TileSpGEMM: A Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs

Yuyao Niu
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
2019211256@student.edu.cn

Zhengyang Lu
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
2021211259@student.edu.cn

Haonan Ji
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
2020211256@student.edu.cn

Shuhui Song
Super Scientific Software Laboratory, China University of Petroleum-Beijing
Beijing, China
2019211805@student.edu.cn

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

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

Abstract

Sparse general matrix-matrix multiplication (SpGEMM) is one of the most fundamental building blocks in sparse linear solvers, graph processing frameworks and machine learning applications. The existing parallel approaches for shared memory SpGEMM mostly use the row-row style with possibly good parallelism. However, because of the irregularity in sparsity structures, the existing row-row methods often suffer from three problems: (1) load imbalance, (2) high global space complexity and unsatisfactory data locality, and (3) sparse accumulator selection.

We in this paper propose a tiled parallel SpGEMM algorithm named TileSpGEMM. Our algorithm sparsifies the tiled method in dense general matrix-matrix multiplication (GEMM), and saves each non-empty tile in a sparse form. Its first advantage is that the basic working unit is now a fixed-size sparse tile containing a small number of nonzeros, but not a row possibly very long. Thus the load imbalance issue can be naturally alleviated. Secondly, the temporary space needed for each tile is small and can always be in on-chip scratchpad memory. Thus there is no need to allocate an off-chip space for a large amount of intermediate products, and the data locality can be much better. Thirdly, because the computations are restricted within a single tile, it is relatively easier to select a fast sparse accumulator for a sparse tile. Our experimental results on two newest NVIDIA GPUs show that our TileSpGEMM outperforms four state-of-the-art SpGEMM methods cuSPARSE, bhSPARSE, NSPARSE and spECK in 139, 138, 127 and 94 out of all 142 square matrices executing no less than one billion flops for an SpGEMM operation, and delivers up to 2.78x, 145.35x, 97.86x and 3.70x speedups, respectively.

CCS Concepts: • **Mathematics of computing → Solvers**; **Mathematical software performance**; • **Computing methodologies → Shared memory algorithms**; **Vector / streaming algorithms**.

Keywords: Sparse matrix, SpGEMM, tiled algorithm, GPU

1 Introduction

Sparse general matrix-matrix multiplication (SpGEMM) operation multiplies two sparse matrices A and B, and gives a resulting sparse matrix C. As a key kernel in the level-3 sparse basic linear algebra subprograms (Sparse BLAS) [44, 45], Combinatorial BLAS [8, 20, 26] and GraphBLAS [36, 68], SpGEMM has a wide range of applications in sparse linear solvers (e.g., algebraic multigrid methods [9, 13, 47–49]), graph processing frameworks [16] (e.g., breath first search [6] and triangular counting [35, 107, 114]), machine learning scenarios (e.g., Markov clustering [7, 97] and pruned deep neural networks [18, 37, 46]) and others (e.g., database [40] and genome assembly [56]).

Because the two input and one output matrices are all sparse, parallelizing SpGEMM is in general more complex than other sparse kernels such as sparse matrix-vector multiplication (SpMV) [15, 28, 52, 81, 83, 89, 90, 94, 115, 120, 121] and sparse matrix-multiple vector multiplication (SpMM) [58,

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508431

61, 72, 112], and thus in recent years has received much attention on a variety of modern parallel platforms such as GPUs [3, 4, 14, 24, 29, 32, 34, 42, 43, 53–55, 66, 74, 75, 79, 80, 92, 93, 96, 106, 109, 110, 117], Xeon Phi [1, 2, 39, 42, 43, 93], domain specific architectures [95, 103, 119], and distributed clusters [10, 12, 13, 22, 23, 62, 63, 108].

Currently, most existing SpGEMM methods on parallel processors use the Gustavson's row-row method [57] as a base. Such approaches in parallel generate the sparse rows of the resulting matrix C by merging sparse rows of B scaled by the nonzeros in the corresponding sparse rows of A. However, although the row-row methods can often bring adequate parallelism, they are hard to resolve three challenging problems: (1) how to address the load balancing problem through a variety of inspecting and binning approaches [3, 80, 92, 93, 96, 106], (2) how to find a good temporary space size to allocate an intermediate matrix for calculating the resulting matrix with good data locality [74, 80, 92, 93, 96], and (3) how to design more efficient sparse accumulator by using dense row [51, 96], heap [4, 80], hash [3, 42, 43, 92, 93, 96], sort [14, 34, 60, 74] and merge [32, 53, 54, 66, 80] for rows of different lengths and sparsity structures.

Even though the three problems have been intensively studied in the recent SpGEMM work, it should be noticed that the hardware resources of modern parallel processors such as GPUs are still largely underused. The reasons are from three aspects: (1) it is hard to efficiently deal with very long rows with thousands of or more nonzeros that not only bring load imbalance but also require much larger space than the capacity of the on-chip scratchpad memory, (2) the size of the temporary space allocated for storing the intermediate products depends on the total number of operations and an initial guess of the size of the final C. In many cases, the intermediate space can be quite large and thus takes much global memory space and long execution time, and (3) the row-based sparse accumulators in the row-row methods cannot exploit 2D spatial structure of matrices, thus may only bring limited data locality. As a result, the existing row-row approaches often deliver unsatisfactory performance.

To better use hardware resources of modern GPUs, we in this paper propose a novel method called TileSpGEMM. In short, TileSpGEMM can be seen as a sparse version of the widely used tiled algorithms in dense general matrix-matrix multiplication (GEMM). Its basic working unit is a sparse tile with a well bounded size (e.g., a 16-by-16 sparse tile contains no more than 256 nonzeros) and can be stored in on-chip scratchpad memory, but not a sparse row of diverse lengths. Thus the above three performance issues in the classic row-row SpGEMM methods can be better resolved: (1) the load imbalance problem caused by uneven row lengths can be largely alleviated, (2) global memory space for storing intermediate products is not required, and (3) sparse accumulator with better data locality is easier to design.

To make the tiled formulation more efficient on modern GPUs, we design a tiled sparse format containing mask information and adaptive nonzero structure in the CSR-style, an adaptive sparse accumulator that optimizes a variety of input sparse tile structures, as well as a three-stage optimization framework for computing tile layout, symbolic and numeric SpGEMM.

By testing all 142 square sparse matrices using no less than one billion floating point operations for both $C = A^2$ and $C = AA^T$ operations from the SuiteSparse Matrix Collection [38] on two latest NVIDIA RTX 3060 and 3090 Ampere GPUs, our TileSpGEMM algorithm significantly outperforms four existing methods: NVIDIA cuSPARSE v11.4 [31], bhSPARSE [80, 82], NSPARSE [93], and spECK [96]. The experimental results show that our method is faster than them on 139, 138, 127 and 94 matrices and achieves up to 2.78x, 145.35x, 97.86x and 3.70x speedups over them, respectively. Compared with the tSparse library [118] using half precision dense tile-wise multiplication on GPU tensor cores, our TileSpGEMM obtains on average 1.98x and up to 4.04x speedups. Also, our experiments show that our tiled sparse format in general takes less space, compared to the standard CSR format.

This work makes the following contributions:

- We avoid three major performance issues of classic row-row SpGEMM.
- We propose TileSpGEMM including a sparse tile data structure and corresponding SpGEMM algorithm.
- We develop optimization techniques for the TileSpGEMM on modern GPUs.
- We achieve significant speedups over existing SpGEMM work on large matrices of various structures.

2 Background

2.1 SpGEMM and Its Row-Row Algorithm

The SpGEMM operation computes $C = AB$, where A, B and C are all sparse matrices. Figure 1 gives an example of multiplying A of eight nonzeros and B of ten nonzeros and getting C of 11 nonzeros.

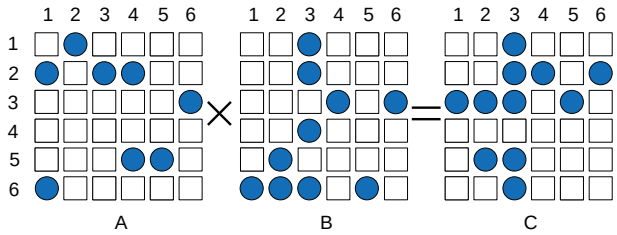

Figure 1. An example of computing $C = AB$.

Currently, most parallel SpGEMM algorithms use Gustavson's row-row formulation [57] shown in Algorithm 1. Such methods exploit the fact that calculating the rows of C

are independent of each other and thus can be parallelized (line 1 in Algorithm 1). Then each nonzero a_{ij} in the row a_{i*} (line 4) is used to scale the nonzeros b_{jk} in the row b_{j*} (line 6) and insert (lines 8-9) or add (line 11) the intermediate products onto the nonzero c_{ik} in the row c_{i*}.

Algorithm 1 A row-row SpGEMM pseudocode for $C = AB$.

```
1:  for each a_i* in A in parallel do ▷ Perf. issue #1. load imbalance among rows
2:      predict size of c_i*          ▷ Perf. issue. #2. space allocation of intermediate products
3:      malloc c_i*
4:      for each nonzero entry a_ij in a_i* do
5:          for each nonzero entry b_jk in b_j* do
6:              value ← a_ij b_jk
7:              if c_ik ∉ c_i* then ▷ Perf. issue #3. sparse accumulator design
8:                  insert c_ik to c_i*
9:                  c_ik ← value
10:             else
11:                 c_ik ← c_ik + value
12:             end if
13:         end for
14:     end for
15: end for
```

2.2 Performance Issues of Row-Row SpGEMM

In spite of the good parallelism of the row-row algorithms, there are still three major performance issues (lines 1, 2-3 and 7-12 highlighted in Algorithm 1, respectively).

The first performance issue (line 1) is that the amount of computations for the rows of C can be very imbalanced, thus it is hard to saturate massively parallel GPUs when a small number of long rows dominate runtime in the row-row style.

The second performance issue (lines 2-3) is that the size of C is unknown in advance and requires a space for storing a possibly large amount of intermediate products, thus it is hard to allocate a proper size at runtime.

The third performance issue (lines 7-12) is that the row-row method inserts nonzeros to random positions in the rows of C of unpredictable sparsity, thus it is hard to design a sparse accumulator to efficiently accumulate new entries.

2.3 Motivation of This Work

To resolve the above three issues, tens of SpGEMM algorithms have been proposed in recent years [1–4, 10, 12–14, 22–24, 29, 32, 34, 39, 42, 43, 53–55, 74, 79, 80, 92, 93, 95, 96, 106, 109, 110, 117, 119]. However, unfortunately, those problems are inherent in the row-row style, meaning that even very smartly designed methods cannot break through the restrictions as long as the row-row computation is still the fundamental pattern. We can take running $C = A^2$ on the 'webbase-1M' matrix of 1,000,005 order listed in Table 2 as an example. It is a classic power-low matrix that easily brings imbalanced computations. Specifically, among the 1,000,005 rows, three need more than 100,000 floating points operations, and 190 need more than 10,000 operations, while the majority of the remaining 999,812 rows only need less than 100 operations. Such imbalanced amount of computations in general lead to underuse of GPU cores when row-row SpGEMM methods are called, since the small amount of rows will dominate the runtime. Furthermore, a large space need to be allocated to store the intermediate products when calculating these rows. At the same time, it is hard to design an efficient accumulator for the long rows, since they in general cannot be fully saved in small on-chip memory.

This fact motivates us to resolve the challenges from the bottom and to explore a novel way for parallel SpGEMM. We in this work select an actually more classic tiled method, which is almost always used in dense GEMM. The difference between our tiled SpGEMM and tiled GEMM are that we only store non-empty tiles in A, B and C, and the tiles are all in sparse form. Because now a sparse tile of fixed size (16-by-16 in this work) is set as the basic working unit and can be stored in fast on-chip memory, it is easier to deal with load imbalance (issue 1), intermediate space allocation (issue 2) and random insertion (issue 3). The experimental results in Figure 7 also prove that our TileSpGEMM obviously outperformed the row-row methods. By resolving the above three challenges, running $C = A^2$ on 'webbase-1M' with our TileSpGEMM is 2.17x, 7.26x, 3.11x and 1.96x faster than with the row-row style methods cuSPARSE, bhSPARSE, NSPARSE and spECK, respectively.

Despite the advantages and clarity of the tiling pattern, making TileSpGEMM efficient is non-trivial and requires addressing three new problems: (1) how to store the most effective information of the sparse tiles, (2) how to efficiently gather sparse tiles needed from A and B to compute the tiles in C, and (3) how to design adaptive tile-wise sparse accumulator. The next section will introduce our methods.

3 TileSpGEMM

3.1 Overview

The TileSpGEMM work consists of a sparse tile data structure and a tiled SpGEMM algorithm using the data structure.

Compared to the standard CSR format using row as the basic working unit, the key feature of the TileSpGEMM is that the input and output sparse matrices are all stored as a number of non-empty sparse tiles of the same size, and the sparse tile now becomes the basic working unit in TileSpGEMM. The tile size is always set to 16-by-16 in this work (convenient to utilize 8-bit unsigned char data type for local indices), indicating that each tile contains no more than 256 nonzeros. For each sparse tile, we store its nonzeros in the CSR style plus row indices and bit masks. Section 3.2 will introduce the sparse tile data structure.

On top of the sparse tile data structure, the TileSpGEMM algorithm includes three steps: (1) finding the possibly non-empty tiles in C, (2) determining the number of nonzeros and the row pointer array of each tile in C, and allocating

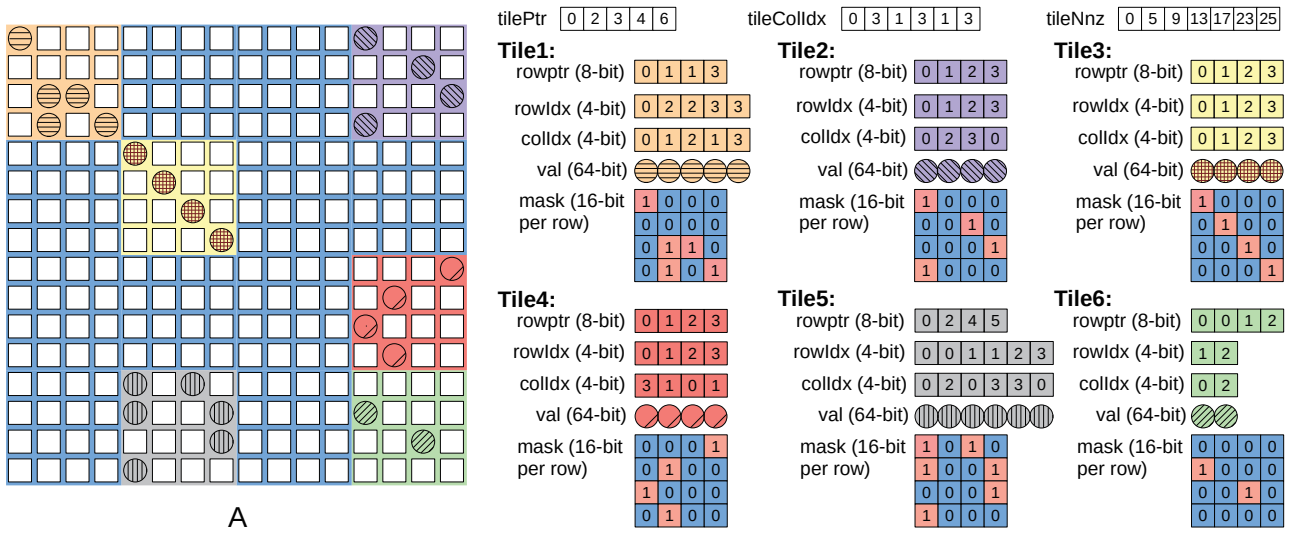

Figure 2. An example matrix A of size 16-by-16 stored in six sparse tiles of size 4-by-4. The tile structure includes three arrays `tilePtr`, `tileColIdx` and `tileNnz` representing the memory offsets of tiles, tile column indices and the offsets for the number of nonzeros in sparse tiles. Meanwhile, each sparse tile consists of five arrays: `rowPtr`, `rowIdx`, `colIdx`, `val` and `mask`.

memory for C, and (3) calculating the row indices, column indices and values of the nonzeros in each tile of C. Besides, several optimization techniques, such as binary search for set intersection, bit mask operations for symbolic SpGEMM, and an adaptive method for selecting sparse or dense accumulator in on-chip memory, are also developed to improve efficiency. Section 3.3 will detail our three-step TileSpGEMM algorithm.

3.2 Sparse Tile Data Structure

The sparse tile data structure stores two levels of tile information. The higher level stores the tile structure of the matrix and consists of three arrays: (1) `tilePtr` of size $tilemA + 1$, where $tilemA$ is the number of tile rows of the matrix, used to store memory offsets of the tiles in tile rows, (2) `tileColidx` of size $numtileA$, where $numtileA$ is the number of sparse tiles, used to store tile column indices, and (3) `tileNnz` of size $numtileA + 1$, used to save the memory offsets for the numbers of nonzeros in the sparse tiles.

On the lower level, the nonzeros in each tile are stored in the CSR style plus row indices and bit masks. We create four arrays to store them: (1) `val`: stores values of all the nonzeros in tile order of size $nnzA$, where $nnzA$ is the number of nonzeros in A, (2) `rowIdx` and `colIdx`: with the size of $nnzA$ as well, stores the row and column indices of each nonzero in the tile. Note that our tile size is always set to 16-by-16 for maximize space utilization, since the row or column index in one tile only needs four bits and can be together stored within an 8-bit unsigned char, (3) row pointer array `rowPtr` saves 16 memory offsets for the nonzeros in the tile, which is different to the normal CSR row pointer array including 16+1 entries. The reason of only using 16 entries is that we want to limit the value in the pointer array to 0–255, which can also be saved in an 8-bit unsigned char. Even though we do not store the 17th entry of the standard pointer array, the value can be extracted from subtracting the last item of `rowPtr` from the corresponding value in `tileNnz` (storing the number of nonzeros of the tiles) if needed, (4) in addition, we set a `mask` array for each tile, since regular bit masks can accelerate symbolic phase and in general only occupies very small space. Here we create a 16-bit unsigned short array of size $numtileA \times 16$ to store the bit masks of sparse tiles. Specifically, we mark 1 in the corresponding column index position for the nonzeros of each row, and mark 0 otherwise. This combined format facilitates the nonzeros in a certain tile of matrix A to be able to access corresponding tile of matrix B and its bit mask at the same time. A small example using 4-by-4 tile can be found in Figure 2.

Note that the reason of setting the tile size to 16-by-16 is that fully utilizing the 8-bit unsigned char for storing indices and pointers and 16-bit unsigned short for bit masks. Compared to 16-by-16 used in this work, other tile sizes (such as 4-by-4 and 8-by-8) cannot saturate 8-bit data type and will bring more complex data packing and unpacking.

3.3 Algorithm Description

The TileSpGEMM method consists of three steps to determine three groups of information of C: (1) overall tile structure arrays `tilePtr` and `tileColidx`, (2) the number of nonzeros array `tileNnz`, row pointer array `rowPtr` and bit mask array `mask` of each sparse tile, and (3) index and value arrays `rowidx`, `colidx` and `val` of nonzeros in each tile.

In the first step, to find the possibly non-empty tiles and to get the sparse tile structure of C, we run an SpGEMM to multiply the high level tile structure of the two input matrices A and B also stored in our tile format. Here we use A' and B' to represent the high level tile layout of A and B. Now the numbers of rows or columns of A' and B' are the numbers of tile rows or tile columns of A and B, and the numbers of nonzeros of the new matrices are the numbers of non-empty tiles of the original matrices. Note that only sparsity structures of matrices A' and B' are needed here. After performing a normal symbolic SpGEMM $C' = A'B'$, the nonzero structure of the resulting matrix C' is generated as the tile structure of C. Figure 3 plots an example of $C' = A'B'$. It can be seen that matrix A' of eight nonzeros multiplies B' of six nonzeros and gets C' of ten nonzeros, which also means that A of eight sparse tiles multiplies B of six sparse tiles and obtains C of ten sparse tiles.

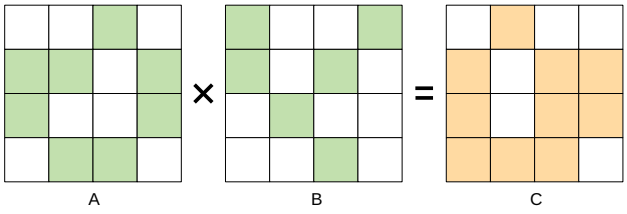

Figure 3. An example of the first step of TileSpGEMM. The step determines the tile structure of C, and each sparse tile is now treated as a nonzero element in a symbolic SpGEMM. The nonzeros in the three matrices represent sparse tiles.

Since the numbers of rows and columns of A' and B' are a proportion of the original numbers of rows and columns of A and B, and the numbers of their sparse tiles are in general much less than the numbers of nonzeros in A and B, the amount of floating point computations and execution time of the first step often only take a low percentage of the entire time of a full SpGEMM. According to our experimental data in Figure 10, the first step normally takes no more than 5% of the total execution time of TileSpGEMM. For simplicity, we directly call the SpGEMM function in the NSPARSE library [93] in this step. The reason of using NSPARSE is that compared to other existing open-source SpGEMM libraries, NSPARSE provides better performance for small cases and a simpler interface to use.

It should also be noticed that the multiplication does not consider tile-wise cancellation, since this stage does not know the number of nonzeros in each tile of C, and could not remove any tile though it is actually empty. In other words, the final C is allowed to store empty tiles.

After this step, the tile structure of C (e.g., the number of sparse tiles *numtileC*, the tile pointer array `tilePtr` and the tile column index array `tileColidx`) has been obtained.

In the second step, on top of the already known sparse tile structure of C, we generate the row pointer array, bit mask array and the number of nonzeros of each tile. Then we can allocate memory space of C in the end of this step.

Here we use C_{ij} to denote the sparse tile in C's ith tile row and jth tile column, and calculate it by multiplying the corresponding sparse tiles A_{ik} in the ith tile row of A and the sparse tiles B_{kj} in the jth tile column of B in a symbolic way. Because empty tiles in A and B should not be involved in the computation, we need to find and match the non-empty tiles with the same indices in the tile rows of A and tile columns of B. This procedure is actually equivalent to a set intersection operation of sparse tiles that appear in both sets (i.e., tile rows of A and tile columns of B).

Algorithm 2 A pseudocode of the 2nd step of TileSpGEMM.

```
1: for i = 0 to numtileC in parallel do
2:     tile_i ← tileRowidx_C[i]
3:     tile_j ← tileColidx_C[i]
4:     lena = tilePtr_A[tile_i + 1] - tilePtr_A[tile_i]
5:     lenb = tilePtr_B[tile_j + 1] - tilePtr_B[tile_j]
6:     if lena < lenb then                    ▷ Binary Search to get the intersection
7:         idx ← tileColPtr_B[tile_j]
8:         pos ← 0
9:         for each value in tileColidx_A[tilePtr_A[tile_i] do
10:            if res =Binary-Search(&tileRowidx_B[idx], value) exists then
11:                atomicAdd(pos, 0 )
12:                matched_posA[pos] = value_index
13:                matched_posB[pos] = res
14:            end if
15:        end for
16:    else
17:        Search tilerowidx_B from tilecolidx_A
18:    end if
19:    for each matched tiles A and B do      ▷ AtomicOr operation to generate the
   maskc
20:        get nnza of A
21:        for k = 0 to nnza do
22:            get maskb and maskc
23:            AtomicOr(maskc, maskb)
24:        end for
25:    end for
26: end for
```

Algorithm 2 shows a pseudocode of the 2nd step of TileSpGEMM, and Figure 4 plots an example of generating two sparse tiles of C. As can be seen in the figure, the tile C_{12} is computed by the sum of products of three tiles in the first tile row of A and two tiles in the second tile column of B. From the basic storage structure, we can extract two arrays `tilecolidx_A`$_{1*}$, including the column indices of the tile row and `tilerowidx_B`$_{*2}$, including the row indices of the tile column. Then we need to find the set intersection of the two index arrays, and multiply the tiles at the corresponding positions. We here assume that the column/row indices in the two arrays are ordered and can use a serial merge-like method by setting two pointers and traversing the two arrays until all the matching tile-pairs are found. However, we in our experiments find that the merging primitive is often slower than binary search approach for set intersection. Specifically, when the two arrays have different sizes, we let one CUDA thread to search each element in the shorter array on the longer array with a typical binary search operation (lines 6-18 in Algorithm 2) to find the tiles matched.

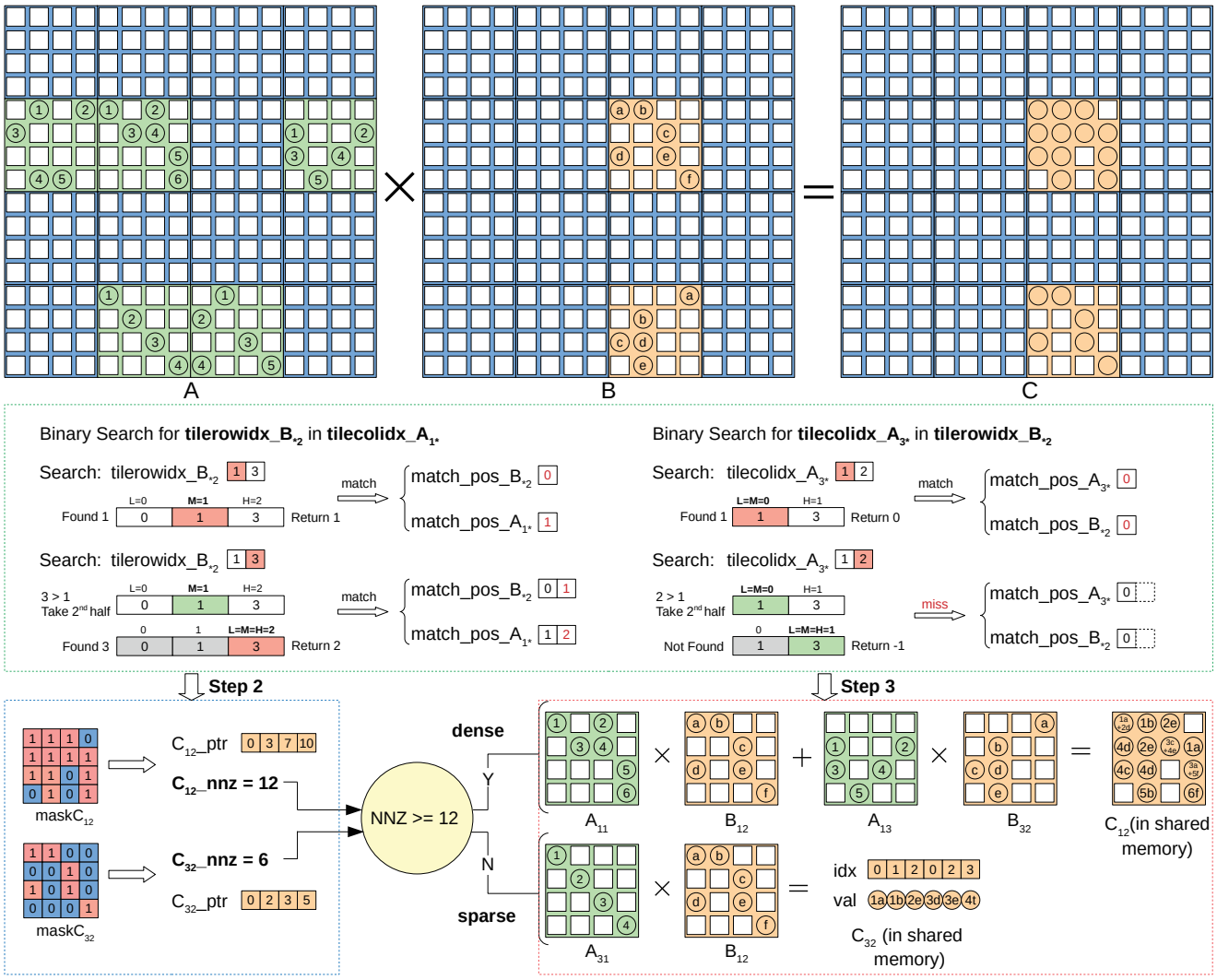

Figure 4. An example of the 2nd and 3rd steps of the TileSpGEMM algorithm multiplying two 16-by-16 sparse matrices stored as sparse tiles of size 4-by-4. In the 2nd step, a binary search approach applied to set intersection gathers sparse tiles needed from A and B. The bit mask operations for symbolic SpGEMM determine the number of nonzeros and the row pointer array of each tile in C. After allocating the resulting matrix, in the 3rd step, an adaptive method for selecting sparse or dense accumulator in on-chip memory optimizes the process of calculating values of the nonzeros in each tile of C. Here a dense accumulator is used for C_{12}, and a sparse accumulator is adopted for C_{32}. Through the two steps and the three optimization techniques, our tiled algorithm speeds up SpGEMM operation. It is worth to note that the threshold of selecting dense or sparse accumulator is 192 for tiles of 16-by-16 in this work, but not this small number 12 for demonstration purpose.

In each search step, we in the on-chip scratchpad memory set two index arrays `match_pos_A`$_{1*}$ and `match_pos_B`$_{*2}$ to record the index positions of intersecting elements respectively (lines 12-13 and the middle part of Figure 4). Since both index arrays are ordered, when one search is completed, the next search range will be narrowed. Specifically, the left bound will be set to the next index of the last matched position, while the right bound is still the final index of the array. For example, in the left part of the middle of Figure 4, the search range of '3' starts at position 3 of `tilecolidx_A`$_{1*}$ because the last element '1' is matched at position 2. After the last element '3' is matched successfully, the intersecting tile pairs that need to be multiplied for generating resulting tile C_{12} have been obtained. It can be seen that tile C_{12} should be calculated by $A_{11} \times B_{12} + A_{13} \times B_{32}$. Similar to C_{12}, the tile C_{32} is calculated through this way.

After completing the set intersection of the tiles in A and B for a certain tile in C, we need to generate its row pointer array and the number of nonzeros for allocating the complete structure of the final C. For speeding up the process,

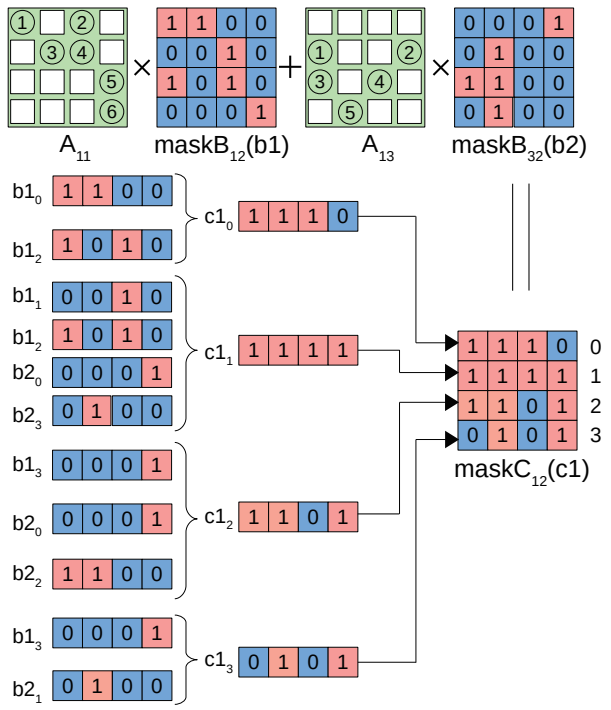

Figure 5. An example of computing bit masks of tiles C_{12}. Each element in tiles of B and C has a bit mask storing 0 or 1 to indicate whether it is a nonzero. For instance, computing C_{12} requires that A_{11} and A_{13} and their corresponding bit masks in B are involved. Here the bit mask `1110` of the first row $c1_0$ in C_{12} is computed from using `AtomicOr` on `1100` ($b1_0$) and `1010` ($b1_2$). When the traversal is complete, the bit mask provides nonzero structure of the tile of C.

we stored bit mask of each tile (recall Section 3.2) to reduce memory transfer cost of repeatedly loading structure information of tiles in B. Similar to the bit masks in A and B, we set a $\texttt{mask\_C}_{ij}$ array to mark the position of nonzeros calculated for tile C_{ij} in C. For each pair of matched tiles A_{ik} and B_{kj}, we load B_{kj}'s bit masks into on-chip memory, traverse all the nonzeros of A_{ik}, and regard the column index of each nonzero as the row index of the bit mask of B_{kj}. Then we in the scratchpad memory use the `AtomicOr` operation for all the matched row masks for getting the result row mask of C_{ij} (lines 19-25). Finally, when all the row masks in $\texttt{mask\_C}_{ij}$ are generated, the row pointer array of C_{ij} is easily calculated by summing 1s in the mask and computing a prefix-sum scan.

In Figure 5, we show an example about how to generate the $\texttt{mask\_C}_{12}$ and determine the number of nonzeros of tile C_{12} (c1). As can be seen, C_{12} is calculated by two sets of matched tiles (A_{11}, B_{12}) and (A_{13}, B_{32}). For the multiplication of A_{11} and B_{12}, we firstly traverse nonzeros of each row in A_{11}. For a_{00} with the column index 0, we can find the first row mask `1100` ($b1_0$) from B_{12} (b1). Then the second entry a_{02} will extract the third row mask `1010` ($b1_2$) since the column index of a_{02} is 2. Next we continue to multiply A_{13} and B_{32} in the same way. After the matching operation, `AtomicOr` operations work for the two matched row masks $b1_0$ and $b1_2$, and the first row mask `1110` ($c1_0$) of C_{12} is finally updated. Similarly, $c1_1$ is generated by four row masks from $b1_1$ and $b1_2$ in tile B_{12}, and $b2_0$ and $b2_3$ in tile B_{32}. The procedure will be repeated until the last row mask $c1_3$ is completed as Figure 5 shows.

Also note that in this step, all the memory requirements are well bounded to no larger than 256 nonzeros and can be completed in on-chip memory. Thus we do not allocate any intermediate array on global memory and save overall space overhead.

So far, we have calculated the number of nonzeros, row pointer array `rowPtr` and bit mask array `mask` of each tile, as well as the total number of nonzeros *nnzC* of the final C. Now we can prepare memory for the `val` and `idx` arrays of size *nnzC* for the third step.

In the third step, a numeric phase is performed to calculate the values and row/column indices of nonzeros in tiles of C. Besides the binary search operations used in the 2nd step, we in this step propose an adaptive method for selecting a sparse or dense accumulator in on-chip memory for each tile to improve performance.

Algorithm 3 A pseudocode of the 3rd step of TileSpGEMM.

```
1:  for i = 0 to numtileC in parallel do
2:      j ← tileRowIdx_C[i]
3:      l ← tileColIdx_C[i]
4:      if the number of tile C_jl <= tnnz then          ▷ sparse accumulator
5:          get ColIdx_C from maskC_jl
6:          for each matched tiles A_jk and B_kl do
7:              for p = 0 to the number of nonzeros of C_jl do
8:                  idxc ← ColIdx_C[p]
9:                  val ← products of two values in A_jk and B_kl
10:                 AtomicAdd(Val_C[idxc], val)
11:             end for
12:         end for
13:     else                                             ▷ dense accumulator
14:         for each matched tiles A_jk and B_jl do
15:             load A_jk and B_kl
16:             AtomicAdd(Val_C, A_jk × B_kl )
17:         end for
18:     end if
19: end for
```

We set a threshold *tnnz* for adaptively selecting a sparse accumulator working on a sparse tile, or a dense one firstly working on a dense full matrix and then saving the result into its sparse form. The reason of using the selection is that in our observation when *tnnz* is larger than 75% of the tile size (i.e., 256 nonzeros), working on dense space often bring better performance. In contrast, for the tiles with less nonzeros, a sparse accumulator is in general faster.

Here the bit masks of the sparse tiles in C obtained from the 2nd step are used to determine the column index of all nonzeros of a tile. In this way, the intermediate products can be directly accumulated at a certain position according to

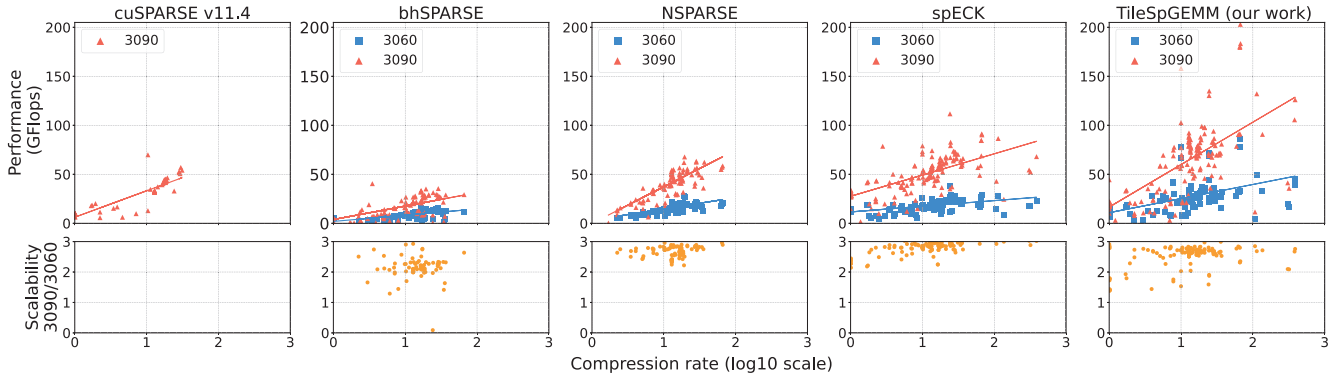

Figure 6. Performance comparison of TileSpGEMM and other four state-of-the-art SpGEMM methods on Geforce RTX 3090 and 3060 GPUs. The five sub-figures on top show double precision performance (in GFlops) of $C = A^2$ and $C = AA^T$, and their linear regression, and the five sub-figures on the bottom show the scalability of the methods on the two GPUs. The x-axis is the compression rate in $\log_{10}$ scale.

the column index instead of allocating a temporary space for them. Also, the `AtomicAdd` operation is used in corresponding position according to the column index to get the resulting values. As shown in figure 4, the tile C_{32} is calculated by a sparse accumulator since the number of nonzeros in it (i.e., 6) is less than 75% of its size (i.e., 16×75%=12).

For the tiles whose number of nonzeros is larger than *tnnz*, the dense accumulator will run on shared memory. Here we directly allocate a dense matrix of tile size (i.e., 256) in on-chip memory and add intermediate products onto it. Note that since the tiles are small enough, TileSpGEMM does not allocated global memory at all for saving space. For example, the tile C_{12} in Figure 4 is calculated by a dense accumulator, since it has 12 nonzeros.

For the implementation of the 2nd and 3rd steps, we assign one warp of 32 CUDA threads for processing one sparse tile on GPUs for saving thread synchronization costs.

After accumulating and saving all sparse tiles of C onto the global memory, the TileSpGEMM algorithm is completed, and all information of a matrix in sparse tile form is obtained.

4 Experimental Results

4.1 Experimental Setup

We use two NVIDIA Ampere GPUs installed in an Ubuntu 18.04 machine as the testbed. The CUDA and GPU driver versions are 11.4 and 470.57.02, respectively. We compare our TileSpGEMM with four state-of-the-art SpGEMM methods cuSPARSE v11.4 [31], bhSPARSE [80, 82], NSPARSE [93] and spECK [96] in double precision, and with tSparse [118] optimized for using GPU tensor cores in half precision. Table 1 lists some specifications of the setup.

As for the dataset, to saturate modern GPUs, we test all 142 sparse square matrices in the SuiteSparse Matrix Collection [38] requiring no less than one billion floating point operations when computing both $C = A^2$ and $C = AA^T$. We also evaluate 18 representative sparse matrices in Table 2 for a more in-depth performance comparison. Note that the first 12 matrices are firstly tested by Williams et al. [105] and are the classic dataset in much subsequent sparse matrix research. As for the comparison with tSparse, we use the 16-matrix dataset in its original paper [118]. The reason is that the dataset may best utilize the half precision tensor cores on GPUs.

Table 1. The two GPUs and six algorithms evaluated.

Two NVIDIA GPUs	Six algorithms
(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.	(1) cuSPARSE v11.4 [31], (2) bhSPARSE [80, 82], (3) NSPARSE [93], (4) spECK [96], (5) tSparse [118], (6) TileSpGEMM (this work).

4.2 Performance Comparison over Existing Work

Figure 6 shows the performance comparison of running double precision $C = A^2$ and $C = AA^T$ on the 142 square matrices with cuSPARSE v11.4 [31], bhSPARSE [80], NSPARSE [93], spECK [96] and our TileSpGEMM on the RTX 3060 and 3090 GPUs.

As can be seen in the top five sub-figures of Figure 6, the vendor supported cuSPARSE v11.4, bhSPARSE and NSPARSE can only complete a portion of matrices in the dataset, while spECK and our TileSpGEMM can finish all 142 matrices. In general, our TileSpGEMM runs faster than other four methods on 139, 138, 127 and 94 matrices respectively on RTX 3090, and on 142, 128, 114 and 92 matrices on RTX 3060 (note that no matrix in our benchmark can be computed with cuSPARSE on RTX 3060).

Besides, our TileSpGEMM has obvious performance advantage over cuSPARSE, bhSPARSE, NSPARSE and spECK.

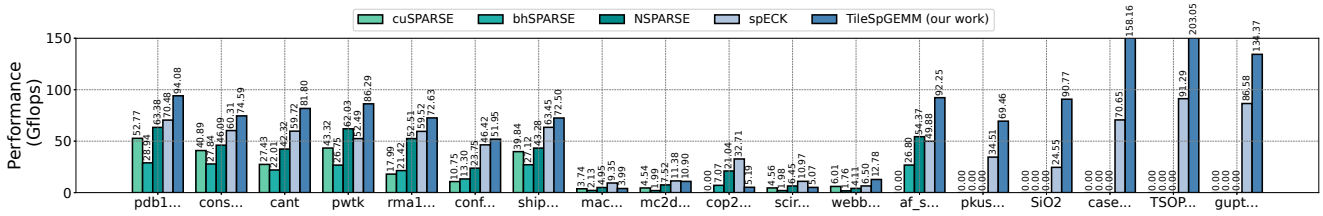

Figure 7. Performance comparison of performing double precision A^2 operation of the 18 representative matrices on Geforce RTX 3090 GPU. The '0.00' on bar areas indicate the corresponding algorithm fails to perform its SpGEMM operation on the matrix.

Table 2. Information of the 18 representative matrices. The compression rate is the ratio of the number of intermediate products (i.e., half of the number of floating point operations) of $C = A^2$ to the number of nonzeros in C.

Matrix A	$n(A)$	$nnz(A)$	$\#flops$ of $C = A^2$	$nnz(C)$	Compression rate
pdb1HYS	36K	4.3M	1.1B	19.6M	28.34
consph	83K	6.0M	927.7M	26.5M	17.48
cant	62K	4.0M	539.0M	17.4M	15.45
pwtk	218K	11.6M	1.3B	32.8M	19.10
rma10	47K	2.4M	313.0M	7.9M	19.81
conf5_4-8x8-05	49K	1.9M	149.5M	10.9M	6.85
shipsec1	140K	7.8M	901.3M	24.1M	18.71
mac_econ_fwd500	206K	1.3M	15.1M	6.7M	1.13
mc2depi	525K	2.1M	16.8M	5.2M	1.60
cop20k_A	121K	2.6M	159.8M	18.7M	4.27
scircuit	170K	1.0M	17.4M	5.2M	1.66
webbase-1M	1M	3.1M	139.0M	51.1M	1.36
af_shell10	1.5M	52.7M	3.68B	142.7M	12.90
pkustk12	94K	7.5M	5.4B	474.8M	5.65
SiO2	155K	11.3M	28.5B	104.8M	136.03
case39	40K	1.0M	8.1B	404.7M	10.00
TSOPF_FS_b300_c2	56K	8.8M	107.9B	805.7M	66.96
gupta3	17K	9.3M	61.4B	270.9M	113.40

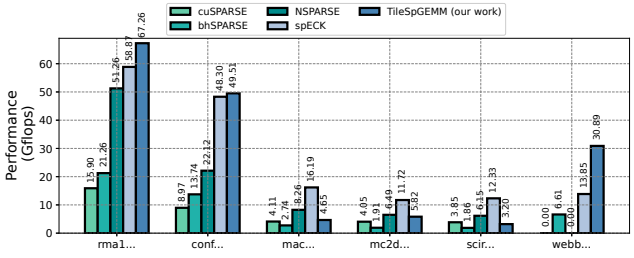

Figure 8. Performance comparison of performing double precision AA^T operation on six asymmetric matrices in the 18 representative matrices on the Geforce RTX 3090 GPU.

First of all, it is worth noting that the peak performance of our algorithm on RTX 3090 GPU can reach up to 203.05 Gflops (at the matrix 'TSOPF_FS_b300_c2'), while the peak of spECK for it is 91.29 Gflops (cuSPARSE, bhSPARSE and NSPARSE cannot calculate it). Also, the overall SpGEMM performance is largely improved. Specifically, the average SpGEMM performance of the five SpGEMM methods are 30.82, 11.54, 37.73, 46.92 and 54.56 Gflops on RTX 3090, respectively, meaning that our TileSpGEMM is on average 1.77x, 4.73x, 1.45x, 1.16x (geometric mean) faster than the other four methods. As for the maximum speedups on RTX 3060, TileSpGEMM achieves up to 2.78x, 145.35x, 97.86x and 3.70x speedups over cuSPARSE, bhSPARSE, NSPARSE and spECK, respectively. On RTX 3060, the performance trend is similar. From the linear regression lines, it can be seen that our TileSpGEMM brings promising performance trend, and may deliver relatively higher SpGEMM throughput for matrices with larger compression rate.

Moreover, the remaining five sub-figures on the bottom show the scalability of RTX 3090 over RTX 3060. Considering the peak computational power and bandwidth of RTX 3090 are around 3x higher over RTX 3060. We expect the SpGEMM methods tested can scale proportionally. As can be seen, bhSPARSE, NSPARSE, spECK and our TileSpGEMM achieve on average 2.12x, 2.66x, 2.82x and 2.53x speedups on RTX 3090 over RTX 3060. The reason of that our scalability is a bit lower than NSPARSE and spECK is because some matrices using our method has more arrays to allocate on GPUs and takes longer time on memory allocation (see Figure 10).

To conduct a more detailed comparison and analysis, we list the 18 representative matrices (see Table 2) and plot their performance bars when computing $C = A^2$ and $C = AA^T$ using the five SpGEMM methods on RTX 3090 GPU. As can be seen in Figure 7, our TileSpGEMM running $C = A^2$ performs in general better than the other four methods on most of the commonly used matrices. In particular, for matrices that fail to be calculated by certain methods due to excessive memory capacity such as 'TSOPF_FS_b300_c2' and 'gupta3', our method naturally avoids the allocation of temporary intermediate products (recall that sparse tiles are calculated in the on-chip shared memory) and reflects obvious advantage. Compared with spECK that completes these two matrices successfully, our method still gets 2.20x, 1.53x speedups, respectively. Furthermore, the matrix 'webbase-1M' with serious load imbalance problem achieves over 12

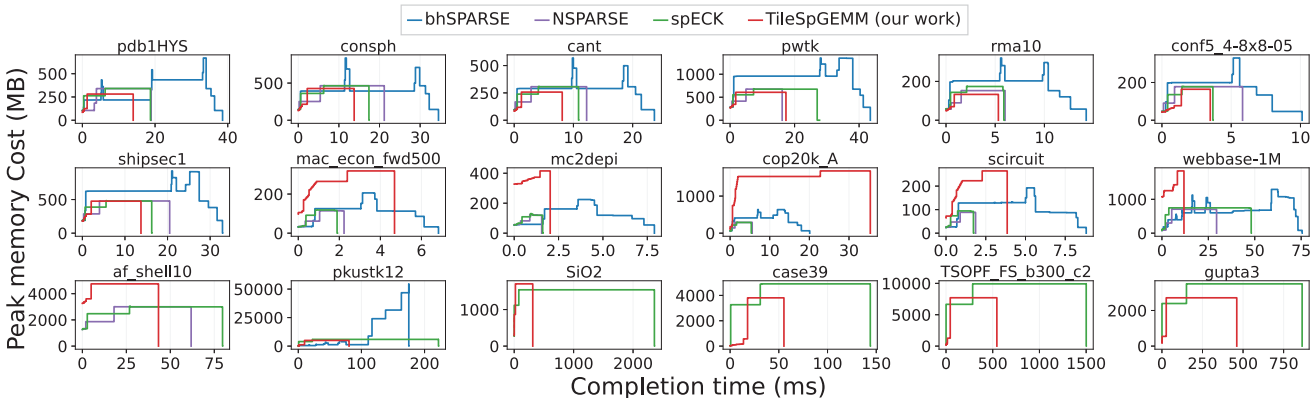

Figure 9. Runtime peak space cost of running $C = A^2$ on 18 representative matrices. The x-axis shows completion time in milliseconds, and the y-axis shows peak memory cost in megabyte.

Gflops performance with our TileSpGEMM, which is a great improvement in recent years.

Figure 8 shows the performance of AA^T operation on the six asymmetric matrices in the 18 representative matrices. As can be seen, the efficiency of our method becomes even more promising when computing AA^T. In particular, the performance of 'webbase-1M' achieves 30.89 GFlops, while cuSPARSE and NSPARSE both fail due to out of memory, and bhSPARSE and spECK give 6.61 and 13.85 GFlops, respectively.

However, for some matrices like 'cop20k_A' and 'scircuit', our algorithm needs more time to complete, because their matrix structures are too sparse and each tile only has little computations. Taking 'cop20k_A' as an example, most tiles in it are very sparse (specifically, 18,705,069 nonzeros distributed in 15,900,566 tiles), which makes the second step of our method (generating and allocating tile structure) dominates overall runtime.

4.3 Comparison of Peak Space Cost at Runtime

Figure 9 plots the memory consumption process of the TileSpGEMM and the other three algorithms (cuSPARSE is a closed source library, thus is not compared here) running $C = A^2$ on the 18 matrices in table 2. As can be seen, in most matrices, bhSPARSE uses the most space, and NSPARSE and spECK are comparable. Our TileSpGEMM often uses less space than the three methods and finishes earlier. For example, for the matrix 'cant', TileSpGEMM uses up to 257 MB space, which is 16%, 16% and 55% less than bhSPARSE, NSPARSE and spECK, respectively. However, for some very sparse matrices like 'cop20k_A', our method allocates a large amount of sparse tiles for local row pointer and bit masks, and thus uses much longer time to allocate and compute. This also reflects the degraded SpGEMM performance shown in Figures 7 and 8.

4.4 Runtime Breakdown of TileSpGEMM Algorithm

Figure 10 shows the runtime breakdown of the TileSpGEMM algorithm. Three algorithm steps and all memory allocation on CPU and GPU at runtime are included. As can be seen, the first step (in red) in general takes less than 5% of the whole runtime. Steps 2 (in green) and 3 (in yellow) account for on average 15% and 70% of the entire time, respectively, and thus are carefully optimized. Besides, memory allocation (in blue) in some cases occupies on average around 20% time, and this also aligns the observation of Gelado and Garland [50].

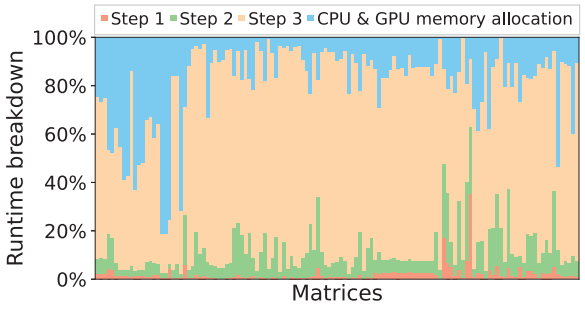

Figure 10. The runtime breakdown of TileSpGEMM.

4.5 Comparison of Space Overhead of Tiled Format

We compare the space cost of our TileSpGEMM with standard CSR, as well as CSB-M and CSB-I designed by Buluç et al. [21, 27] in the Combinatorial BLAS [26]. As shown in Figure 11, our tiled data structure on average takes 31.28 MB less space than CSR, but uses 113.43 and 82.09 MB more space than CSB-M and CSB-I, respectively. The main reason is that our format stores local row pointer (16 unsigned chars) and bit masks (16 unsigned shorts) for each tile. However, it is still worth to save the extra memory space, since it in general makes SpGEMM operation more convenient and faster.

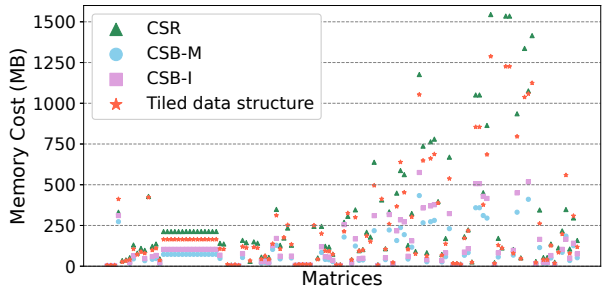

Figure 11. Space cost comparison of our tiled sparse structure, standard CSR and two CSB formats of matrices tested.

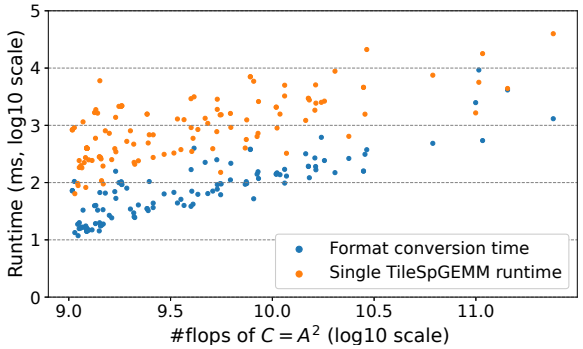

Figure 12. Comparison of the CSR to tiled format conversion time and runtime of a single TileSpGEMM.

4.6 Format Conversion Overhead

We record the overhead of converting a CSR matrix to our tiled data structure, and the comparison of the conversion time and SpGEMM execution time on RTX 3090 GPU is plotted in Figure 12. As can be seen, our conversion time in general does not take longer time than ten single SpGEMM operations. This conversion cost would not affect the efficiency of SpGEMM, since we always assume the matrix is already stored in the tiled format before running TileSpGEMM. This assumption is pretty reasonable in many applications, such as algebraic multigrid (AMG) solvers using the output matrices from an SpGEMM as the input of another SpGEMM in the next round.

4.7 Comparison over tSparse Using Tensor Cores

tSparse [118] is a relatively new SpGEMM work designed for using tensor cores on modern GPUs. It also saves a sparse matrix in tile form, but uses hardware accelerated dense GEMM on tensor cores to multiple tiles, and finally converts the dense resulting tile to its sparse form. Since its open source code currently only supports half precision input and single precision output, we use the same precision setup in our TileSpGEMM for a more intuitive performance comparison.

Figure 13 shows the performance comparison of $C = A^2$ on the 16 sparse matrices listed in the tSparse paper [118] on RTX 3090 GPU. As can be seen, the TileSpGEMM outperforms tSparse on all of the 16 matrices. The average (geometric mean) speedup is 1.98x, and the maximum speedup is 4.04x. To conduct a more comprehensive analysis, we plot the runtime breakdown of the 16 matrices in Figure 14. As can be seen, the 'memory allocation' phase of tSparse takes larger proportion of overall time on many matrices. This is because the memory allocation of C needs to be resized repeatedly during the execution. In addition, for the matrices whose tiles are very sparse (such as 'webbase-1M' and 'cage12'), our TileSpGEMM utilizing sparsity structures of tiles takes much less time on steps 2 and 3 than tSparse. Therefore, although using dense multiplication on hardware accelerated tensor cores can be much faster than normal CUDA cores, recasting sparse tiles into dense ones for using the technique in general wastes sparsity, and thus may be still less efficient than sparse multiplication used in the TileSpGEMM.

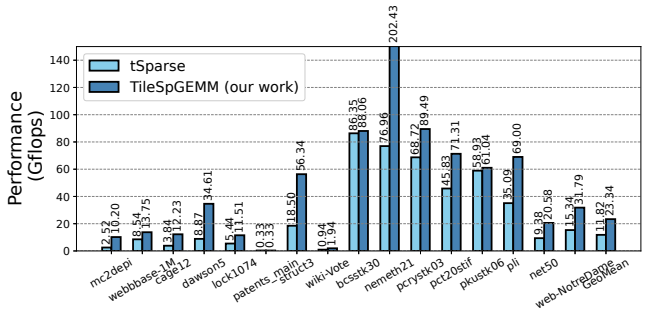

Figure 13. Performance Comparison of tSparse (using half precision tensor cores) and TileSpGEMM (also in half precision) by using the 16-matrix dataset on RTX 3090 GPU.

5 Related Work

Optimizing SpGEMM received much attention in recent years. The first performance issue to resolve is **load imbalance** in the row-row approaches. Some of the existing SpGEMM implementations inspect the number of intermediate products of the rows and group the rows of the similar amount of computations together to balance the load. Liu and Vinter [80, 82] divided the rows into 38 bins and selected different methods for each bin group. Anh et al. [3] used balanced hash methods. Nagasaka et al. [92, 93] used two rounds of binning for symbolic and numeric stages, respectively. Winter et al. [106] emphasized the effectiveness of long rows, and Parger et al. [96] recently developed the spECK library with a light-weight preprocessing method for balancing. Lee et al. [75] developed a block reorganizer to

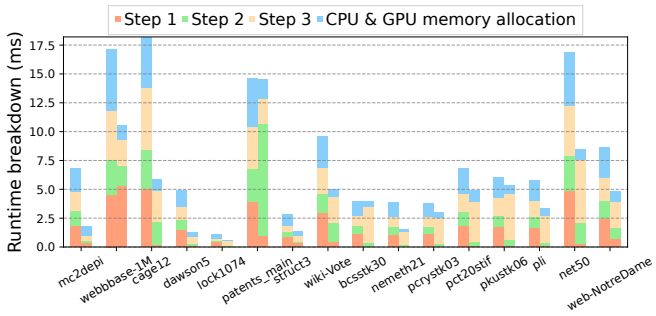

Figure 14. Comparison of runtime breakdown of tSparse and TileSpGEMM both in half precision on the 16 matrices. The bars on the left represent tSparse, and the right ones are TileSpGEMM.

split and gather the computation of blocks in parallel. Compared with those load balancing methods, the basic working unit in our TileSpGEMM is one sparse tile that takes much smaller memory space and in general does not bring any noticed load imbalance.

In addition, **allocating temporary memory space for intermediate products** is the second performance issue to address. For matrices with long rows, this inherent problem of the row-row method can seriously impact performance. Liu and Vinter [80] developed a progressive method to gradually allocate long rows, but suffered from possibly frequent memory copies. Nagasaka et al. [92, 93] allocated enough large space, but their hash operations can be very inefficient on global memory. Recently, the spECK library by Parger et al. [96] gave an effective hybrid implementation for long rows. But the performance degradation is also obvious in cases of high density. In contrast, our TileSpGEMM proposed in this paper does not allocate any temporary space on global memory, since the single sparse tile is small enough to store in on-chip scratchpad memory for even faster performance.

Moreover, the third performance issue on **sparse accumulator (SPA)** received more attention. The earliest SPA method proposed by Gilbert et al. [51] used a dense row to accumulate the nonzeros and then converts them into sparse form. When the number of nonzeros in a row is too few, more "sparse" ways in general bring better performance. One method is the so-called expansion, sorting and compression (ESC) proposed by Bell et al. [14] and is implemented in the CUSP library [33]. To better use GPUs, Dalton et al. further implemented a finer grained ESC variant [34] and used more efficient sorting primitives [32]. Kunchum et al. [74] later developed a multi-lock ESC approach. Another form of SPA uses heap (i.e., priority queue) data structure on CPUs by Azad et al. [4] and on GPUs by Liu and Vinter [80]. When the nonzeros in rows of B are already sorted, the merging methods have been used by Liu and Vinter [80], Dalton et al. [32], Gremse et al. [53, 54] and Ji et al. [66]. Compared to the sorting and merging of $O(n \log n)$ complexity, hash methods of $O(1)$ complexity are also used by Anh et al. [3], Nagasaka et al. [92, 93] and Deveci et al. [42, 43]. Liu et al. [78, 79] recently compared the above SPA methods on GPUs and optimized them by better using registers. Besides the classic SPA, Gu et al. [55] developed a new method using outer product and propagation blocking technique for accumulating nonzeros. Compared with those row-oriented SPA techniques, our TileSpGEMM deals with sparse tiles of well-bounded size, and binary search and dense indexing have been found to be very efficient for accumulating nonzeros.

Besides the above major performance considerations, there have been **several groups of work related to SpGEMM**. Yuster and Zwick [117] developed an algorithm using lower algebraic operations. The sparse matrix-dense matrix multiplication [58, 72, 112] and sparse matrix-sparse vector multiplication [5], are also interesting directions in sparse BLAS, and a series of work in the framework of GraphBLAS [25, 36, 67, 86–88, 111, 113] bring SpGEMM a wide range of graph applications. Also, Xie et al. [109, 110] used deep learning methods for selecting a better storage format for SpGEMM. Pal et al. [95], Zhang et al. [119] and Wang et al. [103] developed hardware accelerators for SpGMM, and Chen et al. [29] developed an SpGEMM method for Sunway processors. Willcock and Lumsdaine [104] proposed methods for further compressing sparse matrices. Knight et al. [71] developed methods for sparse multiplication with low-rank features. Some segmented primitives such as segmented sort and segmented merge are used for accelerating SpGEMM on GPUs [60, 66]. Moreover, compilation techniques have been proven effective for a few important sparse computations [30, 69, 70, 73, 91, 98].

As for **distributed SpGEMM**, Buluç and Gilbert [22–24] for the first time proposed blocking formulations and hypersparse formats. Ballard et al. studied the communication cost for distributed linear algebra [10, 11], and Azad et al. [4] considered multi-level parallelism for distributed SpGEMM. The hypergraph partitioning techniques has been used in shared memory platforms by Akbudak et al. [1, 2, 39] and distributed environments by Ballard et al. [12, 13]. Hussain et al. [62] developed a distributed-memory SpGEMM for extremely large matrices. Xia et al. [108] scaled SpGEMM computation by developing a distribution strategy. Azad et al. [8] improved Combinatorial BLAS and minimized the communication of distributed SpGEMM. Compared to the methods designed for shared memory processors, the data structure of our TileSpGEMM is more like the distributed blocking SpGEMM methods, but optimized for GPUs without concerns on communication costs.

Besides, **tiling techniques** have been well studied for linear algebra operations [19, 85], and tuning small dense tile size is a main method for obtaining higher performance [17, 77, 116]. There also have been a few studies about storing dense small blocks in sparse matrices. The Sparsity library by

Im et al. [64, 65], the OSKI framework by Vuduc et al. [41, 99–101] and the compressed sparse block formats by Buluç et al. [21, 27] are representatives in this direction. Wang et al. [102] and Lu et al. [84] developed tiled methods for sparse triangular solve. Niu et al. [94] recently proposed TileSpMV, a tiled algorithm for optimizing sparse matrix-vector multiplication on GPUs. Hong et al. [59] reordered nonzeros in sparse tiles for sparse matrix-multiple vector multiplication and sampled GEMM. Li et al. [76] proposed a blocked format named HiCOO and various compute kernels for accelerating high dimensional tensors on modern processors. Zachariadis et al. [118] designed the tSparse library to better utilize the GPU tensor cores of 16-bit floating point precision for optimized SpGEMM, which can be applied efficiently in deep learning with best precision demands. Differently from those methods, our method always stores tiles in sparse forms and focuses on SpGEMM operation maintaining sparsity in all three input and output matrices.

6 Conclusion

We in this paper have proposed TileSpGEMM, a tiled algorithm for parallel SpGEMM on GPUs. Our method resolved the three major performance issues, and designed highly efficient data structures for tiled storage and a three-step SpGEMM algorithm for modern GPU architectures. The experimental results on two newest NVIDIA Ampere GPUs show that our TileSpGEMM brought significant speedups over state-of-the-art SpGEMM work, gave good scalability, and saved much memory space at runtime.

Acknowledgement

We deeply appreciate the invaluable comments from all the reviewers. We are also so grateful to Xu Fu for the help of debugging the hardware devices. 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.

References

[1] K. Akbudak and C. Aykanat. Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2258–2271, 2017.

[2] K. Akbudak, O. Selvitopi, and C. Aykanat. Partitioning models for scaling parallel sparse matrix-matrix multiplication. *ACM Trans. Parallel Comput.*, 4(3), 2018.

[3] P. N. Q. Anh, R. Fan, and Y. Wen. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *ICS '16*, 2016.

[4] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[5] A. Azad and A. Buluç. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *IPDPS '17*, 2017.

[6] A. Azad, A. Buluç, and J. R. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *Workshop on Graph Algorithm Building Blocks '15*, pages 804 – 811, 2015.

[7] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç. HipMCL: A high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research (NAR)*, 2018.

[8] A. Azad, O. Selvitopi, M. T. Hussain, J. R. Gilbert, and A. Buluç. Combinatorial blas 2.0: Scaling combinatorial algorithms on distributed-memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):989–1001, 2022.

[9] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang. Challenges of scaling algebraic multigrid across modern multicore architectures. In *IPDPS '11*, pages 275–286, 2011.

[10] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo. Communication optimal parallel multiplication of sparse random matrices. In *SPAA '13*, 2013.

[11] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[12] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Trans. Parallel Comput.*, 3(3), 2016.

[13] G. Ballard, C. Siefert, and J. Hu. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing*, 38(3):C203–C231, 2016.

[14] N. Bell, S. Dalton, and L. N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[15] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09*, pages 1–11, 2009.

[16] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. In *PPoPP '17*, pages 235–248, 2017.

[17] G. Bikshandi, B. B. Fraguela, J. Guo, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua. Implementation of parallel numerical algorithms using hierarchically tiled arrays. In *Languages and Compilers for High Performance Computing*, pages 87–101, 2005.

[18] M. Bisson and M. Fatica. A gpu implementation of the sparse deep neural network graph challenge. In *HPEC '19*, pages 1–8, 2019.

[19] J. C. Brodman, G. C. Evans, M. Manguoglu, A. Sameh, M. J. Garzarán, and D. Padua. A parallel numerical solver using hierarchically tiled arrays. In *LCPC '11*, pages 46–61, 2011.

[20] A. Buluç. *Linear Algebraic Primitives for Parallel Computing on Large Graphs*. PhD thesis, University of California, Santa Barbara, 2010.

[21] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA '09*, pages 233–244, 2009.

[22] A. Buluç and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP '08*, pages 503–510, 2008.

[23] A. Buluç and J. R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *IPDPS '08*, 2008.

[24] A. Buluç and J. R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal of Scientific Computing*, 34(4):170–191, 2012.

[25] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the GraphBLAS API for C. In *Workshop on Graph Algorithm Building Blocks*, 2017.

[26] A. Buluç and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[27] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IPDPS '11*, pages 721–733, 2011.

[28] Y. Chen, A. B. Hayes, C. Zhang, T. Salmon, and E. Z. Zhang. Locality-aware software throttling for sparse matrix operation on gpus. In

USENIX ATC '18, pages 413–425, 2018.

[29] Y. Chen, K. Li, W. Yang, G. Xiao, X. Xie, and T. Li. Performance-aware model for sparse matrix-matrix multiplication on the sunway taihulight supercomputer. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):923–938, 2019.

[30] S. Chou, F. Kjolstad, and S. Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *PLDI '20*, pages 823–838, 2020.

[31] N. Corp. The cusparse library, 2020.

[32] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland. Optimizing sparse matrix operations on gpus using merge path. In *IPDPS '15*, pages 407–416, 2015.

[33] S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014.

[34] S. Dalton, L. Olson, and N. Bell. Optimizing sparse matrix-matrix multiplication for the gpu. *ACM Trans. Math. Softw.*, 41(4), 2015.

[35] T. A. Davis. Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. In *HPEC '18*, pages 1–6, 2018.

[36] T. A. Davis. Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), 2019.

[37] T. A. Davis, M. Aznaveh, and S. Kolodziej. Write quick, run fast: Sparse deep neural network in 20 minutes of development time via suitesparse:graphblas. In *HPEC '19*, pages 1–6, 2019.

[38] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), 2011.

[39] G. V. Demirci and C. Aykanat. Cartesian partitioning models for 2d and 3d parallel spgemm algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 31(12):2763–2775, 2020.

[40] G. V. Demirci and C. Aykanat. Scaling sparse matrix-matrix multiplication in the accumulo database. *Distributed and Parallel Databases*, 38(1):31–62, 2020.

[41] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

[42] M. Deveci, C. Trott, and S. Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *IPDPSW '17*, pages 693–702, 2017.

[43] M. Deveci, C. Trott, and S. Rajamanickam. Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures. *Parallel Computing*, 78:33 – 46, 2018.

[44] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002.

[45] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: A user-level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.

[46] J. A. Ellis and S. Rajamanickam. Scalable inference for sparse deep neural networks using kokkos kernels. In *HPEC '19*, pages 1–7, 2019.

[47] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In *ICCS '02*, pages 632–641, 2002.

[48] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the performance of an algebraic multigrid cycle on hpc platforms. In *ICS '11*, pages 172–181, 2011.

[49] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang. Modeling the performance of an algebraic multigrid cycle using hybrid mpi/openmp. In *ICPP '12*, pages 128–137, 2012.

[50] I. Gelado and M. Garland. Throughput-oriented gpu memory allocation. In *PPoPP '19*, page 27–37, 2019.

[51] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[52] C. Gómez, F. Mantovani, E. Focht, and M. Casas. Efficiently running spmv on long vector architectures. In *PPoPP '21*, page 292–303, 2021.

[53] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015.

[54] F. Gremse, K. Küpper, and U. Naumann. Memory-efficient sparse matrix-matrix multiplication by row merging on many-core architectures. *SIAM Journal on Scientific Computing*, 40(4):C429–C449, 2018.

[55] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad. Bandwidth optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking. In *SPAA '20*, pages 293–303, 2020.

[56] G. Guidi, O. Selvitopi, M. Ellis, L. Oliker, K. Yelick, and A. Buluç. Parallel string graph construction and transitive reduction for de novo genome assembly. In *IPDPS '21*, pages 517–526, 2021.

[57] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[58] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, U. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *HPDC '18*, pages 66–79, 2018.

[59] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP '19*, pages 300–314, 2019.

[60] K. Hou, W. Liu, H. Wang, and W.-c. Feng. Fast segmented sort on gpus. In *ICS '17*, 2017.

[61] G. Huang, G. Dai, Y. Wang, and H. Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC '20*, pages 1–12, 2020.

[62] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad. Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale. In *IPDPS '21*, pages 90–100, 2021.

[63] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad. Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale. In *IPDPS '21*, pages 90–100, 2021.

[64] E.-J. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *ICCS '01*, pages 127–136, 2001.

[65] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[66] H. Ji, S. Lu, K. Hou, H. Wang, Z. Jin, W. Liu, and B. Vinter. Segmented merge: A new primitive for parallel sparse matrix computations. *International Journal of Parallel Programming*, pages 1–13, 2021.

[67] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson. Mathematical foundations of the GraphBLAS. In *HPEC '16*, 2016.

[68] J. Kepner, D. Bader, A. Buluç, J. Gilbert, J. Kepner, T. Mattson, and H. Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. In *ICCS '15*, 2015.

[69] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe. Tensor algebra compilation with workspaces. In *CGO '19*, pages 180–192, 2019.

[70] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1, 2017.

[71] N. Knight, E. Carson, and J. Demmel. Exploiting data sparsity in parallel matrix powers computations. In *PPAM '14*, pages 15–25, 2014.

[72] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.-Y. Oh, L. Oliker, and K. Yelick. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *IPDPS '16*, 2016.

[73] V. Kotlyar, K. Pingali, and P. Stodghill. Compiling parallel code for sparse matrix applications. In *SC '97*, pages 1–18, 1997.

[74] R. Kunchum, A. Chaudhry, A. Sukumaran-Rajam, Q. Niu, I. Nisa, and P. Sadayappan. On improving performance of sparse matrix-matrix multiplication on gpus. In *ICS '17*, 2017.

[75] J. Lee, S. Kang, Y. Yu, Y.-Y. Jo, S.-W. Kim, and Y. Park. Optimization of gpu-based sparse matrix multiplication for large sparse networks. In *ICDE '20*, pages 925–936, 2020.

[76] J. Li, J. Sun, and R. Vuduc. Hicoo: Hierarchical storage of sparse tensors. In *SC '18*, pages 238–252, 2018.

[77] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li. A coordinated tiling and batching framework for efficient gemm on gpus. In *PPoPP '19*, page 229–241, 2019.

[78] J. Liu, X. He, W. Liu, and G. Tan. Register-based implementation of the sparse general matrix-matrix multiplication on gpus. In *PPoPP '18*, page 407–408, 2018.

[79] J. Liu, X. He, W. Liu, and G. Tan. Register-aware optimizations for parallel sparse matrix-matrix multiplication. *International Journal of Parallel Programming*, 2019.

[80] W. Liu and B. Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *IPDPS '14*, pages 370–381, 2014.

[81] W. Liu and B. Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *ICS '15*, pages 339–350, 2015.

[82] W. Liu and B. Vinter. A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85(C):47–61, 2015.

[83] W. Liu and B. Vinter. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Computing*, 49(C):179–193, 2015.

[84] Z. Lu, Y. Niu, and W. Liu. Efficient block algorithms for parallel sparse triangular solve. In *ICPP '20*, 2020.

[85] S. Maleki, G. C. Evans, and D. A. Padua. Tiled linear algebra a system for parallel graph algorithms. In *LCPC '15*, pages 116–130, 2015.

[86] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *HPEC '13*, pages 1–2, 2013.

[87] T. Mattson, T. A. Davis, M. Kumar, A. Buluç, S. McMillan, J. Moreira, and C. Yang. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *GrAPL Workshop '19*, 2019.

[88] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira. GraphBLAS C API: Ideas for future versions of the specification. In *HPEC '17*, 2017.

[89] D. Merrill and M. Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC '16*, pages 678–689, 2016.

[90] D. Merrill and M. Garland. Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. In *PPoPP '16*, 2016.

[91] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout. Sparse computation data dependence simplification for efficient compiler-generated inspectors. In *PLDI '19*, pages 594–609, 2019.

[92] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. High-performance sparse matrix-matrix products on intel KNL and multicore architectures. In *ICPPW '18*, 2018.

[93] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Computing*, 90, 2019.

[94] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan. Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus. In *IPDPS '21*, pages 68–78, 2021.

[95] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *HPCA '18*, pages 724–736, 2018.

[96] M. Parger, M. Winter, D. Mlakar, and M. Steinberger. Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis. In *PPoPP '20*, pages 362–375, 2020.

[97] O. Selvitopi, M. T. Hussain, A. Azad, and A. Buluç. Optimizing high performance markov clustering for pre-exascale architectures. In *IPDPS '20*, pages 116–126, 2020.

[98] M. M. Strout, M. Hall, and C. Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[99] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521–530, 2005.

[100] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *SC '02*, 2002.

[101] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *HPCC '05*, pages 807–816, 2005.

[102] X. Wang, W. Liu, W. Xue, and L. Wu. swsptrsv: A fast sparse triangular solve with sparse level tile layout on sunway architectures. In *PPoPP '18*, pages 338–353, 2018.

[103] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng. Dual-side sparse tensor core. In *ISCA '21*, page 1083–1095, 2021.

[104] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS '06*, pages 307–316, 2006.

[105] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07*, 2007.

[106] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger. Adaptive sparse matrix-matrix multiplication on the gpu. In *PPoPP '19*, pages 68–81, 2019.

[107] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *HPEC '17*, pages 1–7, 2017.

[108] Y. Xia, P. Jiang, G. Agrawal, and R. Ramnath. Scaling sparse matrix multiplication on cpu-gpu nodes. In *IPDPS '21*, pages 392–401, 2021.

[109] Z. Xie, G. Tan, W. Liu, and N. Sun. Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *ICS '19*, pages 94–105, 2019.

[110] Z. Xie, G. Tan, W. Liu, and N. Sun. A pattern-based spgemm library for multi-core and many-core architectures. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):159–175, 2022.

[111] C. Yang, A. Buluç, and J. D. Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *arXiv preprint*, 2019.

[112] C. Yang, A. Buluç, and J. D. Owens. Design principles for sparse matrix multiplication on the GPU. In *Euro-Par '18*, 2018.

[113] C. Yang, A. Buluç, and J. D. Owens. Implementing push-pull efficiently in GraphBLAS. In *ICPP '18*, 2018.

[114] A. Yaşar, S. Rajamanickam, J. Berry, M. Wolf, J. S. Young, and V. ÇatalyÜrek. Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments : (update on static graph challenge). In *HPEC '19*, pages 1–4, 2019.

[115] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas. Speeding up spmv for power-law graph analytics by enhancing locality & vectorization. In *SC '20*, 2020.

[116] K. Yotov, Xiaoming Li, Gang Ren, M. J. S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358–386, 2005.

[117] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.

[118] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 88:106848, 2020.

[119] Z. Zhang, H. Wang, S. Han, and W. J. Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *HPCA '20*, pages 261–274, 2020.

[120] Y. Zhao, J. Li, C. Liao, and X. Shen. Bridging the gap between deep learning and sparse matrix format selection. In *PPoPP '18*, pages 94–108, 2018.
[121] W. Zhou, Y. Zhao, X. Shen, and W. Chen. Enabling runtime spmv format selection through an overhead conscious method. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):80–93, 2020.

A Artifact Description

A.1 Check-list

- **Algorithm**: parallel sparse matrix-matrix multiplication (SpGEMM)
- **Program**: CUDA and C/C++ OpenMP code
- **Compilation**: GPU CUDA code: NVIDIA nvcc and GNU gcc (v11.4 and v7.5.0 as tested, respectively)
- **Binary**: CUDA and OpenMP executables
- **Dataset**: 142 sparse square matrices in the SuiteSparse Matrix Collection requiring no less than one billion floating operations when computing $C = A^2$ and $C = AA^T$ and 18 representative matrices (for a more detailed comparison and analysis).
- **Run-time environment**: Ubuntu 18.04 with CUDA and GPU driver version are 11.4 and 470.57.02 as tested, respectively.
- **Hardware**: Any CUDA GPU with compute capability at least 6.1 (NVIDIA Geforce RTX 3060 and NVIDIA Geforce RTX 3090 as tested) and any Intel CPU (Intel i9-7900X CPU @ 3.30GHz as tested)
- **Output**: execution time in total and breakdown, GFLOPs throughput, and memory cost
- **Experiment workflow**: git clone projects; download the matrices; run the executable; observe the results
- **Publicly available?**: Yes

A.2 How delivered

The source code of TileSpGEMM can be get in this link: https://github.com/SuperScientificSoftwareLaboratory/TileSpGEMM.

A.3 Hardware dependencies

To better reproduce experiment results, we suggest an NVIDIA GPU with compute capability 8.6.

A.4 Software dependencies

Our TileSpGEMM evaluation requires the CUDA GPU driver, nvcc CUDA compiler, and the cuSPARSE library, all of them are included with the CUDA Toolkit. The artifacts have been tested on Ubuntu 18.04/20.04, and are expected to run correctly under other Linux distributions.

A.5 Datasets

At this time of writing, our matrix parser currently only supports input files in the matrix market format (*.mtx), and the matrix list (the datasets we used in our experiment) is in our package. All these matrices in our matrix list are publicly available from the SuiteSparse Matrix Collection, and they can be downloaded from the website:
https://sparse.tamu.edu/

A.6 Installation

Firstly, one must clone the TileSpGEMM code to the local machine.
Then, one must use GNU make to build the executable:
$ make
After that, one will get an executable called *test* and finally can (optionally) download and unpack the datasets for the following tests.

A.7 Experimental workflow

Run our algorithm throught
$ ./test -d 0 -aat 0 <path/to/dataset/mtx>
(Tips: The code takes an optional d=<gpu-device, e.g., 0> parameter that specifies the GPU device to run if multiple GPU devices are available on the machine, and another optional aat=<transpose, e.g., 0> parameter that means computing $C = A^2$ (-aat 0) or $C = AA^T$ (-aat 1)). Then, one can observe the output information on the screen.

In order to record various experimental data more conveniently, we deliver our artifact with two packages. The first one called 'TileSpGEMM' including our source code, datasets and all of our experimental data. Here we also provide six python scripts to generate corresponding figures in our paper. One can run '*reproduce_paper_figure.sh*' , and all of the figures in our paper will be generated in 'Figure' folder. The other one 'Tilespgemm_step' is used for testing the peak space cost of running SpGEMM and generating **Figure 9 in our paper**. Its compiling and running instructions are consistent with our source code.

A.8 Output information

Lines 1-2 outputs the input matrix's information including the path of matrix file, The number of rows, columns and nonzeros.
Line 3 prints the file loading time (in seconds).
Line 4 prints the size of tile used in our TileSpGEMM algorithm.
Line 5 prints the number of floating point operations during the multiplication.
Line 6 prints the runtime of transforming the input matrix from the CSR format to our tiled data structure (in milliseconds) **(Figure 12 in our paper)**.
Line 7 prints TileSpGEMM data structure's space consumption (in million bytes) **(Figure 11 in our paper)**.
Lines 8-14 print execution time (in milliseconds) of the three algorithm steps and all memory allocation on CPU and GPU **(Figure 10 in our paper)**.
Line 15 prints the number of tiles of the resulting matrix C.

Line 16 prints the number of nonzeros of the resulting matrix C.
Line 17 prints TileSpGEMM runtime (in milliseconds) and performance (in GFlOPs) **(Figures 6 and 7 in our paper)**.
Line 18 prints the checking result after comparing our output with the one generated by cuSPARSE.