Adaptive Sparse Tiling for Sparse Matrix Multiplication

Anonymous Author(s)

Abstract

Tiling is a key technique for data locality optimization and is used in all high-performance implementations of dense matrix-matrix multiplication for multicore/manycore CPUs and GPUs. However, the irregular and matrix-dependent data access pattern of sparse matrix multiplication makes it challenging to use tiling to enhance data reuse. In this paper, we devise an adaptive tiling strategy and apply it to enhance performance of two primitives: SpMM (product of sparse matrix and dense matrix) and SDDMM (sampled dense-dense matrix multiplication). In contrast to studies that have resorted to non-standard sparse-matrix representations to enhance performance, we use the standard Compressed Sparse Row (CSR) representation, within which intra-row reordering is performed to enable adaptive tiling. Experimental evaluation using an extensive set of matrices from the Sparse Suite collection demonstrates significant performance improvement over currently available state-of-the-art alternatives - Intel's MKL and Nvidia's cuSPARSE libraries (for SpMM) and the MIT TACO compiler and BidMACH (for SDDMM).

1 Introduction

Tiling is a key technique for effective exploitation of data reuse and is used in all high-performance implementations of dense linear algebra computations, convolutional neural networks, stencil computations etc. While tiling for such regular computations is well understood and is heavily utilized in high-performance implementations on manycore Central Processing Units (CPUs) and Graphics Processing Units (GPUs), the effective use of tiling for sparse matrix multiplication remains relatively unexplored.

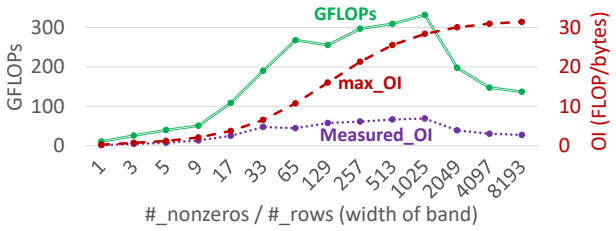

Figure 1. OI and GFLOPs with respect to matrices having different bands

In order to motivate the need for data locality optimization via tiling for sparse-matrix dense-matrix multiplication (SpMM)[1], we present some experimental data on an Intel Xeon Phi processor (KNL, Knights Landing) using the *mkl_scsrmm* routine for SpMM in the Intel MKL library. A number of banded sparse matrices of size $16K \times 16K$ but differing in band-size were used as the sparse matrix argument for the MKL SpMM routine. Fig. 1 presents the performance trend (GFLOPs) as the band-size is varied. Performance improves up to a band-size of 1025 and drops beyond that. The figure also plots the measured operational intensity (OI), the ratio of floating-point operations to the number of bytes of data moved to/from main memory. It may be seen that measured OI increases up to a band-size of 1025 and then drops for larger band sizes. Thus, the performance drop is correlated with increased data movement per operation. As we explain in greater detail later in Sec. 3, this is in contrast to the potential maximum OI, which increases with band-size.

The benefits of tiled execution essentially arise from improved *surface to volume* ratio for tiled versus untiled execution, where the *volume* of a tile represents computational operations in the iteration space, and the *surface* of the tile represents needed data elements to carry out the computations. However, two issues make effective tiling of sparse matrix computations quite challenging:

i) The number of nonzero in some 2D regions (and their row/column positions) may not be conducive to a favorable surface-to-volume ratio for tiles, subject to capacity constraints of cache/scratchpad memories.

ii) Tiling imposes constraints on the execution schedule, with synchronization needed between tiles. This implies that instruction level parallelism can be lower with tiled execution, resulting in more memory access stalls. Thus, even if the total volume of data movement is lower for tiled execution than untiled execution, the increased latency of data access for tiled execution may result in lower overall performance.

Thus, whether tiled or untiled execution is preferable for a 2D block of a sparse matrix depends on the sparsity structure within the block, with the maximal size of 2D blocks for tiled execution being constrained by cache/scratchpad capacity constraints. In this paper, we develop an Adaptive Sparse Tiling (ASpT) approach to tiling two variants of sparse matrix multiplication: SpMM (Sparse-dense Matrix Multiplication) and SDDMM (Sampled Dense Dense Matrix Multiplication). A key idea is that the average number of non-zeros per "active" row/column segments within 2D blocks (i.e., at least

[1]We use SpMM to denote the product of a sparse matrix with a dense matrix, to be distinguished from sparse matrix-matrix multiplication (SpGEMM), where two sparse matrices are multiplied.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

one nonzero) plays a significant role in determining whether tiled or untiled execution is preferable for a 2D block. The sparse matrix is partitioned into panels of rows, with the active columns within each row-panel being either grouped into 2D tiles for tiled execution, or relegated to untiled execution because its active column density is inadequate. In contrast to other prior efforts that have used customized data representations in order to improve performance of sparse matrix operations, we achieve the hybrid tiled/untiled execution by using the standard (unordered) Compressed Sparse Row (CSR) representation of sparse matrices: the nonzero elements in column segments that are to be processed in untiled mode are reordered to be contiguously located at the end in the unordered CSR format.

We demonstrate the effectiveness of the proposed model-driven approach to hybrid-tiled execution of sparse matrix computations by developing implementations for SpMM and SDDMM kernels on GPUs and manycore processors (Intel Xeon Phi). We show significant improvement in achieved performance over available state-of-the-art libraries for these routines on both platforms.

2 Background and Related Work

2.1 Standard Sparse Matrix Representation

CSR representation is one of the most widely used data structures for representing sparse matrices [30, 39]. As shown in Fig. 2 (b,c), the CSR structure is composed of three arrays: row_ptr, col_idx, and values. The value of row_ptr[i] contains the index of the first element of row i. values[] holds the actual numerical values of the nonzero elements, and col_idx[] holds the corresponding column indices. As shown in Fig. 2 (b,c), non-zeros within each row are placed contiguously in col_idx[] and values[]. CSR has two variants, ordered CSR and unordered CSR [16]. In ordered CSR, the column indices within a row are sorted, whereas in unordered CSR the column indices may not be kept in sorted order. Fig. 2 (c) illustrates unordered CSR and Fig. 2 (b) represents the corresponding ordered CSR version.

Double-Compressed Sparse Column (or Row) DCSC (DSCR) [6] is an alternate format, used for ultra-sparse matrices where many rows (columns) may be completely empty. We will consider a DCSC (Fig. 2 (d)) representation where a sparse matrix is partitioned into row panels. Four arrays are maintained: col_ptr[], col_idx[], row_idx[], and values[]. col_idx[] contains the column index and col_ptr[] points to the first element of the corresponding column segment. row_idx[] and values[] store the row indices and the actual non-zero values respectively. Fig. 2 (d) shows the DCSC representation.

Sparse matrices can also be represented using 2D-tiles, as shown in Fig. 2 (e). Like DCSC, the sparse matrix is partitioned into a set of row panels. Each row panel is further divided into 2D tiles. tile_row_ptr[] is used to track the start point of each row within a 2D tile. The col_idx[] and values[] hold the column indices and actual non-zero values respectively.

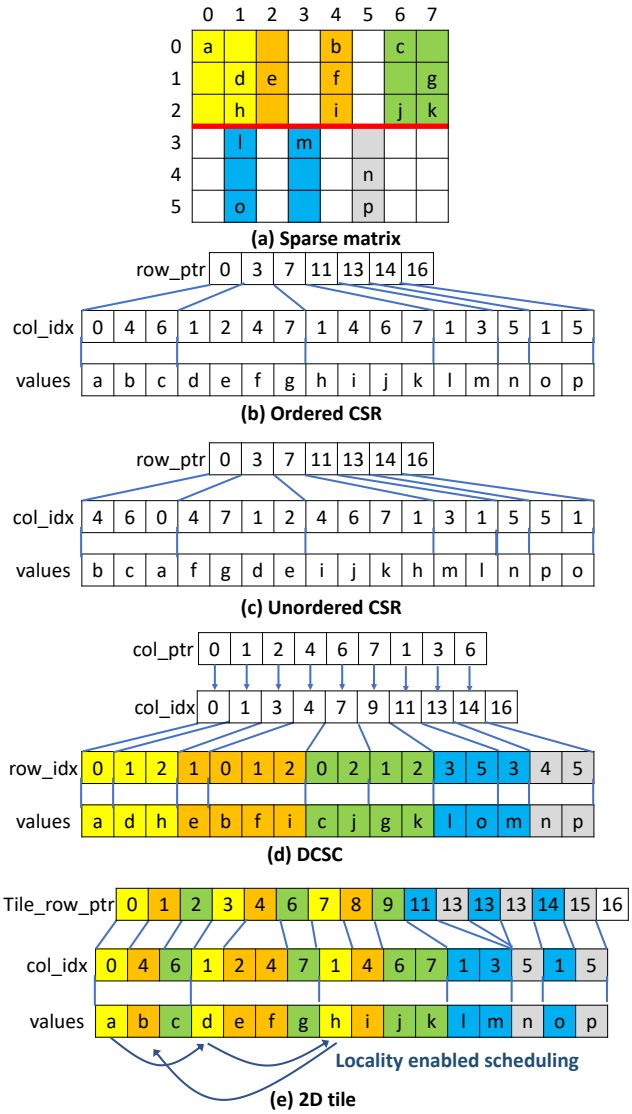

Figure 2. Various data representations for a sparse matrix

2.2 SpMM and SDDMM

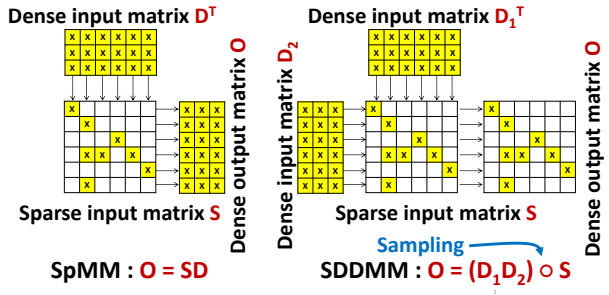

Figure 3. Conceptual view of SpMM and SDDMM

In SpMM, a sparse matrix S is multiplied by a dense matrix D to form a dense output matrix O. Figure 3 (left) shows

the conceptual view of SpMM. Alg. 1 shows the sequential SpMM using CSR representation. SpMM is widely in many applications such as Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) for finding eigenvalues of a matrix [3], Convolutional Neural Networks (CNNs) [14], and graph centrality calculations [28]. SpMM is also one of the core GraphBLAS primitives [7].

In SDDMM two dense matrices *D1* and *D2* are multiplied and the result matrix is then scaled by the input sparse matrix *S* (Hadamard product). Figure 3 (right) shows the conceptual view of SpMM. Alg. 2 shows the sequential SpMM using CSR representation. SDDMM primitive is a key kernel in many algorithms such as Gamma Posson (GaP) [34], Sparse Factor Analysis (SFA) [8], and Alternating Least Squares (ALS) [18].

Both SpMM and SDDMM traverse the rows of S (the outer loop). In SpMM, each element in the i-th row (with column index k) of the input sparse matrix *S* is used to scale the k-th row of *D1[k][:]* and the partial products are accumulated to form the i-th row of output matrix *O[i][:]*. In SDDMM, the dot product of the j-th row of *D1* (i.e., D1[j][:]) and i-th row of *D2* (i.e., D2[i][:]) is computed at nonzero position (i,j) of the sparse matrix S and then scaled with S(i,j) to form O(i,j).

Several recent research efforts have been directed towards the development of efficient SpMV [12, 19, 20, 23, 24, 27, 29–33, 35, 38, 40]. However, very few efforts have focused on SpMM and SDDMM.

In typical real-world applications where SpMM or SDDMM is used, these operations are repeated many times using the same sparse matrix (the values may change, but the sparsity structure does not change). For instance, SpMM is used in Generalized minimum residual (GMRES) [4] where several hundred iterations are required. This usage pattern allows a one time light pre-processing which could enhance the performance, and the cost of this pre-processing is amortized across the iterations. For eg. as explained later, our approach does a onetime reordering of the elements to enhance data locality and reuse.

Algorithm 1: Sequential SpMM (Sparse Matrix Matrix Multiplication)

```
   input : CSR S[M][N], float D[N][K]
   output: float O[M][K]
1  for i = 0 to S.num_rows-1 do
2      for j = S.row_ptr[i] to S.row_ptr[i+1]-1 do
3          for k = 0 to K-1 do
4              O[i][k] += S.values[j] * D[S.col_idx[j]][k];
```

2.3 Related Work

Efforts to optimize SpMM and SDDMM may be grouped into two categories: using standard representation (CSR) or non-standard customized sparse matrix representation.

Intel MKL [37] is one of the most widely used BLAS libraries for multi/many cores. MKL also includes optimized

Algorithm 2: Sequential SDDMM (Sampled Dense Dense Matrix Multiplication)

```
   input : CSR S[M][N], float D1[N][K], float D2[M][K]
   output: CSR O[M][N]
1  for i = 0 to S.num_rows-1 do
2      for j = S.row_ptr[i] to S.row_ptr[i+1]-1 do
3          for k = 0 to K-1 do
4              O.values[j] += D2[i][K] * D1[S.col_idx[j]][k];
5      O.values[j] *= S.values[j];
```

kernels for many sparse matrix computations such as SpMM, SpMV and sparse-matrix sparse-matrix multiplication (SpGEMM).

TACO [17] is a recently developed library using compiler techniques to generate kernels for sparse tensor algebra operation, including SpMM and SDDMM. Generated kernels are already optimized, and OpenMP parallel pragma is used for parallelization.

cuSPARSE [1] provided by NVIDIA also supports SpMM. It offers two different modes depending on access patterns of dense matrices (i.e., row or column major order).

BIDMach [9] is a library for large-scale machine learning, and includes several efficient kernels for machine learning algorithms such as Non-negative Matrix Factorization (NMF) and Support Vector Machine (SVM).

Recently, Yang el al. [39] applied row-splitting [5] and merge-based [23] algorithms to SpMM to efficiently hide global memory latency. Based on the pattern of the sparse matrix, one of two algorithms is applied.

Several efforts have sought to improve SpMM performance by defining new representations for sparse matrices. Variants of ELLPACK have been used to improve performance, such as ELLPACK-R in FastSpMM [25], and SELL-P in MAGMA [3].

Compressed Sparse Blocks (CSB) [2] is a sparse matrix storage format which partitions and stores the matrices in smaller rectangular blocks. SpMM implementation with CSB data representation has been demonstrated to achieve high performance when both SpMM and transposed SpMM ($O = A^T B$) are simultaneously required [2].

Hong et al. [15] developed a hybrid sparse matrix format called RS-SPMM for GPUs. They demonstrated significant performance improvement over alternative SpMM implementations. However, a disadvantage of their approach is that a customized non-standard data structure is used for representing the sparse matrix, making it incompatible with existing code bases and libraries. Applications often use many library functions, which are based on CSR representation. Iterative applications that make repeated use of SpMM interleaved with other sparse matrix operations may incur a high overhead in repeated conversion from standard CSR to the non standard representation [30, 39].

In this paper, we seek to improve SpMM/SDDMM performance without use of any non-standard sparse matrix representations.

3 Overview of ASpT

This subsection provides an overview of ASpT (Adaptive Sparse-matrix Tiling), a strategy for tiled execution of sparse matrix multiplication using the standard Compressed Sparse Row *(CSR)* representation.

We first elaborate on the observed performance trend shown earlier in Fig. 1. Let us consider the execution of the CSR SpMM algorithm (Alg. 1). The outer (i) loop traverses the rows of the sparse matrix S; the middle (j) loop accesses the nonzero elements in row_i of S, and the inner (k) loop updates row_i of the output array O by scaling appropriate rows of the input dense matrix D by the values of the nonzero elements of S in row_i. The total number of non-zero elements for an $N \times N$ banded matrix with band-size B is approximately NB and so the total number of floating point operations for SpMM product with a dense matrix of size $N \times K$ is 2NBK. The total data footprint for the computation (sum of sizes of all arrays) for single-precision (4 bytes per word for the two dense matrices; 8 bytes per nonzero in the sparse matrix, for column index and value; 4 bytes per row pointer in CSR) is 4NK + 4NK + 8NB + 4N, or approximately 8N(K+B). The maximum possible operational intensity (OI), corresponding to complete reuse of data elements in cache/registers is thus $\frac{2NBK}{8N(K+B)} = \frac{1}{\frac{4}{B}+\frac{4}{K}}$. Therefore, as the band-size increases, *max_OI* also increases. The actually achieved (measured) OI first increases as B increases, but then decreases due to limited L2 cache capacity. The Intel Xeon Phi KNL system has a 1Mbyte L2 cache shared by two cores. The inner (k) loop in Alg. 1 traverses K distinct elements, and the middle (j) loop traverses B iterations, resulting in access of a total of BK elements of D. With a banded matrix, the set of column indices for adjacent rows almost completely overlap (except for two elements at the ends of the band), resulting in almost complete reuse of the data from D if sufficient cache capacity is available. For K=128, setting 4*128*B = 512K gives B=1024, consistent with the experimental data that shows a drop in performance when B is raised from 1025 to 2049.

Thus it can be seen that data reuse for the input dense matrix D may suffer significantly if too many other rows are accessed before a row is again referenced (when the next nonzero in the corresponding column of S is accessed). The extent of achieved reuse of D is thus very dependent on the sparsity structure of S. In the extreme case, for Alg. 1, no reuse at all may be achieved for D, while full reuse is achieved for S and O. This is illustrated in Fig. 4(a) - the element $O[i][j]$ in Alg. 1 being placed in a register because of its repeated access in the innermost loop.

In order to achieve better reuse for elements of D, the order of access of the elements of S must be changed. For a banded sparse matrix, full reuse of D can be achieved by accessing S column-wise. But full column-wise access will result in loss of reuse for O. By performing column-wise access within row-panels of S, it is feasible to still achieve full reuse for O in cache, as well as some reuse for D. This corresponds to the use of a DCSC data representations, with the access pattern shown in Fig. 4(b). While this DCSC scheme may be quite superior in terms of minimization of data movement to/from memory, its access pattern may be detrimental to ILP (Instruction Level Parallelism) due to the very small average number of non-zeros within the active columns in a row-panel. Further, the number of register loads/stores increases with this scheme since each operation requires a load-modify-store from/to registers. A third alternative is to use 2D tiling, shown in Fig. 4(c), where access is row-wise within a tile, allowing better register-level reuse for the accumulated results.

The trade-offs between these three alternatives depend on the sparsity structure of the matrix and are very difficult to model analytically due to the complex interplay between the impact of reduction of volume of data access from memory and increase in stall cycles due to reduced ILP. Therefore we used microbenchmarks based on synthetic random matrices to understand the performance trends for the three alternative schemes shown in Fig. 4. Fig. 5 shows the performance (single precision) for different synthetic sparse matrices on an Intel Xeon Phi KNL. The non-zeros in the synthetic matrices are randomly distributed with different sparsity, and *nnz_colseg* (the average number of elements in a column segment in DCSC) is computed as $\frac{T_M \times nnz}{M \times N}$ where T_M is the recommended row segment size for DCSC, and nnz is the number of non-zeros in the sparse matrix with M=128K, N=4K, K=128. To fully exploit L2 cache on KNL, the row panel size is chosen as 512 and 256, for DCSC and 2D tile, respectively (the row panel size is halved in DCSC since the cache is used for both dense input D and dense output$O$ matrices).

In CSR the non-zero elements of the sparse matrix in a row are swept one by one and are multiplied by the corresponding elements in D. The partial results are accumulated in registers and written out to memory at the end of each row. The O elements get a full reuse from registers. However, the reuse of D elements is heavily dependent on the sparsity structure and in general for large sparse matrices the reuse for these elements can be low In other words, with the CSR representation, it is difficult to exploit locality for D. This performance impact is shown in Fig. 5

When the number of elements in a column segment (*nnz_colseg*) is high, the performance can be improved by exploiting reuse of D elements. DCSC targets to improve the reuse of D. In DCSC (Fig. 4), the sparse matrix is partitioned into a set of row panels, each of which has T_M contiguous rows. The size of a row panel (T_M) is chosen such that the corresponding O elements can fit in the L1/L2 cache (or shared memory in case of GPUs), i.e., $T_M \times T_K <$ cache size. Each non-zero column of the row panel is processed sequentially. The D elements corresponding to the column are brought into registers and the partial results are accumulated in the cache. Thus, within a row panel, the D elements get a full reuse from registers and O elements get full reuse from the cache. In DCSC, the reuse for D is increased, but the reuse of O elements is from the cache as opposed to registers in CSR (register accumulations are faster). Hence, as shown in Fig. 5, the performance

of DCSC representation increases with *nnz_colseg*. When *nnz_colseg* is low, the standard CSR representation outperforms DCSC.

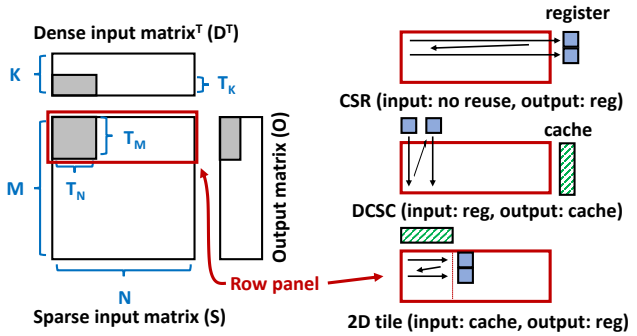

Figure 4. Data Reuse with three different data representations

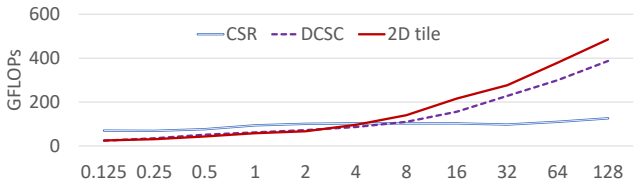

Figure 5. Performance of CSR, DCSC, 2D tile for different synthetic matrices

2D tiling can be used to achieve a good reuse of *D and O* elements. In 2D tiling (in Fig. 4), the sparse matrix is partitioned into a set of row panels which are further subdivided into a set of 2D tiles such that each tile has T_M rows and T_N columns of the sparse matrix. The elements within a 2D tile are represented in CSR format (other formats can also be used). In this scheme, the *D* elements are loaded to cache and the partial accumulations in each row of the 2D tile are done in registers (similar to CSR). However, as shown in Fig. 5, when *nnz_colseg* is low, the performance of 2D tiling scheme is lower than CSR.

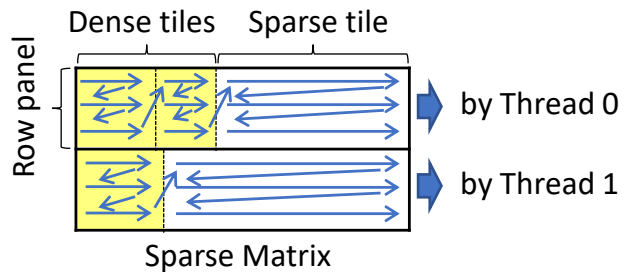

Figure 6. SpMM with ASpT on many cores

Our ASpT scheme is based on the observation that when columns in a 2D tile have sufficiently high *nnz_colseg*, 2D tiling achieves the best performance. When *nnz_colseg* is low, row-wise access with the standard CSR algorithm is best. Our empirical evaluation with synthetic benchmarks did not reveal scenarios where DCSC performance was the best. Therefore, we use a combination of row-wise CSR access and 2D-tiled execution. Fig. 6 shows the high-level idea behind the Adaptive Sparse Tiling (ASpT) approach that we describe in detail in the next section. The sparse matrix is first divided into row-panels, where the row-panel size is determined by cache/scratchpad capacity constraints. Within each row-panel, column segments are classified as sufficiently dense or not (the threshold is dependent on the target system and is determined from the cross-over point between CSR and 2D-Tile performance with the microbenchmarking using synthetic matrices (e.g., Fig. 5). The columns within a row-panel are then reordered so that columns over threshold are placed in 2D tiles (the horizontal sizing of the 2D tiles is explained in the next section), while all columns below threshold are placed at the right end of the row panel in a large group targeted for untiled row-wise CSR execution.

4 SpMM with ASpT

In this section, we present details of the ASpT implementation for SpMM.

4.1 Data Representation

Our SpMM scheme uses the standard CSR representation with additional metadata and is depicted in Fig. 7. Fig. 7 (a) shows the conceptual view of the sparse matrix and Fig. 7, (b) the corresponding CSR representation. Fig. 7, and (c) the conceptual view of the sparse matrix where the heavy column segments are re-ordered. The entire matrix is split into row-panels, where each row-panel contains a set of contiguous rows. A column segment within a column panel is classified as heavy, if it has at least three non-zeros. Fig. 7, (d) shows the CSR (unordered CSR) representation corresponding to Fig. 7 (c). All the heavy columns in a row panel are placed before the light columns. Each reordered row panel can thus be viewed as two segments, where the first segment contains a set of heavy columns and the second segment consists of light columns. The first segment (heavy) is further conceptually subdivided into 2d tiles and the entire second segment is conceptually viewed as a single 2d tile. The width of the 2d-tiles in the heavy segment is selected such that the corresponding elements of *D* fit in the cache (or shared-memory). Note that our approach only performs re-ordering and **not** re-numbering (i.e., actual vertex id are preserved).

Additional metadata called 'tile_row_ptr' keeps track of the start and end pointers of each tile (Fig. 7 (e)) within each row. For example, consider the first row of the re-ordered matrix. The first tile begins at position '0'. Hence, *tile_row_ptr[0]* is '0'. The first element corresponding to the second tile begins at position '2'; hence *tile_row_ptr[1]* is '2', and so on. The number of 2D tiles in each row panel is encoded in *panel_ptr[i]*. For a given row panel, the number of 2D tiles can be obtained by subtracting *panel_ptr[i]* from *panel_ptr[i+1]*.

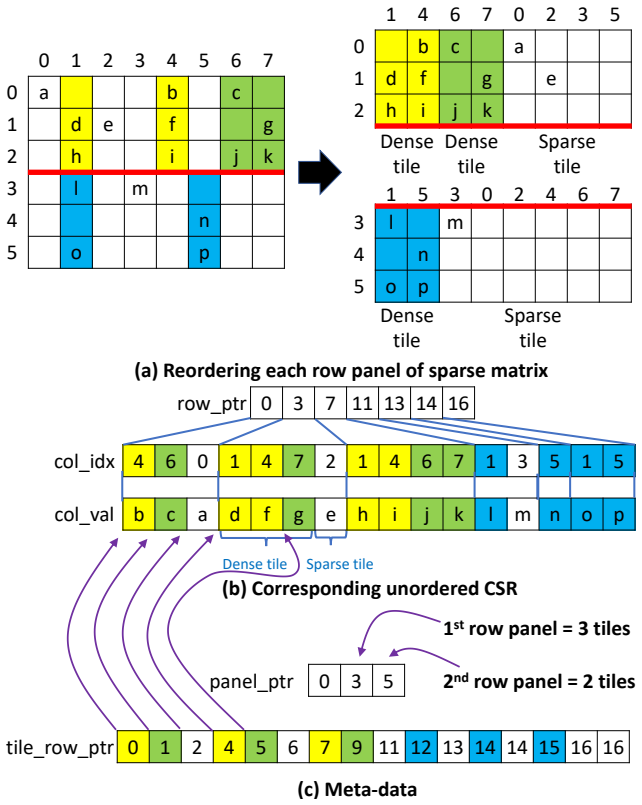

Figure 7. Splitting sparse matrix into heavily clustered row-segments and remainder

4.2 SpMM on Many-Cores

Listing 1. SpMM with ASpT on multi cores

```
1  #pragma omp parallel
2  for row_panel_id=0 to num_row/panel_size-1 do
3   num_tiles = panel_ptr[row_panel_id+1]-panel_ptr[
        row_panel_id];
4   // if tile_id == num_tile-1, sparse tile is processed.
        Otherwise dense.
5   for tile_id=0 to num_tile-1 do
6    for i=0 to panel_size-1 do
7     ptr = panel_ptr[row_panel_id]*panel_size + i*num_tile
          + tile_id;
8     out_idx = i+row_panel_id*panel_size;
9     low = tile_row_ptr[ptr]
10    high = tile_row_ptr[ptr+1];
11    for j = low to high-1 do
12     #pragma simd
13     for k = 0 to K-1 do
14      // inputs is expected to be in cache in dense tiles
15      // output is expected to be in register
16      O[out_idx][k] += col_val[j] * D[col_idx[j]][k];
17     done
18    done
19   done
20  done
21 done
```

Listing 1 shows our SpMM algorithm specialized for Many-core architecture. The row panels of the sparse matrix are distributed among threads (Line 2-21). As mentioned in the previous sub-section, the entire row-panel is split to a set of heavy tiles ($tile\_id < num\_tile - 1$) and a single sparse tile ($tile\_id == num\_tile - 1$). Both the heavy tiles and sparse tiles are processed by the exact same kernel; however, we expect most of the D accesses in heavy tiles to be served by the L2 cache and from memory for the sparse tiles. The tiles within a row panel are processed sequentially (Line 5). The elements of each row within a 2D tile are identified by using *tile_row_ptr[0]* and *panel_ptr[i+1]* (line 7 to 10). The non-zero elements in each row of the 2D-tile are processed sequentially (line 11-18). In order to increase Instruction Level Parallelism (ILP), vectorization is done along K dimension. This also helps to achieve good cache line utilization.

4.3 SpMM on GPUs

Although we can use the same high-level tiling idea for GPUs, the GPU implementation should take advantage of the GPU architecture in order to achieve high performance. The main difference between GPUs and KNL is that GPUs have much more registers per thread. It also has a programmable cache called shared-memory. However, in GPUs, cache per thread ($\frac{cache\ size}{\#\ of\ threads\ on\ SM\ or\ SMs}$) is quite small.

4.3.1 Efficient Utilization of Shared memory and Registers

Contrary to Many-cores, where only a few threads access L1/L2 cache simultaneously, a huge number of threads can access GPU caches at the same time. For instance, on P100, 2K and 112K threads can simultaneously access the same 24KB L1 and 4MB L2 caches respectively. If each thread accesses unique memory locations, then each thread can only use $\frac{24K}{2K} = 12B/threads$ L1 and $\frac{4MB}{112K} = 37B/threads$ L2 cache. However, GPUs have a huge number of registers and shared memory per each Streaming Multiprocessor (SM). For example, P100 has 256KB registers and 64KB shared memory per each SM. The shared memory bandwidth in P100 is higher than L1 cache. Therefore, utilizing these resources for improving locality would be beneficial for performance. In GPUs, only the memory locations with static access patterns can be placed in registers. Since we process each row sequentially, the access pattern of O can be statically determined and these elements can be placed in registers. However, the accesses for D depend on the sparsity structure; hence the accesses are kept in shared-memory.

Mapping the columns of D to shared-memory is non-trivial. Consider Fig. 8 and assume that column 'i' of D is mapped to $i\%4$-th column of the shared-memory. This would result in mapping the first (col idx:2) and third columns (colidx:6) of the yellow tile would be mapped to the same location in shared memory (Fig. 8 (a)) resulting in a conflict. An indirection array can be used to indicate the mapping of column indices to shared-memory. However, this strategy incurs two major overheads: i) extra space requirement of indirection array and ii) most importantly additional overhead for accessing the indirect array. For each non-zero access, the indirection array needs to be accessed to find the element in shared-memory, which is very inefficient. We addressed this issue by re-ordering column indices to remove all mapping

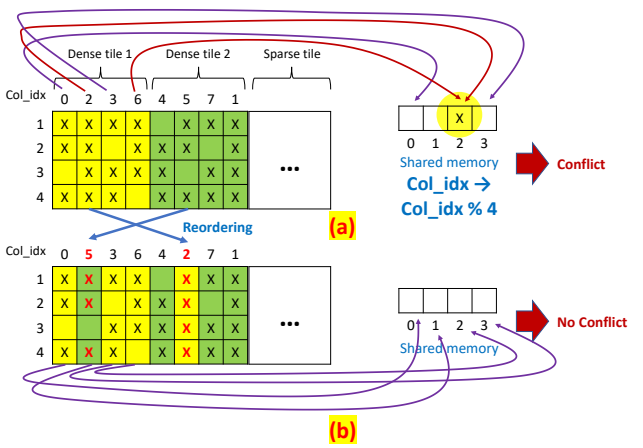

Figure 8. Remove Index Mapping Conflicts to Shared Memory

conflicts in each tile. That is, every column index in the tile is mapped to the different location in the shared memory. By doing so, we can directly access the shared memory using a simple modulo operation. For example, in Fig. 8 (b), 'modulo 4' mapping can be used. This strategy may result in partially filled tiles. If a heavy 2D-tile does not have enough column segments, they are moved to the sparse segment. The reordering can be easily done once during the pre-processing stage.

4.3.2 SpMM algorithm: GPUs

Listing 2. SpMM with ASpT on GPUs (dense tile)

```
1  row_panel_id = tb_idx;
2  row_offset = tid/WARP_SIZE;
3  slice_base = tb_idy*WARP_SIZE;
4  slice_offset = tid%WARP_SIZE;
5  for tile_id=0 to panel_ptr[row_panel_id+1]-panel_ptr[
       row_panel_id]-1 do
6   for i=row_offset to TILE_WIDTH-1 step tb.size()/
        WARP_SIZE do
7    map_id = map_list[panel_ptr[row_panel_id]+tile_id][
         row_offset];
8    sm_D[map_id%TILE_WIDTH][slice_offset] = D[map_id][
         slice_base+slice_offset];
9   done
10  __syncthreads();
11  // processing dense blocks
12  for i=row_offset to panel_size-1 step tb.size()/
        WARP_SIZE do
13   ptr = panel_ptr[row_panel_id]*panel_size + i*(panel_ptr
         [row_panel_id+1]-panel_ptr[row_panel_id]) +
         tile_id;
14   out_idx = i+row_panel_id*panel_size/WARP_SIZE;
15   low = tile_row_ptr[ptr]
16   high = tile_row_ptr[ptr+1];
17   buf_0 = 0;
18   for j=low to high-1 do
19    buf_0 += col_val[j] * sm_D[col_idx[j]%TILE_WIDTH][
          slice_offset];
20  done
21  O[i+row_panel_id*panel_size][slice_base+slice_offset]
        += buf_0;
22  __syncthreads();
23 done
24 done
```

Listing 3. SpMM with ASpT on GPUs (sparse tile)

```
1  row_panel_id = tb_idx;
2  row_offset = tid/WARP_SIZE;
3  slice_id = tb_idy*WARP_SIZE;
4  slice_offset = tid%WARP_SIZE;
5  // processing a sparse block
6  for i=row_offset to panel_size-1 step tb.size()/WARP_SIZE
       do
7   ptr = panel_ptr[row_panel_id]*panel_size + (i+1)*(
       panel_ptr[row_panel_id+1]-panel_ptr[row_panel_id])
       -1;
8   out_idx = i+row_panel_id*panel_size;
9   low = tile_row_ptr[ptr];
10  high = tile_row_ptr[ptr+1];
11  buf_0 = 0;
12  for j=low to high-1 do
13   buf_0 += col_val[j] * D[col_idx[j]][slice_base+
         slice_offset];
14  end
15  O[i+row_panel_id*panel_size][slice_base+slice_offset] +=
        buf_0;
16 end
```

Listing 2 and 3 show the SpMM-ASpT GPU algorithm for dense and sparse tiles respectively. Our GPU algorithm is similar to that of Many-cores. The different threads in a warp are mapped along K to avoid thread divergence and to achieve good load balance. For heavy 2D-tiles the corresponding elements of D is brought to shared-memory, whereas for light 2D-tiles the shared-memory is not used. Each 2D-tile is processed by a thread block.

For processing both dense and sparse tiles, row panel id, row offset, and slice index are first computed (line 1-4 in Listing 2 and 3). Then, for dense tiles, all the threads in a thread block collectively bring the corresponding elements of D to shared memory (line 6-9 in Listing 2). *map_id* (line 7 in Listing 2) keeps track of the original column index, and is used to access elements of D.

The rest of the code (Lines 12-23 in Listing 2 and Lines 6-15 in Listing 3) is very similar to Multi-Core algorithm. Different warps process different rows within a row-panel, and the threads within a warp are distributed along K. The results are accumulated in registers and written out to global-memory at the end of each row (Line 21 in Listing 2 and Line 15 in Listing 3)). The only difference between processing dense and sparse tiles is whether elements of D is served by shared-memory or cache/global memory.

4.4 Parameter Selection

The key parameters that affect performance for ASpT are i) the threshold of the number of non-zeros in a column segment to be classified as heavy and ii) the tile sizes (T_M, T_N, and T_K). These parameters were empirically determined using synthetic matrices described in Section 3. Fig. 5 shows the performance of CSR, DCSC and 2D tile as a function of column density. The threshold for classifying a column segment as heavy is chosen as the minimum column density, at which 2D-tile outperforms CSR. The reasoning for this is based on the fact that our heavy segment is processed by our 2D-tile algorithm, whereas the light segment, even though represented as a single 2D-tile, is processed by CSR

algorithm. Thus, if the length of column segment is less than the crossover point in Fig. 5 then it is better to process that column using CSR algorithm, and 2D-tile algorithm otherwise. The tile sizes were also chosen empirically such that the data footprint of the tile fits in L2 cache (512 KB per core) for KNL (i.e. $(T_M + T_N) \times T_K \times sizeof(word) \times \frac{\#\_of\_threads}{\#\_of\_cores} = 512K$). For our experiments, the L1 cache was too small to exploit locality (and thus did not give great benefits). We explored different (T_M, T_N, T_K) subjected to the L2 footprint constraint, and selected the best performing parameters. The best performance was obtained when $T_M = T_N$, $T_K = K$, and $\frac{\#\_of\_threads}{\#\_of\_cores} = 2$.

We followed similar steps for selecting GPU parameters. Since D elements are kept in shared-memory, the tile sizes T_K and T_N are constrained by the shared-memory capacity. The shared-memory size per thread block was selected such that full occupancy was maintained (for P100 we assigned 32KB of Shared-Memory per thread block of size 1024). Since O elements are kept in registers T_K and T_M are constrained by the register capacity. Thread coarsening [21, 22] was also employed to improve performance.

5 SDDMM with ASpT

In SDDMM, two dense matrices are multiplied and the resulting matrix is then scaled using an element wise multiplication (Hadamard product) with a sparse matrix. Since the sparsity structure of the input and output sparse matrices is the same, we can optimize SDDMM by forming dense matrix products only on locations corresponding to non-zero locations in the input sparse matrix, and employ this optimization is employed by the existing implementation such as [9, 17].

Listing 4. Part of SDDMM on multi cores

```
#pragma simd
for j = low to high-1 do
 #pragma simd reduction
 for k = 0 to K-1 do
  // D2 is expected to be in cache in dense tiles
  // output_col_val is expected to be in register
  O[j] += D2[out_idx][k] * D[col_idx[j]][k];
 done
 O[j] *= col_val[j];
done
```

Listing 5. Part of SDDMM on GPUs (dense tile)

```
buf_D2 = D2[i+row_panel_id*panel_size][slice_base+
     slice_offset];
for j=low to high-1 do
 buf_O = buf_D2 * sm_D[col_idx[j]%TILE_WIDTH][
      slice_offset];
 for k=WARP_SIZE/2 downto 1 step k=k/2 do
  buf_O += __shfl_down(buf_output, k);
 done
 if slice_offset == 0 then
  O[j] += buf_O * col_val[j];
 end
done
```

SDDMM on Many-Cores can be implemented by substituting the box (line 11-18) in Listing 1 with Listing 4. For SDDMM, the K dimension is not tiled, and the tile sizes $T_M and T_N$ are chosen such that both *D1 and D2* fit in L2 cache. The For loop in Line 11 computes the dot product of *D1 and D2*. The inner For loop (Line 14) which corresponds to K is vectorized. Note that unlike SpMM, SDDMM requires reduction across K dimension and is implemented by specifying the 'reduction clause' in Line 17. Line 19 scales the result by multiplying it with the corresponding element in the input sparse matrix (S).

Listing 6. Part of SDDMM on GPUs (sparse tile)

```
buf_D2 = D2[i+row_panel_id*panel_size][slice_base+
     slice_offset];
for j=low to high-1 do
 buf_O = buf_D2 * D[col_idx[j]][slice_base+slice_offset];
 for k=WARP_SIZE/2 downto 1 step k=k/2 do
  buf_O += __shfl_down(buf_O, k);
 done
 if slice_offset == 0 then
  O[j] += buf_O * col_val[j];
 end
done
```

SDDMM on GPUs for dense and sparse tiles can be implemented by replacing the box in Listing 2 and 3 with Listing 5 and 6 respectively. The only different between Listing 5 and 6 is in Line 13 where elements of *D1* is served by shared-memory in the dense version and global memory in the sparse version. The elements of *D2* corresponding to the row are kept in registers for both dense and sparse tiles (line 11 in Listing 5). Since the K dimension is mapped across threads and the corresponding O elements are kept in registers which are private to a thread, we use warp shuffling for reduction (line 14-16 in Listing 5). The accumulated output value is scaled and written back to global memory (line 17-19 in Listing 5).

6 Experimental Evaluation

This section details the experimental evaluation of our strategy for SpMM and SDDMM on two different architectures:

- Nvidia P100 GPU (56 Pascal SMs, 16GB global memory with bandwidth of 732GB/sec, 4MB L2 cache, and 64KB shared memory per each SM)
- Intel Xeon Phi (68 cores with 1.40 GHz, 16GB MCDRAM with bandwidth of 384GB/sec, 34MB L2 cache)

For GPU experiments the code was compiled using NVCC 9.1 with -O3 flag and was run with ECC turned off.

For Intel Xeon Phi, the code was compiled with Intel ICC 18.0.0 with -O3 and -MIC-AVX512 flags. The clustering mode was set to 'All-to-All' and the memory mode was set to 'cache-mode' (to fit big datasets).

We only include the kernel execution time for all experiments. Preprocessing time and data transfer time from CPU to GPU or disk to RAM are not included. The impact of preprocessing overhead is reported separately. All tests were run 5 times, and average numbers are reported.

6.1 Datasets and Comparison Baseline

For experimental evaluation, we selected all matrices in SuiteSparse (also called University of Florida Sparse Matrix Collection) [11] which has at least 10K rows and 10K

columns and 100k non-zeros. These matrices have different matrix characteristics and are from diverse application domains.

For SpMM on KNL, we compared ASpT with Intel MKL [37], CSB [2], and TACO [17], which has the current state-of-the-art SpMM implementations.

For SpMM on GPUs, we compare ASpT with Nvidia cuSPARSE. cuSPARSE offers two modes, and we compare against the better performing one. We do not compare ASpT with MAGMA [3] and CUSP [10] as they are consistently outperformed by cuSPARSE (more than 40% on average).

For SDDMM on Many-cores (KNL), we compared ASpT with TACO which is shown to outperform Eigen [13] and uBLAS [36] by orders of magnitude [17]. For SDDMM on GPUs, we compared ASpT with BIDMach [9] which offers the state-of-the-art SDMMM implementation.

We evaluate ASpT with single precision (SP) and double precision (DP) with the number of vectors set to 32 and 128 (K=32,128). However, only SP is used for SDDMM evaluation on GPUs since BIDMach does not support DP for SDDMM.

6.2 SpMM

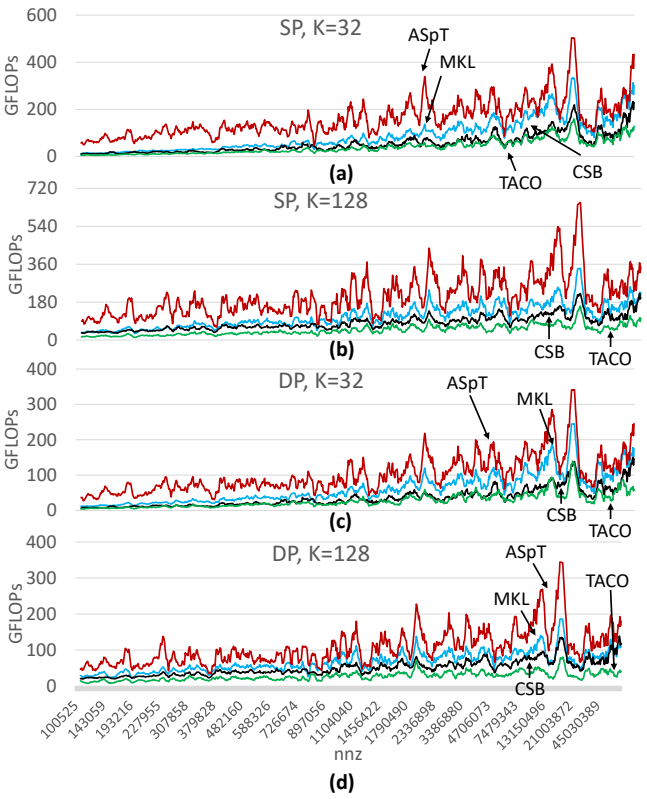

Figure 9. SpMM results on KNL

Fig. 9 (a-d) shows the SpMM performance with SP and DP for different K widths on KNL. Each point in Fig. 9 (a-d) is the average value around 10 neighboring values (to increase visibility). The matrices are sorted in ascending order by the number of non-zeros. As shown in Fig. 9, TACO is superseded by CSB which is outperformed by MKL. For all configurations, ASpT outperforms other implementations. Our performance is improved when K is increased from 32 to 128. For matrices having a small number of non-zeros, performance is low. This is because concurrency is very low or there is not enough data reuse (i.e., $\frac{nnz}{\#\_of\_cols}$ is small).

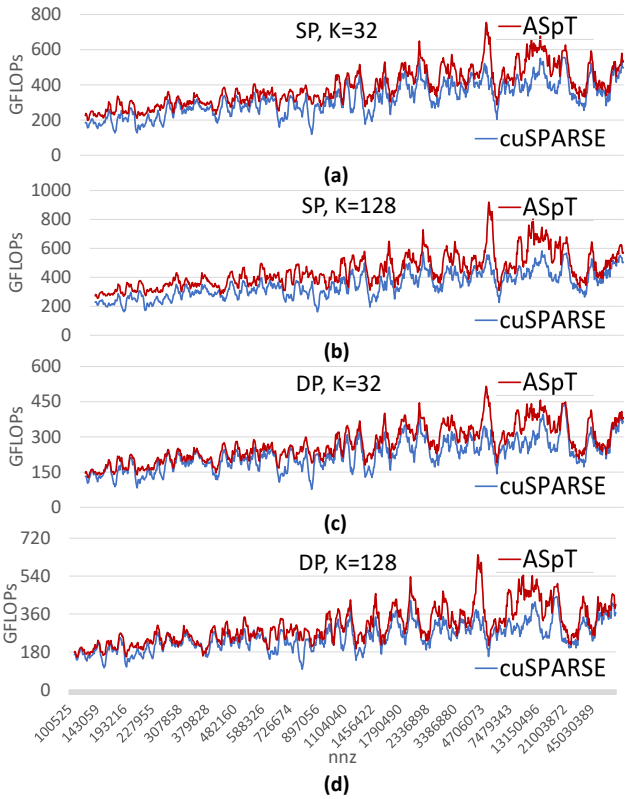

Figure 10. SpMM results on P100

Fig. 10 (a-d) shows the SpMM performance on P100. The performance gap between ASpT and cuSPARSE is higher for higher K widths. Performance of cuSPARSE is not much improved when K is increased. cuSPARSE at most achieves 685 GFLOPs (SP, K=128 with tsyl201 dataset), whereas ASpT achieves more than 900 GFLOPs when the number of non-zeros is around 4M.

6.3 SDDMM

Fig. 11 shows the SDDMM performance on KNL. The performance trend is similar to SpMM in Fig. 9, but the absolute GFLOPs are lower. ASpT consistently outperforms TACO across all cases. Fig. 12 presents SDDMM performance on GPUs. In GPUs The performance trend is similar to that of SpMM in Fig. 10 with lower absolute GFLOPs. Both BIDMach and ASpT improve when K is increased, and ASpT significantly outperforms BIDMach across all the cases.

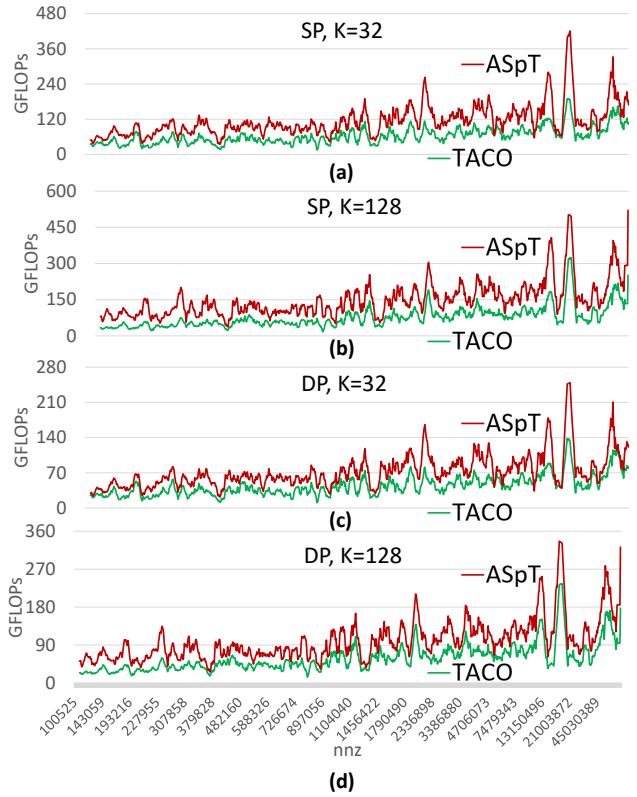

Figure 11. SDDMM results on KNL

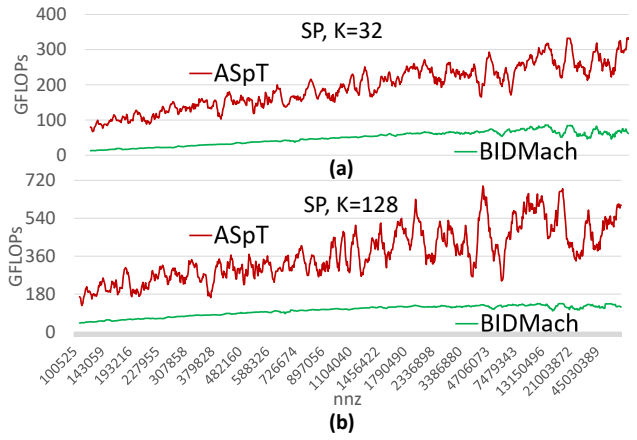

Figure 12. SDDMM results on P100

6.4 Preprocessing Overhead

Constructing additional meta-data and reordering the column indices incur additional overhead. Fig. 13 shows the preprocessing time normalized to the execution time of one ASpT SpMM with K=128 for single precision (red curve) and double precision (blue curve). Note that typical applications involving SpMM and SDDMM execute a large number of iterations (e.g., [26] for SpMM and [41] for SDDMM). Hence, our preprocessing overhead is negligible. DP precision has less overhead, as the preprocessing time for SP and DP is similar but the SpMM time for DP is higher than that of SP. The pre-processing time for SDDMM overhead can be computed by $\frac{SpMM\_overhead \times one\_iter\_SDDMM(time)}{one\_iter\_SpMM(time)}$, for brevity we don't report the same.

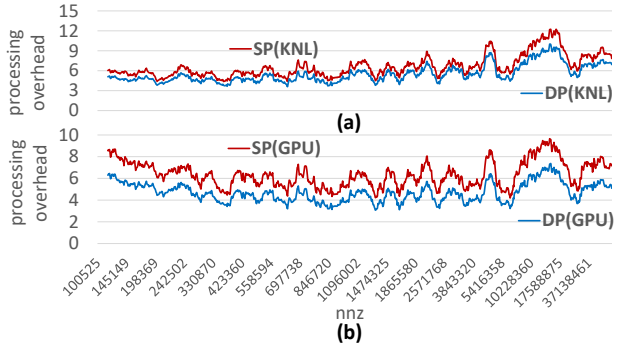

Figure 13. Pre-processing time (SpMM)

7 Conclusion

SpMM and SDDMM are key kernels in many machine learning applications. In contrast to other efforts that use customized sparse matrix representations to achieve high performance, this paper targets efficient implementation of these primitives using the standard CSR sparse matrix representation so that incorporation into applications can be facilitated. An adaptive 2D tiled approach exposes higher memory reuse potential and an efficient reordering scheme enables efficient execution of 2D tiles. The ASpT based SpMM algorithm achieves speedup up to 10.19x with a geomean of 1.63x on an Intel Xeon Phi KNL and achieves speedup up to 24.21x with a geomean of 1.35x on GPUs. The ASpT based SDDMM algorithm achieves speedup up to 30.15x with a geomean of 1.93x on the KNL and achieves speedup up to 13.74x with a geomean of 3.60x on GPUs.

References

[1] 2018. The API reference guide for cuSPARSE, the CUDA sparse matrixlibrary.(v9.1 ed.). http://docs.nvidia.com/cuda/cusparse/index.html.

[2] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1213–1222.

[3] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. 2015. Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product. In *Proceedings of the Symposium on High Performance Computing*. Society for Computer Simulation International, 75–82.

[4] Allison H Baker, John M Dennis, and Elizabeth R Jessup. 2006. On improving linear solver performance: A block variant of GMRES. *SIAM Journal on Scientific Computing* 27, 5 (2006), 1608–1626.

[5] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 18.

[6] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *Parallel and Distributed*

Processing, 2008. IPDPS 2008. IEEE International Symposium on. IEEE, 1–11.

[7] Aydin Buluc, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 643–652.

[8] John Canny. 2002. Collaborative filtering with privacy. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. IEEE, 45–57.

[9] John Canny and Huasha Zhao. 2013. Bidmach: Large-scale learning with zero memory allocation. In *BigLearn workshop, NIPS*. 117.

[10] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. *Version 0.5. 0* (2014).

[11] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[12] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 769–780.

[13] Gaël Guennebaud, Benoıt Jacob, Philip Avery, Abraham Bachrach, Sebastien Barthelemy, et al. 2010. Eigen v3.

[14] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[15] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient Sparse-matrix Multi-vector Product on GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 66–79. https://doi.org/10.1145/3208040.3208062

[16] Christoph W Keßler and Craig H Smith. 1999. The SPARAMAT approach to automatic comprehension of sparse matrix computations. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*. IEEE, 200–207.

[17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 77.

[18] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 8 (2009), 30–37.

[19] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 339–350.

[20] Yongchao Liu and Bertil Schmidt. 2015. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. IEEE, 82–89.

[21] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 455–466.

[22] Alberto Magni, Christophe Dubach, and Michael FP O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 11.

[23] Duane Merrill and Michael Garland. 2016. Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 43, 2 pages. https://doi.org/10.1145/2851141.2851190

[24] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. *HiPEAC* 5952 (2010), 111–125.

[25] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M Garzón. 2013. Fastspmm: An efficient library for sparse matrix matrix product on GPUs. *Comput. J.* 57, 7 (2013), 968–979.

[26] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409* (2016).

[27] Juan C Pichel, Francisco F Rivera, Marcos Fernández, and Aurelio Rodríguez. 2012. Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems* 36, 2 (2012), 65–77.

[28] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2015. Regularizing graph centrality computations. *J. Parallel and Distrib. Comput.* 76 (2015), 106–119.

[29] Markus Steinberger, Andreas Derlery, Rhaleb Zayer, and Hans-Peter Seidel. 2016. How naive is naive SpMV on the GPU?. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 1–8.

[30] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing*. ACM, 13.

[31] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 353–364.

[32] Xiangzheng Sun, Yunquan Zhang, Ting Wang, Xianyi Zhang, Liang Yuan, and Li Rao. 2011. Optimizing SpMV for diagonal sparse matrices on GPU. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 492–501.

[33] Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. 2013. Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 26.

[34] Michalis K Titsias. 2008. The infinite gamma-Poisson feature model. In *Advances in Neural Information Processing Systems*. 1513–1520.

[35] Francisco Vázquez, José-Jesús Fernández, and Ester M Garzón. 2011. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 23, 8 (2011), 815–826.

[36] Joerg Walter, Mathias Koch, et al. 2006. uBLAS. *Boost C++ software library available from http://www. boost. org/doc/libs* (2006).

[37] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon PhiâĎć*. Springer, 167–188.

[38] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. CVR: efficient vectorization of SpMV on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 149–162.

[39] Carl Yang, Aydin Buluc, and John D Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. *arXiv preprint arXiv:1803.08601* (2018).

[40] Hiroki Yoshizawa and Daisuke Takahashi. 2012. Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUs. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*. IEEE, 130–136.

[41] Huasha Zhao, Biye Jiang, John F Canny, and Bobby Jaros. 2015. Same but different: Fast and high quality gibbs parameter estimation. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1495–1502.