A Novel Data Transformation and Execution Strategy for Accelerating Sparse Matrix Multiplication on GPUs

Peng Jiang
The University of Iowa, USA
peng-jiang@uiowa.edu

Changwan Hong
MIT, USA
changwan@mit.edu

Gagan Agrawal
The Ohio State University, USA
agrawal@cse.ohio-state.edu

Abstract

SpMM (multiplication of a sparse matrix and a dense matrix) and SDDMM (sampled dense-dense matrix multiplication) are at the core of many scientific, machine learning, and data mining applications. Because of the irregular memory accesses, the two kernels have poor data locality, and data movement overhead is a bottleneck for their performance. To overcome this issue, previous works have proposed using tiling and data reorganization to enhance data reuse. Despite their success in improving the performance for many sparse matrices, we find that the efficacy of existing techniques largely depends on how the non-zeros are distributed in a sparse matrix. In this work, we propose a novel *row-reordering* technique to improve data locality for SpMM and SDDMM on GPUs. The goal of such row reordering is to place similar rows close to each other, allowing them to be processed together, and thus providing better temporal locality for the values of the dense matrix. We focus on performing the row-reordering efficiently, by using a hierarchical clustering procedure optimized by locality-sensitive hashing. We also investigate when row-reordering is useful, and what factors the performance gains from our method are correlated to. Experimental evaluation using 1084 sparse matrices from SuiteSparse collection and Network Repository shows that our technique achieves up to 2.91x speedup for SpMM and up to 3.19x speedup for SDDMM against the state-of-the-art alternatives on an Nvidia P100 GPU.

CCS Concepts • **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Massively parallel systems**;

Keywords Sparse Matrix Multiplication, GPUs

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374546

1 Introduction

Many scientific, machine learning, and data mining applications involve multiplying a sparse matrix with one or multiple dense matrices, which is commonly abbreviated as SpMM. For example, graph convolution, which is the most basic operation in Graph Neural Networks [22], is a SpMM, where the sparse matrix represents the edges of a graph and the dense matrix stores the feature vector of each vertex. Similarly, gradient descent for solving the Collaborative Filtering problem [18], where the computation of the gradient in each iteration involves an SDDMM (sampled dense-dense matrix multiplication).

Accelerating sparse matrix multiplication on GPUs has been studied for many years. Most of the early works are focused on developing efficient SpMV (sparse matrix-vector multiplication) and SpGEMM (sparse matrix-matrix multiplication) [5, 7, 10, 11, 13, 25]. Only recently, more attention has been given to SpMM and SDDMM [16, 17, 23, 31, 33]. Hong *et al.* [16, 17] have argued that simply implementing SpMM as multiple invocations to highly-optimized SpMV will not deliver good performance because that results in no data reuse with the sparse matrix. Traditional vertex-reordering techniques for improving data locality for SpMV (e.g., [42], [39] and [27]) will not work for SpMM either, because there is little spatial locality among the corresponding elements in different rows of the dense matrix. Hong *et al.* proposed an *adaptive-tiling* technique to improve data locality for SpMM and SDDMM [17]. The main idea is that they divide the rows of the sparse matrix into *row panels*. Next, for each panel of consecutive rows, they reorder the columns to set *densely populated* columns apart from the *sparsely populated* ones. For densely populated columns, they use a traditional tiling approach, as sufficient data reuse is likely to amortize the tiling overhead. For sparsely populated columns, they do not apply tiling (i.e., a row-wise implementation) since tiling overhead cannot be offset by data reuse [16]. This adaptive-tiling technique can achieve good performance for SpMM and SDDMM because it saves data movement costs while computing with the dense tiles.

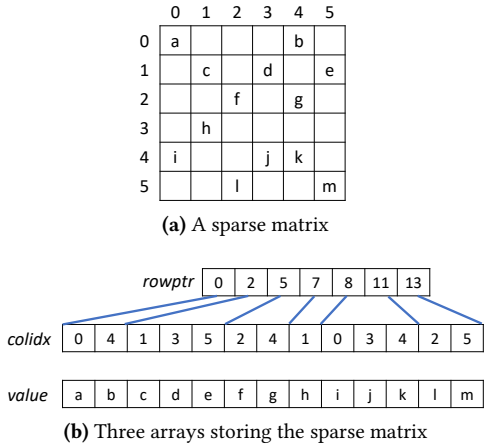
Hong *et al.* show that their technique achieves good average speedup against best alternatives on GPUs for SpMM and SDDMM on 975 matrices from the SuiteSparse collection [15].

However, many of the matrices from the SuiteSparse collection show less than 10% speedup or even a slowdown [17]. The reason is that in these matrices there are few dense columns in each row panel, which leads to few or no dense tiles. In other words, the effectiveness of the adaptive-tiling approach depends on the sparsity structure of the input data. If a sparse matrix already has good data reuse among consecutive rows, adaptive-tiling is an efficient implementation to exploit the data locality. However, if a sparse matrix has poor data reuse among consecutive rows, adaptive-tiling cannot improve the data locality.

We find the poor data locality issue is quite common for real-world data. For 351 of the 1084 matrices from the SuiteSparse collection and the Network Repository [34], the adaptive-tiling method has less than 1% of the nonzeros in the dense tiles, and therefore, has little or no advantage over a simple row-wise implementation.

The goal of this work is to achieve consistently higher performance for SpMM and SDDMM. To this end, we focus on the cases where adaptive-tiling is not effective. We propose to use a *row-reordering* procedure to group similar rows together. More specifically, we first reorder the rows of the sparse matrix such that rows with non-zeroes at similar locations are put in the same row panel. This brings more non-zeros to the dense tiles and increases data reuse in the shared memory. Next, we reorder the rows of the remaining sparse tiles in a similar way. This allows rows with similar non-zeroes to be processed simultaneously and improves temporal locality in the cache when computing with the sparse tiles. Note that row-reordering is different from vertex-reordering, which is commonly used to improve data locality for SpMV or graph algorithms. Vertex-reordering permutes the elements in the dense vector so that consecutive memory accesses to the dense vector have good spatial locality. This is not likely to work for SpMM or SDDMM since the dense input matrix may have hundreds or thousands of columns and there is little spatial locality among elements in a column no matter how you reorder them. Our row-reordering method does not change the indices of the dense matrix, but changes the processing order of the nonzeros of the sparse matrix. In this sense, row-reordering can be considered an aggressive tiling to the sparse matrix.

An important aspect of our work is achieving row reordering efficiently. For this purpose, we use a *hierarchical clustering* procedure. To reduce the clustering overhead, we use a *locality sensitive hashing* procedure to first generate candidate pairs that may have similarities larger than a threshold [28], and we only consider the clustering of the candidate pairs, which are far fewer than all possible combinations of the rows. Our preprocessing procedure takes an average of 69 seconds for sparse matrices with $10^4 \sim 10^7$ rows. For many real applications, this pre-processing overhead can be amortized across iterations (e.g., in gradient descent for solving collaborative filtering), or be considered as offline step (e.g., in inference on graph neural networks).



(a) A sparse matrix

(b) Three arrays storing the sparse matrix

Figure 1. Compressed Sparse Row Representation of a Sparse Matrix

To demonstrate the efficacy of our proposed method, we use 1084 sparse matrices collected from the SuiteSparse collection and the Network Repository to compare the performance of our GPU kernels with Nvidia's cuSPARSE and ASpT (Adaptive Sparse Tiling) [17], which are two state-of-the-art alternatives for computing SpMM and SDDMM on GPUs. The experiments show that our technique achieves up to 2.91x and average 1.19x speedup for SpMM, and up to 3.19x and average 1.49x speedup for SDDMM against the state-of-the-art alternatives.

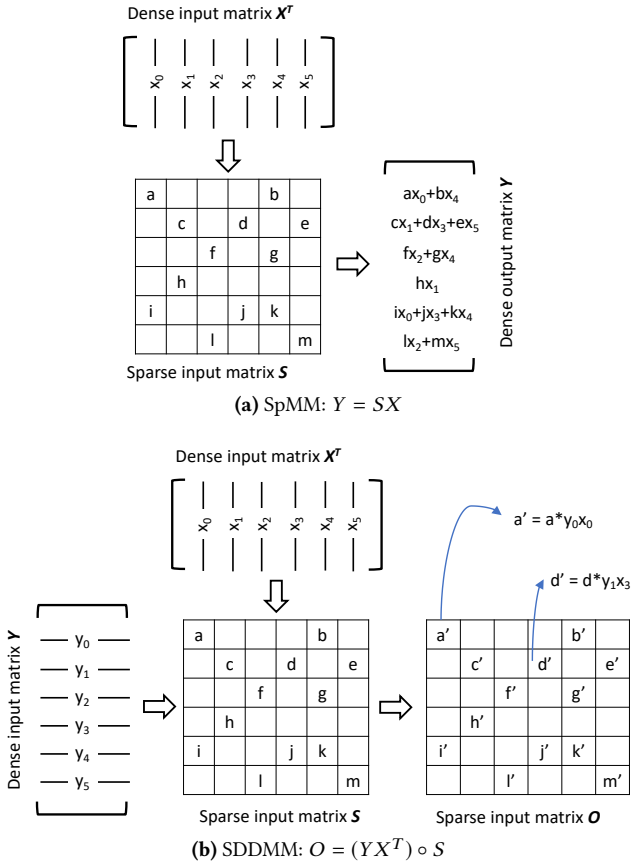
2 Background

This section gives background on the Compressed Sparse Row representation of a sparse matrix and introduces the idea of Adaptive Sparse Tiling (ASpT) from Hong *et al.* [16] for accelerating SpMM and SDDMM.

2.1 Compressed Sparse Row Representation

The compressed sparse row (CSR) representation is one of the most widely used data structures for storing sparse matrices [17, 36]. As shown in Fig 1, CSR format contains three arrays: *rowptr*, *colidx* and *value*. The value of *rowptr*[*i*] is the index of the first element of row *i* in array *colidx* and *value*. The array *colidx* stores the column indices of the nonzeros in the sparse matrix row by row. The corresponding values of these nonzeros are stored in the array *value*. For example, in Fig 1b, the second element of *rowptr* is 2, which indicates that the column index and the value of the first nonzero in the second row of the sparse matrix is in *colidx*[2] and *value*[2].

With CSR format, we can easily access a row of the sparse matrix – *rowptr*[*i*] is the starting position of row *i* in array



(a) SpMM: $Y = SX$

(b) SDDMM: $O = (YX^T) \circ S$

Figure 2. A conceptual view of SpMM and SDDMM

Algorithm 1: A Row-wise Implementation of SpMM

```
   input  : S[M][N], X[N][K]
   output : Y[M][K]
1  for i = 0 to S.nrows − 1 do
2      for j = S.rowptr[i] to S.rowptr[i + 1] − 1 do
3          for k = 0 to K − 1 do
4              Y[i][k]+ = S.value[j] ∗ X[S.colidx[j]][k];
```

colidx, and *rowptr*$[i+1]$-1 is the ending position. For example, if we want to access the second row of the sparse matrix in Fig 1a, we know its starting position is *rowptr*$[1] = 2$ and its ending position is *rowptr*$[2] = 5$, so we can iterate over *colidx*$[2:4]$ and *values*$[2:4]$ to get the column indices and values of the nonzeros in the second row.

2.2 SpMM and SDDMM

In SpMM, a sparse matrix S is multiplied by a dense matrix X to form a dense output matrix Y. As shown in Fig 2a, a row of the output matrix Y is the weighted sum of certain rows of the dense input matrix X, which are selected based on the non-zeros in the sparse matrix S. A naive implementation

Algorithm 2: A row-wise implementation of SDDMM

```
   input  : S[M][N], X[N][K], Y[M][K]
   output : O[M][N]
1  for i = 0 to S.nrows − 1 do
2      for j = S.rowptr[i] to S.rowptr[i + 1] − 1 do
3          for k = 0 to K − 1 do
4              O.value[j]+ = Y[i][k] ∗ X[S.colidx[j]][k];
5          O.value[j]∗ = S.values[j];
```

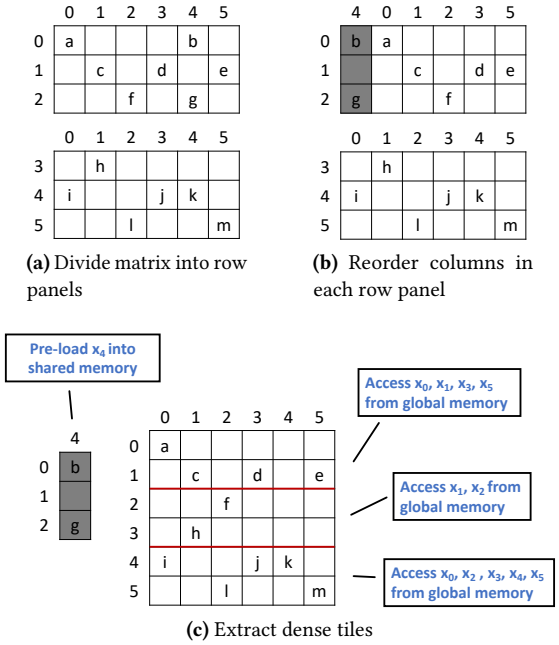
of SpMM can compute the output row-by-row, as shown in Alg 1. To compute the ith row of Y, this method iterates over the non-zeros in the ith row of S. And for each non-zero, suppose its column index is $c = colidx[j]$ and its value is $v = values[j]$, the algorithm selects the cth row of the dense input matrix X, scales it by v, and adds the scaled vector to the result. SpMM is widely used in many applications such as Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) for finding eigenvalues of a matrix [4], and graph centrality calculations [35]. It is also the most basic operation in Graph Neural Networks (GNNs), which are emerging deep learning models for non-Euclidean data [22, 40].

In SDDMM, two dense matrices X and Y are multiplied and the result matrix is then scaled by an input sparse matrix S (Hadamard product). As shown in Fig 2b, an element in the output matrix ($O[i][j]$) is nonzero only if the corresponding element in the input sparse matrix ($S[i][j]$) is nonzero, and the value of $O[i][j]$ is the inner product of the ith row of X and jth column of Y scaled by $S[i][j]$. A naive implementation of SDDMM is shown in Alg 2. The SDDMM primitive is used for efficient implementation of many applications such as Gamma Poisson (GaP) [37], Sparse Factor Analysis (SFA) [8], and Alternating Least Squares (ALS) [24].

2.3 Adaptive Sparse Tiling

As shown in Alg 1 and Alg 2, both SpMM and SDDMM involve traversing the nonzeros of the sparse matrix S and accessing the data of the input dense matrices. A straightforward GPU implementation can assign a warp to process a row of the sparse matrix (i.e., an iteration of the i loop in Alg 1 and Alg 2), with each thread in the warp processing a column of the dense matrix (i.e., an iteration of the k loop in Alg 1 and Alg 2). The threads in a warp iterate over the nonzeros in that row and accumulate the results, which corresponds to the j loop in Alg 1 and Alg 2.

There is good data locality for the matrix Y in both SpMM and SDDMM with a row-wise implementation as the data in Y is reused among the nonzeros in a row of the sparse matrix. The problem is that, because of the sparsity of S, the memory accesses to the dense matrix X are irregular and have poor data locality, making data movement overhead a bottleneck to the performance. Hong *et al.* proposed an *adaptive sparse tiling* (ASpT) technique to improve the data

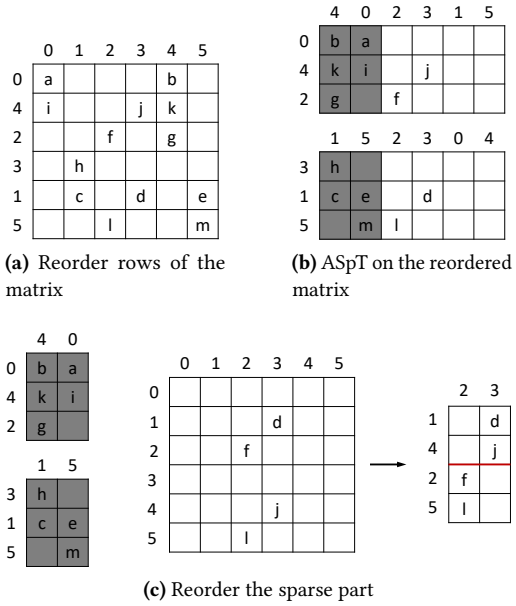


(a) Divide matrix into row panels

(b) Reorder columns in each row panel

(c) Extract dense tiles

Figure 3. An illustration of adaptive sparse tiling



(a) Reorder rows of the matrix

(b) ASpT on the reordered matrix

(c) Reorder the sparse part

Figure 4. An illustration of row-reordering

locality for SpMM and SDDMM. We now give a brief introduction to their main idea, and then point out the limitation of their method as a motivation for our work.

The first step of ASpT is to divide the sparse matrix into *panels* of consecutive rows. As shown in Fig 3a, suppose the panel size is 3, the sparse matrix in Fig 1a will be divided into two panels, each containing three rows. The second step is to sort the columns in each row panel according to the number of nonzeros. As shown in Fig 3b, because column 4 in the first row panel has two nonzeros and all the other columns have only one nonzero, column 4 becomes the first column after reordering. And because all the columns in the second row panel have only one nonzero, the order of the columns is unchanged. Suppose in each row panel a column is considered a dense column if it has at least two nonzeros, the only dense column in this example is column 4 of the first row panel (depicted in shadow). The traversal of the nonzeros is then separated into two parts: one for the dense tiles and one for the remaining nonzeros. For the dense tiles, they pre-load the corresponding rows of the input dense matrix X into the shared memory, and change all the memory accesses to X in the global memory to the shared memory. For the sparse part, they use a warp to process each row and put several warps processing consecutive rows into a *thread-block* as shown in Fig 3c. Suppose a thread-block has two warps here. The processor needs to load a row of X from global memory for each nonzero. The total number of memory accesses to the global memory in this case is $1 + 11 = 12$.

Consider the computation on the original sparse matrix in Fig 1a. Suppose each row is processed by a warp and two warps are grouped in a thread-block. Because there are no identical columns between two consecutive rows, each thread-block needs to load the data of the input dense matrix from global memory for each nonzero. The total number of memory accesses to global memory is 13. ASpT improves the performance of SpMM and SDDMM for this sparse matrix because one global memory access is saved.

However, in the above example, only 2 out of the 13 nonzeros are in the dense tile, which leads to only 1 less access to global memory. As the saving of memory access is guaranteed in the dense tiles, the more nonzeros we have in the dense tiles, the better the performance ASpT can achieve. The problem is that, for many sparse matrices, the proportion of nonzeros in the dense tiles is small after applying ASpT. In fact, more than 30% of the matrices we selected from SuiteSparse collection and the Network Repository for our experiment have less than 1% of nonzeros in the dense tiles. This is the problem we are addressing in this paper.

3 Improving Data Locality with Row-Reordering

In this section, we first give an overview of our main idea of using row-reordering to improve data locality for SpMM and SDDMM. Then, we describe the algorithm for performing row-reordering in detail.

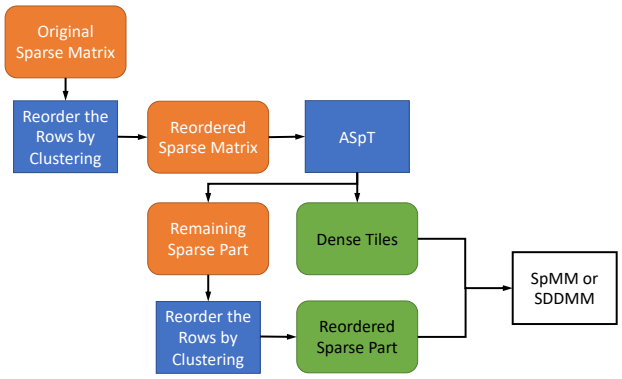


Figure 5. Workflow of using row-reordering for SpMM or SDDMM

3.1 Overview of Row-Reordering

Consider again the sparse matrix in Fig 1a. Row 0 has two identical columns (i.e. location of non-zeroes) with row 4, while row 1 has one identical column with row 5. If we exchange row 1 and row 4, we can get a reordered matrix as shown in Fig 4a. Applying ASpT to the reordered matrix will give us a tiled matrix as shown in Fig 4b. We can see that the number of nonzeros in the dense tiles increases to 9. This means that more data movement from the global memory can be saved during the execution. Also, because the number of nonzeros in each column of the dense tile may be increased (e.g., the first column of the first row panel in Fig 4a has 3 nonzeros), the data reuse in shared memory is improved, which leads to better amortization of the overhead of pre-loading data into the shared memory.

Row-reordering can also help improve data locality for the sparse part – we can perform another round of row-reordering to bring similar rows in the sparse part together. Consider the sparse matrix in Fig 4b. We can separate it into two dense tiles and a new sparse matrix for which we can put rows 1 and 4 together and similarly, row 2 and row 5 together, as shown in Fig 4c. Suppose two consecutive rows are processed by a thread-block of two warps. Then for each thread-block only one memory access to the global memory is needed if there is no cache eviction.

In this example, 6 accesses to the global memory are needed in total: 4 for the dense tiles and 2 for the sparse part. Compared with ASpT on the original sparse matrix (Fig 3c), row-reordering saves 5 more memory accesses and hence leads to better performance.

The above example illustrates our main idea of using row-reordering to improve data locality for SpMM and SDDMM. For a sparse matrix, we first reorder the rows in a way that rows with identical columns are close to each other. Then, ASpT is applied to extract dense tiles from the reordered matrix. These dense tiles are given to a GPU kernel that utilizes shared memory to cache the data of the input dense matrix. Next, we perform another row-reordering step to the sparse matrix comprised of the remaining nonzeros to group rows with identical columns together. This leads to a better cache performance for SpMM or SDDMM on GPUs. The workflow is summarized in Fig 5.

3.2 Row-Reordering by Clustering

We have shown that reordering the rows of the sparse matrix can improve data locality for SpMM and SDDMM. The remaining problem is how to reorder the rows such that similar rows are consecutive. We achieve this by using a *hierarchical clustering* procedure.

Each row of the sparse matrix can be considered as a set of column indices. Two rows are similar if they have many non-zeros in identical columns. A natural definition of the similarity between two rows would be the *Jaccard similarity* between the two sets representing the two rows:

$$J(S_i, S_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}.$$

For example, row 0 of the sparse matrix in Fig 1a has nonzeros in column 0 and 4 (i.e., $S_0 = \{0, 4\}$), while row 4 has nonzeros in column 0, 3 and 4 (i.e., $S_0 = \{0, 3, 4\}$). The similarity between row 0 and 4 is $|S_0 \cap S_4|/|S_0 \cup S_4| = 2/3$.

Intuitively, two rows with a large similarity should be put close to each other to allow more data reuse among the rows of the dense matrix. Initially, we consider each row as a cluster by itself. Then, in each iteration, we select the two clusters that have the largest similarity and group them into one. The similarity between the two clusters is the similarity between the *representing rows* of the two clusters. The representing row of a cluster can be defined in the following way: if a cluster has only one row, then it is the representing row; if two clusters are grouped together, then the representing row of the resulting cluster is the representing row of the larger cluster between the two; if the clusters are of the same size, we choose the row with the smaller index as the representing row of the resulting cluster. Whenever the size of a cluster reaches a threshold, we output the rows in it and remove the cluster. Because any two rows in the cluster have a large similarity, which means they have many columns in common, there will be good data reuse for the dense input matrix X in SpMM and SDDMM. The procedure ends when there is no cluster left.

Although the clustering procedure above is effective in improving data locality for SpMM and SDDMM, it is infeasible for a large number of rows. Suppose that a sparse matrix has 1M rows, which is quite common in real-world data. The clustering procedure will need to compute and store $1M \times 1M = 1T$ similarity values, which will require a large number of computing cycles and easily exceeds the memory capacity of a common computer. Therefore, we use *locality sensitive hashing* (LSH) to reduce the clustering overhead [28]. The idea of LSH is to hash the nodes to be

clustered into different buckets such that nodes in the same buckets are likely to be very similar while nodes in different buckets are not similar. The pairs of nodes within the buckets are considered candidate pairs for clustering. Because the pairs of nodes within buckets are much fewer than the pairs of all nodes, LSH reduces the memory consumption and computation complexity of the clustering procedure. The performance of LSH is controlled by two parameters: *siglen* (the length of the signature) and *bsize* (the size of the band). The larger the *siglen*, the more accurate the hashing. The smaller the *bsize*, the more likely two nodes will be hashed into the same bucket. The total time complexity is $siglen \times nnz + siglen/bsize \times N + d_{max} \times E$ where *nnz* is the number of nonzeros in the sparse matrix, N is the number of rows, d_{max} is the number of nonzeros in the longest row, and E is the number of candidate pairs generated. We will use LSH as a black-box function in our clustering algorithm. Readers are referred to Chapter 3 of [28] for details of this technique.

Alg 3 shows our clustering-based row-reordering procedure. First, we use LSH to get a list of candidate pairs of rows that are likely to have large values of similarity score. These candidate pairs are put in a priority queue (*sim_queue*) to enable a fast querying of the pair with the largest similarity. We use a union-find data structure to maintain the clusters. Basically, it stores a cluster as a tree and merges two clusters by adding one tree as the child of another tree. A detailed description of this data structure can be found in Chapter 21 of [12]. Initially, each row is a cluster of itself. The algorithm selects the pair (i, j) with the largest similarity and tries to merge the two clusters they belong to in each iteration. If i and j are the representing rows of the two clusters, we merge the smaller cluster into the large one. If the resulting cluster exceeds the threshold size, we delete the cluster and stop considering it for future clustering. If i or j is not the representing row of a cluster, we get their representing rows by calling the `root` function. The `root` function keeps finding the parent of a node in the tree representing a cluster until it finds the root whose parent is itself. The update of the parents in line 9 is an optimization. It brings the subtree closer to the root, which reduces the overhead of future queries. If i and j are in two different clusters and the two clusters are not already a candidate pair, we add their similarity to the similarity queue. The algorithm continues until there is no candidate pair in the similarity queue or there is no cluster left. Finally, we output the row indices cluster by cluster to generate the reordered rows.

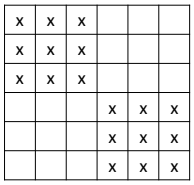
Suppose that the number of rows to be clustered is N and the number of candidate pairs is E. Because the 'then' branch in line 15 takes constant time, and each time the 'while' loop takes the 'then' branch, the number of clusters is reduced by at least one, the total time complexity of the 'then' branch is $O(N)$. Similarly, the loop cannot take the 'else' branch in line 25 for more than E time. Since we always merge the smaller cluster into the larger one, the `root` function cannot have more than $O(log(N))$ iterations. The updates of the similarity queue and the candidate pairs in line 29

Algorithm 3: Reorder the rows by clustering

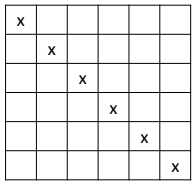
```
   input  : S[M][N], siglen, bsize, threshold_size
   output : reordered_rows
   /* use LSH to get candidate pairs for clustering        */
1  candidate_pairs = LSH(S, siglen, bsize);
2  sim_queue = MaxHeap(candidate_pairs);
   /* initially, each row is a cluster of itself           */
3  cluster_id = Array([0, 1, ..., S.nrows − 1]);
4  cluster_sz = Array([1, 1, ..., 1]);
   /* track if a cluster has been deleted                  */
5  deleted = Array([false, false, ..., false]);
6  func root(i):
       /* find the representing row of a cluster           */
7      while i ≠ cluster_id[i] do
8          cluster_id[i] = cluster_id[cluster_id[i]];
9          i = cluster_id[i];
10     return i;
11 while !sim_queue.empty() and nclusters > 0 do
       /* get the pair with the largest similarity         */
12     i, j = sim_queue.top();
13     sim_queue.pop();
       /* if i and j are the representing rows, merge the
          smaller cluster into the larger one              */
14     if i == cluster_id[i] and j == cluster_id[j] then
15         if deleted[i] or deleted[j] then continue;
16         if cluster_sz[i] < cluster_sz[j] then
17             cluster_id[i] = j; nclusters −= 1;
18             if cluster_sz[j] ≥ threshold_size then
19                 deleted[j] = true; nclusters −= 1;
20         else
21             cluster_id[j] = i; nclusters −= 1;
22             if cluster_sz[i] ≥ threshold_size then
23                 deleted[i] = true; nclusters −= 1;
24     else
25         i = root(i); j = root(j);
26         if deleted[i] or deleted[j] then continue;
           /* if i, j are not in the same cluster, add their
              similarity to the queue                      */
27         if i ≠ j and (i, j) not in candidate_pairs then
28             sim_queue.insert(J(S_i, S_j), i, j);
29             candidate_pairs.insert((i, j));
   /* output the row indices cluster by cluster to generate
      reordered_rows                                       */
30 clusters = Map();
31 for (i = 0; i < S.nrows; i++) do
32     clusters[root(i)].insert(i)
33 for cl in clusters do
34     reordered_rows.append(cl.value)
```

Clusters:	Candidate pairs:
{0}, {1}, {2}, {3}, {4}, {5}	[(0, 4), (2, 4)]
{0, 4}, {1}, {2}, {3}, {5}	[(2, 4)]
{0, 4}, {1}, {2}, {3}, {5}	[(2, 0)]
{0, 2, 4}, {1}, {3}, {5}	[]

Figure 6. A demonstration of the clustering algorithm on the sparse matrix in Fig 1a



(a) A well clustered matrix



(b) An extremely scattered matrix

Figure 7. Sparse matrices for which row-reordering does not help

and 30 have a time complexity of $O(log(E))$. Also, popping out a pair from the similarity queue in line 13 and line 14 has a time complexity of $O(log(E))$. Therefore, the total time complexity of the clustering procedure (not including LSH) is $O(Elog(N) + (N+E)log(E) + N)$. Suppose that E, the number of candidate pairs generated by LSH, is proportional to N. The time complexity is $O(Nlog(N))$, which means that the clustering is almost as fast as sorting the N rows.

Take the sparse matrix in Fig 1a as an example. Suppose LSH generates two candidate pairs: row 0 and row 4 which have Jaccard similarity of 2/3, and row 2 and row 4 which have Jaccard similarity of 1/4. The clustering procedure is shown in Fig 6. Initially, each row is a cluster of itself. The underscored number indicates the representing row of a cluster. In the first iteration, the candidate pair $(0, 4)$ is popped out as it has the largest similarity. The cluster of row 4 is merged into the cluster of row 0. In the second iteration, the pair $(2, 4)$ is popped out. Because 4 is not a representing row, we find the representing row of the cluster that row 4 belongs to, which is row 0, and then put the pair $(2, 0)$ into the similarity queue. In the last iteration, the pair $(2, 0)$ is popped out. Because cluster 2 is smaller than cluster 0, we merge cluster 2 into cluster 0. The clustering stops as there is no candidate pair left, and the algorithm returns $[0, 2, 4, 1, 3, 5]$ as the reordered rows of the sparse matrix. Suppose the row panel size for ASpT is 3. We achieve the same tiling results as shown in Fig 4b. For the remaining sparse part, if LSH generates either pair $(1, 4)$ or pair $(2, 5)$, the clustering algorithm will group $1, 4$ or $2, 5$ together and reorder the sparse part as shown in Fig 4c.

4 Effectiveness of Row-Reordering

The effectiveness of row-reordering in improving data locality for SpMM and SDDMM depends on the distribution of nonzeros in the sparse matrix. This section describes the cases where row-reordering may not be effective and gives a simple method to avoid row-reordering for these cases.

Broadly, there are two types of cases where row-reordering is not helpful. The first case is that the matrix is already well clustered. For example, Fig 7a shows a sparse matrix that has identical rows in three consecutive rows. If we apply ASpT with a row panel size of 3 to this matrix, it will have perfect data reuse – all of the nonzeros are put into two dense tiles. In this case, reordering the rows cannot improve data reuse, and because our row-reordering algorithm uses a small set of candidate pairs for clustering, it may even damage the clusters and reduce the data reuse.

Another case is when the sparse matrix is too scattered (i.e., there are few or no similar rows in the sparse matrix). For example, there is no data reuse among different rows in the diagonal matrix shown in Fig 7b, no matter how the rows are reordered.

Our row-reordering algorithm detects the second case automatically. When a sparse matrix has few similar rows, LSH will generate few candidate pairs, which in turn causes few or no reordering of the rows. For the first case, we can check how the rows in a sparse matrix are clustered by computing the average similarity between two contiguous rows. For example, in the sparse matrix shown in Fig 7a, the average Jaccard similarity between two contiguous rows is

$$\frac{1}{5}(J(S_0, S_1) + J(S_1, S_2) + \ldots + J(S_4, S_5)) = 0.8$$

We want to avoid row-reordering for this case because the reordering may damage the clusters and reduce the data reuse among consecutive rows. Also, LSH will generate a large amount of candidate pairs for neighboring rows, leading to a large row-reordering overhead.

As shown in the workflow (Fig 5), our technique involves two rounds of row-reordering. The first round is on the original sparse matrix for increasing the number of nonzeros in the dense tiles. We determine whether to do the first round of row-reordering by computing the proportion of nonzeros in the dense tiles for the original matrix. If the proportion is larger than a threshold, we skip the row-reordering. For the second round of row-reordering, because the data reuse depends on the scheduling of threads on the GPU, it is hard to have an accurate model to predict whether row-reordering will improve the data reuse. We use the average Jaccard similarity between two consecutive rows as an indicator. If the average similarity is greater than a threshold, we skip the second round of row-reordering.

Although more complicated models can be created to achieve a more accurate prediction of when row-reordering is beneficial, a simple method to determine whether to do row-reordering in real applications is by *trial-and-error*. In

an online scenario, one can perform row-reordering in the first iteration and do SpMM or SDDMM on both the reorder matrix and the original matrix. If the reordered matrix is faster, keep the row-reordering for the rest of iterations; otherwise, discard the row-reordering and use the original sparse matrix for the rest of computation. In an offline scenario, the method is similar – one can try both reordered and non-reordered matrix for SpMM or SDDMM, and use the reordered matrix in deployment if it is faster.

5 Experiments

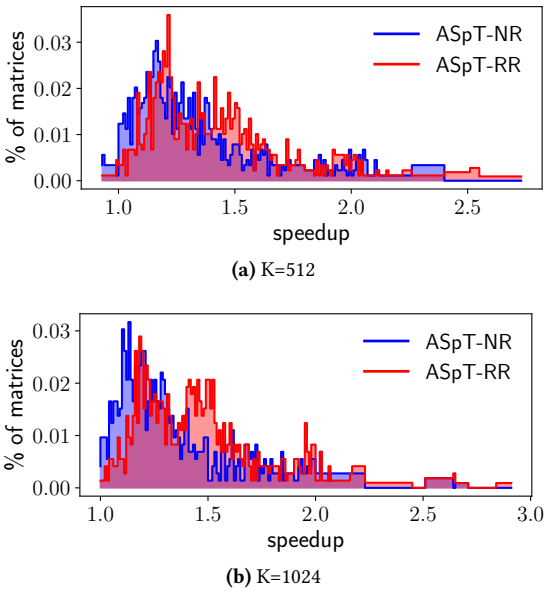
In this section, we apply our row-reordering technique to a diverse set of sparse matrices and evaluate its effectiveness in improving the performance of SpMM and SDDMM.

5.1 Experimental Setup

Platform: Our experiments are conducted on an Nvidia P100 GPU. The GPU has 56 Pascal SMs, 16GB global memory with a bandwidth of 732GB/sec, 4MB L2 cache, and 64KB shared memory on each SM. The code is compiled using NVCC 10.1 with O3 optimization. The GPU is connected to an Intel Xeon CPU E5-2680 v4. The CPU has 14 physical cores running at 2.40GHz. We only include the kernel execution time for all experiments. Preprocessing time and data transfer time from CPU to GPU or disk to RAM is not included. The impact of preprocessing overhead is reported separately. All tests were run five times, and average numbers are reported. Because the differences across executions were small, we do not show the range of times of the reported results.

Datasets: For experimental evaluation, we selected matrices having at least 10K rows, 10K columns, and 100K nonzeros from the SuiteSparse collection [15] and the NetworkRepository [34]. Although each of these repositories had about 900 matrices that meet the criteria, many of them are identical and there are totally 1084 distinct matrices from the two datasets. The matrices in these two datasets are mostly real-world sparse matrices and graphs found in scientific computing and graph processing/mining applications.

Baseline: To show the effectiveness of row-reordering, we compare the performance of ASpT without row-reordering (`ASpT-NR`) and ASpT with row-reordering (`ASpT-RR`). ASpT-NR is claimed to be the state-of-the-art for SpMM and SDDMM. It shows an average speedup of 1.35x over Nvidia cuSPARSE for SpMM and 3.6x over BIDMach [9] for SDDMM [17]. Despite the good speedups on average, ASpT-NR achieves less than 10% speedup or even slowdown compared with cuSPARSE for SpMM on more than 25% of the matrices they selected from the SuiteSparse collection. Therefore, we also compare performance with cuSPARSE for SpMM. We do not compare with MAGMA [4] and CUSP [14], which are two other alternatives for computing SpMM, as they are consistently outperformed by cuSPARSE (more than 40% on average). We do not compare with BIDMach for SDDMM as it is consistently outperformed by `ASpT-NR`.



(a) K=512

(b) K=1024

Figure 8. Speedups of ASpT-RR and ASpT-NR against cuSPARSE: SpMM

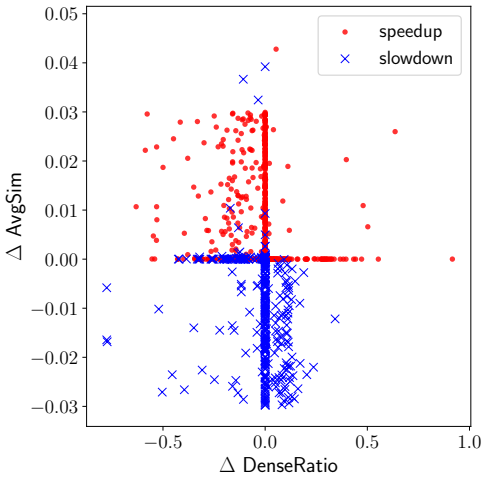


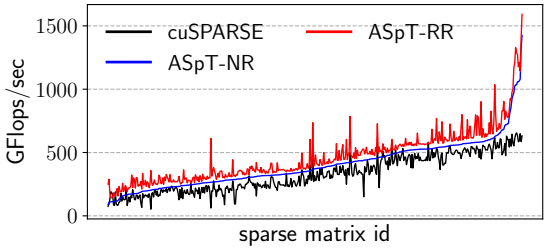
Figure 9. Effectiveness of row-reordering on different sparse matrices

5.2 Performance of SpMM

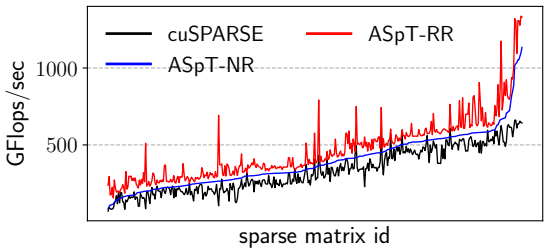
We multiply the sparse matrices with two randomly generated dense matrices with 512 and 1024 columns, respectively. Fig 8 is a summary of speedups of ASpT (with and without row-reordering) over cuSPARSE on all of the 1084 matrices. Overall, the results validate that row-reordering, together with adaptive tiling, helps provide consistent and higher

Table 1. Speedups of ASpT-RR against the faster one of cuSPARSE and ASpT-NR: SpMM

		percentage	
		K=512	K=1024
slowdown	0%~10%	1%	0%
speedup	0%~10%	40%	28.8%
	10%~50%	53.1%	65.3%
	50%~100%	4.8%	4.9%
	>100%	1.1%	1%



(a) K=512



(b) K=1024

Figure 10. Throughputs of SpMM with cuSPARSE, original Adaptive Sparse Tiling with No Reordering (ASpT-NR) and Adapxtive Sparse Tiling with Row-Reordering (ASpT-RR).

speedups over cuSPARSE. Specifically, we can see that row-reordering decreases the proportion of matrices showing slowdown or less than 10% speedup over cuSPARSE, and increases the proportion of matrices showing 50% ~ 100% speedups.

As explained in §4, the effectiveness of row-reordering can vary across matrices depending on how clustered the original sparse matrix is. Fig 9 gives an experimental illustration of the different effectiveness. The x-axis in the figure ($\Delta DenseRatio$) is the change of the ratio of nonzeros in the dense tiles after row-reordering. The y-axis ($\Delta AvgSim$) is the change of average similarity between consecutive rows in the sparse part. We can see that for matrices that both of the two criteria are increased, the performance of SpMM is improved (compared with original ASpT without row-reordering). For matrices that both of the two criteria are decreased, the performance of SpMM is decreased. For the matrices that show an increase in one of the criteria and decrease on the other, the performance can be improved or

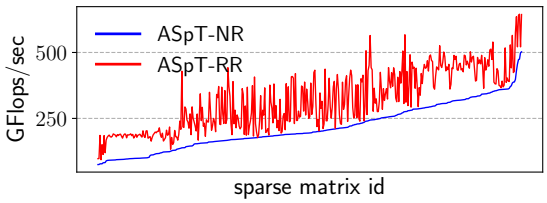
Table 2. Speedups of ASpT-RR against ASpT-NR: SDDMM

		percentage	
		K=512	K=1024
speedup	0%~10%	11.3%	7.0%
	10%~50%	44.4%	47.4%
	50%~100%	33.8%	35.7%
	>100%	10.5%	9.9%

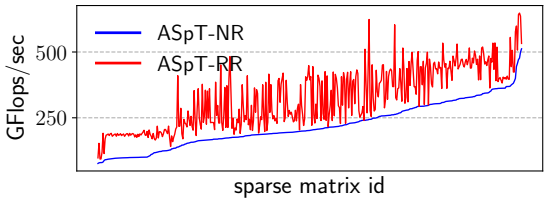
decreased. There are 613 (out of the 1084) matrices showing performance improvement for SpMM with $K = 512$ after row-reordering. Most of the cases are near the two axes, i.e., one of the criteria remains about the same. To show the advantage of row-reordering, we also reordered the sparse matrix with METIS [21] which is a graph-partitioning-based vertex-reordering tool, and give the reordered sparse matrix to the original ASpT. It turns out all of the sparse matrices show slowdown for SpMM with $K = 512$ and $K = 1024$ after being reordered by METIS. This validates our argument that vertex-reordering does little help to SpMM.

In real applications, we can avoid the slowdown by choosing not to performance row-reordering for such matrices. For evaluation, we select the matrices that need row-reordering from the 1084 matrices based on the simple strategy described in §4. First, the ratio of nonzeros in the dense tiles of ASpT before row-reordering is computed for all matrices. In our experiment, we find that for all matrices that show slowdown after row-reordering, the origin ratios of nonzeros in the dense tiles are greater than 10%. So we set the threshold to 10% – if the ratio is greater than 10%, we skip the first round of row-reordering. After applying ASpT, we compute the average similarity between two consecutive rows of the remaining sparse part to decide whether to do the second round of row reordering. We find that for all matrices that show slow down after row-reordering, the average similarities are greater than 0.1, which indicates that the rows of the sparse part are already well clustered. So we skip the second round of row-reordering if the average similarity is greater than 0.1. There are 416 (out of 1084) matrices that need at least one of the two rounds of row-reordering.

Table 1 is a summary of speedups of ASpT-RR over the faster one of ASpT-NR and cuSPARSE on the 416 matrices that need row-reordering. The speedups are up to 2.73x for $K = 512$, and up to 2.91x for $K = 1024$. The median speedup is 1.12x for $K = 512$ and 1.14x for $K = 1024$. The geometric mean speedup is 1.17x and 1.19x for $K = 512$ and 1024, respectively. The same results are displayed as throughput in Fig 10. We sort the 416 matrices in the x-axis by the throughput of ASpT-NR so that the lines can be clearly separated. The figure shows that row-reordering brings consistent speedup to SpMM with ASpT.



(a) K=512



(b) K=1024

Figure 11. Throughputs of SDDMM with original Adaptive Sparse Tiling with No Reordering (ASpT-NR) and Adaptive Sparse Tiling with Row-Reordering (ASpT-RR).

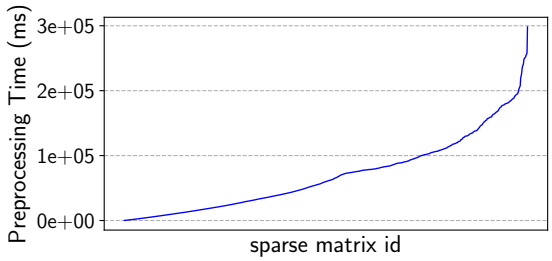


Figure 12. Preprocessing time for the 416 matrices that need row-reordering.

5.3 Performance of SDDMM

We use randomly generated dense matrices with 512 and 1024 columns for testing SDDMM. Nvidia's cuSPARSE does not support this operation, and the original ASpT has the best performance among all available alternatives, so we only compare our implementation with ASpT without row-reordering. Table 2 is a summary of speedups of ASpT-RR over ASpT-NR on the 416 matrices that need at least one round of row-reordering. The speedups are up to 3.19x for $K = 512$ and 2.95x for $K = 1024$. The median speedup is 1.45x for both $K = 512$ and $K = 1024$. The geometric mean speedup is 1.48x and 1.49x for $K = 512$ and 1024, respectively. The throughput of SDDMM with ASpT-RR and ASpT-NR on the 416 matrices are shown in Fig 11. We sort the matrices in the x-axis by the throughput of ASpT-NR so that the lines are clearly separated. The results validate that our row-reordering technique improves the performance of SDDMM.

Table 3. Ratios of preprocessing time to actual computation time for the 416 matrices that need row-reordering: SpMM

		percentage	
		K=512	K=1024
ratio	0x~5x	24.8%	90.9%
	5x~10x	61.1%	5.3%
	10x~100x	12.7%	3.1%
	>100x	1.4%	0.7%

5.4 Preprocessing Overhead

The row-reordering procedure (Alg 3) is conducted on the CPU. There are mainly two parts in the procedure: 1) generating candidate pairs of rows by LSH and 2) clustering the rows based on the candidate pairs. The first part is embarrassingly parallel; we use OpenMP to accelerate the loops. The second part is inherently sequential, but it is fast as we explained in §3.2.

We configure Alg 3 with $siglen = 128$, $bsize = 2$, and $threshold\_size = 256$ in all our experiments. For the 416 matrices that need row-reordering, our preprocessing procedure (including one or two rounds of row-reordering + ASpT) takes 157 ms (milliseconds) to 298 seconds as shown in Fig 12. The average preprocessing time for these matrices is 69.38 sec, and the median preprocessing time is 59.58 sec. The preprocessing time varies for different matrices as they have a different number of rows with different similarities and thus LSH generates a different number of candidate pairs for clustering. Because the first part of Alg 3 is embarrassingly parallel, we expect the preprocessing time can be further reduced if implemented on GPUs.

The ratios of the preprocessing time to the actual computation time for the 416 matrices that need row-reordering are summarized in Tables 3 and 4 for SpMM and SDDMM, respectively. We can see that, when $K = 512$, more than 85% of the matrices have preprocessing time that is less than 10x of the actual computation time of SpMM, and more than 90% matrices have preprocessing time that is less than 10x of the actual computation time of SDDMM. When $K = 1024$, more than 90% of the matrices have preprocessing time that is less than 5x of the actual computation time of SpMM and SDDMM. For iterative applications (e.g., gradient descent for solving collaborative filtering problem) that involve hundreds of iterations of SDDMM, the preprocessing overhead of our method can be amortized. For applications where data reordering can be performed offline (e.g., reordering a graph for graph neural network inference), our row-reordering method incurs little overhead at compile-time.

6 Related Work

This section summarizes the most closely related work on accelerating sparse matrix multiplication.

Nvidia's cuSPARSE [1] is a highly-optimized library for sparse matrix computation on GPUs. It supports many sparse

Table 4. Ratios of preprocessing time to actual computation time for the 416 matrices that need row-reordering: SDDMM

		percentage	
		K=512	K=1024
ratio	0x∼5x	33.2%	95.7%
	5x∼10x	61.3%	2.4%
	10x∼100x	4.5%	1.7%
	>100x	1.0%	0.2%

matrix computations including SpMV, SpMM and sparse matrix-matrix multiplication (SpGEMM). The library offers two different modes depending on the access patterns of dense matrices (i.e., row-major or column-major). BIDMach [9] is a library for large-scale machine learning. It provides efficient GPU implementation for several machine learning kernels such as Non-negative Matrix Factorization (NMF), Support Vector Machine (SVM), as well as SDDMM. Yang *et al.* [41] applied row-splitting [6] and merge-based [30] algorithms to SpMM to efficiently hide global memory latency. Based on the pattern of the sparse matrix, one of the two algorithms is applied. More recently, Hong *et al.* [17] proposed an Adaptive Sparse Tiling (ASpT) technique to improve the data reuse for SpMM and SDDMM. They achieve significant performance improvement over cuSPARSE and BIDMach, which were the state-of-the-art for SpMM and SDDMM before their work. Their technique also preserves the standard CSR representation of sparse matrix, facilitating the incorporation of the kernels into real applications.

Intel's MKL [2] is a widely-used library for computing math functions on multi/many-core CPUs. It provides optimized kernels for many sparse matrix computations, including SpMM, SpMV, and SpGEMM. TACO [23] is a recently developed library using compiler techniques to generate code for sparse tensor algebra operations including SpMM and SDDMM. Although the authors intend to support code generation for GPUs, their current version still only supports CPUs and they use OpenMP for parallelization.

Several efforts have sought to improve the performance of sparse matrix multiplication by reordering the input data and defining new representations for sparse matrices. For example, variants of ELLPACK have been used to improve performance (e.g., ELLPACK-R in FastSpMM [33], and SELL-P in MAGMA [4]). OSKI [38] uses register blocking to enhance data reuse in registers/L1-cache which improves the SpMV performance. When the nonzero elements are highly clustered, register blocking can reduce the data footprint of the sparse matrix. Compressed Sparse Blocks (CSB) [3] is another sparse matrix storage format that exploits register blocking. The sparse matrix is partitioned and stored as small 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 [3]. Hong *et al.* developed a hybrid sparse matrix format called RS-SpMM to improve data locality for SpMM on GPUs [16]. Their sparse matrix format enables significant performance improvement over alternative SpMM implementations, but it is incompatible with existing code bases and libraries. These works based on new sparse matrix representation assume the nonzeros in the sparse matrix are somewhat clustered. For matrices that do not have the block or cluster structures (e.g., the matrix in Fig 1a which we use as a motivating example), these techniques may not be very helpful.

Vertex-reordering is a widely explored technique to improve data locality for SpMV and graph algorithms [10, 19–21, 26, 29, 32, 39, 42, 43]. Yzelman *et al.* [42] show that reordering by recursive hypergraph-based sparse matrix partition can enhance cache locality, and thus performance. Oliker *et al.* [32] demonstrate that the performance of conjugate gradient (CG) and incomplete factorization (ILU) preconditioning can be improved by several reordering techniques such as METIS graph partitioning [21], which enhance locality. GOrder [39] and ReCALL [27] try to reduce the preprocessing overhead using greedy strategies for graph algorithms. The key idea is to number/index the vertices such that vertices with many common neighbors are assigned indices which are close to each other to improve data locality. These vertex-reordering techniques are unlikely to help SpMM or SDDMM because the dense matrix may have hundreds or thousands of columns – this implies little spatial locality among the elements in a column no matter how the vertices are reordered. Different from vertex-reordering, which changes the order of rows in the dense matrix, our row-reordering technique reorders the rows of the sparse matrix, which can be considered as a form of aggressive tiling to improve the data reuse of the dense matrix.

7 Conclusion

In this paper, we have developed a row-reordering technique to improve the performance of two important sparse matrix multiplication kernels: SpMM and SDDMM. Our work is based on the observation that existing techniques for SpMM and SDDMM can deliver good performance only when consecutive rows of the sparse matrix have non-zeroes at identical columns. For sparse matrices that do not have this property, we present a clustering-based row-reordering procedure to put similar rows close to each other, hence increasing the data reuse for SpMM and SDDMM. The experimental evaluation shows that our row-reordering technique can achieve up to 2.91x and average 1.19x speedup over the state-of-the-art alternative for SpMM. We also achieve up to 3.19x and average 1.49x speedup over the state-of-the-art alternative for SDDMM. Our method, when used in conjunction with adaptive tiling, helps deliver more consistent and higher performance improvements over cuSPARSE.

References

[1] 2019. The API reference guide for cuSPARSE, the CUDA sparse matrix library. https://docs.nvidia.com/cuda/cusparse/index.html Version 10.1.168.

[2] 2019. IntelÂő Math Kernel Library. https://software.intel.com/en-us/mkl

[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 1213–1222. https://doi.org/10.1109/IPDPS.2014.125

[4] 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 (HPC '15)*. Society for Computer Simulation International, San Diego, CA, USA, 75–82. http://dl.acm.org/citation.cfm?id=2872599.2872609

[5] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 781–792. https://doi.org/10.1109/SC.2014.69

[6] N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/1654059.1654078

[7] A. Benatia, W. Ji, Y. Wang, and F. Shi. 2016. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. In *2016 45th International Conference on Parallel Processing (ICPP)*. 496–505. https://doi.org/10.1109/ICPP.2016.64

[8] J. Canny. 2002. Collaborative filtering with privacy. In *Proceedings 2002 IEEE Symposium on Security and Privacy*. 45–57. https://doi.org/10.1109/SECPRI.2002.1004361

[9] John F. Canny and Huasha Zhao. 2013. BIDMach: Large-scale Learning with Zero Memory Allocation.

[10] Linchuan Chen, Peng Jiang, and Gagan Agrawal. 2016. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2854038.2854046

[11] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 115–126. https://doi.org/10.1145/1693453.1693471

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[13] Mayank Daga and Joseph L. Greathouse. 2015. Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices. In *Proceedings of the 2015 IEEE 22Nd International Conference on High Performance Computing (HiPC) (HIPC '15)*. IEEE Computer Society, Washington, DC, USA, 64–74. https://doi.org/10.1109/HiPC.2015.55

[14] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/ Version 0.5.0.

[15] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[16] 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

[17] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

[18] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. 2010. *Recommender Systems: An Introduction* (1st ed.). Cambridge University Press, New York, NY, USA.

[19] Peng Jiang and Gagan Agrawal. 2018. Conflict-free Vectorization of Associative Irregular Applications with Recent SIMD Architectural Advances. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 175–187. https://doi.org/10.1145/3168827

[20] Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2016. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/2925426.2926285

[21] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392. https://doi.org/10.1137/S1064827595287997

[22] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=SJU4ayYgl

[23] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[24] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (Aug. 2009), 30–37. https://doi.org/10.1109/MC.2009.263

[25] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 247–256. https://doi.org/10.1145/1941553.1941587

[26] Christopher D. Krieger and Michelle Mills Strout. 2013. A Fast Parallel Graph Partitioner for Shared-Memory Inspector/Executor Strategies. In *Languages and Compilers for Parallel Computing*, Hironori Kasahara and Keiji Kimura (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–204.

[27] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna. 2017. ReCALL: Reordered Cache Aware Locality Based Graph Processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 273–282. https://doi.org/10.1109/HiPC.2017.00039

[28] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of Massive Datasets* (2nd ed.). Cambridge University Press, New York, NY, USA.

[29] John Mellor-Crummey, David Whalley, and Ken Kennedy. 2001. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *Int. J. Parallel Program.* 29, 3 (June 2001), 217–247. https://doi.org/10.1023/A:1011119519789

[30] 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

[31] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan. 2018. Sampled Dense Matrix Multiplication for High-Performance Machine Learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. 32–41. https://doi.org/10.1109/HiPC.2018.00013

[32] Leonid Oliker, Xiaoye S. Li, Parry Husbands, and Rupak Biswas. 2002. Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations. *SIAM Rev.* 44 (2002), 373–393.

[33] Gloria Ortega, Francisco VÃązquez, Inmaculada GarcÃŋa, and Ester M. GarzÃşn. 2013. FastSpMM: An Efficient Library for Sparse Matrix Matrix Product on GPUs. *Comput. J.* 57, 7 (05 2013), 968–979. https://doi.org/10.1093/comjnl/bxt038 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/57/7/968/1007133/bxt038.pdf

[34] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. http://networkrepository.com

[35] Ahmet Erdem Sariyuce, Erik Saule, Kamer Kaya, and Umit V. Catalyurek. 2015. Regularizing Graph Centrality Computations. *J. Parallel Distrib. Comput.* 76, C (Feb. 2015), 106–119. https://doi.org/10.1016/j.jpdc.2014.07.006

[36] 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 (ICS '17)*. ACM, New York, NY, USA, Article 13, 11 pages. https://doi.org/10.1145/3079079.3079086

[37] Michalis K. Titsias. 2008. The Infinite Gamma-Poisson Feature Model. In *Advances in Neural Information Processing Systems 20*, J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis (Eds.). Curran Associates, Inc., 1513–1520. http://papers.nips.cc/paper/3309-the-infinite-gamma-poisson-feature-model.pdf

[38] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16 (jan 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071

[39] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1813–1828. https://doi.org/10.1145/2882903.2915220

[40] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[41] Carl Yang, Aydin BuluÃğ, and John Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU *(Euro-par)*.

[42] A. Yzelman and R. Bisseling. 2009. Cache-Oblivious Sparse Matrix Vector Multiplication by Using Sparse Matrix Partitioning Methods. *SIAM Journal on Scientific Computing* 31, 4 (2009), 3128–3154. https://doi.org/10.1137/080733243

[43] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 369–380. https://doi.org/10.1145/1950365.1950408