Jigsaw: Accelerating SpMM with Vector Sparsity on Sparse Tensor Core

Kaige Zhang*, Xiaoyan Liu*, Hailong Yang, Tianyu Feng, Xinyu Yang, Yi Liu, Zhongzhi Luan and Depei Qian
School of Computer Science and Engineering, Beihang University, Beijing, China
{kaige.zhang,liuxiaoyan,hailong.yang,ty_feng,ltyxy,yi.liu,07680,depeiq}@buaa.edu.cn

ABSTRACT

As deep learning models continue to grow larger, model pruning is employed to reduce memory footprint and computation complexity, which generates a large number of sparse matrix-matrix multiplication (SpMM) with unstructured sparsity (e.g., vector sparsity). However, leveraging GPU especially the newly integrated sparse tensor core (SpTC) to accelerate SpMM is quite challenging due to the unstructured sparsity. Unfortunately, existing works fail to fully exploit the SpTC on GPU due to the difficulty of satisfying the stringent requirement for restricted sparsity (e.g., 2:4 sparsity). In this paper, we propose *Jigsaw*, a novel method to utilize SpTC for accelerating SpMM with vector sparsity. Specifically, we propose the *multi-granularity sparsity reorder* method to transform the sparse data for satisfying the sparse pattern supported on SpTC. In addition, we propose a *reorder-aware storage format* for the transformed sparse data to better adapt to the parallelism of SpTC. Moreover, we propose corresponding optimizations to better exploit the SpTC for further accelerating SpMM. The experiment results demonstrate that *Jigsaw* outperforms state-of-the-art SpMM implementations and achieves promising speedup over cuBLAS.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; • **Computing methodologies** → **Parallel programming languages**; • **General and reference** → **Performance**.

KEYWORDS

sparse matrix reordering, sparse tensor core, deep learning optimization, sparse matrix-matrix multiplication

ACM Reference Format:
Kaige Zhang*, Xiaoyan Liu*, Hailong Yang, Tianyu Feng, Xinyu Yang, Yi Liu, Zhongzhi Luan and Depei Qian. 2024. *Jigsaw*: Accelerating SpMM with Vector Sparsity on Sparse Tensor Core. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673108

∗ Both authors contributed equally to this paper.

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673108

1 INTRODUCTION

Deep neural network (DNN) models have achieved unprecedented success in a large number of applications, such as image classification [15], object detection [3], and natural language processing [6, 24]. Particularly, with the enabling of large language models (LLM), applications such as question-answering system [16], multi-modal data processing [13], embodied intelligence [19], and personal intelligence assistant [21] have received abrupt adoption, impacting various aspects of human society. The success of LLM can be attributed to the massive number of model parameters (e.g.,1 trillion with GPT-4), which has led to a significant demand for computation power. To address the ever-increasing computation demand, researchers and practitioners resort to model compression techniques to reduce the computation and memory requirements while maintaining model accuracy, thereby enhancing model inference efficiency.

Model pruning is one of the most popular and effective methods for model compression. Model pruning methods eliminate the unimportant weights by setting them to zero, and then transform the dense computation into sparse computation. For example, matrix multiplication, a commonly used operator in DL models [23, 24] and ML algorithms [18, 26], will be transformed into sparse matrix multiplication (SpMM) after pruning. In order to achieve structured sparsity with better data locality (and thus better computation performance) after pruning, a popular pruning method is to zero out the weights at block granularity [14]. Different pruning methods use different block shapes. Among all block pruning methods, 1D block pruning (i.e., vector pruning), has been proven to achieve a better tradeoff between sparsity and accuracy [9].

However, exploiting model sparsity by efficiently performing SpMM on mainstream inference platforms, such as GPUs, remains challenging. The architecture advantage of GPU relies on SIMT computation and coalesced memory access, which is not friendly to sparse computation due to its irregular data access and low computation intensity, especially for the fine-grained sparsity derived from vector pruning [4]. Vector pruning typically generates an unstructured distribution of non-zero vectors, which can easily lead to poor data locality and warp divergence. Moreover, the compression of sparse data leads to a large number of complex indirect indexing, introducing additional overhead. Therefore, the achieved performance of SpMM on GPUs is quite limited, and far from optimal. Especially for SpMM in DNNs, the performance potential is constrained due to the below-extremely-high sparsity of input matrix [8, 28].

Meanwhile, various hardwares have been proposed to realize the performance potential of sparse DNNs by accelerating sparse computation such as SpMM [12, 17, 25]. Recently, NVIDIA GPUs

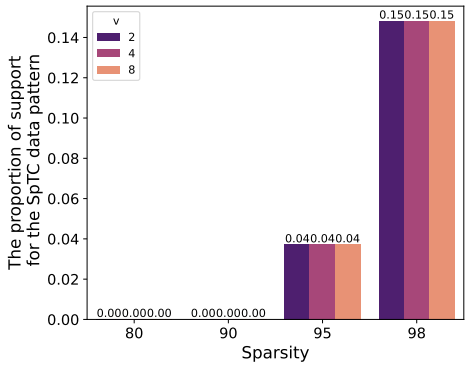

Figure 1: The proportion of matrices in the DLMC [7, 14] dataset that support for the SpTC sparse pattern. The vector width v is 2, 4, and 8

introduced the Sparse Tensor Core (SpTC), that is specifically designed for a particular class of sparse computations on Ampere and afterwards architectures. To use SpTC, it requires the left-handed matrix to follow 50% sparsity with a 2:4 structured pattern. For instance, FP16 SpTC supports at most two non-zero elements within every four consecutive elements in one row. If the above sparse pattern is satisfied, SpTC can transform the SpMM into a smaller matrix multiplication and achieve 2× speedup compared to the dense Tensor Core [17, 20].

Unfortunately, it is very difficult to exploit the performance advantage of SpTC for accelerating SpMM due to the restricted sparse pattern supported. Based on our observation of Google sparse dataset DLMC [7, 14], even for matrices with 98% sparsity, the proportion of matrices satisfying the 2:4 sparse pattern remains quite low, which only reaches around 15%, as shown in Figure 1. Existing works [2, 5, 28] cannot efficiently utilize the SpTC to further improve the performance of SpMM, leaving scarce computation resources on table. The above limitation results in a large fraction of sparse DNNs can yet benefit from the SpTC for improved inference performance.

In order to utilize SpTC, existing works have to co-design the pruning approach for specific sparse patterns, such as VENOM [2] and DFSS [5]. However, for a large number of sparse models, especially sparse models, it is still challenging to effectively utilize SpTC for acceleration. Additionally, works like SparTA [28] decompose sparse matrices into parts that can be accelerated using SpTC and the remaining parts, and then accelerate the outputs of the two kernels to get the final result. However, this approach introduces additional overhead due to kernel decomposition and may generate plenty of underutilized SpTCs (e.g., the proportion of non-zero elements in a single SpTC is much less than 50%).

To address the limitations of existing works in utilizing SpTC acceleration for vector sparsity, we have to tackle the following three challenges: *1)* how to transform unstructured sparse data to meet SpTC sparsity requirements while improving SpTC utilization, *2)* how to design the sparse format for transformed sparse data to ensure computational correctness while keeping the sparse format GPU-architecture friendly, and *3)* how to design the SpMM kernel to fully exploit GPU parallelism and bandwidth.

In this paper, we propose *Jigsaw*, a novel method to fully utilize SpTC for vector-sparse SpMM acceleration. To solve the first challenge, we propose *multi-granularity sparsity reorder*, transforming the data to satisfy the SpTC sparsity requirement by column reordering while maximizing the SpTC utilization and skipping the computation of all-zero column vectors. To solve the second challenge, we propose a *reorder-aware storage format*, using a multi-granularity approach to store the transformed data hierarchically to facilitate different parallel levels of the GPU to access different granularities of data. We also propose corresponding optimizations to solve the third challenge, including bank conflict elimination, overlapping memory and computation, and optimizing the metadata loading for SpTC.

Specifically, this paper makes the following contributions:

- We propose the *multi-granularity sparsity reorder*. We reorder the sparse data by two granularities. First, at coarse granularity, we tighten the non-zero vectors together, and then we reorder the vectors at SpTC granularity to satisfy the 2:4 requirements.
- We propose the *reorder-aware storage format* that stores sparse matrices hierarchically to adapt the parallelism hierarchy for GPU. We also record the information before the reordering to ensure the computation's correctness.
- We implemented *Jigsaw* with a series of GPU optimizations, including bank conflict elimination, overlapping memory and computation, and optimizing the metadata loading for SpTC. The experiment results demonstrate that *Jigsaw* achieves superior performance compared to SOTA SpMM implementations and cuBLAS.

2 BACKGROUND

2.1 GPU Architecture and Optimization

GPU Computation Hierarchy - We use NVIDIA GPU as an example to introduce the architecture of GPU. In the CUDA programming model, the computing hierarchy of a GPU can be divided into grids, thread blocks, and threads. A grid is a collection of thread blocks, and a thread block is a collection of threads. The entire GPU is organized based on these three levels of computation. GPUs comprise stream multiprocessors (SMs). For example, the A100 GPU has 108 SMs. The entire grid of a kernel will be executed on these 108 SMs. Each thread block is assigned to execute on one SM. Each SM can execute multiple thread blocks simultaneously, e.g., 32 thread blocks in A100. However, the actual number of thread blocks that can coexist in an SM is usually less than the limit due to restrictions on the shared memory and register usage of the thread blocks. Every group of 32 threads is called a *warp*. Each SM has four warp schedulers, which are responsible for selecting ready warps from thread blocks and issuing instructions from them. Threads within a warp execute the instruction concurrently. When the current warp experiences delay lasting multiple clock cycles (warp stalls) due to memory access overhead or resource contention, the warp scheduler can opt to execute other warps instead. If no warp is available for scheduling within the current SM, the entire SM enters a stall state, resulting in wasted hardware resources on the GPU.

GPU Memory Hierarchy - The memory hierarchy of a GPU can be divided into global memory, shared memory, and registers. Global memory has the largest space but comes with high access latency. Shared memory and registers offer faster access speeds, but the physical resources limit the amount available to each thread block and thread. In the GPU, DRAM serves as global memory. The data accesses to DRAM are cached in the L2 cache. All SMs can access both the DRAM and the L2 cache. Furthermore, within each SM, there is an L1 cache. The L1 cache is a memory accessed independently within the SM. A part of the L1 cache is designated as shared memory, exclusively accessible within the thread block. In A100, the amount of shared memory used by each thread block is limited to 164KB. Each thread has its independent register resources, with a maximum allocation of 256 registers per thread.

GPU Performance Optimization - Fully utilizing GPUs' memory hierarchy and bandwidth is an essential optimization for accelerating sparse computation. Here, we concisely introduce two effective optimizations.

1) Avoiding Shared memory bank conflict: shared memory is divided into 32 banks based on its address, each responsible for four consecutive bytes of memory. Accessing shared memory involves these 32 banks. Each bank can only read or write data from its designated region once at a time. If multiple accesses within a single memory access hit the same bank, the access will be split into multiple memory accesses to the bank. This splitting of accesses significantly reduces the memory bandwidth available for concurrent operations. This scenario is known as *bank conflict*. Ideally, accessing shared memory involves accessing consecutive memory addresses that cover all 32 banks of shared memory. This setup eliminates bank conflicts and maximizes memory coalescing, fully utilizing memory bandwidth.

2) Overlapping data access and computation: in architectures prior to A100, the double buffering technique was commonly used to overlap data access and computation, as certain warps accessing memory could schedule other warps ready to perform calculations. However, in this approach, accessing global memory still required assistance from registers. Specifically, data from global memory would first be stored in registers before being transferred to shared memory. This method consumes register resources required for computation and still needs to stall within warps, thus not achieving true parallelism. The A100 architecture introduces asynchronous copy instructions, which support direct data transfer from global memory to shared memory without the register's help. As a result, it can effectively overlap memory access and computation, reducing the number of warp stalls.

2.2 Sparse Tensor Core

Since the Ampere architecture, NVIDIA GPUs have introduced the third generation of tensor core units, known as the sparse tensor cores, providing hardware acceleration for 2:4 structured sparsity. After compressing 2:4 structured sparse matrices, data occupancy, and bandwidth are reduced by half, and sparse tensor cores double computational throughput by skipping zeros.

The sparse tensor cores can only be issued by inserting the Parallel Thread eXecution (PTX) assembly MMA calls. In contrast, dense tensor cores support both assembly insertion and legacy `wmma.mma` instruction. Figure 2 illustrates the PTX instruction `mma.sp` assembly format for invoking sparse tensor cores, where A, B, C, and D represent fragments of the sparse matrix, dense matrix, and output matrix, respectively. E denotes the metadata, and F selects the thread ID providing the metadata. Figure 3 illustrates a matrix following the 2:4 sparse pattern. By removing the zero values from the matrix and compressing it, the size reduces from $M \times K$ to $M \times K/2$. Metadata preserves the position of each nonzero element within the original four consecutive elements. For example, for the first row of the matrix, metadata should be (0, 3) and (1, 2). During computation, the hardware uses a selector based on metadata to choose elements from matrix B that match the K/2 nonzeros.

Table 1: Ampere architecture supports sparse tensor core

Precision	Supported shapes
tf32	m16n8k16, m16n8k8
f16/bf16	m16n8k16, m16n8k32
u8/s8	m16n8k32, m16n8k64
u4/s4	m16n8k64, m16n8k128

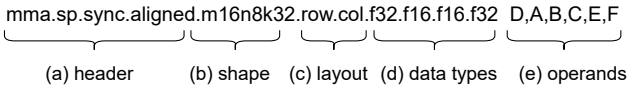

Figure 2: The `mma.sp` PTX instruction performs $D = A \times B + C$ SpMM. A contains the compressed sparse matrix with nonzero values of structured 2:4 data pattern. B is dense matrix. Operand E is the sparse metadata containing the column indexes of nonzero values in A, and operand F selects the thread ID providing the metadata.

As shown in Table 1, the sparse tensor core supports various sparsity precisions and shapes. For the fp16 type, two options are supported: m16n8k16 and m16n8k32. For the m16n8k32 type of MMA, both fragment A and fragment B require a register vector of length four of the uint32 type. According to the microbenchmarks conducted by this study [22], experiments demonstrate that the m16n8k32 type of sparse tensor core can maintain the same latency and bandwidth as dense MMA of the same size. However, the m16n8k16 size tensor core instead decreases the overall throughput. Therefore, we ultimately choose the m16n8k32 type of sparse tensor core.

3 METHODOLOGY

3.1 Overview

Figure 4 illustrates the overview of *Jigsaw*, including *multi-granularity sparsity reorder*, *reorder-aware storage format*, and kernel design and optimizations. We formulate the sparse matrix multiplication as $C = A \times B$, where the output matrix C has a shape of $M \times N$, the Left-Hand-Side(LHS) **sparse** matrix A has a shape of $M \times K$, and the Right-Hand-Side(RHS) **dense** matrix B has a shape of $K \times N$.

Multi-granularity sparsity reorder - We reorder the sparse data by two granularities, *BLOCK_TILE* (purple box in Figure 4 (a)) and *MMA_TILE* (yellow box in Figure 4 (a)). First, at the *BLOCK_TILE*

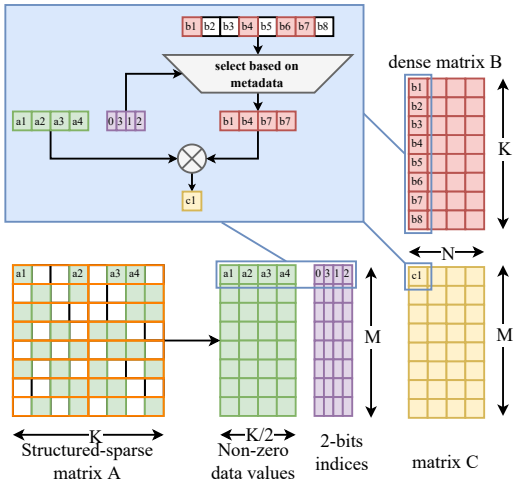

Figure 3: Sparse $M \times N \times K$ GEMM using SpTC

granularity, we move the zero columns (white box in Figure 4 (a)) to the end, allowing us to skip the computation of zero columns to reduce the computational workload for SpMM. Second, we tile *BLOCK_TILE* into *MMA_TILE* granularity and then perform column reorder inside each *MMA_TILE*. After reordering, each *BLOCK_TILE* will follow the SpTC sparse pattern. In *Jigsaw*, each block computes *BLOCK_TILE*, and each SpTC mma operation computes *MMA_TILE*. The details of *multi-granularity sparsity reorder* will be introduced in Section 3.2. Due to the stationary of sparse weight matrices during inference, the reorder only takes one-time light preprocessing, whose cost can be amortized over inferences.

Reorder-aware storage format - We compress the reordered sparse matrix by *reorder-aware storage format*. Specifically, we use *col_idx_array* to store the original column indexes before the *BLOCK_TILE* granularity reorder, as indicated by the red array in the Figure 4 (b). Then, we use *block_col_idx_array* to store original column indexes before the *MMA_TILE* granularity reorder, as indicated by the yellow array in the Figure 4 (b). Finally, we use *sptc_col_idx_array* to store the metadata used by SpTC. The *reorder-aware storage format* will be introduced in detail in Section 3.3.

Kernel design and optimizations - Figure 4 (c) illustrates the computation process of the *Jigsaw* kernel. Each thread block is responsible for a matrix C tile of size *BLOCK_TILE_M* $\times$ *BLOCK_TILE_N*. Within each thread block, each warp is responsible for matrix C tile of size *WARP_TILE_M* $\times$ *WARP_TILE_N*. Each warp will perform MMA operations multiple times.

We also explore the memory hierarchy of GPU to optimize memory access. During the computation, we will load the B tile (colored in red) needed by the thread block into shared memory. Note that each *MMA_TILE* requires the same data (but with a different column order) of B inside one thread block. Thus, matrix B can be used by different warps in one thread block. To leverage SpTC, we need to store data of A and B in fragments. Note that sparse data A is reordered at the granularity of *MMA_TILE*, we load matrix B data into fragments according to the reordering information stored in our sparse format data. After the warp completes the computation,

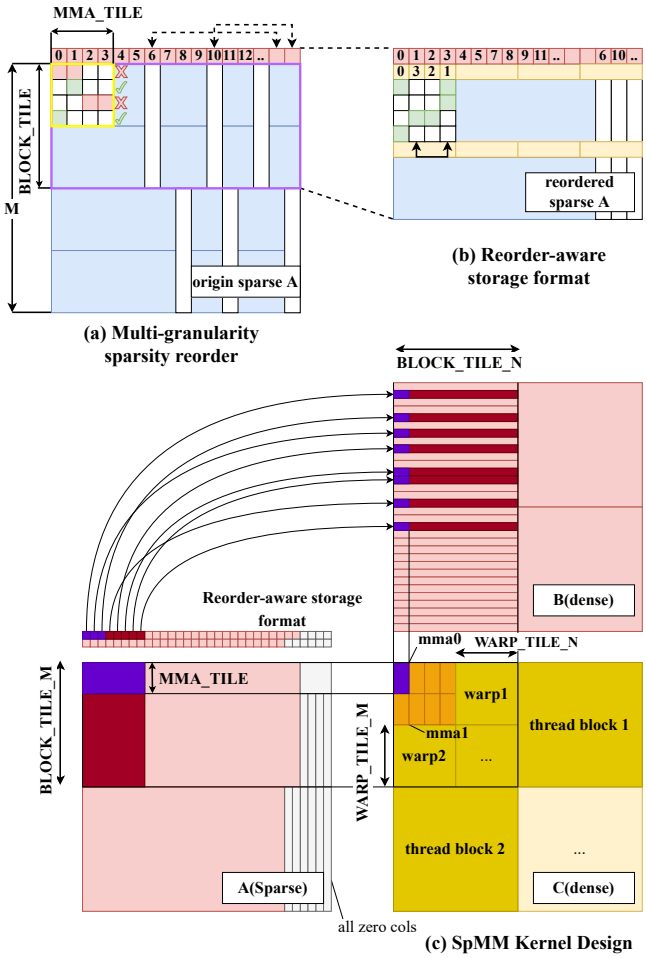

Figure 4: Overview of *Jigsaw* (We use 1:2 sparsity for more concise illustration, the actual sparse pattern of SpTC is 2:4)

we store the matrix tile C in registers and directly write them back to GMEM.

We propose several optimizations to improve the performance of data loading and computation. Our optimizations include employing pipelining to break the data dependency and overlap the memory and computation process, introducing efficient bank conflict elimination optimization, and exploring interleaving metadata loading patterns. The detailed optimizations will be introduced in Section 3.4.

3.2 Multi-granularity Sparsity Reorder

The *multi-granularity sparsity reordering* reorders the sparse matrix A by two granularities, *BLOCK_TILE* and *MMA_TILE*.

First, we perform column reorder on the sparse matrix A at the *BLOCK_TILE* granularity to reduce the number of calls to SpTC and also improve the utilization of SpTC. As illustrated in Figure 5 (a), we first extract the zero columns and place them at the end of the *BLOCK_TILE*, thereby skipping a portion of data computation by SpTC and avoiding utilizing SpTC to compute the data whose

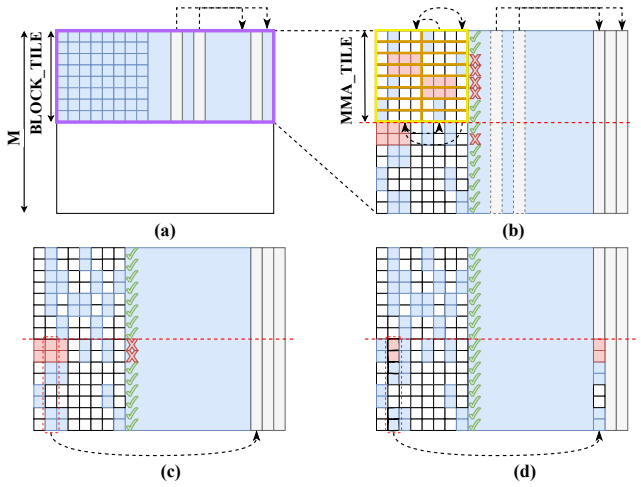

Figure 5: *Multi-granularity sparsity reorder* (*MMA_TILE*=8 × 8 and SpTC data requirement of 2:4 for example)

sparsity is higher than 50%. After removing the zero columns, we compress the remaining columns in this *BLOCK_TILE* and update the *col_idx_array* of the *reorder-aware storage format*.

Then, we perform column reorder at the *MMA_TILE* granularity to perform a *MMA_TILE* that meets the requirements of SpTC, as shown in Figure 5 (b). In Figure 5, we use *MMA_TILE* in as 8 × 8 for illustration, while we use *MMA_TILE* in 16 × 16 in our implementation.

Algorithm 1 shows the *MMA_TILE* granularity reorder algorithm, using a *MMA_TILE* in 16 × 16 for example. Firstly, we combine each of the four columns into a group, resulting in four groups in total. For each group, if the number of non-zero elements in each row does not exceed 2, then this group meets the 2:4 sparse pattern, and we refer to such columns group that follow the 2:4 sparse pattern as a *compatible column group*. If all groups in *MMA_TILE* are *compatible column group*, then the entire *MMA_TILE* meets the 2:4 sparse pattern requirements of SpTC. Thus, we need to search the possible combinations as much as possible to find enough *compatible column group* to achieve the success of the *MMA_TILE* granularity reorder. While achieving this goal, we also need to reduce the complexity of the search in order to keep the reorder overhead within acceptable levels.

Firstly, we need to identify all possible *compatible column groups* that meet the SpTC 2:4 sparse pattern. Specifically, we iterate through all combinations of four columns and determine their compatibility based on the number of non-zero elements in each row (line 2-8 in Algorithm 1). After finding all possible *compatible column groups*, we need to select some of them to cover the entire 16 columns of *MMA_TILE*. A simple and intuitive method is to iterate through all *compatible column groups* to find a valid solution. However, considering the high time complexity of this method, we use a bidirectional search for acceleration. First, we iterate all disjoint *compatible column groups* with four columns and combine them into *compatible column groups* with 8 columns (line 9-13). Then, we find two disjoint *compatible column groups* with

Algorithm 1 *MMA_TILE* reorder

Input: *MMA_TILE* A[16][16]
Output: reordered *MMA_TILE* A[16][16]
1: Get all possible *compatible column groups*
2: **for** any 4 different columns i, j, k, w in A **do**
3: Bool compatibleFlag ← 1
4: **for** 16 rows from 1 to 16 **do**
5: **if** more than two nonzeros in A[row][i], A[row][j], A[row][k], A[row][w] **then**
6: compatibleFlag ← 0
7: **if** compatibleFlag = 1 **then**
8: these four columns form a *compatible column group*
9: Using bilateral search to find the final divide solution
10: **for** every two possible *compatible column groups* (group g_1 and group g_2) **do**
11: **if** no common columns between g_1 and g_2 **then**
12: get new *compatible column group* with 8 columns
13: $G = [g_1, g_2]$
14: **for** any two *compatible column group with 8 cols* (group G_1 and group G_2) **do**
15: **if** $|G_1 \cup G_2| = 16$ **then**
16: column_reorder(*MMA_TILE*, G_1, G_2)
17: **return** Reorder Success
18: **return** Reorder Failure

8 columns and reorder the *MMA_TILE* (line 14-17). Figure 5 (b) to (c) shows a successful reorder for the upper *MMA_TILE* inside the Figure.

If we cannot find any disjoint *compatible column groups* with 8 columns, the algorithm fails to reorder the *MMA_TILE*. To tame this problem, we propose a reorder retry method. We will try to move the column that appears least frequently in all *compatible column groups* with 4 columns to the end and set it to an all-zero column in *BLOCK_TILE*, as shown in Figure 5 (c) to (d). Then, we will reapply the reorder algorithm to see if it succeeds. If it does not, we will continue to move a column until the reordering algorithm is successful. To ensure that SpMM still executes correctly after moving the columns, we use *col_idx_array* and the *block_col_idx_array* in the *reorder-aware storage format* to record the moved columns. After the reordering algorithm, all *MMA_TILE* meets the SpTC data requirement.

3.3 Reorder-aware Storage Format

We propose the *reorder-aware storage format* to compress the sparse matrix after the *multi-granularity sparsity reordering*.

The *reorder-aware storage format* comprises three levels of index arrays. Precisely, it consists of the top-level *col_idx_array*, the middle-level *block_col_idx_array*, and the innermost *sptc_col_idx_array* (or *metadata*). The top-level *col_idx_array* stores the column indices of non-zero columns in *BLOCK_TILE* granularity. The middle-level *block_col_idx_array* stores each column's indices within *MMA_TILE* granularity. The innermost *sptc_col_idx_array* stores the column indices of each non-zero element after compression within *MMA_TILE*.

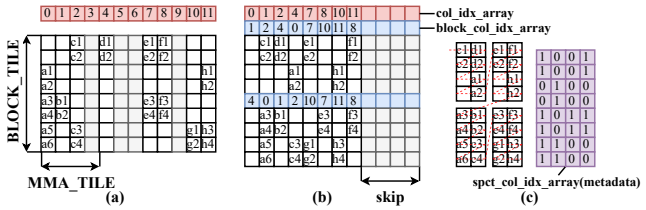

Figure 6: *Reorder-aware storage format* (*MMA_TILE*=4 × 4 and SpTC required sparse pattern of 1:2 for example)

Figure 6 shows an example of *reorder-aware storage format*. The red array of Figure 6 (a) represents the original matrix's column indices. During the *BLOCK_TILE* granularity reorder of *multi-granularity sparsity reorder*, the all-zero columns 3, 5, 6, and 9 are placed at the end of *MMA_TILE*. We then record the indices of the remaining columns in *col_idx_array*, as shown in the red arrays in Figure 6 (b).

Next, we record the reordered columns in the *block_col_idx_array* after the *MMA_TILE* granularity reorder, as shown in the blue array in Figure 6 (b). In this example, after *BLOCK_TILE* granularity reorders, columns 7, 8, 10, and 11 compose one *MMA_TILE*, where elements $e1$ in column 7 and $f1$ in column 8 do not meet the 1:2 sparse pattern requirement. The *MMA_TILE* granularity reorder algorithm changes the columns reorder into 7, 10, 11, and 8, which meets the sparse requirement.

Finally, we compress the reordered sparse data inside each *MMA_TILE* and generate *metadata* used by SpTC. As shown in Figure 6 (c), we generate *metadata* of 0 or 1 for each non-zero element based on its position in the original two elements in *sptc_col_idx_array*. The compressed sparse matrix will be stored contiguously, as depicted in Figure 6 (c), assuming that each 4 × 2 matrix block is stored continuously in a Z-shaped swizzle pattern, for example. In our implementation, each compressed 16×8 matrix block is continuously stored in a Z-shaped swizzle pattern.

3.4 Kernel Optimization

3.4.1 Bank Conflict Elimination. We eliminate bank conflicts caused by writing to and reading from matrix *B* in shared memory. Figure 7 (a) shows that each thread block reads a 64×64 matrix *B*. The matrix *B* is stored in global memory in row-major format. Thus, matrix *B* in shared memory is also stored in row-major format to utilize vectorized memory access and apply asynchronous memory access. Tiles of different colors represent 8 half-type elements, each corresponding to 4 different shared memory banks. In `mma.sp.m16n8k32`, each warp needs to read a 32×8 tile *B* from shared memory, which can be achieved using a `ldmatrix.x4` instruction. The `ldmatrix.x4` instruction reads in 4 stages, loading 8×8 tile *B* each stage. We have to ensure no bank conflict within each 8×8 tile *B*. To avoid bank conflicts during data retrieval, we pad 4 banks after each row, ensuring that the 8×8 tile B covers 32 banks.

The reorder at the *MMA_TILE* granularity will change the row order of tile *B*, as shown in Figure 7 (b). Ideally, `ldmatrix` reading rows 0-7 would not cause bank conflicts. However, if the row order after reordering is 0, 2, 3, 8, 9, 12, 5, 6, and 15, conflicts will occur

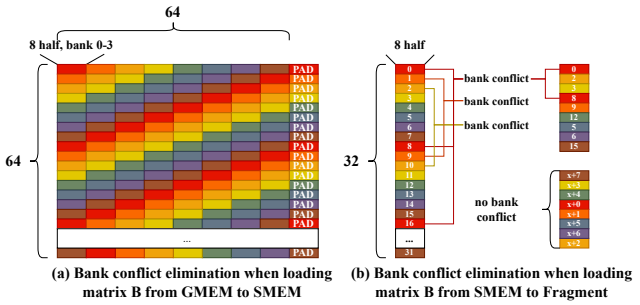

Figure 7: Bank conflict elimination

between 0 and 8. Similarly, conflicts will also arise if rows 1 and 9, 2 and 10, 3 and 11, etc., form *compatible column groups* with 8 columns. To address this issue, we identify and prefer reorder schemes at the *MMA_TILE* granularity that do not cause bank conflicts, avoiding grouping rows from the same bank into the same *compatible column group*.

3.4.2 Overlap Memory and Computation Using Pipelining. Memory and computation overlapping is a common optimization to improve GPU instruction throughput. Unfortunately, establishing a two-stage pipeline naively for loading matrices and SpTC computation is inefficient for SpMM. Since matrix *A* is reordered at the *BLOCK_TILE* granularity, matrix *B* needs to be loaded into SMEM according to the *A*'s column indices. Consequently, we access matrix *B* indirectly through *col_idx_array*. Thus, when loading matrix *B*, severe warp stall can occur if *col_index_array* is not ready in SMEM, leading to poor overlapping efficiency.

We propose to deepen the pipeline to relieve the data dependency between *col_idx_array* and matrix *B*. Figure 8 illustrates the deepened pipeline for data loading and computation, where dashed lines represent data dependencies. During the current computation iteration at step *n*, matrices *A* and *B* at step (*n*+1) will be loaded into shared memory, while the *col_index_array* for step (*n*+2) will be loaded. Throughout the data loading process, we use 128-bit memory access instructions and coalesce memory accesses to multiples of the L1/L2 cache line size to minimize cache line wastage. We also utilize CUDA asynchronous copy to improve the overlapping efficiency.

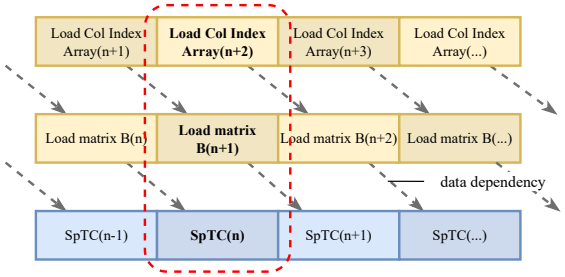

Figure 8: Overlapping memory and computation using pipeline

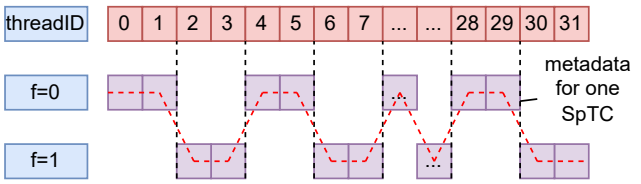

Figure 9: Metadata interleave loading pattern of instruction `mma.sp`

3.4.3 Efficient Metadata Interleave Loading Pattern. In SpTC with the 2:4 sparse format, every four elements contain two non-zero elements, recorded by two column indices. Each column index is represented using a 2-bit compressed number format. Thus, `m16n8k32` MMA has 16×16 column indices (metadata used by SpTC). After compression using 2 bits, those column indices can be stored in 16 integers. To use SpTC, we need to load the metadata to the register of each thread by a specified pattern. As shown in Figure 9, taking F=0 in instruction `mma.sp` for example, only thread 0, 1, 4, 5, ..., 28, and 29 need to load metadata, leading to extra branch instruction and warp diverge. To avoid the branch, one naive way is making other threads load useless data, which results in throughput waste.

To tame this problem, we change the layout of *sptc_col_idx_array* and use `ldmatrix` to load the metadata of SpTC. We find that by changing F, 32 threads can provide metadata for two SpTC operations. As shown in Figure 9, we store the metadata needed for two SpTC operations continuously in an interleaved pattern, where each thread is mapped to the metadata of 32-bit length (one integer). In the *reorder-aware storage format*, we organize metadata in the *sptc_col_idx_array* based on the interleaved pattern. Specifically, the metadata required for two consecutive SpTC operations is stored in 32 integers. Thus, we can leverage one `ldmatrix` to load 32 integer metadata needed by twice SpTC, which optimizes memory throughput. After that, we call the `mma.sp` with F=0 and F=1 to ensure that the two instructions use the corresponding metadata.

4 EVALUATION

4.1 Experimental Setup

Our experiments are conducted on a platform equipped with two NVIDIA A100 GPUs(108 Ampere SMs, 40GB). The CPU of the platform is Intel(R) Xeon(R) Gold 6336Y CPU. Similar to prior works [1, 4, 14], we construct benchmarks from the DLMC [7] sparse dataset, replacing each nonzero element with a 1-D vector with different width.

We compare *Jigsaw* with SOTA SpMM optimization implementations, including CLASP [1], Magicube [14], Sputnik [8] and SparTA [28], as well as with the dense counterpart cuBLAS. Specifically, CLASP [1] proposes the column vector format. The length of the dense vector stored in the format, represented by a private vector(pv), supports 2, 4, and 8. In CLASP, different pv have a significant impact on CLASP performance. Therefore, for all evaluation scenarios, we execute CLASP with pv=2, 4, and 8 and select the best result as its performance. For cuBLAS, we use cuBLASHGemm for its performance. For Sputnik [8], the sparse matrix is converted to CSR format. We use the half-precision version of Sputnik. For Magicube [14], we choose the L16-R16 version. For SparTA [28], we implement a half-precision version of the kernel based on cuSparseLt and Sputnik. Specifically, we split the matrix into two parts, one part following the 2:4 sparse pattern and the remaining part, and execute both parts of the kernel using cuSparseLt (FP16) [17] and Sputnik (FP16), respectively. To better tradeoff between parallelism and shared memory usage, we empirically tune the size of *BLOCK_TILE* (16, 32, and 64) to achieve the best performance. For the above tile sizes, the shared memory usage per thread block is 21.25KB, 24.83KB, and 27.65KB, respectively.

Table 2: Average and maximum(avg/max) speedup *Jigsaw* achieved compared to cuBLAS and SOTA SpMM implementations

Sparsity	v	cuBLAS	CLASP	Magicube	Sputnik	SparTA
80%	2	0.77/1.27	1.13/1.97	2.90/6.47	1.91/3.84	1.56/3.14
	4	0.89/1.34	1.32/1.90	2.68/6.25	2.23/4.49	1.71/3.16
	8	1.00/1.67	1.38/1.90	1.75/2.50	2.71/5.25	1.77/2.85
90%	2	1.00/1.58	1.09/1.53	3.09/8.62	1.65/2.43	1.89/3.15
	4	1.13/1.95	1.26/1.60	2.77/6.14	1.91/3.46	1.99/2.98
	8	1.35/1.85	1.36/1.89	1.71/2.44	2.39/4.65	2.17/3.09
95%	2	1.19/1.73	1.08/1.55	3.03/7.40	1.46/2.09	2.18/3.04
	4	1.44/2.83	1.28/1.62	3.01/7.08	1.74/2.60	2.43/3.16
	8	1.78/4.12	1.34/1.77	1.70/2.56	2.11/3.83	2.68/3.59
98%	2	1.43/1.89	1.15/1.69	3.31/8.77	1.40/1.73	2.56/3.46
	4	1.72/4.14	1.28/1.76	3.22/8.43	1.60/2.38	2.81/3.61
	8	2.14/5.45	1.31/1.85	1.70/2.82	1.87/3.68	3.09/4.46

We utilize the **Duration** metric obtained from NVIDIA Nsight Compute as the execution time of the kernel (excluding format conversion time, kernel launch time, and CPU-GPU data transfer time). Nsight Compute employs the same configuration for executing *Jigsaw* and other baselines, locking the GPU at a fixed frequency to ensure stable results across all evaluation scenarios.

4.2 Performance Improvement

Figure 10 illustrates their speedup compared to cuBLAS on the A100 GPU. Our kernel's average and maximum speedups compared to cuBLAS and sparse implementations for different sparsities and column vector lengths(v) are shown in Table 2.

As for the performance of sparse implementations, the results indicate that as the sparsity increases, most kernels' performance improves. This is because, with increasing sparsity, more workload can be skipped. For CLASP [1], it surpasses cuBLAS at approximately 95% sparsity. SparTA [28] leverages the high-performance kernel of cuSparseLt [17], building upon Sputnik's foundation. However, its performance does not consistently surpass that of Sputnik. Specifically, SparTA's performance is not directly affected by sparsity. When sparsity is low, SparTA exhibits superior performance to Sputnik due to the high utilization of cuSparseLt, particularly in SpTC operations. However, as sparsity increases, cuSparseLt's SpTC utilization decreases, resulting in more redundant computations in SparTA. Consequently, SparTA's overall performance may deteriorate compared to Sputnik as sparsity increases.

As the length of the column vector(v) increases, the speedup of *Jigsaw* becomes obvious. For *Jigsaw*, a larger column vector increases the likelihood of forming all-zero columns. Therefore, with v=8, more workload can be skipped compared to v=2. CLASP uses `mma.m8n8k16`, and when v=8, the MMA utilization rate can

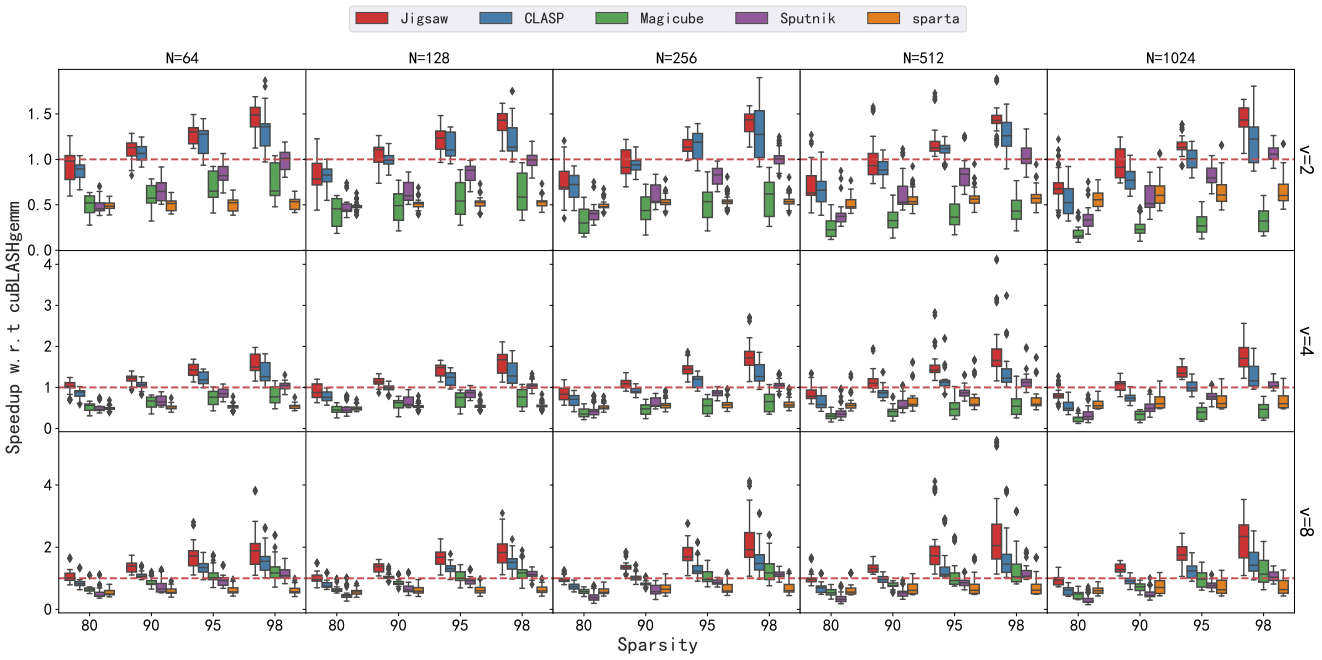

Figure 10: The Performance of SpMM on A100. The reported speedup is normalized to cuBLASHgemm. N is the number of columns of the output matrix

reach 100%; when v=4, the utilization rate is only 50%; and when v=2, the utilization rate is only 25%. Therefore, a larger v yields better performance.

As shown in Table 2, at v=8, Magicube's performance is significantly improved. Nsight Compute shows that Magicube is additionally optimized for v=8, with an average reduction of 50% in shared memory bank conflict, an average reduction of 10% in the total number of instructions, and an average reduction of 50% in inter-instruction waits, as compared to the case of v=4 and v=2. However, our kernel further reduces the number of instructions by 85% and the wait time between instructions by a further 50%. This still allows us to achieve twice the maximum acceleration ratio.

We identify and analyze the outliers shown in Figure 10. We found that when M=2048 and K=2048, cuBLAS shows 3× performance degradation between N=256 and N=512. This performance degradation leads to the appearance of outliers. We analyze the cuBLAS kernel using Nsight Compute. We find that cuBLAS launches 6× more than the expected number of thread blocks, causing kernel issues with a large number of memory access requests, resulting in significant warp stalls and performance degradation.

Sputnik is developed on the V100 architecture. Our experiments exhibit significant performance gaps compared to cuBLAS. This is primarily due to the introduction of faster tensor cores and efficient asynchronous memory access instructions in the A100 architecture. Sputnik does not fully leverage the performance acceleration provided by the hardware resources, thus achieving performance similar to cuBLAS only at 98% sparsity.

CLASP offers a SOTA fine-grained SpMM implementation, which extends the support of vectorSparse [4] to the Ampere architecture [20]. However, its performance still exhibits gaps compared to cuBLAS. In most cases, *Jigsaw*' average performance surpasses CLASP, except for specific instances of sparsity at N=256 and v=2. This is because CLASP and *Jigsaw* launch a few thread blocks when the matrix size is small, failing to utilize all SMs on the A100. In CLASP, each thread block is responsible for a smaller matrix block size than *Jigsaw*, resulting in CLASP launching more thread blocks and better-utilizing hardware resources. However, CLASP and *Jigsaw* saturate the hardware resources as the matrix size increases. Due to the smaller matrix block size per thread block in CLASP, data reuse is poorer compared to *Jigsaw*. Therefore, *Jigsaw* exhibits better overall performance.

4.3 Multi-granularity Sparsity Reorder Performance

Assuming that reordered data can satisfy the 2:4 sparse data pattern while maintaining the K no bigger than the original matrix (without severe reorder retry), we define it as a successful reorder. We selected sparse matrices from the DLMC [7] random pruning dataset with sparsity levels ranging from 80% to 98% and processed them according to the data processing methods described in Section 4.1. We then calculated the success rates of these data after being reordered using *multi-granularity sparsity reorder* at different granularities of BLOCK_TILE = 16, 32, and 64, as shown in Figure 11.

A critical factor affecting the success rate of the entire matrix reorder is the handling of failures at the *MMA_TILE* granularity reorder. When a reorder fails at the *MMA_TILE* granularity, as

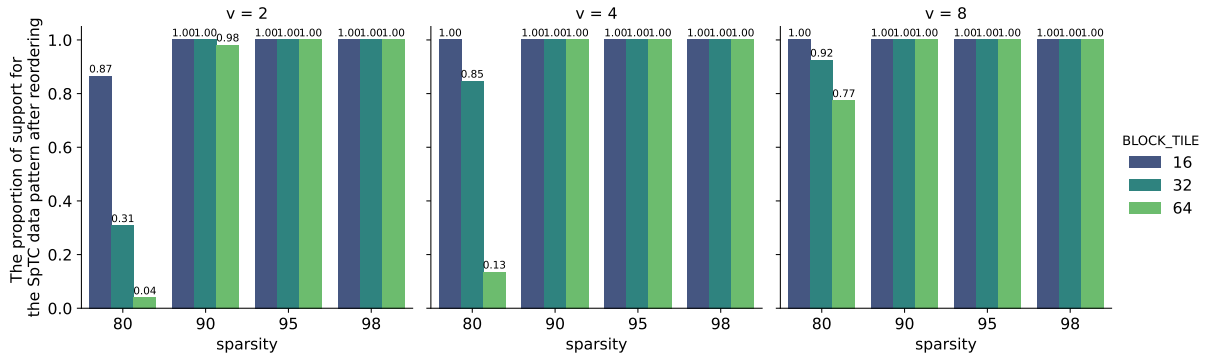

Figure 11: The proportion of support for the SpTC data pattern after reordering

depicted in Figure 5 (c), we choose the least compatible column and place it after the *BLOCK_TILE*. Consequently, as sparsity increases, the matrix's all-zero columns increase, allowing for more tolerance for failures in rearranging matrices at the *MMA_TILE* granularity.

At 80% sparsity, we find that as *BLOCK_TILE* increases, the success rate of reordering decreases. This is because it is more challenging to form all-zero columns when *BLOCK_TILE* increases, reducing the number of all-zero columns extracted by *BLOCK_TILE*. Additionally, as v decreases, the reorder success rate decreases. This is because as v decreases, the length of the 1-D vector decreases, making it more challenging to form all-zero columns, thus reducing the success rate of rearrangement.

Moreover, the size of the matrix, especially the number of columns in sparse matrices, also significantly impacts the success rate of rearrangement. As the number of columns increases, more all-zero columns are obtained, resulting in a higher success rate of rearrangement. We analyzed the failure cases of rearranging matrices with 80% sparsity, v=2, and *BLOCK_TILE*=16 and found that in such cases, the size of K does not exceed 128 (in the DLMC dataset, K ranges from 64 to 4,608).

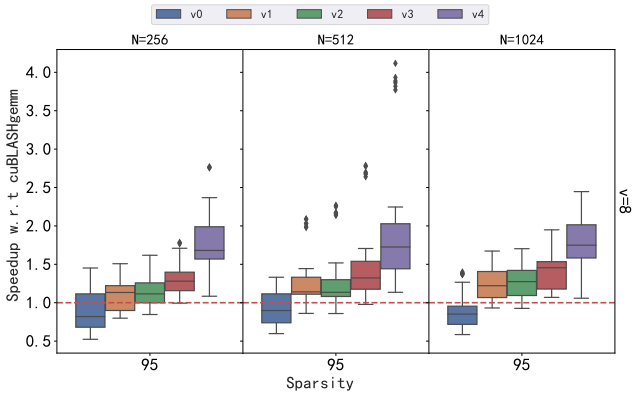

Figure 12: Ablation experiment of SpMM on A100. The reported speedup is normalized to cuBLASHgemm. V0 to V4 versions are detaild in 4.4

4.4 Ablation Study

In this section, we demonstrate the effectiveness of our optimization through ablation experiments. We select data from the DLMC [7] dataset with sparsity levels of 95% for evaluation, replacing each non-zero element with a 1-D vector of v=8. The evaluation results are shown in Figure 12. With this ablation study, we can see that all the optimization methods discussed in Section 3.4 are very effective.

We introduce the kernels used in the ablation study. The v0 version of the kernel is the basic kernel but does not apply the *shared memory bank conflict elimination* optimization. The v1 version further eliminates the shared memory bank conflict by padding matrix B in shared memory. The v0 and v1 versions both employ asynchronous copy to parallelize memory access and computation but do not deepen the pipeline further to break the data dependency between accessing *col_idx_array* and matrix B. The v2 version employs our *overlap communication and calculation using pipelining* optimization based on v1. The v3 version incorporates the *efficient metadata interleave loading pattern* on top of v2. Kernels for v0, v1, v2, and v3 versions only support *BLOCK_TILE*=64, while the kernels used in Section 4.2 supports *BLOCK_TILE*=16, 32, and 64. We consider the results from Section 4.2 as the v4 version of the kernel.

Considering the kernel's v0, v1, v2, and v3 versions, there is an overall improvement in average performance. Specifically, the v0, v1, v2, v3, and v4 versions of our kernel can ultimately achieve an average performance of 0.89×, 1.20×, 1.23×, 1.40× and 1.82× speedup relative to cuBLAS.

We use Nsight Compute to examine the effectiveness of *shared memory bank conflict elimination* optimization. We select evaluation scenarios with 95% sparsity, v=8, and M=512, N=512, K=512. Because this scale of the problem is large enough to distinguish the performance of various kernel versions and demonstrate the optimization effect. In v1, the shared memory bank conflicts are reduced by 99.48% compared to the v0 version, resulting in nearly 2× speedup, as shown in Figure 12.

Compared to v1, v2 breaks the data dependency between accessing *col_idx_array* and matrix B. As shown in the Nsight Compute profiling result, in v1, the metric *warp long scoreboard* is 1.82. In contrast, in v2, the *warp long scoreboard* is 0.87.

Table 3: Average speedup of *Jigsaw* compared to VENOM and cuSparseLt

	VENOM			cuSparseLt		
Sparsity/v	32	64	128	32	64	128
80%	1.91×	1.63×	1.50×	2.10×	2.12×	2.01×
90%	1.53×	1.37×	1.33×	2.16×	2.19×	2.08×
95%	1.32×	1.22×	1.21×	2.19×	2.21×	2.15×
98%	1.22×	1.14×	1.15×	2.31×	2.32×	2.28×

By comparing the v3 and v2, we demonstrate the effectiveness of the *efficient metadata interleave loading pattern* optimization. Nsight Compute profiling results show that in v3, the access instructions to shared memory are reduced by 7.78%. The *warp short scoreboard* metric decreased by 9.65%, indicating fewer warp stalls due to shared memory access. We stored metadata in *sptc_idx_array* according to an interleave pattern, allowing all threads to access SMEM continuously and coalesce efficiently.

V4 shows significant improvements compared to v3. This is because kernels v0, v1, v2, and v3 only support reordering with *BLOCK_TILE*=64, while v4 supports multiple sizes. When *BLOCK_TILE*=16 and 32, more zero columns can be skipped, resulting in noticeable performance gains.

4.5 Applying to Sparse Matrices without Reordering

We also evaluate the performance of *Jigsaw* on sparse matrices that satisfy the SpTC requirement without reordering. We compare *Jigsaw* with VENOM [2] and cuSparseLt [17], which both use pruning methods to accelerate SpMM with SpTC. By leveraging the pruning methods, both VENOM and cuSparseLt can achieve a 2:4 sparse pattern. For *Jigsaw*, the pruned matrices from these methods can be directly computed without reordering. Thus, we evaluate *Jigsaw*, VENOM and cuSparseLt on sparse matrices pruned by VENOM. As shown in Table 3, the results indicate that our method achieves better performance over VENOM and cuSparseLt across all settings. This is because *Jigsaw*'s GPU optimizations are more efficient. In addition, the *reorder-aware storage format* in *Jigsaw* supports better data reuse and is more conducive to parallel processing.

4.6 Overhead Analysis

In this section, we analyze the memory overhead introduced by our method. For simplicity, we do not calculate the saved memory usage after deleting blank columns. Assuming the original matrix size is $M \times K$ and of half-precision type. If using the dense storage format accepted by cuBLAS, the matrix occupies a memory space of $2M \times K$ bytes. Firstly, after reordering and compression, the size of the matrix becomes $\frac{M \times K}{2}$, occupying $M \times K$ bytes. Next is the overhead introduced by the *reorder-aware storage format*. The *col_idx_array* occupying $\frac{4M \times K}{BLOCK\_TILE}$ bytes of space. Furthermore, the *block_col_idx_array*, occupying $\frac{4M \times K}{MMA\_TILE}$ bytes of space. Finally, the *sptc_col_idx_array* occupying $\frac{M \times K}{8}$ bytes of space. Altogether, this amounts to $\frac{5M \times K}{8} + \frac{4M \times K}{BLOCK\_TILE} + \frac{4M \times K}{MMA\_TILE}$ bytes of space. We empirically set *BLOCK_TILE* to 16, 32, 64 with *MMA_TILE* is 16. For the above tile sizes, the total memory usage compared to dense representation is 56.25%, 50%, and 46.87%.

4.7 Applying to Wider Range of Sparsity

The sparsity ratio of the matrices evaluated in Section 4.2 ranges from 80% to 98%. When the sparsity ratio is greater than 80%, our *multi-granularity sparsity reorder* technique can skip most of the unnecessary computations, and reorder the sparse data into a 2:4 sparse pattern, thereby efficiently utilizing the SpTC to accelerate SpMM. Additionally, our *reorder-aware storage format* can effectively reuse sparse data, reducing the overhead of data reloading. Furthermore, the GPU optimizations adopted in our approach also help to achieve efficient computation. The above reason explains the performance speedup achieved by our approach for sparsity ratio larger than 80%.

To achieve performance speedup for a wider range of sparsity (e.g., below 80%), we need to augment the reorder algorithm and better utilize GPU cores in *Jigsaw* (e.g., utilizing dense tensor cores as well as CUDA cores). For denser data tile, we can use dense tensor cores, which does not require metadata generation and still achieves performance acceleration. For sparser data tile, even with a few non-zero elements, using SpTC can result in low resource utilization and inefficient computation. Alternatively, we can accelerate the sparser data tiles using CUDA cores. We leave the above improvements of *Jigsaw* for future work.

5 RELATED WORK

In scientific computing, numerous optimization efforts are being made to target sparse matrix multiplication. The NVIDIA cuSparse library provides a high-performance cuda-core SpMM kernel. ASpT [10] introduces the *adaptive tiling* approach, which partitions sparse matrices based on row similarity and uses shared memory to accelerate SpMM. However, the sparsity in deep learning and the size of matrices are relatively minor compared to those in scientific computing [8]. Therefore, approaches tailored for highly sparse matrices are unsuitable for deep learning scenarios.

Some efforts focus on optimizing SpMM for DL workloads. Gale et al. propose Sputnik [8], which utilizes the CSR data structure and introduces optimization methods such as the *1-D tiling scheme* and *Row-swizzle Load Balance* to achieve vectorized memory access and load balancing. Huang et al. propose GE-SpMM [11] to optimize GNN, employing the *warp merging* strategy to reduce redundant data loading. However, none of the above efforts has exploited tensor cores to accelerate SpMM on GPU, leaving the performance opportunity to further boost SpMM on table.

Using tensor cores is a promising approach to accelerate SpMM on GPUs. vectorSparse [4], CLASP [1], and cuBLAS utilize the Tensor Core, while VENOM [2] and cuSparseLt [17] leverage the Sparse Tensor Core. vectorSparse [4] proposes the *TCU-based 1-D Octet tiling* method, using vectorized memory access. However, it primarily targeted the V100 architecture, thus it outperformed cuBLAS on the A100 architecture only at a high sparsity level. CLASP [1] continued the work of vectorSparse [4] on the A100. TiledCSR [27] leverages the *Row Shuffle Algorithm* and dense tensor core to accelerate SpMM. These works fail to exploit the performance opportunities enabled by SpTC. VENOM [2] proposes pruning weight

matrices to meet the requirements of SpTC data format, however cannot be directly applied to arbitrarily fine-grained sparse data without pruning. *Jigsaw* enhances the utilization of SpTC acceleration for fine-grained sparse data through *multi-granularity sparsity reorder*, *reorder-aware storage format*, and SpMM kernel design. *Jigsaw* can directly apply to any fine-grained sparse matrix and achieve practical acceleration compared to cuBLAS.

6 CONCLUSION

In this paper, we introduce *Jigsaw*, a novel approach designed to maximize the utilization of SpTC for accelerating vector-sparse SpMM computations. Our method comprises three essential contributions, including *multi-granularity sparsity reorder*, *reorder-aware storage format*, and corresponding optimizations.

Through the performance evaluation on the sparse matrices with different sizes and sparsity from DLMC on an NVIDIA A100 GPU, we demonstrate that *Jigsaw* achieves practical speedup compared to cuBLAS and SOTA SpMM implementations. We believe this work will motivate future research on accelerating various sparsity patterns and dynamic sparsity for deep learning models.

ACKNOWLEDGMENTS

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB3001801), National Natural Science Foundation of China (No. 62322201, 62072018, U23B2020 and U22A2028), and the Fundamental Research Funds for the Central Universities (YWF-23-L-1121 and JKF-20240198). Hailong Yang is the corresponding author.

REFERENCES

[1] Roberto L Castro, Diego Andrade, and Basilio B Fraguela. 2022. Probing the efficacy of hardware-aware weight pruning to optimize the spmm routine on ampere gpus. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 135–147.

[2] Roberto L Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B Fraguela, and Torsten Hoefler. 2023. VENOM: A Vectorized N: M Format for Unleashing the Power of Sparse Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[3] Yukang Chen, Jianhui Liu, Xiangyu Zhang, Xiaojuan Qi, and Jiaya Jia. 2023. Voxelnext: Fully sparse voxelnet for 3d object detection and tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 21674–21683.

[4] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. 2021. 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*. 1–14.

[5] Zhaodong Chen, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and Yuan Xie. 2023. Dynamic n: M fine-grained structured sparse attention mechanism. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 369–379.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[7] Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The State of Sparsity in Deep Neural Networks. *arXiv e-prints* arXiv:1902.09574 (2019). https://arxiv.org/abs/1902.09574

[8] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[9] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[10] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

[11] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC41405.2020.00076

[12] Guyue Huang, Zhengyang Wang, Po-An Tsai, Chen Zhang, Yufei Ding, and Yuan Xie. 2023. RM-STC: Row-Merge Dataflow Inspired GPU Sparse Tensor Core for Energy-Efficient Sparse Acceleration. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 338–352.

[13] Jing Yu Koh, Daniel Fried, and Russ R Salakhutdinov. 2024. Generating images with multimodal language models. *Advances in Neural Information Processing Systems* 36 (2024).

[14] Shigang Li, Kazuki Osawa, and Torsten Hoefler. 2022. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[15] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*. 10012–10022.

[16] Antoine Louis, Gijs van Dijck, and Gerasimos Spanakis. 2024. Interpretable long-form legal question answering with retrieval-augmented large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 22266–22275.

[17] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378* (2021).

[18] Hatem Moumni and Olfa Hamdi-Larbi. 2021. An Efficient Parallelization Model for Sparse Non-negative Matrix Factorization Using cuSPARSE Library on Multi-GPU Platform. In *Algorithms and Architectures for Parallel Processing: 21st International Conference, ICA3PP 2021, Virtual Event, December 3–5, 2021, Proceedings, Part II*. Springer-Verlag, Berlin, Heidelberg, 161–177. https://doi.org/10.1007/978-3-030-95388-1_11

[19] Yao Mu, Qinglong Zhang, Mengkang Hu, Wenhai Wang, Mingyu Ding, Jun Jin, Bin Wang, Jifeng Dai, Yu Qiao, and Ping Luo. 2024. Embodiedgpt: Vision-language pre-training via embodied chain of thought. *Advances in Neural Information Processing Systems* 36 (2024).

[20] NVIDIA. 2021-1-30. NVIDIA NVIDIA Ampere Architecture Whitepaper. online. https://www.nvidia.com/en-us/data-center/a100/ Accessed March 30, 2021.

[21] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. arXiv:2304.03442 [cs.HC]

[22] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2023. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (Jan. 2023), 246–261. https://doi.org/10.1109/tpds.2022.3217824

[23] Aravind Vasudevan, Andrew Anderson, and David Gregg. 2017. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 19–24.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[25] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1083–1095.

[26] Jonghye Woo, Fangxu Xing, Jerry L Prince, Maureen Stone, Arnold D Gomez, Timothy G Reese, Van J Wedeen, and Georges El Fakhri. 2021. A deep joint sparse non-negative matrix factorization framework for identifying the common and subject-specific functional units of tongue motion during speech. *Medical image analysis* 72 (2021), 102131.

[27] Zeyu Xue, Mei Wen, Zhaoyun Chen, Yang Shi, Minjin Tang, Jianchao Yang, and Zhongdi Luo. 2023. Releasing the Potential of Tensor Core for Unstructured SpMM using Tiled-CSR Format. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 457–464.

[28] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. {SparTA}:{Deep-Learning} Model Sparsity via {Tensor-with-Sparsity-Attribute}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 213–232.