TileSpMSpV: A Tiled Algorithm for Sparse Matrix-Sparse Vector Multiplication on GPUs

Haonan Ji
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
haonan_ji@yeah.net

Huimin Song
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
2020215949@student.cup.edu.cn

Shibo Lu
Northeastern University
Boston, U.S.
lu.shib@northeastern.edu

Zhou Jin
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
jinzhou@cup.edu.cn

Guangming Tan
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
tgm@ict.ac.cn

Weifeng Liu
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
weifeng.liu@cup.edu.cn

ABSTRACT

Sparse matrix-sparse vector multiplication (SpMSpV) is an important primitive for graph algorithms and machine learning applications. The sparsity of the input and output vectors makes its floating point efficiency in general lower than sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpGEMM). Existing parallel SpMSpV methods focused on various row- and column-wise storage formats and merging operations. However, the data locality and sparsity pattern of the input matrix and vector are largely ignored.

We in this paper propose TileSpMSpV, a tiled algorithm for accelerating SpMSpV on GPUs. Firstly, tile-wise storage structures are developed for fast positioning a group of nonzeros in matrix and vectors. Then, we develop the TileSpMSpV algorithm on top of the storage structures. In addition, to accelerate directional optimization breadth-first search (BFS) by using TileSpMSpV, we propose a TileBFS algorithm including three kernels called Push-CSC, Push-CSR and Pull-CSC. In the experiments running on a high-end NVIDIA GPU and using 2757 sparse matrices, the TileSpMSpV algorithm outperforms TileSpMV, cuSPARSE and CombBLAS by a factor of on average 1.83, 17.18 and 17.20 (up to 7.68, 1050.02 and 235.90), respectively. Moreover, our TileBFS algorithm outperforms Gunrock and GSwitch by a factor of on average 2.88 and 4.52 (up to 21.35 and 1000.85), respectively.

KEYWORDS

Sparse matrix, SpMSpV, BFS, Tiling, GPU

ACM Reference Format:
Haonan Ji, Huimin Song, Shibo Lu, Zhou Jin, Guangming Tan, and Weifeng Liu. 2022. TileSpMSpV: A Tiled Algorithm for Sparse Matrix-Sparse Vector Multiplication on GPUs. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545028

1 INTRODUCTION

The sparse matrix-sparse vector multiplication (SpMSpV) operation $y = Ax$ multiplies a sparse matrix A with a sparse vector x and gives a resulting sparse vector y. The operation is particularly useful in graph processing and machine learning. Specifically, SpMSpV is a fundamental primitive in the GraphBLAS standard [13, 26], the Combinatorial BLAS package [5, 9], and some graph processing frameworks such as GraphMat [47], GraphPad [2] and GraphBLAST [51]. Also, a number of graph algorithms, such as breadth-first search (BFS) [3], betweenness centrality [46] and reverse Cuthill-McKee (RCM) ordering [4], can be accelerated by fast SpMSpV. In addition, as the sparsity of graphs and deep neural networks is better exploited, SpMSpV is implemented in more graph and AI accelerators [1].

The SpMSpV routine can be seen as a special case both of sparse matrix-vector multiplication (SpMV) [35, 40, 58] and of sparse matrix-matrix multiplication (SpGEMM) [25, 33, 34, 41, 50]. Compared to SpMV, the two vectors involved in SpMSpV are both sparse, and compared to SpGEMM, SpMSpV multiplies a sparse matrix with a sparse vector, but not with another sparse matrix of possibly a large number of columns. As a result, to compute SpMSpV, it is in general less efficient to just call an SpMV (mostly needs to first convert the input sparse vector to its dense form, and wastes space and calculations on zeros) or an SpGEMM (mostly needs to run the Gustavson's row-row method [19], and encounters very bad data locality since each non-empty row of the multiplier has only one element).

Therefore, designing efficient specific algorithms for SpMSpV received much attention. Yang et al. [53] worked on the compressed sparse row (CSR) format and computed SpMSpV in an SpGEMM-like style. Azad et al. [3] and Li et al. [30] used the compressed sparse column (CSC) format and merged sparse columns together in various load balanced ways. Using the knowledge of directional optimization BFS algorithm [6], Li et al. [31] selected running SpMV or SpMSpV according to the sparsity of the input vector. As can be seen, developing variants of the basic CSR or CSC formats and SpMSpV algorithms, as well as adaptively selecting an algorithm among them, are for now the major strategy for parallel SpMSpV.

However, existing work, whether using CSR or CSC, largely ignored exploiting local sparsity in the input sparse matrix. For example, merging sparse columns of A to a sparse y normally needs much larger space over the size of on-chip memory, and working on the off-chip global memory makes merging or sorting very slow. Moreover, when using SpMSpV for BFS on an unweighted graph, storing the position indices for the nonzeros (i.e., edges) in the integer data type may not be space efficient. Furthermore, it is well known that no one matrix storage formulation works for any sparsity structure [28], but there currently lacks work considering effective format for SpMSpV.

In this paper, we propose TileSpMSpV, a tiled algorithm for SpMSpV on GPUs. The key feature of the TileSpMSpV is that the sparse matrix A is stored in sparse tiles containing values, indices and bitmasks, and the sparse vectors x and y are also saved in tiles. Depending on the sparsity of the input vector, three compute kernels called Push-CSC, Push-CSR and Pull-CSC are implemented and automatically selected accordingly. Also, to save the storage space for not storing too many zeros in the very sparse tiles, we extract the nonzeros from such tiles into a separate submatrix for subsequent computations. Through combining the tiled data structures and the kernels, we develop the TileSpMSpV algorithm, as well as a BFS algorithm called TileBFS on top of it.

In our experiments using an NVIDIA Geforce RTX 3090 Ampere GPU and all the 2757 sparse matrices in the SuiteSparse Matrix Collection [14], our TileSpMSpV outperforms TileSpMV [40], cuSPARSE and the Combinatorial BLAS library [3, 9] by a factor of on average 1.83 (up to 7.68), 17.18 (up to 1050.02) and 17.20 (up to 235.90), respectively. Moreover, our TileBFS algorithm is faster than Gunrock [48] and GSwitch [37] on 93.12% and 70.80% matrices, respectively. Speedup-wise, TileBFS outperforms Gunrock and GSwitch by a factor of on average 2.88 (up to 21.35) and 4.52 (up to 1000.85), respectively.

This work makes the following contributions:

- We develop tiled storage structures for the sparse matrix and vectors involved in SpMSpV.
- We design a tiled sparse algorithm called TileSpMSpV and a directional optimization BFS algorithm called TileBFS.
- We evaluate our algorithms on latest NVIDIA GPU and significantly outperform existing work.

2 BACKGROUND

2.1 SpMSpV

Because the one matrix and two vectors are all sparse in SpMSpV, the routine can be executed from two directions shown in Algorithms 1 and 2, respectively.

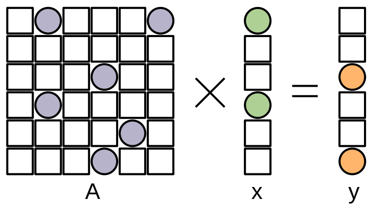


Figure 1: An example of SpMSpV that multiplies a 6-by-6 sparse matrix A with a sparse vector x of two nonzeros and gets a sparse vector y of two nonzeros.

The one listed in Algorithm 1 in parallel computes dot product (i.e., corresponding element of y) of each sparse row a_{i*} of the matrix and the vector x (line 3), which is like the row-wise SpMV method, and logically needs to ensure the element in x requested is nonzero (line 4). The method is also called matrix-driven.

The other shown in Algorithm 2 works from the sparse vector side. Each nonzero in x finds the corresponding column a_{*j} (line 2), scales the nonzeros in the column, and merges the results into y (lines 3-4). It is also called vector-driven method.

Algorithm 1 A pseudocode of row-wise SpMSpV for $y = Ax$

```
1: for each $a_{i*}$ in the matrix $A$ in parallel do
2:     $y_i \leftarrow 0$
3:     for each nonzero entry $a_{ij}$ in $a_{*j}$ do
4:         if $x_j != 0$ then
5:             $y_i \leftarrow y_i + a_{ij} \times x_j$
6:         end if
7:     end for
8: end for
```

Algorithm 2 A pseudocode of column-wise SpMSpV for $y = Ax$

```
1: $y \leftarrow 0$
2: for each nonzero entry $x_j$ in the vector $x$ do
3:     for each nonzero entry $a_{ij}$ in $a_{*j}$ do
4:         $y_i \leftarrow y_i + a_{ij} \times x_j$
5:     end for
6: end for
```

2.2 BFS

BFS is one of the most studied traversal algorithms in graph computations. The algorithm starts from a source vertex in the graph and accesses all reachable vertices through multi-layer traversal. Each layer of the traversal can be implemented by an SpMSpV operator, in which the sparse matrix A is derived from the adjacency matrix of the graph, and the sparse vector x is the active vertex set (also called frontier) of the current layer. The resulting vector y will be the new set of vertices found in this round.

Figure 2 is an example of using SpMSpV to implement the first iteration of BFS. The graph on the left (A's graph form) is an undirected graph composed of six vertices. We set the active vertex set of the current layer to {1} (the green area), and the result of this traversal is the vertex set {2, 3, 4}. Now we can use SpMSpV for

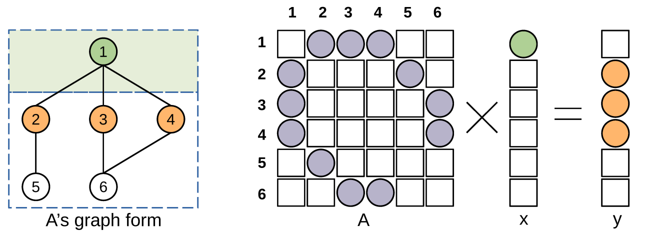


Figure 2: An example of running the first iteration of BFS on the graph (left) by using SpMSpV (right).

translating this process. The graph can be represented by a 6-by-6 adjacency matrix A. The active vertex set can be represented by a vector x whose length is 6 (the number of vertices of the graph). The x_0 corresponding to the active vertex 0 is set to nonzero, and the rest elements of x are set to zeros. The vector y is obtained by multiplying A and x. As can be seen, the resulting vector y now contains three nonzeros at positions 2, 3 and 4, which correspond to the vertices traversed. Algorithm 3 shows the procedure.

Algorithm 3 A pseudocode of BFS using SpMSpV

```
1: while ISNEW($visited$)=1 do
        ▷ The ISNEW is used to determine whether there is a new vertex in the $visited$ vector
2:     SPMSPV($A$, $x$, $y$)
3:     for each nonezero entry $y_i$ in parallel do
4:         $visited_i \leftarrow y_i$
5:     end for
6:     MEMCPY($x$, $y$, $n$)
7: end while
```

3 TILESPMSPV

3.1 Overview

Our TileSpMSpV extends the tiled storage formats proposed in the TileSpMV [40] and TileSpGEMM [41], and adds extra tiled information for sparse matrices and sparse vectors. This design not only exploits sparsity of the matrix and the vectors, but also better utilizes the SIMD/SIMT execution model of GPUs (e.g., 32 threads in a CUDA warp). We further extend TileSpMSpV for BFS by adding a tiled dense bitmask storage format for fast symbolic operations in BFS. Section 3.2 will explain the storage structure of the matrix and vectors for SpMSpV and BFS in detail.

In our TileSpMSpV algorithm, tile is the minimum working unit, and the sparsity of the vector determines the amount of computations required. So, we develop a tile indexing method for the sparse vector which achieves $O(1)$ time complexity to access the sparse vector and to eliminate time wasted on multiplication with empty tiles. Section 3.3 will detail the algorithm of TileSpMSpV.

On top of the TileSpMSpV algorithm, we implement a BFS algorithm called TileBFS. The algorithm will choose a proper tile size to compress and store both matrices and vectors. Furthermore, we propose three new directional optimization methods (i.e., Push-CSR, Push-CSC and Pull-CSC), and choose the best algorithm to traverse each layer of vertices for avoiding the unnecessary amount of computations brought by a single traversal method. Section 3.4 will introduce the algorithm of TileBFS in detail.

3.2 Storage Structures

3.2.1 Storage Structure of Sparse Matrix. When preprocessing the input sparse matrix, TileSpMSpV divides it into sparse tiles of size nt-by-nt, and takes a sparse tile as the basic working unit. nt is usually 16, 32 or 64. If nt is set to 16, a single unsigned char can store indices, and the first and last four bits will contain the row and column indices, respectively. Besides, one warp can be used to process a sparse tile (i.e., two threads works for each row). For a matrix of size m-by-n, we can partition it into (m/nt)*(n/nt) sparse tiles. Those sparse tiles with nonzeros are called 'non-empty tiles', and the sparse tiles without nonzeros inside are called 'empty tiles'. From the matrix's point of view, we treat the non-empty tile as a nonzero element and store it in the CSR format, meaning that only nonzeros inside are saved.

Besides, there can be a number of very sparse tiles (imagine that a tile only contains a couple of nonzero) should not be stored into regular storage formats for saving the cost of maintaining the tile information. In such case, we extract the nonzeros in the very sparse tiles and save them into a new matrix in the COO format.

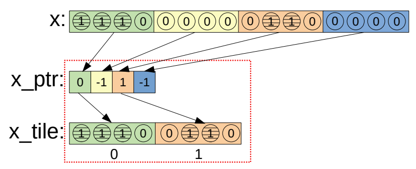


Figure 3: An example of a sparse vector using the tiled storage format.

3.2.2 Storage Structure of Sparse Vector. We store the input vector in the similar tile style, which effectively reduces the data storage space and removes the computations on zeros. For a vector of length n, we will divide it into n/nt tiles. We remove the empty tiles from the vector, store the non-empty tiles tightly, and store the original positions of the non-empty tiles through an index array. The vector of this storage structure contains two arrays: (1) `x_ptr`: an index array of length n/nt that records the type and location of vector tiles. If it is a non-empty tile, the index position will be recorded. If it is an empty tile, the corresponding position will be marked as -1; (2) `x_tile`: a value array that stores the elements of non-empty vector tiles, and its length is that nt multiplies the number of non-empty tiles.

As shown in Figure 3, a vector x of length 16 with five nonzeros can be divided into four tiles of length four. Through traversing the entire x and calculate the `x_ptr` array, the second and fourth tiles do not have nonzeros and are marked as -1. Then the rest tiles are marked as 0, 1, 2, ..., in the order of the non-empty tiles in the vector. In this way, we can get the original x value by the formula `x_ptr`$[x\_id/nt]$ + `x_id`%nt according to the position `x_id` of the input vector.

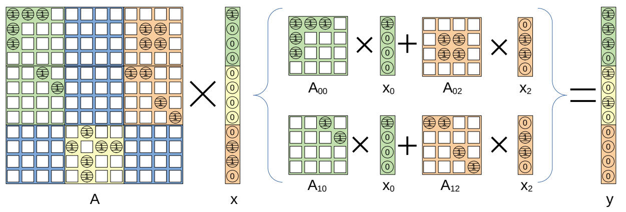


Figure 4: An example describing SpMSpV of a sparse matrix A multiplied by a sparse vector x. The vector x is stored as two arrays `x_ptr` and `x_tile` (as explained in Figure 3). The tile vectors in `x_tile` are respectively multiplied by the tile of the corresponding column index, and the results with the same row index are added together.

3.2.3 Auxiliary Data Structure for TileBFS. The BFS algorithm developed only records the access relationship between vertices and does not need to store the actual values correspondingly. Thus the tile format of the adjacency matrix can be further compressed. To adapt the possibly extremely sparse input vector at the beginning of BFS, we provide two forms of SpMSpV algorithms for the BFS algorithm: CSR-SpMSpV and CSC-SpMSpV. When using SpMSpV in the CSR form, we still store the non-empty tiles in the CSR format. Besides, the non-empty tiles use a binary bitmask to record whether the elements in a tile are zero, and then compress the nt binary bits of each row into an unsigned integer. As for CSC-SpMSpV, we use the CSC format for non-empty tile storage, where nt binary bits of each column are compressed into an unsigned integer within the tile. It is worth noting that when the graph is an undirected graph, these two compression methods will obtain same arrays, which can save about half of the storage space. In addition, two arrays are required to record the currently visited vertex set (input vector) and the processed vertex set (mask vector) respectively. These two vectors firstly are stored as dense tiled vectors of length n/nt, where each value in the vector represents a tile vector of size nt*1. The vector is then converted to its sparse form. We will maintain both two vector formats during the BFS running, and this conversion time is negligible (see the execution time of BFS in the experimental section of the paper).

We in Figure 5 give an example: the top left graph G showing the iterative process of BFS contains 16 vertices, which can be stored as a 16-by-16 adjacency matrix A. The matrix A is divided into 16 tiles of 4-by-4 size, including nine non-empty tiles and seven empty tiles. When using the CSC format, the tile is compressed by column, the entire matrix is denoted as $A1$; when using CSR format, with row-wise compression within a tile, the entire matrix is denoted as $A2$. Not just matrices, each vector tile is compressed individually to ensure data format consistency. Each of them is represented by four binary bits which can be compressed into an unsigned integer.

3.3 TileSpMSpV Algorithm

According to the sparse storage format aforementioned, the non-empty tiles in the matrix are stored as three arrays in the CSR style. All non-empty tiles in each nt rows are treated as a group, called row tiles. We use a CUDA warp of 32 threads to process each row tile of the matrix. For each non-empty tile in the row tile, 32 threads work together. Firstly, we will load the corresponding tile data into the GPU shared memory. Then according to the column position $tile\_col\_id$ of the tile, we find the actual storage position in x_tile_pos of the corresponding tile in the vector tile index array `x_ptr`, that is, x_tile_pos = `x_ptr`$[tile\_col\_id/nt]$. If x_tile_pos equal to -1, the corresponding matrix tile does not need to be calculated. If x_tile_pos does not equal to -1, the real vector nonzero elements are stored in the consecutive nt units starting from the $x\_tile \times nt$ position in x and these nt units will be loaded into the GPU shared memory. Algorithm 4 shows a pseudocode of the TileSpMSpV algorithm in the CSR style for $y = Ax$.

Figure 4 plots a calculation process of TileSpMSpV. The two non-empty tiles A_{00} and A_{02} of the first row tiles are multiplied by the corresponding vector tiles x_0 and x_2 and added with the result vector to obtain y_0. The second row tiles is calculated as the first row tiles to obtain y_1. The calculation is performed only when the vector tiles corresponding to each non-empty tiles have nonzero elements. Therefore, the third row tiles do not have to be calculated. Finally, the entire vector y is given.

Algorithm 4 A pseudocode of TileSpMSpV in the CSR form.

```
1:  for $ti = 0$ to 31 in parallel do
2:      $tile\_colid = A\_tile\_colid[tile\_id]$
3:      $x\_offset = x\_ptr[tile\_colid]$
4:      if $x\_offset == -1$ then continue
5:      end if
6:      $x\_offset = x\_offset * nt$
7:      $csr\_offset = A\_csr\_offset[tile\_id] * nt$
8:      for $i = A\_Row\_Ptr[csr\_offset + ti/2] + ti\%2$ to $i < A\_Row\_Ptr[csr\_offset + ti/2 + 1]$ do
9:          $col\_id = A\_Col\_id[i]$
            //calculate the sum of each row in the tile
10:         $sum\_x += x\_tile[x\_offset + col\_id] * A\_Val[i]$
11:     end for
        //calculate the sum of each row in the row tile
12:     $sum\_x$ += __SHFL_DOWN_SYNC(0xffffffff, $sum\_x$, 1)
        //__SHFL_DOWN_SYNC is for shuffling register data within a warp of CUDA
13:     $sumsum$ += __SHFL_DOWN_SYNC(0xffffffff, $sum\_x$, $ti$)
14: end for
```

3.4 TileBFS Algorithm

On top of the TileSpMSpV algorithm and data structure, we further develop a BFS approach called TileBFS. We first generate the adjacency matrix A in its tiled form of the given graph. To improve parallelism and facilitate atomic operations, nt will be appropriately changed according to the order of A. Specifically, if the order is greater than 10,000, A will be divided into 64-by-64 tiles; otherwise A will be divided into 32-by-32 tiles. After the non-empty tiles in the adjacency matrix are stored as bitmasks, the binary sparse vectors corresponding to one column or row will be represented by numbers of two data types: 32 corresponds to the bit length of the unsigned integer, and 64 corresponds to unsigned long long integer. In the input vector x and mask vector m that also stored in the bitmask format, '1' corresponds to the vertex that has been

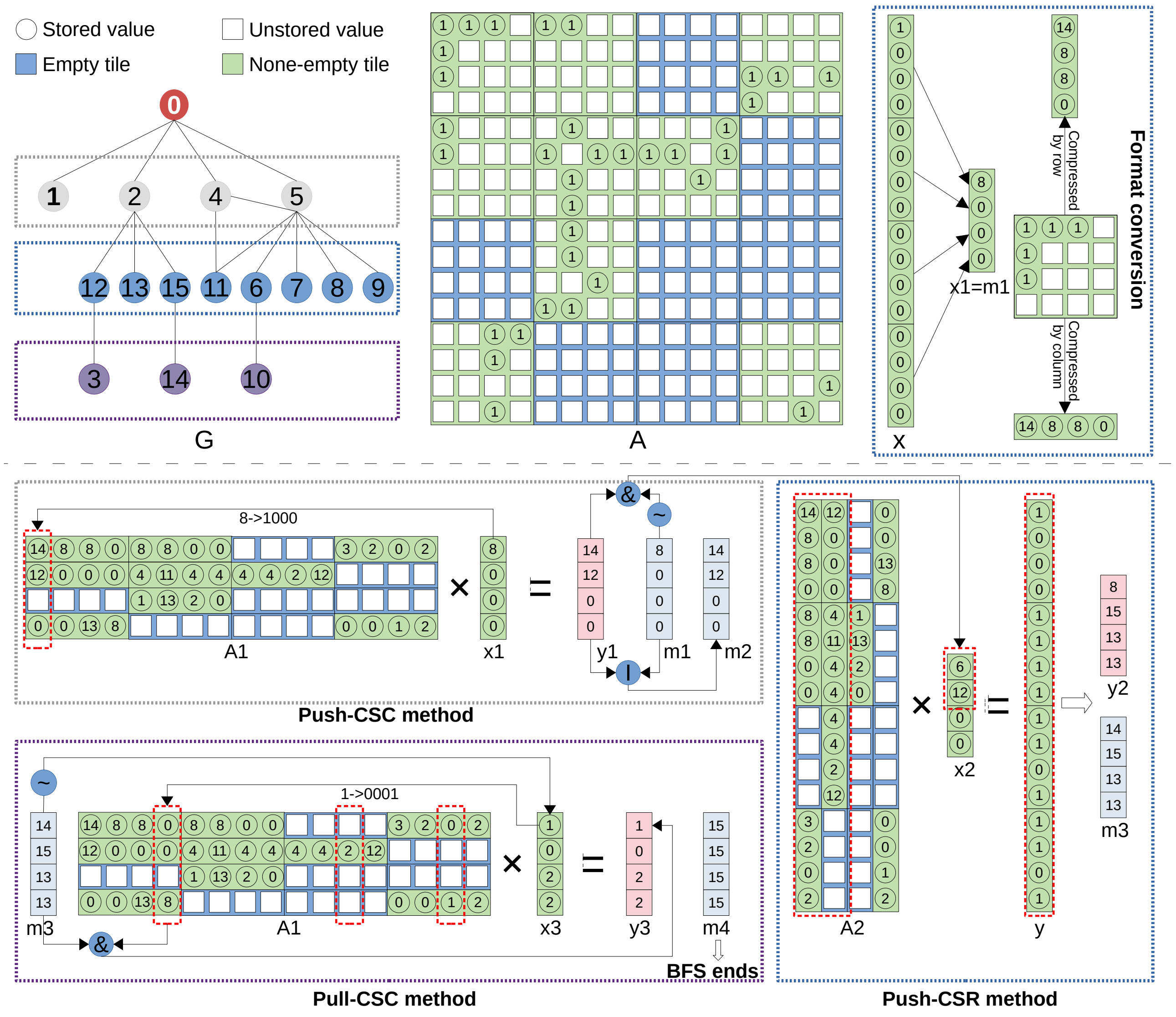


Figure 5: An example of BFS algorithm with a three-layer iterations using three methods starting from source vertex 0. The graph G is represented as a adjacency matrix A of size 16-by-16, which is divided into 4-by-4 tiles. Each non-empty tile is compressed horizontally or vertically to obtain $A1$ and $A2$. The current visited vertex set and the processed vertex set are compressed to represent xi and mi, respectively (i means the ith iteration). The first iteration adopts the Push-CSC method, selects the first column of $A1$ according to the nonzero position of $x1$. Then we use `OR` operation to operates the selected data and vector $y1$, store the result into the vector $y1$, and update the x and m vectors. In the second iteration, the Push-CSR method is adopted, and the nonzeros of each row of $A2$ and the corresponding nonzeros of $x2$ are used for `AND` operation to obtain $y2$. The third iteration uses the Pull-CSC method to find the corresponding column in $A1$ according to the nonzero element position loc in $x3$ obtained by $m3$, and conducts the `AND` operation with $m3$. If the result is not zero, set the result $y[loc]$ to 1. After three iterations, all vertices are traversed, and the BFS algorithm ends.

visited, and '0' otherwise. Both matrices and vectors are stored as bitmasks, and the matrix multiplication operation is now converted to semiring operations, where the AND operation represents multiplication, and the OR operation represents addition. Figure 5 shows the detailed process of the TileBFS algorithm from the data storage structure to one BFS iteration using three different methods will introduced later on. Here we will start BFS from the source vertex '0', and finish a total of three levels of traversal.

In TileBFS, the input vector x usually undergoes a process of increasing density first and then decreasing with some fluctuations. Because the density of matrix and vector will greatly affect the amount of calculations in the multiplication, it is difficult for a single operation method to maintain high performance when dealing with vectors of different sparsity. From the view of linear algebra, when the number of nonzeros in x is small, merging the corresponding columns in the CSC form perform better. On the contrary, CSR and SpMV may be more efficient when x is pretty dense. This performance difference due to density is also the linear algebra basis of the directional optimization BFS algorithm [6], which is also called the push-pull optimization in graph algorithms [52].

In order to further accelerate TileBFS, we design three types of SpMSpV methods according to the density of the input vector and the number of vertices visited:

(1) When the sparsity of the input vector x is less than 0.01 and the number of unvisited vertices is large, we will use a method called Push-CSC,
(2) When the sparsity of the input vector x is greater than or equal to 0.01 and the number of unvisited vertices is large, we will use a method called Push-CSR,
(3) When the number of unvisited vertices is small, we will use a method called Pull-CSC.

The Push-CSC method is driven from the vector and finds the tiles in the matrix. Firstly, we get corresponding matrix column tiles according to the positions of the nonzeros in the input vector, and then merge the corresponding matrix column tiles into the result vector tile. We use a 32-thread warp to process the nonzeros of a vector, where 32 threads will process consecutive nt tiles of the matrix. Then we store all non-empty column tiles into the resulting vector by atomic OR operation. This not only reduces the runtime caused by frequent replacement of data in on-chip memory, but also ensures that the work load of a single thread would not be too large. As shown in Figure 5, $x1$ is the input vector and the first digit '8' in it represents vector {1, 0, 0, 0}. This binary vector has only one nonzero, and its position corresponds to the first column of the sparse matrix $A1$. We can simplify the multiplication of $A1$ and $x1$ to the AND-OR operations to obtain the target vector $y1$ {14, 12, 0, 0}, which indicates that the vertices contained in the current layer and the previous layer are '0, 1, 2, 4, 5'. At this time, $m1$ records the vertex set contained in the 'upper layer' in the tree form of G before the operation. By performing bitwise OR operation on the upper layer vertex set $m1$ and the current layer vertex set $y1$, we can obtain their union $m2$ which records all the vertices visited so far. By inverting $m1$ and performing the AND operation with $y1$, the result sparse vector $x2$ {6, 12, 0, 0} shows that current layer contained vertices '1, 2, 4, 5'. Algorithm 5 shows a pseudocode of this procedure.

Algorithm 5 A pseudocode of warp level Push-CSC.

```
1: for $ti = 0$ to 31 in parallel do
2:     $sum\_x = A[(blk\_id << 6) + x\_id\%nt]$
3:     $blk\_y\_rowid = columnid[blk\_id]$
4:     $sum =$ (NOT ($mask[blk\_y\_rowid]$ AND $sum\_x$)) AND $sum\_x$
5:     AtomicOr($y[blk\_rowid]$, $sum$)
6:     AtomicOr($flag[blk\_rowid]$, $sum$)
7: end for
```

The Push-CSR method is matrix-driven and finds the vector from row tiles in the matrix. By multiplying each row tile of the matrix by the corresponding vector tiles, we can obtain the result vector tiles of the corresponding position of each row tile. We use a 32-thread warp to process a row tile of a matrix, and cyclically compute in the units of 32 threads. In a loop, each thread processes the nonzero elements in a matrix tile until all matrix tiles in the row tile are calculated. For row tiles which is very long, the load will be unbalanced and seriously affect the performance. Therefore, we introduce the method of splitting long row tiles for optimization and use multiple warps to process them for better load balancing. To reduce the amount of computations, we only compute corresponding matrix tiles when the input vector tiles are non-empty.

In the Push-CSR method part in Figure 5, we only need to compute the first two columns circled by the red box in the matrix $A2$. For example, the nonzero elements in the vector $x2$ correspond to the vector {6, 12} of length 2, and the first two elements in the first row of $A2$ form the vector {14, 12}. By conducting AND and OR on the two vectors, the value of the first item of the result vector y is '1'. Each bit of the vector y can be calculated like this.

Algorithm 6 A pseudocode of warp level Push-CSR.

```
1: for $ti = 0$ to 31 in parallel do
2:     $x\_i = x[columnid[blk\_id + ti]]$
3:     if $x\_i == 0$ then continue
4:     end if
5:     $sum\_x = 0$
6:     for $rj = 0$ to $nt$ do
7:         if $A[((blk\_id + ti) << 6) + rj]$ AND $x\_i$ !=0 then
8:             $sum\_x| = 1 << (nt - 1 - rj)$
9:         end if
10:    end for
11:    $sum =$ ( NOT ($mask[blk\_rowid]$ AND $sum\_x$)) AND $sum\_x$
12:    AtomicOr($y[blk\_rowid]$, $sum$)
13:    AtomicOr($flag[blk\_rowid]$, $sum$)
14: end for
```

The Pull-CSC method is also vector-driven and finds the matrix saved in the CSC style by elements in vector. Firstly, we calculate the input vector according to the mask vector, then several corresponding matrix column tiles can be found according to the position of the nonzeros of the input vector. The corresponding matrix column tiles will be added to the mask vector. If the resulting vector has a nonzero, the parent vertices of the unvisited vertices must have been visited, and the vertices can be added to the vector recording the visited vertices. We use a 32-thread warp to process the nonzero elements of a vector. These 32 threads will process consecutive tiles of size nt, and each thread processes a tile of vector by using AND operations. If obtained a nonzero element, the corresponding vertex will be recorded as a visited vertex, and the message will be synchronized to other threads to stop the operation of the warp.

In the Pull-CSC method part in Figure 5, the vector $x3$ can be obtained by bitwise inversion of each nonzeros in the vector $m3$. The element '1' in the first item of the vector $x3$ is the result of the compression of the vector {0, 0, 0, 1}. So it corresponds to the fourth column in the matrix $A1$ (0, 0, #, 8). By using the AND-OR operations of this vector with the vector $m3$, we can get the result {0, 0, 0, 1}.

If there exists nonzero, the same position of $y3$ will be set to '1' according to the position of the input vector at $x3$. After performing AND-OR operations with $m3$ for each column corresponding to the nonzero elements of vector $x3$ in A, we obtain the complete vector $y3$. Then $y3$ and $m3$ are computed by OR operation to get $m4$. In $m4$, all binary bits are 1, which means that all vertices have been visited. Up to now, the TileBFS algorithm is completed.

Algorithm 7 A pseudocode of warp level Pull-CSC.

```
for $ti = 0$ to 31 in parallel do
    if $x\_id = -1$ then break
    end if
    $sum\_x = A[(blk\_id << 6) + x\_id\%nt]$
    $blk\_y\_rowid = columnid[blk\_id]$
    $sum = ($ NOT $(mask[blk\_y\_rowid]$ AND $sum\_x))$ AND $sum\_x$
    if $sum\_x != 0$ then
        $sum = sum$ Or $1 << (nt - 1 - x\_id\%nt)$
        AtomicOr$(flag[blk\_rowid], sum)$
        $x\_id = -1$
    end if
end for
```

In order to deal with the very sparse part extracted from the matrix (recall Section 3.2.1), we use GSwitch [37] to traverse this part and to complete each iteration. The operation is like multiplying two matrices with the same input vector, and merge the results into one output vector. Although the very sparse cases do not show frequently in the SuiteSparse dataset, we can see this hybrid optimization can greatly reduce unnecessary space and computing time once it is required.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

Our experimental platform is a Linux machine with two NVIDIA Ampere GPUs, a Geforce RTX 3060 and a Geforce RTX 3090, installed. The CUDA version is 11.4, and the GPU driver version is 495.29.05.

In order to compare the performance of TileSpMSpV more comprehensively, we compare the algorithms of SpMV and SpMSpV. For SpMV, we compared with the TileSpMV [40] algorithm and the cusparse?bsrmv() kernel in the cuSPARSE library. For SpMSpV, we did our best to implement the GPU version of the SpMSpV-bucket algorithm in the CombBLAS library [3], which is hundreds of times faster than its CPU version, and compared the performance with it. For BFS, we compare our TileBFS with the BFS kernels in the Gunrock framework proposed by Wang et al. [48], and in the GSwitch framework proposed by Meng et al [37]. In our experiments, we use all the optimizations in the two frameworks, including push-pull, ignore-weight, directed, etc., for a fair comparison. Table 1 lists the specifications of the testbed used and algorithms involved.

Our evaluation dataset for SpMSpV contains all the 2757 sparse matrices from the SuiteSparse Matrix Collection [14]. Inside the dataset, 2081 sparse matrices are square and are used for testing BFS. We also list 12 representative matrices in Table 2 to in-depth analyze the performance of the algorithms tested.

Table 1: The specifications of the machine and the seven algorithms evaluated.

	Algorithm	Machine specification
SpMSpV	(1) TileSpMV [40]	(1) NVIDIA Geforce RTX 3060 (Ampere), 3,584 CUDA cores @ 1.78 GHz, 12 GB GDDR6, B/W 360.0 GB/s, (2) NVIDIA Geforce RTX 3090 (Ampere), 10,496 CUDA cores @ 1.70 GHz, 24 GB GDDR6X, B/W 936.2 GB/s.
	(2) cuSPARSE v11.4 BSR	
	(3) CombBLAS [3]	
	(4) TileSpMSpV (this work)	
BFS	(1) Gunrock [48]	
	(2) GSwitch [37]	
	(3) TileBFS (this work)	

Table 2: Information of the 12 representative matrices.

Matrix	Size	#nonzeros	#tiles (16*16)	#tiles (32*32)	#tiles (64*64)
af_5_k101	503K x 503K	17M	257K	110K	55K
cant	62K x 62K	4M	62K	20K	8K
cavity23	4K x 4K	144K	2K	1K	1K
pdb1HYS	36K x 36K	4M	50K	19K	8K
fullb	199K x 199K	11M	31K	112K	220K
ldoor	952K x 952K	46M	998K	574K	380K
in-2004	1M x 1M	27M	1M	641K	363K
msdoor	415K x 415K	20M	484K	288K	191K
roadNet-TX	1M x 1M	3M	1M	740K	464K
ML_Geer	1M x 1M	110M	1M	694K	332K
333SP	3M x 3M	22M	8M	7M	7M
dielFilterV2clx	607K x 607K	25M	2M	1M	481K

4.2 Performance Comparison of SpMSpV

By benchmarking the 2757 matrices on RTX 3090, we compare our TileSpMSpV algorithm with TileSpMV, cuSPARSE and CombBLAS. Figure 6 shows the performance of the four algorithms under different vector sparsity. Among them, vectors with different sparsity are generated randomly with random seeds 1. It represents that the result of our experiment can be reproduced in other comparison scenarios.

As can be seen from Figure 6, our algorithm shows the best performance on most matrices. Specifically, we achieve speedups of on average (geometric mean) 1.10x, 1.65x, 2.20x, 2.38x (up to 1.42x, 4.85x, 12.14x, 12.34x) over TileSpMV, and speedups of on average 7.58x, 13.78x, 22.43x, 24.95x (up to 275.17x, 647.04x, 1452.73x, 1825.17x) over cuSPARSE, the speedups of on average 13.46x, 14.85x, 20.06x, 20.43x (up to 235.65x, 304.30x, 190.25x, 213.67x) over CombBLAS at vector sparsity of 0.1, 0.01, 0.001 and 0.0001.

As the vector becomes more and more sparse, the amount of computation required by SpMSpV decreases, while the necessary computation overhead in the algorithm remains unchanged, and the resource utilization of GPU decreases. In this case, TileSpMSpV has a significant performance improvement over TileSpMV due to its way of quickly locating the nonzero positions of sparse matrices and avoiding a lot of computations. For example, the matrix 'TSOPF_RS_b2383' has only 0.25% nonzero vector tiles, and reaches 12.34x, 161.01x and 55.60x speedups over TileSpMV, cuSPARSE and CombBLAS, respectively.

When the vectors are denser, the performance of TileSpMSpV is still higher than that of the SpMV algorithms (TileSpMV and cuSPARSE), and the performance improvement is more obvious on large matrices. Our highest performance occurs on the 'trans5' matrix, which can reach 79.43 GFlops, and is 1.23x, 145.03x and 232.75x faster than TileSpMV, cuSPARSE and CombBLAS. Because the number of non-empty tiles here is far less than in the other

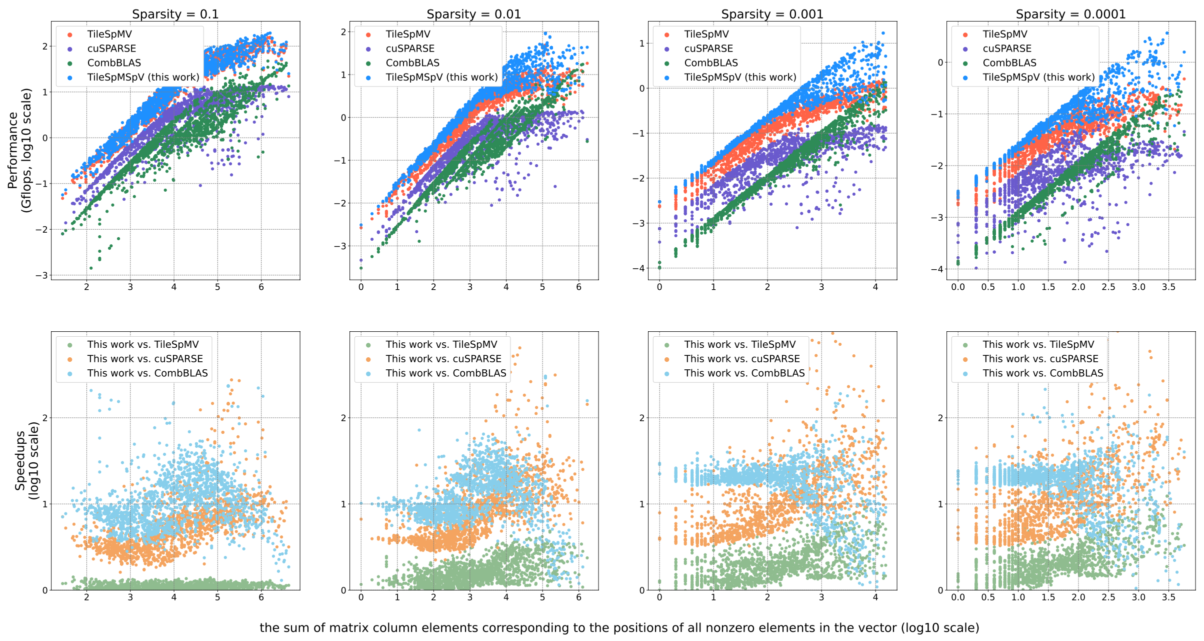
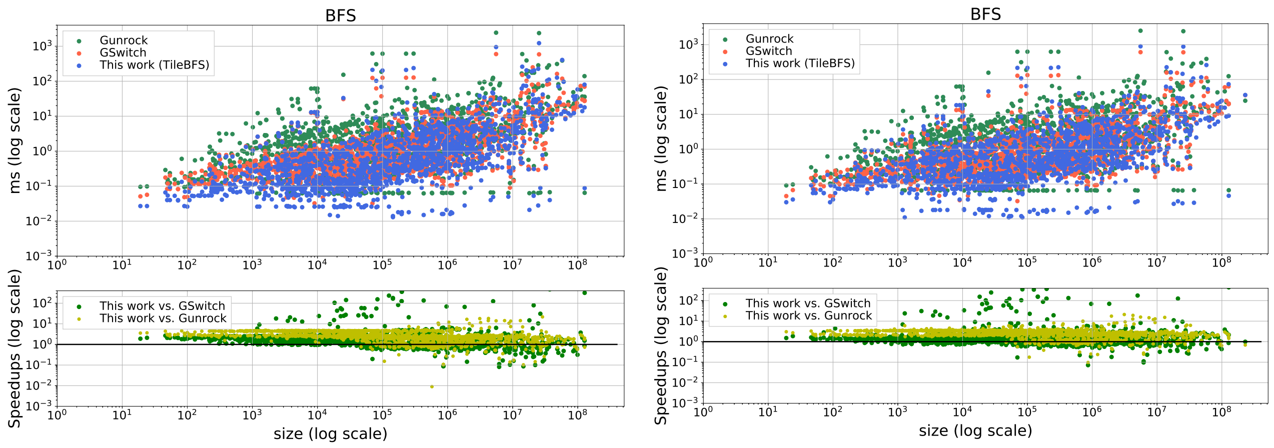


Figure 6: Performance comparison of TileSpMSpV, TileSpMV, cuSPARSE and CombBLAS method with four different sparsity. The four sub-figures on top show performance (in GFlops), and the four sub-figures on the bottom show the speedups of our algorithm over TileSpMV, cuSPARSE and CombBLAS.



(a) BFS performance and speedups on RTX 3060

(b) BFS performance and speedups on RTX 3090

Figure 7: The two sub-figures on top show the performance of Gunrock, GSwitch and TileBFS on the two GPUs. The two sub-figures on the bottom show the speedups of TileBFS over Gunrock and GSwitch on the two GPUs.

matrices of the same scale, the nonzeros of the calculated matrix are relatively concentrated (only 0.00018% non-empty tiles in total). So the runtime can be reduced by avoiding a large number of tile information accesses. Also, TileSpMSpV reduces the overhead of matrix tiles and ensures the efficiency by using the COO format to store nonzeros if extremely sparse. For example, the 'cryg10000' matrix has 2.19% of non-empty tiles before extracting the COO data. After adopting the extra COO format, 1.10% of non-empty tiles are moved out, and the performance is improved by 1.6x.

4.3 Performance Comparison of BFS

We use the two GPUs, i.e., RTX 3060 and RTX 3090, for comparing our TileBFS algorithm with the BFS algorithms in Gunrock and GSwitch on the 2081 square matrices. The results are shown in Figure 7. As can be seen, our algorithm has better performance in most

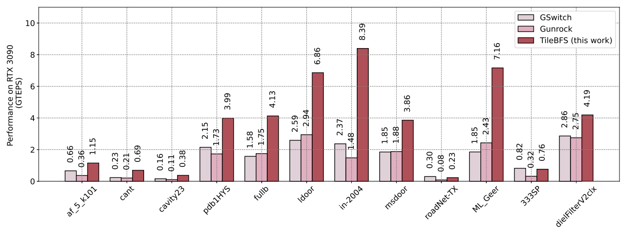


Figure 8: Performance comparison of 12 representative matrices on RTX 3090 GPU

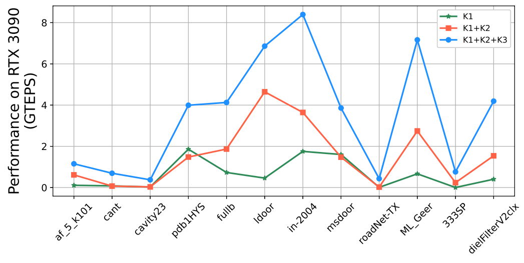


Figure 9: Comparison of BFS performance using three-direction optimization step by step of the representative matrices.

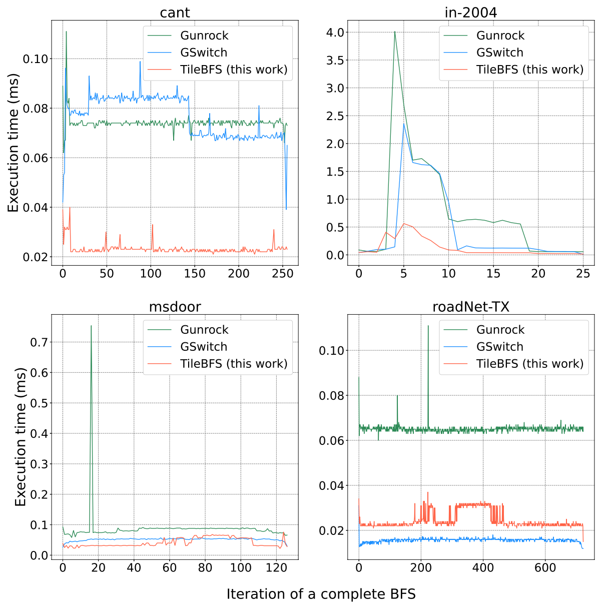


Figure 10: The iteration time comparison of Gunrock, GSwitch and TileBFS.

matrices than the other two algorithms. On RTX 3060, the average speedup (geometric mean) of TileBFS over Gunrock is 3.03x, and the maximum speedup is 21.70x. On 92.25% matrices of the dataset, TileBFS is faster than Gunrock. Compared to GSwitch, the average speedup is 4.35x, and the maximum speedup is 837.36x. On 73.00% matrices, TileBFS is faster than GSwitch. On RTX 3090, the average speedup of TileBFS over Gunrock is 2.74x, and the maximum speedups is 21.01x. On 93.99% matrices, TileBFS is faster. As for GSwitch, the average speedup is 4.69x, and the maximum speedups is 1164.35x. TileBFS is faster than GSwitch on 68.6% matrices.

Table 2 lists 12 typical matrices for more detailed experimental analysis, and Figure 8 shows BFS performance of the three algorithms on the matrices. In particular, our algorithm performs well on matrices with less non-empty tiles occupation and dense distribution of nonzeros in the tiles, the matrix 'ldoor' is an example. For these matrices, we not only use less memory through data compression and improve memory access efficiency, but also increase the parallelism of the tile-oriented CUDA kernels. Compared with Gunrock and GSwitch, here we obtain the average speedups of 2.33x and 2.65x. The scalability of our algorithm can be seen in the two sub-figures of Figure 7. On RTX 3090, the BFS performance on large-scale matrices is significantly improved over on RTX 3060. For example, the BFS performance of the matrix 'dielFilterV3real' on RTX 3090 has a speedup of 2.42x over on RTX 3060.

4.4 Directional Optimization Analysis

We test the performance of step-wise stacking of the three directional optimization BFS kernels: Push-CSC (K1), Push-CSR (K2) and Pull-CSC (K3). Figure 9 shows the performance of BFS using one, two, and three directional optimization methods, respectively. It can be seen that because we divide the input data of BFS and the fluctuation trend of the number of unvisited vertices in a fine-grained manner and formulate an appropriate switching timing. The performance is significantly improved with the increase in the number of optimized kernels in the direction of use. On the matrix 'dielFilterV2clx', the speedups of 2.32x and 7.91x can be achieved.

4.5 Iteration Time Analysis

Figure 10 shows the comparison of time of each iteration step of four representative matrices processed by Gunrock, GSwitch and TileBFS. As can be seen, when the runtime of the three algorithms are in the same trend, our algorithm is often obviously faster and more stable than the other algorithms. Thanks to our kernel selection methods, the execution time of TileBFS can be effectively controlled, in particular when the compute time of the other methods nearly reach their peak (see the matrices 'in-2004' and 'msdoor'). Moreover, right before the end of the traversal, TileBFS sometimes switches to the Pull-CSC kernel, and may consume a bit longer time at a certain iteration before the end (see the matrices 'msdoor' and 'cant'). But this approach actually saves time when the input vector is very dense. For the matrix 'roadNet-TX', our TileBFS is much faster than Gunrock, but is slower than GSwitch. The reason is that the matrix has a large amount of non-empty tiles leading to low compute effciency, and dynamic vector sparsity makes the kernels Push-CSC and the Push-CSR switch back and forth during the calculation. But the switching time is far less than GSwitch and Gunrock in terms of the trend.

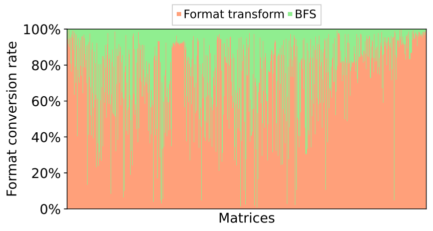


Figure 11: Comparison of preprocessing time and a BFS time of the representative matrices.

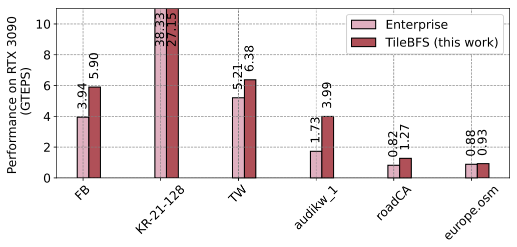


Figure 12: Performance comparison of 6 representative matrices on RTX 3090 GPU

4.6 Format Conversion Overhead

We show a comparison of the time converted a CSR matrix to tiled format and a BFS execution time in Figure 11 on RTX 3090. The time for format conversion does not exceed a single BFS processing time in normal cases, and does not exceed 10x of a single BFS processing time in most cases. This conversion efficiency is acceptable, since after an input graphs is converted to the tile format once, it can be continuously traversed many times from different sources.

4.7 Comparison over Enterprise

To our knowledge, Enterprise [32] is the first BFS algorithm that performs different load balancing for different out-degrees of the frontiers in BFS and reduces random access to memory. Figure 12 shows the performance comparison of TileBFS and Enterprise on RTX 3090 using sparse matrices listed in its original paper. It can be seen that TileBFS outperforms Enterprise on most matrices. The average speedup is 1.39, and the maximum speedup is 2.31. Especially on the matrix *audikw*1 which had low percentage of nonzero tiles, our algorithm greatly reduces the memory overhead and achieves higher speedups.

5 RELATED WORK

There have been **a few studies focusing on parallel SpMSpV**. Li et al. [30] and Yang et al. [53] developed SpMSpV methods using merge primitives on CPUs and GPUs, respectively. In the Combinatorial BLAS package [5, 9], Azad et al. [3] developed an SpMSpV method by using MPI+OpenMP for distributed environments. Li et al. [31] recently proposed an algorithm to select either SpMV or SpMSpV for a certain sparsity of an input vector. Yavits and Ginosar [55] developed an acceleration architecture for SpMSpV used in machine learning. Burkhardt [10] recently gave theoretically analysis for using SpMSpV in BFS.

Many **graph processing frameworks**, such as GraphLab [36], GraphMat [47], MultiGraph [22], PowerGraph [18], CuSha [27], GraphReduce [44], PowerLyra [11], Ligra [45], GSwitch [37], NXgraph [12], GraphPhi [43], and GraphBLAST [51] have been designed for the development of fast graph applications.

Accelerating BFS may be the most important task of parallel graph processing. The directional optimization BFS algorithm proposed by Beamer et al. [6] has been used in many subsequent accelerating techniques. Merrill et al. [38] first developed fast BFS on GPUs. Hong et al. [24],Liu and Huang [32], Wen and Zhang [49] , Zhang and Lin [56] , Li and Becchi [29] also implemented BFS algorithms for GPUs.

There also has been a few works on **graph computations on various hardware platforms**. For distributed environments, Buluç and Madduri [8], Hong et al. [23], Besta et al. [7] and Faisal et al. [15] proposed some new graph algorithms and frameworks. In contrast, processing big graphs on a single machine is accelerated by Nguyen et al. [39], Gera et al. [16], and Han et al. [21]. Also, designing and evaluating accelerators for graph processing received attention by Ahn et al. [1], Ham et al. [20], Pal et al. [42] and Zhang et al. [57].

Utilizing sparse linear algebra for accelerating graph problems is another efficient way received much attention. GraphBLAS [17, 26] is the representative in this direction. Yang et al. [51] designed the GraphBLAST package. Sundaram et al. [47] in the GraphMat package reduced complexity of the use of graph analytics through matrix operations. Yang et al. [54], designed sparse matrix storage formats for fast graph processing. Yang et al. [52] discussed several techniques for accelerating BFS by using sparse matrix operations.

Moreover, TileSpMSpV and TileBFS are the latest work that extends and improves the **tiled formats and algorithms** recently proposed in our TileSpMV [40] and TileSpGEMM [41].

6 CONCLUSION

In this paper, we have proposed the tiled storage structures for sparse matrices and vectors, as well as the TileSpMSpV and the TileBFS algorithms on GPUs. Compared to the existing work, we improve the data locality through tile-wise data accesses and the execution efficiency by proposing three vector-friendly traversal kernels. Our experiments showed that the TileSpMSpV and TileBFS bring obvious performance advantages over TileSpMV, cuSPARSE and CombBLAS, and two representative BFS methods Gunrock and GSwitch, respectively.

ACKNOWLEDGMENTS

We deeply appreciate the invaluable comments from all the reviewers. We are also so grateful to Ke Meng, Yuyao Niu, Shuhui Song, Xu Fu and Zhengyang Lu for the help in the experimental test. Weifeng Liu is the corresponding author of this paper. This research was supported by the National Natural Science Foundation of China under Grant No. 61972415, 61972377, 62032023, and the National Key R&D Program of China under Grant No. 2021YFB0300600.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA '15*.

[2] Michael J. Anderson, Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Theodore L. Willke, and Pradeep Dubey. 2016. GraphPad: Optimized Graph Primitives for Parallel and Distributed Platforms. In *IPDPS '16*.

[3] Ariful Azad and Aydin Buluç. 2017. A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm. In *IPDPS '17*.

[4] Ariful Azad, Mathias Jacquelin, Aydin Buluç, and Esmond G. Ng. 2017. The Reverse Cuthill-McKee Algorithm in Distributed-Memory. In *IPDPS '17*.

[5] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John R. Gilbert, and Aydın Buluç. 2022. Combinatorial BLAS 2.0: Scaling Combinatorial Algorithms on Distributed-Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 989–1001.

[6] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12*.

[7] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *HPDC '17*.

[8] Aydin Buluç and Kamesh Madduri. 2011. Parallel Breadth-First Search on Distributed Memory Systems. In *SC '11*.

[9] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.

[10] Paul Burkhardt. 2021. Optimal Algebraic Breadth-First Search for Sparse Graphs. *ACM Transactions on Knowledge Discovery from Data* 15, 5 (2021).

[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *EuroSys '15*.

[12] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An efficient graph processing system on a single machine. In *ICDE '16*.

[13] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4 (2019).

[14] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[15] S. M. Faisal, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Global graphs: A middleware for large scale graph processing. In *Big Data '14*.

[16] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing Large Graphs on GPUs with Unified Memory. *Proc. VLDB Endow.* 13, 7 (2020).

[17] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2007. High-Performance Graph Algorithms from Parallel Sparse Matrices. In *LLC '07*.

[18] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI '12*.

[19] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978), 250–269.

[20] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO '16*.

[21] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: A Fast Parallel Graph Engine Handling Billion-Scale Graphs in a Single PC. In *KDD '13*.

[22] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. 2017. MultiGraph: Efficient Graph Processing on GPUs. In *PACT '17*.

[23] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: A Fast Distributed Graph Processing Engine. In *SC '15*.

[24] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *PACT '11*.

[25] Haonan Ji, Shibo Lu, Kaixi Hou, Hao Wang, Zhou Jin, Weifeng Liu, and Brian Vinter. 2021. Segmented Merge: A New Primitive for Parallel Sparse Matrix Computations. *Int. J. Parallel Program.* 49, 5 (2021), 732–744.

[26] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, Matrices, and the GraphBLAS: Seven Good Reasons. *Procedia Computer Science* 51 (2015), 2453–2462.

[27] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-Centric Graph Processing on GPUs. In *HPDC '14*.

[28] Daniel Langr and Pavel Tvrdík. 2016. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2016), 428–440.

[29] Da Li and Michela Becchi. 2013. Deploying Graph Algorithms on GPUs: An Adaptive Solution. In *IPDPS '13*.

[30] Haoran Li, Harumichi Yokoyama, and Takuya Araki. 2018. Merge-Based Parallel Sparse Matrix-Sparse Vector Multiplication with a Vector Architecture. In *HPCC/SmartCity/DSS '18*.

[31] Min Li, Yulong Ao, and Chao Yang. 2021. Adaptive SpMV/SpMSpV on GPUs for Input Vectors of Varied Sparsity. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021), 1842–1853.

[32] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-First Graph Traversal on GPUs. In *SC '15*.

[33] Junhong Liu, Xin He, Weifeng Liu, and Guangming Tan. 2019. Register-Aware Optimizations for Parallel Sparse Matrix-Matrix Multiplication. *International Journal of Parallel Programming* (2019).

[34] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *IPDPS '14*.

[35] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS '15*.

[36] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. In *UAI'10*.

[37] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *PPoPP '19*.

[38] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *PPoPP '12*.

[39] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *SOSP '13*.

[40] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *IPDPS '21*.

[41] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs. In *PPoPP '22*.

[42] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. 2021. SparseAdapt: Runtime Control for Sparse Linear Algebra on a Reconfigurable Accelerator. In *MICRO '21*.

[43] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. 2018. Graphphi: Efficient Parallel Graph Processing on Emerging Throughput-Oriented Architectures. In *PACT '18*.

[44] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *SC '15*.

[45] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP '13*.

[46] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling Betweenness Centrality Using Communication-Efficient Sparse Matrix Multiplication. In *SC '17*.

[47] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (2015), 1214–1225.

[48] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *PPoPP '16*.

[49] Hao Wen and Wei Zhang. 2019. Improving Parallelism of Breadth First Search (BFS) Algorithm for Accelerated Performance on GPUs. In *HPEC '19*.

[50] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2022. A Pattern-Based SpGEMM Library for Multi-Core and Many-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2022), 159–175.

[51] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear. algebra-based graph framework on the GPU. *ACM Trans. Math. Software* 48, 1 (2022).

[52] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Implementing Push-Pull Efficiently in GraphBLAS. In *ICPP '18*.

[53] Carl Yang, Yangzihao Wang, and John D. Owens. 2015. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU. In *IPDPSW '15*.

[54] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. 2011. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. *Proc. VLDB Endow.* 4, 4 (2011), 231–242.

[55] Leonid Yavits and Ran Ginosar. 2018. Accelerator for Sparse Machine Learning. *IEEE Computer Architecture Letters* 17, 1 (2018), 21–24.

[56] Feng Zhang, Heng Lin, Jidong Zhai, Jie Cheng, Dingyi Xiang, Jizhong Li, Yunpeng Chai, and Xiaoyong Du. 2018. An Adaptive Breadth-First Search Algorithm on Integrated Architectures. *J. Supercomput.* 74, 11 (2018), 6135–6155.

[57] Feng Zhang, Weifeng Liu, Ningxuan Feng, Jidong Zhai, and Xiaoyong Du. 2019. Performance Evaluation and Analysis of Sparse Matrix and Graph Kernels on Heterogeneous Processors. *CCF Transactions on High Performance Computing* (2019).

[58] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *PPoPP '18*. 94–108.