BRP-SpMM: Block-Row Partition Based Sparse Matrix Multiplication with Tensor and CUDA Cores

Yukang Dong, Wenbin Jiang, Xinhai Shen, Haihong Guo, Zhiyuan Shao, Hai Jin
National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Clusters and Grid Computing Lab, Huazhong University of Science and Technology, China
{ykdong, wenbinjiang, d202381421, guohh, zyshao, hjin}@hust.edu.cn

2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3315-3237-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/IPDPS64566.2025.00085

***Abstract*—Sparse-Dense Matrix Multiplication (SpMM) is a fundamental computational operation in various domains, and leveraging Tensor cores or CUDA cores on GPU to accelerate SpMM has become common practice. While Tensor cores offer notable advantages in dense matrix multiplication, their efficiency significantly decreases when handling sparse matrices. Besides, the computational power of CUDA cores should not be overlooked. Nevertheless, the differences in supported data formats present challenges in effectively leveraging both types of cores to accelerate SpMM. To this end, we propose BRP-SpMM, a block and row partition approach designed for efficient SpMM on GPU. BRP-SpMM partitions the sparse matrix into two parts: TC Block and Residual Row part, which are computed on Tensor cores and CUDA cores separately. Meanwhile, a customized storage format is proposed to manage these two distinct parts. Our two GPU kernels incorporate several advanced techniques including load balance, register remapping, and 1-D tiling. BRP-SpMM can achieve higher memory access efficiency and more rational computing resource utilization of GPU. Extensive experiments on modern NVIDIA A800 GPU show that BRP-SpMM outperforms SOTA libraries by up to 2.9× (on average 2.1×). Furthermore, BRP-SpMM accelerates end-to-end GNN training by up to 1.9× compared to popular frameworks.**

***Index Terms*—Sparse-Dense Matrix Multiplication (SpMM), Tensor cores, CUDA cores, GPU, GNN.**

I. Introduction

Sparse-Dense Matrix Multiplication (SpMM) is a crucial operation across various domains such as scientific computing [3], [13], machine learning [23], [31], etc. However, there is no strict and formal definition of the sparse matrix. Wilkinson gives the most popular one: the sparse matrix is any matrix with enough zero elements that it pays to take advantage of them [7]. Therefore, whether the matrix with around 50% sparsity (the proportion of zero elements within a matrix), such as the weight matrix in SpDNN, or the matrix with over 90% sparsity, such as the adjacency matrix of a graph, both can be referred as the sparse matrix. Meanwhile, different strategies may be required depending on the level of sparsity [4], [22], [24], [27], [35], as using Tensor cores on GPU to process sparse matrices with high sparsity still presents significant challenges.

Nowadays, GPUs are equipped with two primary types of computing units: CUDA cores and Tensor cores [14]. CUDA cores are the fundamental computing unit, supporting a wide range of data types and storage formats, and are capable of efficiently handling both dense and sparse matrices. However, Tensor cores are specifically designed for processing dense matrices, and directly using them for SpMM operations often results in low efficiency. Meanwhile, Tensor cores only support *blocked formats*, when using Tensor cores to process sparse matrices with high sparsity, the first step in existing works [20], [22], [33] is always looking for dense or almost dense blocks of the input sparse matrix, this step can be referred to as sparse matrix condense.

TC-GNN [33], as a pioneering study focusing on SpMM operations in Graph Neural Networks (GNNs) [26], [29] using Tensor cores, introduced Sparse Graph Translation (SGT) to condense the sparse matrix. After condensation, the sparse matrix consists of dense blocks, referred to as TC Blocks, which serve as the basic processing unit for Tensor cores. However, the number of non-zero elements in each TC Block is unpredictable, leading to inefficient memory access. Furthermore, some TC Blocks contain very few non-zero elements, resulting in wasted computational resources on Tensor cores. DTC-SpMM [22], a state-of-the-art work for SpMM built upon TC-GNN, focuses primarily on optimizing runtime GPU kernels, including techniques such as double buffer, load balance, register remapping, and pipeline strategies. Although this approach yields significant performance improvements, the issues related to TC Blocks persist, leaving considerable room for further optimization. Additionally, the excessive optimizations result in a bloated GPU kernel function, placing excessive pressure on the instruction cache of the GPU.

Meanwhile, the champion of Graph Challenge in 2022 [32] partitioned the sparse matrix into dense and sparse parts, assigning computations to Tensor cores and CUDA cores, respectively. In this process, a similarity-based reordering strategy was employed to identify dense blocks within the sparse matrix. Although this work was designed for SpDNNs [5], focusing on weight matrices with low sparsity (50%) and dimensions no larger than 256×256, it suggests that by customizing task partitioning to divide the sparse matrix into two parts and utilizing different computational units for processing, higher efficiency can be achieved.

However, there are several challenges exist. First, an effective partition approach for the sparse matrix with higher sparsity and larger dimensions is crucial, and the reordering strategy may be unsuitable due to its time-consuming nature. During partitioning, a critical aspect is controlling the number of non-zero elements in each dense block to achieve higher memory access efficiency. Second, a novel and efficient stor-

1530-2075/25/$31.00 ©2025 IEEE
DOI 10.1109/IPDPS64566.2025.00085

age format for these two parts is necessary, as Tensor cores and CUDA cores require different data formats. Third, the GPU runtime kernels for these two parts need to be highly optimized while maintaining a minimal program size.

To this end, targeting sparse matrices with high sparsity, we introduce BRP-SpMM, a Block-Row Partition approach for efficient SpMM on GPU. This approach partitions the sparse matrix into two parts: the TC Block part and the Residual Row part, which are computed using Tensor cores and CUDA cores, respectively. The TC Block is the dense block of the sparse matrix, serving as the basic processing unit for Tensor cores, with size 16×8. Specifically, we introduce *Fixed Non-zero* strategy to fix the number of non-zero elements in each TC Block and any non-zero elements that cannot form a complete *TC Block Group* are partitioned into the Residual Row part. Then, a novel and efficient storage format is introduced to manage these two parts, with a space requirement only slightly larger than the standard Compressed Sparse Row (CSR) format. Furthermore, two runtime GPU kernels are designed to process these two parts: the TC Block kernel and the Residual Row kernel. For the TC Block kernel, we incorporate two key techniques: load balance and register remapping. The load balance technique ensures that each thread block on the GPU processes the same amount of tasks. Meanwhile, our novel register remapping strategy greatly reduces the number of atomic write-back operations. For the Residual Row kernel, due to the minimal number of non-zero elements per residual row, we employ a one-dimensional tiling strategy to maximize thread resource utilization. Notably, we maintained a minimal program size while implementing two distinct GPU kernels.

We use eight graph datasets to evaluate the performance of SpMM. Extensive experiments on modern NVIDIA A800 GPU show that BRP-SpMM outperforms the most widely used libraries cuSPARSE [16] by up to 2.9× (on average 2.1×). Notably, BRP-SpMM delivers an average speedup of 1.2× and up to 1.8× compared to the state-of-the-art SpMM method, DTC-SpMM [22]. Detailed analysis shows that BRP-SpMM improves memory access efficiency and optimally utilizes GPU computational resources. Furthermore, BRP-SpMM is successfully applied to end-to-end training of GNN [26], outperforming existing popular frameworks by up to 1.9×.

The contributions of this paper are summarized as follows:

- We propose BRP-SpMM, a Block-Row Partitioning approach for efficient SpMM on GPU, which condenses and partitions sparse matrices into two parts: TC Block part and Residual Row part.
- A customized storage format is introduced to manage these two parts efficiently.
- We propose two highly optimized GPU runtime kernels that utilize several advanced techniques to separately employ Tensor cores and CUDA cores.
- Experimental results indicate that BRP-SpMM achieves significant speedups over four representatives and SOTA SpMM works on diverse real-world matrices.

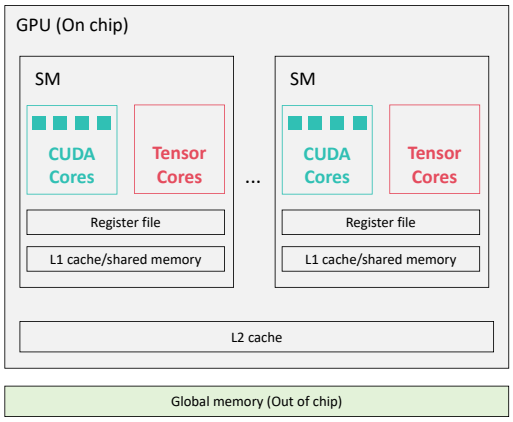


Fig. 1. GPU Architecture.

II. Background and Related Work

A. NVIDIA GPU architecture

NVIDIA GPUs [14], [15] are highly parallel devices consisting of numerous streaming multiprocessors (SMs), each equipped with CUDA cores, Tensor cores, and additional units, as depicted in Fig. 1. In GPU kernel execution, threads are grouped into blocks and grids. Upon kernel launch, these thread blocks are distributed among available SMs, ensuring maximal parallelism while minimizing potential resource contention. GPUs feature a large but high-latency global memory accessible to all SMs, along with an L2 cache shared across SMs and L1 caches dedicated to individual SM. Both CUDA cores and Tensor cores are computational units, but they differ in the data types and formats they support. CUDA cores handle a wide range of standard data types, including FP16, INT32, FP32, and FP64. While Tensor Cores also support FP16 and FP64, for 32-bit data types, the computation of C=A×B must use mixed precision, with matrices A and B in TF32, and matrix C in FP32 [15]. For sparse matrix operations, CUDA cores accommodate widely-used storage formats such as Coordinate (COO) and Compressed Sparse Row (CSR), whereas Tensor cores require inputs to be in NVIDIA-defined block formats like m16n8k8 and m8n8k4. NVIDIA provides two main APIs for programming on Tensor cores: WMMA in high-level C code [15] and *mma* in low-level PTX code [18]. Since Tensor cores are primarily optimized for accelerating GEMM (General Matrix Multiply) operations, these APIs do not cover all necessary operations for sparse computations, such as atomic writes. Consequently, most existing works [22], [24], [35] leveraging Tensor cores for SpMM utilize a combination of *mma* PTX APIs and standard CUDA C code. In this paper, we follow a similar approach.

B. Related Work about SpMM

Sparse-Dense Matrix Multiplication (SpMM) involves multiplying a $M \times K$ sparse matrix A with a $K \times N$ dense matrix B, resulting in a dense $M \times N$ output matrix C, expressed as $C_{M\times N} = A_{M\times K} \times B_{K\times N}$. Numerous studies have proposed

optimization techniques for SpMM on GPUs, focusing on both CUDA cores and Tensor cores.

For SpMM using CUDA cores, the widely-used NVIDIA cuSPARSE library [16] provides high-performance CUDA-core SpMM kernels and supports standard CSR and COO formats. ASpT [2] introduced an adaptive tiling method to partition the sparse matrix into dense and sparse parts to utilize the shared memory efficiency. GE-SpMM [4] introduced the coalesced row caching and coarse-grained warp merging method to ensure efficient coalesced access to the GPU global memory. Sputnik [27] introduced one-dimensional tiling and reverse offset memory alignment strategies to efficiently handle sparse matrix, which is widely regarded as the state-of-the-art method on CUDA cores. RoDe [19] divides each row of the sparse matrix into segments of size 32, achieving more efficient memory access. Additionally, the Row-Reordering [9], [21] strategy is often used to enhance data locality.

Using Tensor cores to accelerate SpMM has gained significant traction in recent years. NVIDIA's cuSPARSELt [17] is a high-performance CUDA library using Tensor cores, focusing on 1:2 or 2:4 structured sparsity. VectorSparse [35] introduced column-vector-sparse-encoding and Tensor-Core-based 1D Octet Tiling for Volta GPU architecture under FP16 precision, which has a smaller grain size under the same reuse rate compared with block sparsity. Magicube [24] proposed a novel data format called SR-BCRS and several crucial online optimizations for sparse attention operation in Transformer [25], [28], [34]. Meanwhile, the Sparse Deep Neural Network (SpDNN) Graph Challenge [5] also presents some SpMM works [6], [11], [30], [32] focused on sparse weight matrix in neural networks with low sparsity. In contrast, TC-GNN [33] focuses on optimizing general SpMM operations within Graph Neural Network (GNN) [8], [26], [29] workloads. In this paper, we refer to the SpMM kernel in TC-GNN as TCGNN-SpMM. In TCGNN-SpMM, the sparse matrix is the adjacency matrix of a graph, which is highly sparse (sparsity>90%) and can be very large, reaching $1M \times 1M$ in size. DTC-SpMM [22], the latest work, further improves upon TCGNN-SpMM by proposing efficient storage formats, reordering methods, and several runtime optimizations. Unfortunately, DTC-SpMM has not made any additional advancements in sparse matrix condensing techniques and missed many optimization opportunities.

III. Analyses and Motivation

A. Brief introduction of TCGNN-SpMM

TCGNN-SpMM [33] introduces Sparse Graph Translation (SGT) to condense the sparse matrix, as shown in Fig. 2. Colored small squares denote non-zeros in the sparse matrix. To simplify the explanation, every four rows are grouped into a row window. Within each window, if a local column contains no non-zeros, that local column is condensed, as seen in columns 2, 3, 5, 7, 8, 10, 12, 14, and 15 in window $w0$. After condensed, each 4×4 tiling represents a TC block, which is the basic unit processed by the Tensor cores. Each window may require padding the local columns on the right side to form a complete TC block. In practical implementation, the window size is 16, and the TC block size is 16×8. Then, Tensor cores are used for the calculation process.

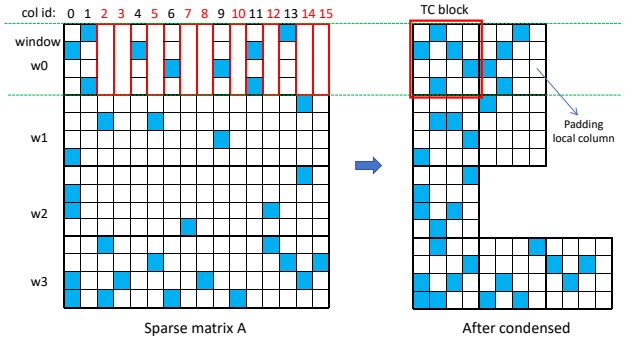


Fig. 2. Sparse Graph Translation (SGT) in TCGNN-SpMM.

B. Challenges and opportunities

a) Inefficient memory access: The main limitation of the SGT method is the unpredictable non-zero elements in each TC block, leading to unaligned memory access and inefficient L2 cache usage. As shown in Figure 3, when each dense block is 128 bytes and the starting address is aligned to 128, GPU memory access efficiency is maximized. In contrast, misaligned starting addresses and unpredictable dense block sizes can significantly reduce memory access efficiency. The L2 cache on the GPU is primarily used to load data from global memory, with a cache line size of 32 bytes, which represents the smallest data load unit. Therefore, when the size of a dense block is a multiple of 32 bytes, L2 cache utilization reaches its peak. However, the current SGT method clearly falls short of achieving this. Thus, optimizing memory access efficiency during building dense blocks is crucial for improving SpMM computational performance.

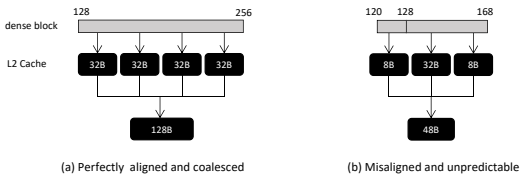


Fig. 3. Memory access patterns on GPU. (a) Perfectly aligned and coalesced. The dense block size is fixed at 128B. (b) Misaligned and unpredictable. We use 8+32+8B as an example, in practice, both the dense block size and starting address are unpredictable in SGT.

b) Compute unit mismatched: As shown in Fig. 2, the second TC block in w_1 contains only a single non-zero element, leading to a significant underutilization of the Tensor cores. Tensor cores are designed to handle dense matrix operations and achieve high throughput, but their potential is not fully leveraged when faced with blocks of this type. This inefficiency suggests that an alternative strategy may be beneficial. Specifically, reverting incomplete TC blocks like this to their original row form and computing them

using CUDA cores, could result in more efficient resource utilization and improved overall performance. This hybrid approach would allow us to balance the strengths of both Tensor cores and CUDA cores.

c) Bloated runtime GPU kernel: The GPU kernel function of TCGNN-SpMM [33] utilizes the WMMA C API but does not contain advanced optimization techniques. In contrast, DTC-SpMM [22] uses the low-level *mma* PTX API and incorporates numerous runtime optimizations, such as double buffering, vectorized global memory accesses, register remapping, and load balancing. Despite their extensive efforts, this results in a significantly large size kernel function, with program size reaching approximately 300 lines. VectorSparse [35] proposes a key guideline that the kernel function should reduce program size to avoid overflow of the instruction cache. Thus, the challenge lies in how to optimize the kernel function while minimizing the program size effectively.

IV. BRP-SpMM Design

A. Overview

BRP-SpMM is designed to fully exploit the computational power of the GPU for SpMM, which partitions the sparse matrix into the TC Block part and Residual Row part, leveraging Tensor cores and CUDA cores for computation separately. As shown in Fig. 4, BRP-SpMM consists of three key components. ❶ First, we introduce an innovative Block-Row Partition (BRP) approach that reorganizes the input sparse matrix by partitioning it into two distinct parts. ❷ Next, a novel storage format is proposed to efficiently manage these two parts. ❸ Finally, two highly optimized GPU runtime kernels are introduced, integrating load balancing, register remapping, and 1-D tiling strategies. These two kernels are executed sequentially, and their computed results are summed to obtain the final results. In the following sections, we will explain each of these components in detail.

B. Block-Row Partition

To fully utilize the computational resources of the GPU, we partition the sparse matrix into two parts: the TC Block part and the Residual Row part. As depicted in Fig. 1, our Block-Row Partition (BRP) approach builds upon the SGT in TCGNN-SpMM [33]. We retain the concepts of row window and TC Block. Each TC Block has a size of $h \times w$=16×8. Compared to SGT, our approach introduces two key improvements:

a) Fixed Non-zero: The most significant issue with SGT is the variable number of non-zero elements within each TC block, which limits the memory access efficiency of GPU. To address this issue, we introduce the *Fixed Non-zero* strategy, where the number of non-zero elements per column in each TC block is fixed, denoted by f. We observed that when the matrix is highly sparse, even after compression using the SGT method, most local columns (the columns within a TC Block) contain only a single non-zero element. Thus, we set f=1, meaning the entire TC Block contains exactly w non-zero elements. This approach minimizes the growth in the number of TC Blocks while maintaining a fixed number of

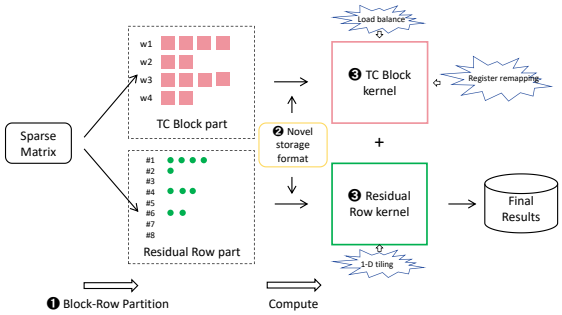


Fig. 4. The design overview of BRP-SpMM.

non-zero elements per TC Block, leading to improved GPU memory access efficiency. During the implementation, if a local column contains multiple non-zero elements, that column is further split to ensure each local column has only one non-zero element, as shown in columns 1 and 11 of window *w0*, and the first local column in *w3* (yellow rectangle) in Fig. 5. However, the last TC block in each window may be incomplete, with fewer non-zero elements, which can reduce computational efficiency on GPU. Based on this analysis, we further propose the concept of the *TC Block Group*.

b) TC Block Group: The concept of the *TC Block Group* means that every g TC Blocks form a *TC Block Group*. During partitioning, the TC Block part is composed of multiple *TC Block Group*s, while any non-zero elements that cannot form a complete *TC Block Group* are assigned to the Residual Row part. In Fig. 5, each colored square represents a non-zero element, with pink and green squares denoting the non-zero elements in the TC Block part and Residual Row part, respectively. For simplicity, we set the row window size to 4, the TC Block size to 4×4, and define a *TC Block Group* as two TC Blocks ($g = 2$). In row window 0 (*w0*), there is one *TC Block Group* in the TC Block part and two non-zero elements in the Residual Row part. Row windows *w1* and *w2* illustrate special cases: *w1* contains only the TC Block part, while *w2* includes only the Residual Row part. In *w2*, there are five non-zero elements in the Residual Row part. Although these five elements could form a TC Block, they cannot form a complete *TC Block Group*, so all five elements are assigned to the Residual Row part. This case also applies to the last five non-zero elements in *w3*.

In practice, we set g=4, which means that each *TC Block Group* contains a fixed number of 32 non-zero elements ($f \times w \times g$=32). For the *float* data type, 32 *float* values occupy 128 bytes. During the data loading process, the starting address of the *TC Block Group* is aligned to a multiple of 128, ensuring optimal access alignment. Furthermore, since 128 bytes is a multiple of the L2 cache line size (32 bytes), this configuration maximizes memory access efficiency, achieving the ideal state shown in Figure 3(a).

C. Storage format

We have made further improvements to the storage format, requiring up to four arrays for each of the two parts, as shown

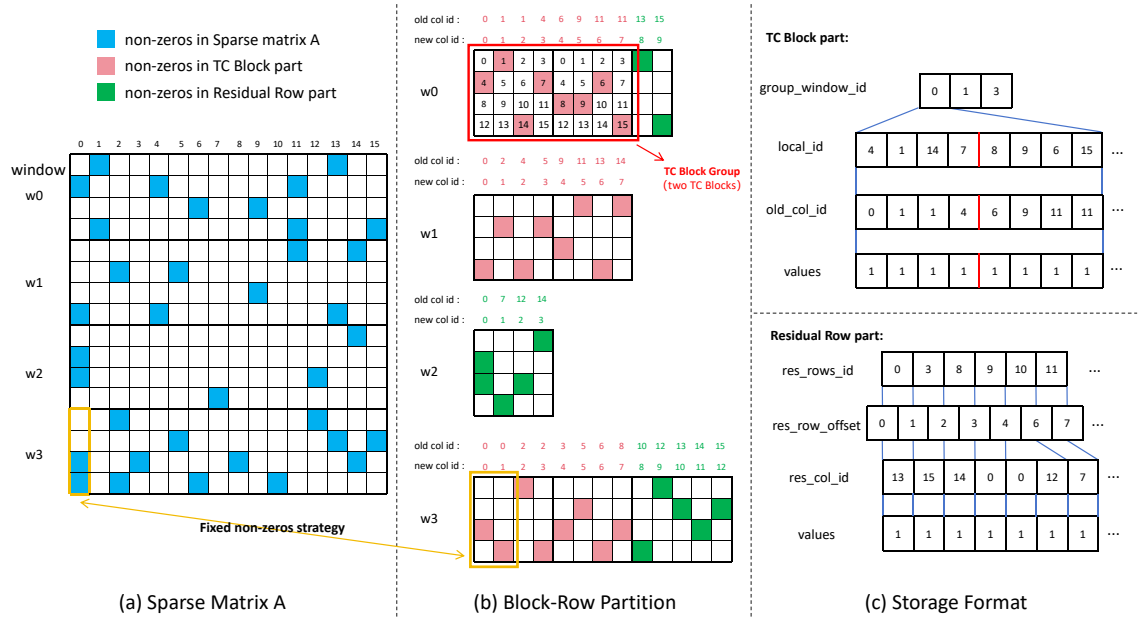


Fig. 5. The workflow of Block-Row Partition approach and storage format. (Each column in the TC Block contains exactly one non-zero element. Any non-zero elements that cannot form a complete *TC Block Group* are assigned to the Residual Row part.)

in Figure 5(c). In the TC Block part: ❶ The *group_window_id* array stores the row window index of each *TC Block Group*. ❷ The *local_id* array stores the local index of each non-zero element in the TC Block. The values in *local_id* range from 0 to (h×w-1). For instance, in Figure 5, where the TC Block size is set to h×w = 4×4, the *local_id* values range from 0 to 15. For example, in *w0*, the *local_id* for the four non-zero elements in the first TC Block are 4, 1, 14, and 7 in column order. ❸ The *old_col_id* array stores the old column index of each non-zero element, corresponding one-to-one with the *local_id* array. This array is necessary for retrieving the corresponding row in the dense matrix during computation. ❹ The *values* array stores the value of each non-zero element. This array is optional and can be omitted if all non-zero elements are the same, such as when all values are 1.

For the Residual Row part, since not every row of the sparse matrix is a Residual Row, we extend the standard CSR format, which includes the *row_offset*, *col_id*, and *values* arrays, by adding a *res_rows_id* array to specify the row containing non-zero elements. For clarity, we explain each array in detail: ❶ The *res_rows_id* array stores the row index of each Residual Row. ❷ The *res_row_offset* array stores the offset of each Residual Row. The number of non-zero elements in *res_rows_id*[i] is given by *res_row_offset*[i] - *res_row_offset*[i-1] (where $i \geq 1$). ❸ The *res_col_id* array stores the old column index of each non-zero element in the Residual Row part. ❹ The *values* array serves the same function as the one in the TC Block part.

Next, we estimate the storage space required for our customized format. In our work, the exact number of TC Blocks cannot be determined without knowing the distribution of non-zero elements, as the distribution varies significantly within different row windows. Therefore, we provide a rough estimate of the required storage space. Assume the sparse matrix has dimensions $n \times n$ with a total of nnz non-zero elements and the TC Block size is $h \times w$, with each *TC Block Group* containing g TC Blocks. After condensing and partitioning the sparse matrix using the BRP method, the number of non-zero elements in the TC Block and Residual Row parts is denoted as nnz_t and nnz_r, respectively, with $nnz_t + nnz_r = nnz$.

The number of *TC Block Group*s is $\frac{nnz_t}{g \times w}$, so the total size of the four arrays needed for the TC Block part is $size_t = \frac{nnz_t}{g \times w} + 3 \times nnz_t$. When $g \times w = 4 \times 8$, this equals $3.03 \times nnz_t$, which can be approximated as $size_t \approx 3nnz_t$. For the Residual Row part, in the extreme case where every row of the sparse matrix is a Residual Row, the upper limit of the res_row array size is n. Thus, the upper limit of the total size for the four arrays in the Residual Row part is $size_r = n + (n+1) + nnz_r + nnz_r = 2n + 1 + 2nnz_r$. The total required storage size is approximately $3nnz_t + 2nnz_r + 2n = 2nnz + nnz_t + 2n$. In comparison, the storage space required by the CSR format is approximately $2nnz + n$, meaning that our method requires only about $nnz_t + n$ additional storage (about 50% more compared to CSR), which is quite efficient.

Furthermore, we make every effort to accelerate the execution of the Block-Row Partition (BRP) approach. To reduce preprocessing overhead, we leverage CUDA to accelerate the BRP process and integrate it with the format conversion process. During partitioning, the relevant data of two parts is directly stored in our customized format, rather than per-

forming format conversion as a separate step. This integration significantly reduces preprocessing overhead. The pseudo-code is provided in Alg. 1. Block-Row Partition takes *row_offset* in CSR format and coordinate list in COO format as input. We partition nnz in each row window and get two arrays. *tcblk_offset* marks the starting index of nnz belongs to TC Block part in each row window and *res_offset* is similar but for the Residual Row part (lines 3). Then, the index range of nnz in each row window can be obtained (lines 8-9, lines 14-16, lines 23-25). Therefore, computations within the $for\ loop$ can be parallelized, eliminating the need for atomic operations (lines 10-12, lines 17-19, lines 27-30, lines 33-36). Note that two sorting operations are employed (line 21, line 38). The first converts to column-major order in each row window, while the second reverts to row-major order since the processing of the Residual Row part resembles CSR compression (lines 38-41).

After obtaining the TC Block part and the Residual Row part, we use Tensor cores and CUDA cores to compute these two parts separately. In the next two sections, we present a detailed explanation of our GPU kernel design.

D. TC Block kernel

In this section, we detail the design of our TC Block GPU kernel that follows the Block-Row Partition approach.

We begin by briefly outlining the overall execution flow, as presented in Alg. 2. The kernel starts by calculating the necessary indices (lines 4-6). Following this, shared memory arrays and registers are allocated (lines 8-9 and 11-12). Subsequently, *old_col_id* is loaded into shared memory (line 14). Meanwhile, all elements of A_tile are initialized to zero, and non-zero values are set using the *local_id* array (lines 15-16). At this stage, all required data is prepared. Next, the "cvt.rna.tf32.f32" assembly instruction is employed to load A_tile and B_tile into registers, converting the data type from float32 to tf32 (lines 18 and 20). Then, four *mma.m16n8k8* instructions are called to utilize Tensor cores for computation (lines 22-33). Finally, atomic write operations are used to store the results in the output matrix C.

The advantages of our TC Block kernel lie in the load balance strategy, which ensures that each thread block handles an equal amount of work, and the new register mapping strategy, which reduces the number of atomic write-back operations. Meanwhile, the program size remains as small as possible. Hereafter, a detailed explanation is provided:

a) Load balance: As the concept of *TC Block Group* is introduced in this paper, which means g TC Blocks form a *TC Block Group*. In the computation process, we assign a thread block on GPU to handle a *TC Block Group*. Meanwhile, by adopting the *Fixed Non-zero* strategy in the BRP method, the non-zero elements within each *TC Block Group* remain fixed and identical. Thus, each thread block processes the same number of non-zero elements, achieving near-perfect load balance. In the specific implementation, we use shared memory to build A_tile. First, we allocate a block of shared memory with a size of $h \times w \times g$ and initialize all its elements to zero. Subsequently, we set the non-zero elements using

Algorithm 1 Block-Row Partition (BRP) CUDA pseudo code.

```
1: Input: Sparse Matrix A(row_offset, row_idx, col_idx)
2: Output: group_window_id, local_id, old_col_id, res_row_id,
   res_row_offset, res_col_id
3: tcblk_offset,res_offset=Partition(row_offset);
4: // One thread block handles one row window
5: tid=threadIdx.x;
6: window_id=blockIdx.x;
7: // Generate group_window_id
8: grp_start=tcblk_offset[window_id]/(g*w);
9: grp_end=tcblk_offset[window_id+1]/(g*w);
10: for i=tid to grp_end-grp_start by ThrdperBlk do
11:   group_window_id[grp_start+i]=window_id;
12: end for
13: // Index range of nnz in current row window
14: win_start=row_offset[window_id*h];
15: win_end=
      row_offset[min(window_id*h+h,num_nodes)];
16: nnz_window=win_end-win_start;
17: for i=tid to nnz_window by ThrdperBlk do
18:   nnz_window_id[win_start+i]=window_id;
19: end for
20: // Column-major order in each row window
21: MutiKeySort(
      Tuple(nnz_window_id, col_idx, row_idx));
22: // Index range of nnz in TC Block part
23: TCBlkPart_start=tcblk_offset[window_id];
24: TCBlkPart_end=tcblk_offset[window_id+1];
25: TCBlkPart=TCBlkPart_end-TCBlkPart_start;
26: // Generate local_id and old_col_id
27: for i=tid to TCBlkPart by ThrdperBlk do
28:   local_id[TCBlkPart_start+i]=
        row_idx[win_start+i]%h*w+tid%w;
29:   old_col_id[TCBlkPart_start+i]=
        col_idx[win_start+i];
30: end for
31: // Get residual nnz, coo format
32: ResPart=nnz_window-TCBlkPart;
33: for i=tid to ResPart by ThrdperBlk do
34:   res_row[res_offset[window_id]+i]=
        row_idx[i+win_end-ResPart];
35:   res_col[res_offset[window_id]+i]=
        col_idx[i+win_end-ResPart];
36: end for
37: // Row-major order
38: MutiKeySort(Tuple(res_row,res_col));
39: res_row_offset=GetCSRFormat(res_row);
40: res_row_id=Unique(res_row); // Deduplication
41: res_col_id=res_col;
```

the corresponding *local_id*. During this process, the starting addresses corresponding to *local_id* are straightforward to compute (line 6 in Alg.2). Please note that in this process, we do not allocate an array in shared memory to store *local_id*; instead, the data is accessed directly from global memory. Since the *local_id* array is only used once, allocating additional shared memory is unnecessary.

As shown in Figure 6, assuming that *w0* consists of two *TC Block Groups*, we assign two thread blocks to process this window. The warps within each thread block share the same A_Tile from the *TC Block Group*, while each warp retrieves different B_Tiles from the dense matrix B. Each warp

Algorithm 2 TC Block GPU kernel pseudo code.

```
1: Input: group_window_id, local_id, old_col_id, values, Dense Matrix B, g=4, N
2: Output: Matrix C
3: // One thread block handles one TC Block Group.
4: bid = blockIdx.x;
5: window_id = group_window_id[bid];
6: eIdx_start = bid×g × w;
7: // Allocate shared memory. h × w = 16 × 8, g = 4.
8: __shared__ A_tile [g × h × w];
9: __shared__ old_col_id_smem [w × g];
10: // Allocate registers.
11: uint32_t RA[16], RB[8];
12: float RC[4];
13: // Initialization.
14: Load(old_col_id_smem);
15: InitZeros(A_tile); // Init all elements in A_tile to 0.
16: Setnnz(A_tile, local_id, eIdx_start);
17: // Load A_tile to registers. Omit index calculation.
18: asm("cvt.rna.tf32.f32 %RA, %A_tile");
19: // Load B_tile to registers. Omit index calculation.
20: asm("cvt.rna.tf32.f32 %RB, %B");
21: // Tensor cores compute. Four mma.m16n8k8 PTX APIs.
22: mma1688(RC[0], RC[1], RC[2], RC[3])
23:          RA[0], RA[1], RA[2], RA[3])
24:          RB[0], RB[1]);
25: mma1688(RC[0], RC[1], RC[2], RC[3])
26:          RA[4], RA[5], RA[6], RA[7])
27:          RB[2], RB[3]);
28: mma1688(RC[0], RC[1], RC[2], RC[3])
29:          RA[8], RA[9], RA[10], RA[11])
30:          RB[4], RB[5]);
31: mma1688(RC[0], RC[1], RC[2], RC[3])
32:          RA[12], RA[13], RA[14], RA[15])
33:          RB[6], RB[7]);
34: // Write back.
35: AtomicWrite(C, RC);
```

processes four *mma* operations to produce partial results for the C_Tile, and each C_Tile aggregates the results from thread blocks 0 and 1 to ensure the correctness of the computation. This is why the kernel is required to use atomic operations during the write-back process (line 35 in Alg. 2).

b) Register remapping: As outlined in Alg. 2, within each thread block, four consecutive *mma.m16n8k8* instructions are used for computation. Although the existing work, such as DTC-SpMM, also utilizes four *mma* instructions per kernel, our register mapping strategy is entirely different, as illustrated in Fig. 7. In DTC-SpMM, Fig. 7(a), the four *mma.m16n8k4* instructions are relatively independent, sharing the same A_frag but producing four distinct C_frag. The advantage of this approach is that it reduces shared memory usage, as only A_frag needs to be loaded into shared memory, while B_frag is directly fetched from global memory using *float4*. This allows additional shared memory for double buffering and pipeline strategy, which can overlap the loading of A_frag with Tensor cores computations. However, the major drawback of this method is that atomic operations are required when writing back each of the four C_frag, significantly reducing kernel execution efficiency.

In our register mapping strategy, as shown in Fig. 7(b),

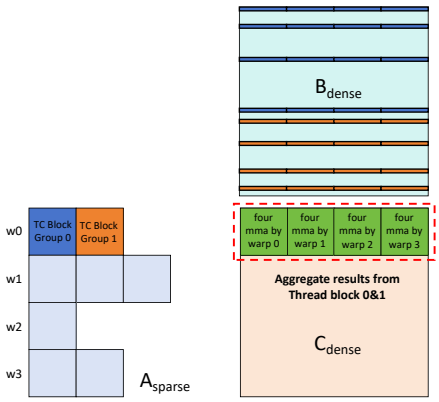


Fig. 6. Thread-block view of TC Block kernel in BRP-SpMM with load balance. (Each thread block is responsible for managing a *TC Block Group*.)

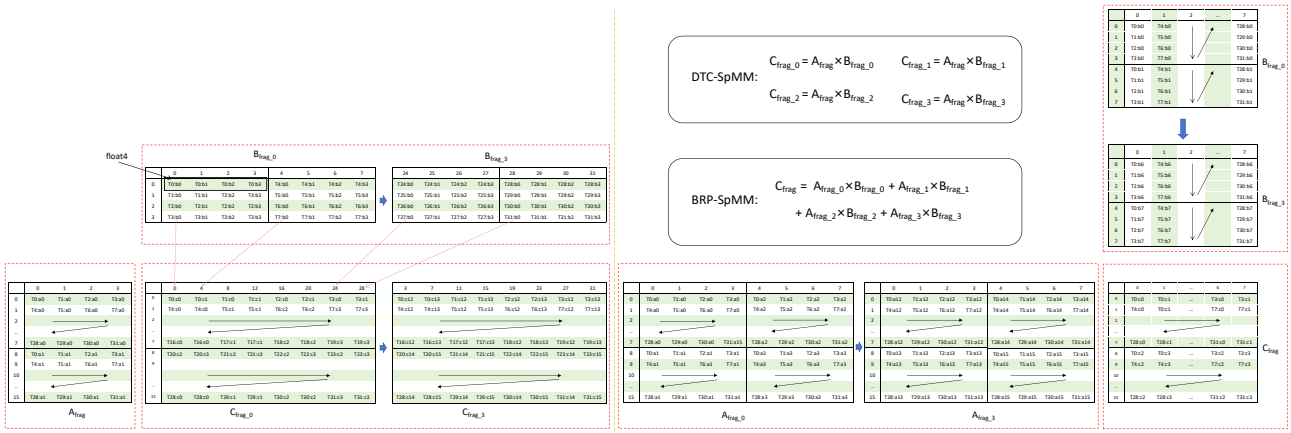
following the *TC Block Group* concept ($g = 4$), four *mma.m16n8k8* instructions are used to compute a single C_{frag}, utilizing four different A_{frag} and B_{frag}. It should be noted that the *mma.m16n8k8* instructions are executed in units of warps (32 threads). Each warp needs to load a 16×8 A_{tile} and a 8×8 B_{tile}, resulting in a 16×8 C_{tile}. When the data type is 32-bit, each thread requires 4, 2, and 4 registers to store A_{tile}, B_{tile}, and C_{tile}, respectively. In our approach, we invoke four *mma.m16n8k8* instructions consecutively since we set $g = 4$. Consequently, each thread requires 16, 8, and 4 registers to store A_{tile}, B_{tile}, and C_{tile} (lines 11 and 12 in Alg. 2). Additionally, we utilize the register RC to accumulate intermediate computation results of C_{tile}, thereby reducing the number of atomic write-back operations. Overall, the number of atomic operations in our method is only $1/4$ that of DTC-SpMM.

However, a limitation of our approach is the increased shared memory usage, as four A_{frag} fragments are loaded into shared memory simultaneously. In our implementation, the shared memory usage per thread block is approximately 2KB, which is acceptable for the A800 GPU. Additionally, we forgo the use of double buffer and pipeline strategies to prevent further increases in shared memory consumption and to keep the program size more compact. In practice, the program size of our TC Block kernel is only one-third that of the DTC-SpMM kernel.

E. Residual Row kernel

In the BRP algorithm, the number of non-zero elements in the Residual Row part is significantly less than in the TC Block part, and the number of non-zero elements per row window is always less than 32 (we set $g \times w = 4 \times 8$). Therefore, to maximize the utilization of thread resources, we follow the one-dimensional tiling strategy in Sputnik [27]. Alg.3 presents the implementation of our Residual Row GPU kernel.

The one-dimensional tiling strategy means that when building the thread block, we only set the number of threads along the *threadIdx.y* direction to N, while fixing *threadIdx.x* to 1. During computation, the *res_col_id* and *values* required for



(a) DTC-SpMM. Four *mma.m16n8k4*.

(b) BRP-SpMM (Ours). Four *mma.m16n8k8*.

Fig. 7. Register mapping strategy. (T0:a0 denotes thread 0 and register a0, indicating that thread 0 loads the corresponding elements into register a0.)

each residual row are loaded into shared memory, with each thread computing a single value in the output matrix C. This implies that each thread utilizes the data from one row of the Residual Row part in the sparse matrix and the corresponding column of the dense matrix. Thus, N threads compute one row and N columns of the output matrix, achieving maximum utilization of the thread resources. The *res_col_id* and *values* data in shared memory are reused N times. In this kernel, the load balance strategy is not considered, as the non-zero elements for each residual row are relatively small and do not vary significantly. Meanwhile, line 5 in Alg.3 demonstrates the role of the *res_rows_id* array in reducing invalid kernel launches. Besides, to simplify the expression, we omitted the values array in Alg.2 and Alg.3.

It should be noted that the TC Block kernel and the Residual Row kernel are executed serially on the GPU, with the TC Block kernel completing its execution before the Residual kernel starts. Within the Residual Row kernel, the final results from both parts are aggregated. Therefore, we use the '+=' operator for the write-back operation in the Residual Row kernel (line 22 in Alg.3).

Algorithm 3 Residual Row GPU kernel pseudo code.

```
1: Input: res_rows, row_offset, col_id, Dense Matrix B
2: Output: Matrix C
3: // Calculate indices.
4: bid = blockIdx.x;
5: x = res_rows[bid];
6: // One-dimensional thread block.
7: tid = threadIdx.y;
8: // Allocate shared memory.
9: __shared__ col_id_smem[32];
10: // Start and end index calculate.
11: start_idx = row_offset[bid];
12: end_idx = row_offset[bid+1];
13: // Fill shared memory.
14: col_id_smem[tid] = col_id[start+tid];
15: // Perform computations utilizing CUDA cores.
16: result=0;
17: for i = start_idx to end_idx do
18:   offset = col_id_smem[i]*N+tid;
19:   result += B_dev[offset];
20: end for
21: // Write back. Aggregate results from TC Block part.
22: C[x*N + tid] += result;
```

V. Evaluation

We evaluate BRP-SpMM in comparison with four representative works: (1) cuSPARSESpMM [16], the most widely used SpMM method from the vendor's cuSPARSE library (v12.4); (2) Sputnik [27], a state-of-the-art sparse library designed for deep learning, which leverages CUDA cores on GPU; (3) TCGNN-SpMM [33], a pioneering work focused on SpMM in GNNs using Tensor cores; and (4) DTC-SpMM [22], a state-of-the-art implementation of SpMM on Tensor cores. These works can be grouped by their computing units: cuSPARSE and Sputnik utilize CUDA cores, while TCGNN-SpMM and DTC-SpMM rely on Tensor cores.

The experiments are conducted on an NVIDIA A800 GPU (Ampere architecture, CC 8.6, with 80GB memory). The codes are compiled using NVCC from CUDA 12.4. Each test is repeated 100 times, and the average results are presented.

Our experiments utilize eight widely-used graph datasets from the Deep Graph Library (DGL) [12], with a detailed summary of their statistics presented in Table I. It is worth noting that all datasets exhibit sparsity levels exceeding 99%.

TABLE I
Statistics of Datasets

Dataset	Rows	nnz	Dataset	Rows	nnz
CoauthorCS	18K	164K	Questions	49K	307K
PubMed	20K	89K	Flickr	89K	899K
AmazonRatings	24K	186K	ogbn-arxiv	169K	1.16M
CoauthorPhysics	34K	496K	Yelp	717K	14.0M

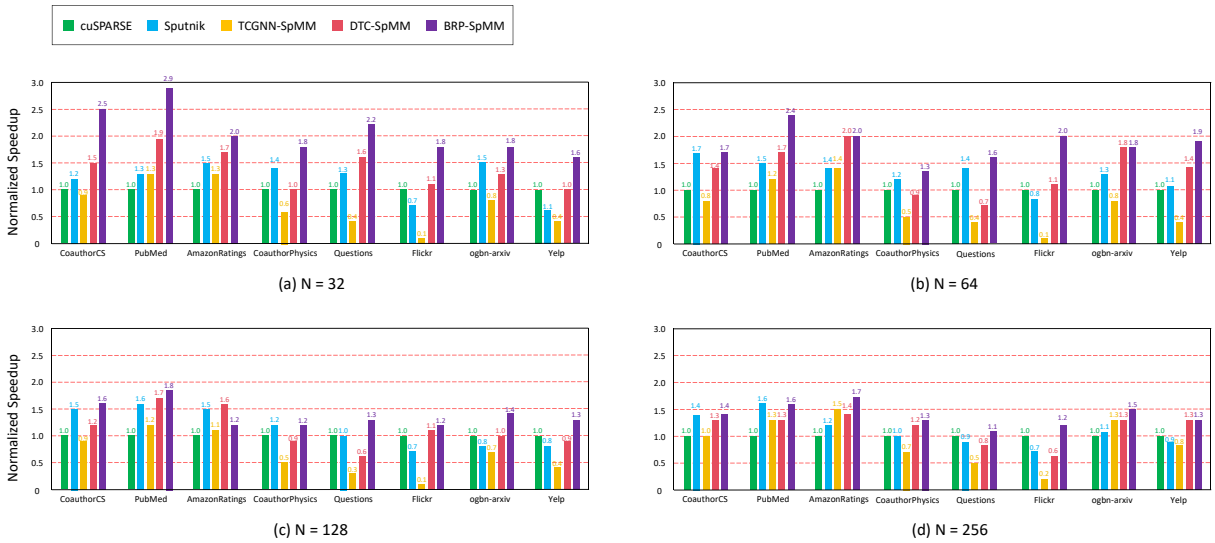


Fig. 8. SpMM performance comparison on A800 GPU. (Normalized speedup to cuSPARSE. N is the columns of dense matrix B.)

A. SpMM performance comparison

As shown in Fig. 8, we present normalized speedup compared to cuSPARSE with different N settings (the number of columns of dense matrix B), including 32, 64, 128, and 256.

The overall results demonstrate that BRP-SpMM delivers significant performance improvements over four representive works. Notably, when N=32, BRP-SpMM achieves an average speedup of 2.1× compared to cuSPARSE, with a maximum speedup of 2.9× on the PubMed dataset. Additionally, compared to the SOTA DTC-SpMM, BRP-SpMM achieves an average speedup of 1.2× across all scenarios, with a maximum speedup of 1.8× on the Flickr dataset when N=64.

In detail, Sputnik demonstrates relatively stable performance, achieving an average normalized speedup of 1.2×. In contrast, TCGNN-SpMM shows the poorest performance among all methods, delivering only 0.75× of cuSPARSE on average, and in the worst case, on the Flickr dataset, it reaches a speedup of just 0.1×, meaning it is 10× slower than cuSPARSE. DTC-SpMM, the state-of-the-art among the four baseline works, achieves an average speedup of approximately 1.3× to cuSPARSE. This highlights that using Tensor cores to process sparse matrices with high sparsity requires extensive optimization of the kernel function; otherwise, performance degradation may occur. It should be noted that Sputnik exclusively utilizes CUDA cores, while DTC-SpMM relies solely on Tensor cores. When comparing these two works, we find that DTC-SpMM does not gain a substantial lead, indicating that the computational power of CUDA cores should not be overlooked. Furthermore, BRP-SpMM achieves average speedups of 2.1×, 1.8×, 1.4×, and 1.4× compared to cuSPARSE when N=32, 64, 128, and 256, respectively. We observe that our method achieves optimal performance when N=32. This is attributed to the thread block configuration of 32×4, where each thread block contains four warps, with each warp processing 8 columns of the output matrix using the *mma.m16n8k8* PTX API. As a result, thread resources are maximally utilized when N=32.

Nevertheless, it is important to highlight that cuSPARSE and Sputnik have the advantage of requiring no additional preprocessing overhead and supporting standard COO and CSR formats. In contrast, TCGNN-SpMM, DTC-SpMM, and BRP-SpMM require extra format conversion overhead.

B. Preprocessing overhead

The only preprocessing overhead in this paper is format conversion cost, specifically converting from commonly used CSR formats to our customized format. Since both BRP-SpMM and DTC-SpMM are built on TCGNN-SpMM, we compare the preprocessing overhead of these three works in Table II. TCGNN-SpMM relies on the CPU for format conversion, resulting in the longest overhead, whereas both DTC-SpMM and our approach leverage the GPU, achieving significant acceleration. The preprocessing overhead of our work is the lowest across all eight datasets. Overall, we observe that as the dataset size increases, the advantage of our work becomes more pronounced. On the largest dataset, Yelp, the format conversion speed of our work is 212.9× faster than TCGNN-SpMM and 1.9× faster than DTC-SpMM.

In our BRP approach, preprocessing is executed concurrently across all row windows. In the TC Block part, the local row index is obtained by performing a column-major sorting of the non-zero elements. Since the number of non-zero elements in each TC Block is fixed, this significantly reduces the time spent on index calculation. A single thread block handles a row window. For the residual part, we eliminate empty rows and apply compression using the CSR format. In DTC-SpMM, the primary source of time consumption in preprocessing is the

TABLE II
PREPROCESS OVERHEAD STATISTICS. (THE VALUES IN PARENTHESES INDICATE THE SPEEDUP COMPARED TO TCGNN-SPMM.)

Datasets	Preprocess overhead (ms)		
	TCGNN-SpMM (Device: CPU)	DTC-SpMM (Device: GPU)	BRP-SpMM (Device: GPU)
Co.CS	129.81	2.57 (50.5×)	2.32 (60.0×)
PubMed	75.85	1.58 (48.0×)	1.10 (69.0×)
Ama.Rat.	157.44	2.40 (65.6×)	2.01 (78.3×)
Co.Phy.	394.82	6.08 (64.9×)	3.84 (102.8×)
Questions	264.01	5.57 (47.4×)	2.95 (89.5×)
Flickr	945.75	16.00 (59.1×)	6.08 (155.5×)
ogbn-arxiv	1392.59	9.50 (146.6×)	8.26 (168.6×)
Yelp	15076.68	131.20 (114.9×)	**70.82 (212.9×)**

calculation of TCLocalId, which accounts for over 50% of the total time. This is largely due to the difficulty in determining the number of non-zero elements in each TC block, resulting in a complex process for identifying thread index.

Thus, the *Fixed Non-zero* and *TC Block Group* strategies not only enhance the computational efficiency of SpMM but also improve preprocessing speed, achieving dual benefits.

C. Detailed comparison of two parts

In this segment, we perform a detailed comparison between the TC Block part and the Residual Row part. As shown in Figure 9, we separately evaluate the proportion of non-zero elements within each part as well as the corresponding kernel execution time. The results clearly show that the TC Block part significantly occupies a dominant position both in terms of the non-zero element proportion and kernel execution time.

Upon closer observation, for non-zero elements, the average proportion of the TC Block part is 88.5%, while for kernel execution time, the average proportion of the TC Block part is 85.9%, which is only slightly less than the former. This indicates that our proposed BRP partitioning method effectively performs task division, thereby maximizing the utilization of GPU computing resources.

However, we have also identified a potential issue: as the dataset size increases, the proportion of non-zero elements in the Residual Row part tends to decrease. For instance, in the largest dataset, Yelp, this proportion drops to around 5%. This is due to our BRP algorithm setting fixed values for g and w at 4 and 8, respectively. As a result, within each window, i.e., every 16 rows of the matrix, the Residual Row part can contain a maximum of 31 non-zero elements. In large datasets, this constraint leads to an overly small proportion of non-zero elements in the Residual Row part. This issue can potentially be mitigated by adopting an adaptive f and g value, but further research is required to determine the optimal configuration.

D. L2 cache hit rate

In GPUs, the L2 cache plays a crucial role in improving memory access efficiency. The L2 cache hit rate refers to the proportion of successful data retrievals from the L2 cache during computation tasks. A higher L2 cache hit rate implies that more memory requests are being served directly from the L2 cache, bypassing the need to access global memory. Therefore, the L2 cache hit rate is an important metric for evaluating the memory access efficiency of algorithms.

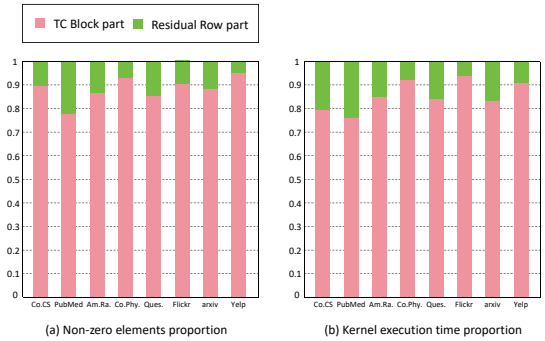


Fig. 9. Detailed comparison of TC Block part and Residual Row part.

Given the pivotal role of the TC Block kernel in our work, we conduct a comparative analysis of its L2 cache hit rate against cuSPARSE and DTC-SpMM, as shown in Fig. 10. The experimental results reveal that our TC Block kernel consistently demonstrates a remarkable advantage across eight diverse datasets. On average, our TC Block kernel outperforms cuSPARSE by 13.2% and DTC-SpMM by 8.6%, underscoring its superior performance. Notably, on the Yelp dataset, our TC Block kernel achieves an extraordinary enhancement of over 20% in L2 cache hit rate. This significant improvement not only reflects the efficacy of our kernel but also underscores the advantages of our Block-Row Partitioning (BRP) approach.

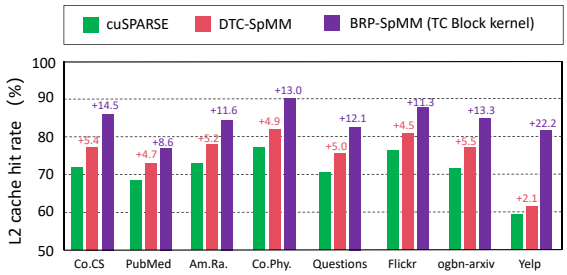


Fig. 10. L2 cache hit rate comparison. (N = 32. The numbers above the bars represent the growth compared to cuSPARSE.)

E. Storage Analysis

As shown in Fig. 11, we compare the memory space requirements for two widely used sparse formats, CSR and COO, along with three customized formats employed in TCGNN-SpMM, DTC-SpMM, and BRP-SpMM. The results demonstrate that the CSR format exhibits a clear advantage, requiring the least memory space, while TCGNN-SpMM, which performs the worst, necessitates the most memory. The difference between the other three methods is minimal. BRP-SpMM is very close to the COO format and, compared to the CSR format, requires at most 50% additional memory, consistent with our previous theoretical analysis in IV-C.

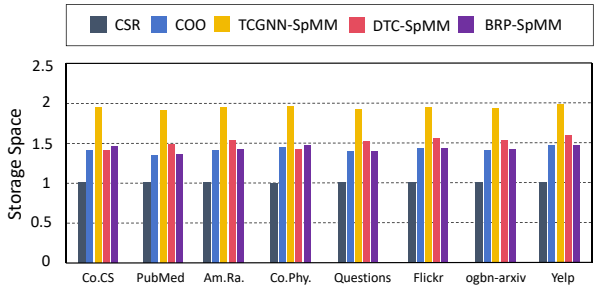


Fig. 11. Comparison of memory space requirements for different storage formats. (Normalized proportion to CSR format. Lower is better.)

F. Case Study: End-to-end Performance for GNN

BRP-SpMM is well-suited for end-to-end GNN training. We use the Graph Convolutional Neural Network (GCN) [26] model as the backbone. The forward propagation formula of GCN can be simplified as:

$$H^{(l+1)} = \sigma[A \times H^{(l)} \times W^{(l)}] \tag{1}$$

where σ is the activation function; A denotes the adjacency matrix of the graph; $H^{(l)}$ denotes the feature matrix and $W^{(l)}$ denotes the weight matrix in layer l. Since A is a sparse matrix, H and W are dense matrices, the operation $A \times H$ represents a typical SpMM operation.

We conduct an in-depth analysis of the source code for the Deep Graph Library (DGL v2.2.0) [12] and PyTorch-Geometric (PyG v2.6.0) [10] frameworks and identify a key difference in their handling of SpMM operations. DGL implements a customized *SparseMatrix* class to represent sparse matrices, while PyG directly utilizes the *SparseTensor* class from PyTorch [1]. To eliminate any potential impact caused by model architecture, we implement the basic two-layer GCN model using PyTorch. The only difference lies in the interfaces used for the SpMM operation, allowing us to isolate and evaluate the effect of different SpMM methods on GNN performance. The GCN model is evaluated on three representative graph datasets: Pubmed, ogbn-arxiv, and Yelp. BRP-GCN is compared with two popular GNN training frameworks, DGL and PyG, as well as two state-of-the-art methods, TC-GNN (which utilizes TCGNN-SpMM) and DTC-GCN (which utilizes DTC-SpMM). Fig. 12 presents a comparison of the overall training time for these methods. On A800 GPU, BRP-GCN achieves 1.13×, 1.51×, 1.36×, and 1.49× average speedups over DTC-GCN, TC-GNN, DGL, and PyG, respectively. Notably, BRP-GCN achieves up to 1.87× speedup over DGL on the Yelp dataset when N=128.

VI. Limitation

Our method is primarily constrained by the sparsity and dimension of the sparse matrix. In the BRP algorithm, we employed the *Fixed Non-zero* strategy and set f=1, meaning that each column in the TC Block contains only one non-zero element. This configuration may not be suitable for matrices with moderate sparsity, such as 50% to 80%. Additionally,

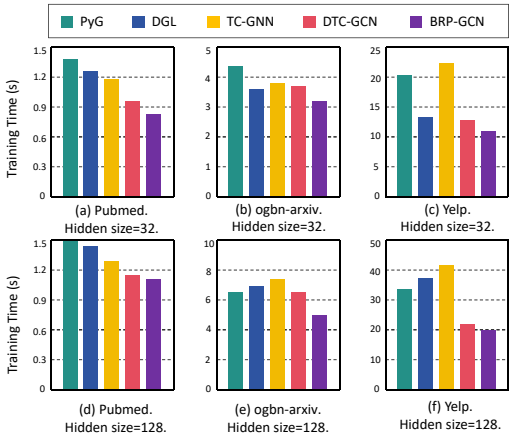


Fig. 12. End-to-end two-layer GCN training (500 epochs). The format conversion time is not excluded, as it is nearly negligible.

as the matrix dimension increases, the number of non-zero elements in the Residual row part becomes too small, limiting the effective utilization of the GPU's computational resources.

However, these constraints could potentially be mitigated by adjusting the values of f and g. One promising direction is introducing an adaptive approach that dynamically adjusts these parameters based on the number of non-zero elements in a given row window. Although this solution appears promising, it requires further investigation and research.

VII. Conclusion

In this paper, we propose BRP-SpMM, a novel block-row partitioning approach designed for efficient SpMM on GPU, specifically targeting sparse matrices with high sparsity. Our method divides the sparse matrix into two distinct parts, which are processed separately by the Tensor cores and CUDA cores, maximizing the utilization of GPU resources. Extensive experiments on modern A800 GPU have demonstrated that BRP-SpMM achieves substantial performance improvements compared to the state-of-the-art SpMM methods.

The most significant contribution of this work lies in offering a new perspective on how to efficiently process sparse matrix with high sparsity. By carefully partitioning the workload to fully exploit the computational power of modern GPU, we have advanced the state of research in sparse matrix operations. While our method shows strong performance, there is potential optimization for sparse matrices with different sparsity levels and large dimensions. In summary, BRP-SpMM sets a new benchmark for GPU-accelerated sparse matrix multiplication and opens up promising avenues for future research.

VIII. Acknowledgments

This research was supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501400, and the National Natural Science Foundation of China under Grant No. 62372199.

REFERENCES

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Neural Information Processing Systems (NeurIPS 2019)*, pages 8024–8035, 2019.

[2] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP 2019)*, pages 300–314. ACM, 2019.

[3] F. Vázquez, E.M. Garzón, and J.J. Fernández. A matrix approach to tomographic reconstruction and its implementation on GPUs. *Journal of Structural Biology*, 170:146–151, 2010.

[4] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2020)*, pages 1–12. IEEE/ACM, 2020.

[5] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett and Sid Samsi. Sparse Deep Neural Network Graph Challenge, 2019. IEEE HPEC. Available: https://ieeexplore.ieee.org/document/8916336/.

[6] Shui Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *2023 IEEE High Performance Extreme Computing Conference (HPEC 2023)*, pages 1–7. IEEE, 2023.

[7] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse Matrices in MATLAB: Design and Implementation. *SIAM journal on matrix analysis and applications*, 13(1):333–356, 1992.

[8] Kiran Koshy Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. Attention-based Graph Neural Network for Semi-supervised Learning. *arXiv preprint arXiv:1803.03735*, 2018.

[9] Eunji Lee, Yoonsang Han, and Gordon Euhyun Moon. Exploiting Tensor Cores in Sparse Matrix-Multivector Multiplication via Block-Sparsity-Aware Clustering. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2024)*, pages 1181–1183. IEEE, 2024.

[10] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[11] Ming Dun, Xu Zhang, Huawei Cao, Yuan Zhang, Junying Huang, and Xiaochun Ye. Adaptive Sparse Deep Neural Network Inference on Resource-Constrained Cost-Efficient GPUs. In *Proceedings of the High Performance Extreme Computing Conference (HPEC 2023)*, pages 1–7. IEEE, 2023.

[12] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[13] Mohammed Heyouni and A. Essai. Matrix Krylov subspace methods for linear systems with multiple right-hand sides. *Numerical Algorithms*, 40:137–156, 2005.

[14] NVIDIA. Ampere GPU Architecture Whitepaper, 2020. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[15] NVIDIA. CUDA C Programming Guide, 2024. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[16] NVIDIA. cuSPARSE Library Documentation, 2024. Available: https://docs.nvidia.com.

[17] NVIDIA. cuSPARSELt: A High-Performance CUDA Library for Sparse Matrix-Matrix Multiplication, 2024. Available: https://docs.nvidia.com/cuda/cusparselt/.

[18] NVIDIA. PTX ISA 8.5: CUDA Toolkit Documentation, 2024. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[19] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. A Row Decomposition-based Approach for Sparse Matrix Multiplication on GPUs. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2024)*, pages 377–389, 2024.

[20] Paolo Sylos Labini, Massimo Bernaschi, Werner Nutt, Francesco Silvestri, and Flavio Vella. Blocking Sparse Matrices to Leverage Dense-Specific Multiplication. In *12th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3 2022)*, pages 19–24. IEEE, 2022.

[21] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *Proceedings of the 25th Symposium on Principles and Practice of Parallel Programming (PPoPP 2020)*, pages 376–388. ACM, 2020.

[22] Ruibo Fan, Wei Wang, and Xiaowen Chu. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)*, pages 253–267. ACM, 2024.

[23] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing Graph Neural Networks: A Survey from Algorithms to Accelerators. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2022.

[24] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient Quantized Sparse Matrix Operations on Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2022)*, pages 1–15. IEEE/ACM, 2022.

[25] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long Range Arena : A Benchmark for Efficient Transformers. *arXiv preprint arXiv:2011.04006*, 2020.

[26] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR 2017)*, pages 1–14, 2017.

[27] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2020)*, pages 1–14. IEEE/ACM, 2020.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Proceedings of the Neural Information Processing Systems (NeurIPS 2017)*, pages 5998–6008, 2017.

[29] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *Proceedings of the International Conference on Learning Representations (ICLR 2018)*, pages 1–12, 2018.

[30] Jie Xin, Xianqi Ye, Long Zheng, Qinggang Wang, Yu Huang, Pengcheng Yao, Linchen Yu, Xiaofei Liao, and Hai Jin. Fast Sparse Deep Neural Network Inference with Flexible SpMM Optimization Space Exploration. In *Proceedings of the High Performance Extreme Computing Conference (HPEC 2021)*, pages 1–7. IEEE, 2021.

[31] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient Transformers: A Survey. *ACM Computing Surveys (CSUR)*, 55(6):1–28, 2023.

[32] Yufei Sun, Long Zheng, Qinggang Wang, Xiangyu Ye, Yu Huang, Pengcheng Yao, Xiaofei Liao, and Hai Jin. Accelerating Sparse Deep Neural Network Inference Using GPU Tensor Cores. In *Proceedings of the High Performance Extreme Computing Conference (HPEC 2022)*, pages 1–7. IEEE, 2022.

[33] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC 2023)*, pages 149–164. USENIX, 2023.

[34] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for Longer Sequences. In *Proceedings of the Neural Information Processing Systems (NeurIPS 2020)*, pages 17283–17297, 2020.

[35] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient Tensor Core-Based GPU Kernels for Structured Sparsity under Reduced Precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2021)*, pages 1–14. IEEE/ACM, 2021.