
	GT01R	5	1.60K x 1.60K	20.37K (13)	2D inviscid fluid
	raefsky4	3	6.59K x 6.59K	111.4K (17)	Container buckling problem
	bmw7st_1	6	23.6K x 23.6K	229.1K (10)	Car body analysis
	pwtk	6	36.3K x 36.3K	289.3K (8)	Pressurized wind tunnel
	RM07R	7	545K x 545K	1.504M (28)	3D viscous turbulence
	audikw_1	3	314K x 314K	4.471M (14)	AUDI crankshaft model

Table IV. Overview of test block matrices used in experimental evaluation.

poorly on KNC due to its single-threaded design.

On the Kepler architecture, we found that our algorithms performed best with larger block sizes as they were able to achieve better reuse of x. Also, larger block sizes enable better use of cache. For a block size of 2x2, a warp will cover 8 blocks; in each loop iteration (see line 13 of Algorithm 2 and line 12 of Algorithm 3), a warp must fetch 8 block vectors (i.e. 16 elements or 128 bytes, which is one cache line). For a block size of 4x4, a warp will cover only 2 blocks; in each loop iteration, it will then only need to fetch 2 block vectors (i.e. 8 elements or 64 bytes, which is only half a cache line). The next block vector will be brought into cache as part of the memory request, and the next iteration of the loop will then not need to request it.

B. *Impact of Block Sizes*

To understand how the algorithms perform on manycore architectures with matrices of different block sizes, we partitioned the GT01R test matrix (a discretization from a 2D inviscid fluid simulation – see Table IV) into blocks with sizes from 2 to 32, placing nonzero elements in the appropriate aligned blocks and filling in zeros. The results are shown in Figure 9.

On the Sandy Bridge architecture, the performance of our algorithm generally increases with block size. This is expected, as larger block sizes offer improved temporal cache reuse for x.

Performance of our algorithm on KNC is poor for small block sizes, sustaining under 50% of peak bandwidth. This is likely due to the performance of gather/scatter instructions on the KNC architecture. We found that for block sizes < 16, not including 8, the Intel compiler emits gather

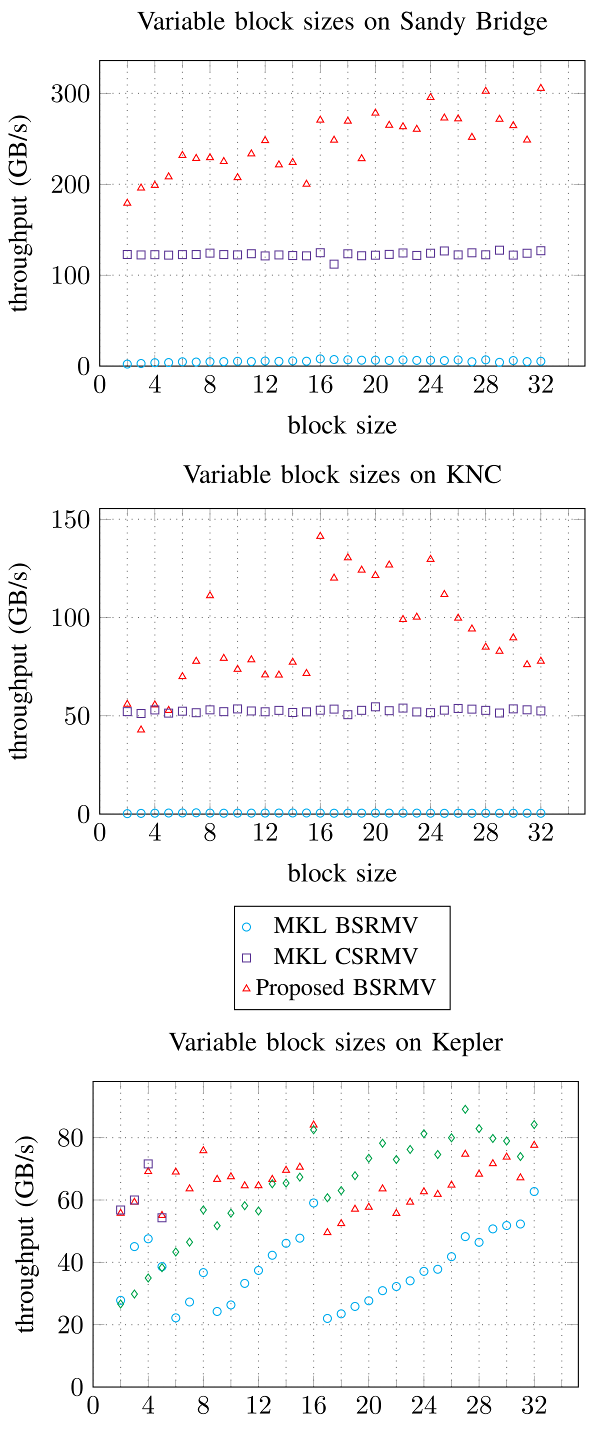


Figure 9. Performance of SpMV algorithms running on the Kepler GPU, Xeon Phi KNC, and Sandy Bridge architectures for the GT01R test matrix divided into variable block sizes.

instructions for fetching elements from $\boldsymbol{A}.val$, despite these accesses being consecutive and unit-strided. For a block size of 8, the innermost loop of the algorithm is perfectly vectorized and no gather instructions are emitted, yielding higher performance as expected. For block sizes $\geq$ 16, the compiler fuses the middle loops and emits no gather instructions, again yielding higher performance. Further investigation is required to determine why the compiler produces gather/scatter instructions for smaller block sizes and whether pragmas can be used to prevent the compiler from emitting unnecessary gather/scatters. Performance will also likely improve with the introduction of gather/scatter-related AVX-512 instructions in the Knight's Landing (KNL) architecture.

On Kepler, our by-block and by-column algorithms perform comparably for block sizes ≤ 5 (the maximum block size the by-block algorithm can handle). The by-block algorithm appears to perform marginally faster, likely due to reduced integer opterations (both r and c are constant, as discussed in III-D1). For block sizes from 6 to 16, the by-column algorithm performs optimally, and for block sizes greater than 16, the row-per-thread algorithm performs best. Note that for block sizes greater than 16, the by-column and row-per-thread algorithms operate similarly in that a warp covers a single column at a time, with each thread assigned to a row within a block. However, the row-per-thread algorithm exhibits better cache performance for $\boldsymbol{x}$, as it loads an entire block vector (16+ elements at a time) into shared memory in a coalesced manner, while the by-column algorithm accesses only a single element from $\boldsymbol{x}$ at a time.

The cuSPARSE algorithm appears to have been optimized for block sizes that are powers of two. While our algorithms also exhibit performance drops for the block size after each power of two, they are less significant, and by using a combination of our three algorithms we achieve fairly consistent performance across block sizes.

C. *Scalability*

We also examined how our CPU-oreinted algorithm scales over a varying number of cores within a single node. For Sandy Bridge, we tested the algorithm with up to 8 cores on a single socket, and used two sockets to test with up to 16 cores. We also tested how hyperthreading and the usage of multiple hardware threads would affect performance. Results are shown in Figure 10.

Performance appears to scale nearly linearly with additional cores within a node, even across sockets. While we performed no tests with multiple nodes, we expect this algorithm will continue to scale well due to a lack of inter-thread communication.

Using multiple threads enhanced performance on both Sandy Bridge and KNC architectures. Hyperthreading on Sandy Bridge yielded slight performance gains, while using

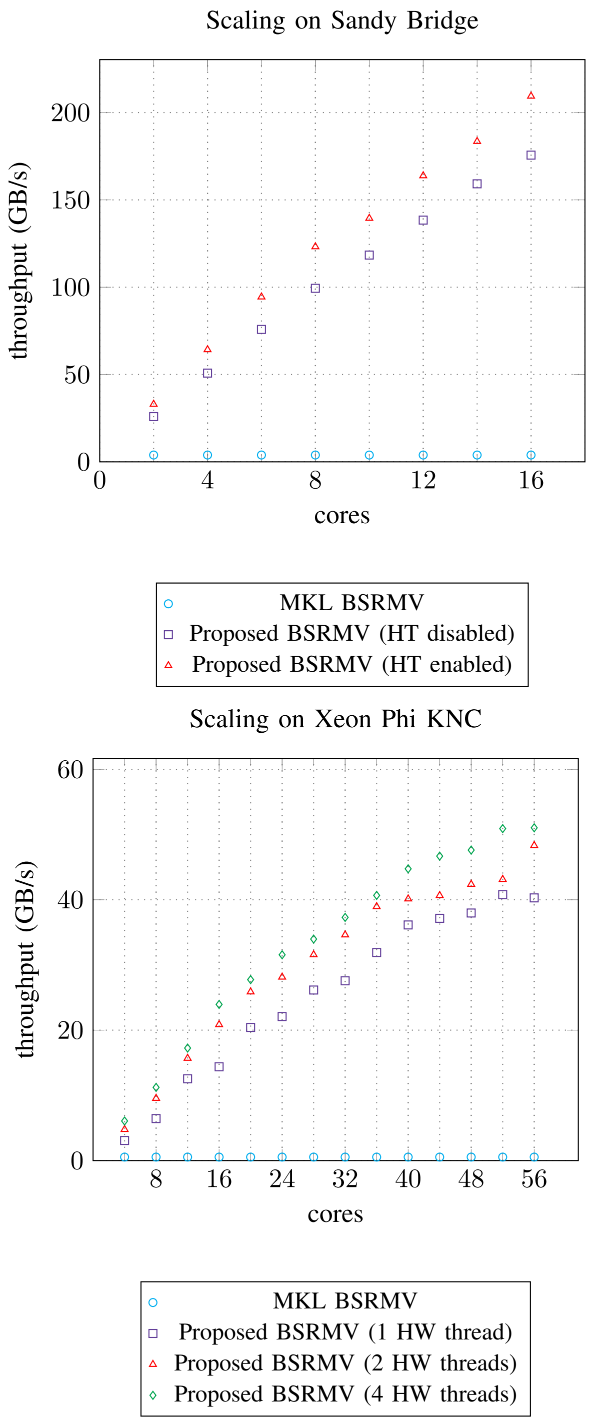


Figure 10. Scaling of our algorithm on a Sandy Bridge Xeon E5-2670 processor and a KNC Xeon Phi using the GT01R test matrix. We tested the impact of hyperthreading (HT) on the CPU and the impact of hardware (HW) thread usage on the Xeon Phi.

multiple hardware threads on KNC yielded more significant improvements.

V. Conclusions and Future Work

By optimizing memory access patterns and minimizing visible latencies, the algorithms presented in this paper are able to achieve high bandwidth utilization and outperform the vendor-optimized block sparse matrix-vector implementations on the Intel Sandy Bridge and NVIDIA Kepler architectures. However, the implementation of our algorithm on the Intel Xeon Phi KNC many-integrated-core architecture is lacking. Additional optimizations may be made to further improve the throughput of our algorithms.

Our algorithm typically utilizes only 15-20% of the Xeon Phi's theoretical bandwidth. As discussed in Section IV-B, this appears to be caused by the compiler emitting gather/scatter instructions when they should not be used, and may be addressable by using pragmas or refactoring the algorithm to avoid this. Additionally, a cooperative threading strategy may be developed so that KNC hardware threads may work on consecutive block rows and achieve increased temporal cache locality with $\boldsymbol{x}$. In the current algorithm, hardware threads on a core operate independently on separate block rows. As KNC is an in-order architecture and this severely limits instruction throughput, a better thread cooperation strategy will benefit performance.

The performance of the GPU algorithms may be optimized as well, as they used only 25-50% of the K80's theoretical 240GB/s bandwidth in our tests. However, full utilization will not be possible without enabling NVIDIA GPU Boost on the GPU and disabling ECC, which we were not able to do in our test setting.

To improve performance for iterative solvers when nonzero blocks are unevenly distributed between rows, a preliminary analysis stage may be introduced in which the algorithm groups rows by $nnzb/row$. The multiplication can then be executed in a multi-pass style, where each pass includes rows of a certain length, in order to better distribute load between cores.

Data structure transformations may be required to further improve performance by a significant margin. Specifically, blocks may be grouped into and processed as tiles in order to potentially (a) further reduce the sizes of the row_ptr and col_idx arrays and (b) improve cache performance for $\boldsymbol{x}$. Such an optimization may be possible without changing the basic BCRS format by adding metadata pointing to tiles in the matrix.

VI. Acknowledgements

We would like to thank Travis Fisher for reviewing a draft of this paper. We would also like to thank Carter Edwards, Simon Hammond, and Christian Trott for reviewing our work and offering technical assistance.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 18.

[2] T. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1:1 – 1:25.

[3] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, *Efficient sparse matrix-vector multiplication on x86-based many-core processors*, in Proceedings of the 27th international ACM conference on International conference on supercomputing, ACM, 2013, pp. 273–282.

[4] R. Shahnaz and A. Usman, *Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers.*, Int. Arab J. Inf. Technol., 8 (2011), pp. 130–136.

[5] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*, PhD thesis, Citeseer, 2003.

[6] R. W. Vuduc and H.-J. Moon, *Fast sparse matrix-vector multiplication by exploiting variable block structure*, in High Performance Computing and Communications, Springer, 2005, pp. 807–816.

[7] 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 (2009), pp. 178–194.