electronics

Article

Tensor Core-Adapted Sparse Matrix Multiplication for Accelerating Sparse Deep Neural Networks

Yoonsang Han [1], Inseo Kim [2], Jinsung Kim [2,*] and Gordon Euhyun Moon [1]

[1] Department of Computer Science and Engineering, Sogang University, Seoul 04107, Republic of Korea; han14931@sogang.ac.kr (Y.H.); ehmoon@sogang.ac.kr (G.E.M.)
[2] School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, Republic of Korea; inseo764@cau.ac.kr
* Correspondence: kimjsung@cau.ac.kr

Abstract: Sparse matrix–matrix multiplication (SpMM) is essential for deep learning models and scientific computing. Recently, Tensor Cores (TCs) on GPUs, originally designed for dense matrix multiplication with mixed precision, have gained prominence. However, utilizing TCs for SpMM is challenging due to irregular memory access patterns and a varying number of non-zero elements in a sparse matrix. To improve data locality, previous studies have proposed reordering sparse matrices before multiplication, but this adds computational overhead. In this paper, we propose Tensor Core-Adapted SpMM (TCA-SpMM), which leverages TCs without requiring matrix reordering and uses the compressed sparse row (CSR) format. To optimize TC usage, the SpMM algorithm's dot product operation is transformed into a blocked matrix–matrix multiplication. Addressing load imbalance and minimizing data movement are critical to optimizing the SpMM kernel. Our TCA-SpMM dynamically allocates thread blocks to process multiple rows simultaneously and efficiently uses shared memory to reduce data movement. Performance results on sparse matrices from the Deep Learning Matrix Collection public dataset demonstrate that TCA-SpMM achieves up to 29.58× speedup over state-of-the-art SpMM implementations optimized with TCs.

Keywords: sparse matrix multiplication; tensor cores; sparse deep neural networks; load balancing; data movement

check for updates

Citation: Han, Y.; Kim, I.; Kim, J.; Moon, G.E. Tensor Core-Adapted Sparse Matrix Multiplication for Accelerating Sparse Deep Neural Networks. *Electronics* **2024**, *13*, 3981. https://doi.org/10.3390/electronics13203981

Academic Editor: David Defour

Received: 12 September 2024
Revised: 5 October 2024
Accepted: 8 October 2024
Published: 10 October 2024

1. Introduction

Matrix–matrix multiplication is a key computation in the training of deep neural networks (DNNs), including convolutional neural networks (CNNs), graph neural networks (GNNs), and transformers. In practice, matrix–matrix multiplication involving weights and neuron values dominates over all computations in the training of DNNs [1,2]. Therefore, NVIDIA's Tensor Cores (TCs) have recently emerged to accelerate General Matrix–Matrix Multiplication (GEMM), also known as dense matrix–dense matrix multiplication, by increasing throughput and reducing memory footprint through the use of lower-precision floating-point formats [3–5]. Moreover, with the increasing size of DNNs, various pruning and sparsification techniques have been widely adopted to reduce the number of parameters and computational operations in DNNs [6,7]. Therefore, sparse matrix–dense matrix multiplication (SpMM) has become an essential computational kernel, occupying a substantial portion of the operations in sparse DNNs [8]. Furthermore, training GNNs typically requires performing numerous SpMM operations to multiply a large sparse matrix, derived from the adjacency matrix, with the feature vectors of nodes to aggregate neighbor information within the graph [9,10]. However, unlike GEMM, which can straightforwardly use tiling or blocking techniques to achieve high performance, SpMM is inherently performance-limited on GPUs. As the number of non-zero elements on the left-hand-side (LHS) sparse matrix varies irregularly across rows, achieving good load balance for SpMM on GPUs is challenging. Furthermore, as the non-zero elements are irregularly scattered, irregular memory access leads to poor performance due to reduced cache hit rates on GPUs.

Therefore, several previous studies have sought to reorder the sparse matrix based on its sparsity patterns to enhance data locality, which repeatedly accesses a specific set of memory locations within a short period [11,12]. Data locality is associated with the amount of data movement from/to memory, and improving data locality reduces data movement costs and increases data reuse. In terms of both energy and execution time, as technology trends have made the cost of data movement significantly higher than the cost of performing arithmetic operations on GPUs, improving data locality is crucial for accelerating the GPU kernel. The primary goal of sparse matrix reordering for SpMM is to rearrange the matrix to facilitate the efficient extraction of high-density sub-matrices. By extracting the high-density sub-matrices from the reordered sparse matrix, these sub-matrices can then be used to perform blocked dense matrix–dense matrix multiplications with the right-hand-side (RHS) dense matrix using TCs. However, reordering the sparse matrix is very computationally demanding because the sparsity patterns of non-zero elements in each row must be compared by measuring the similarity between rows. Furthermore, the overhead of the reordering process increases as the size of the matrix grows. In general, the preprocessing time required for reordering the sparse matrix can exceed the actual execution time required for performing the SpMM operation [9]. Therefore, reordering the sparse matrix incurs significant overhead for the entire SpMM operation. Moreover, for the adjacency matrix of a directed graph, where the same row and column indices represent the same node, reordering only the row indices of the adjacency matrix results in incorrect edge information between nodes. To perform accelerated SpMM without a reordering process, NVIDIA's cuSPARSE Block-SpMM API [13] can be used for blocked dense matrix–dense matrix multiplication utilizing TCs. However, cuSPARSE Block-SpMM requires storing the elements of the sparse matrix in blocks using the Block-Ellpack format, which can be considered additional overhead. Compared to the memory-efficient compressed sparse row (CSR) format, which stores only the non-zero elements in a sparse matrix, the Block-Ellpack format becomes increasingly inefficient in terms of memory usage as the sparsity and irregularity of the matrix increase. This is because the Block-Ellpack format requires storing all remaining zero elements in non-zero blocks, even if a block contains only one non-zero element.

In this paper, we propose a novel Tensor Core-Adapted SpMM kernel (called TCA-SpMM) that efficiently leverages TCs using the memory-efficient CSR format without requiring the reordering of the sparse matrix. The main goal of this paper is to explore the feasibility of using TCs for SpMM by fully exploiting their architectural characteristics, particularly their high throughput of floating-point operations. As TCs use fragments (i.e., sub-matrices) as input for GEMM computation, the data structures that store non-zero elements in CSR format cannot be directly used with TCs. The basic idea behind our approach is to utilize TCs for performing the dot product of two vectors within each innermost loop of the original SpMM algorithm using the CSR format. Hence, we first transform the dot product of two vectors into a smaller blocked matrix–matrix multiplication. After transforming the two vectors into matrices, the transformed matrices are used to perform accelerated matrix–matrix multiplication on TCs using the MMA (matrix multiply-and-accumulate) operation. As a result, the diagonal elements of the resulting matrix are accumulated to obtain the final output element for the dot product of the two vectors. In the context of GPU computing, the irregular distribution of non-zero elements in the sparse matrix makes parallel decomposition challenging, leading to load imbalance and a reduced degree of parallelism. It is obvious that the number of operations required for each dot product of a row vector in the LHS sparse matrix and a column vector in the RHS dense matrix is inconsistent, as the number of operations for each dot product depends on the number of non-zero entries in each row vector of the sparse matrix. Therefore, to achieve good load balancing across CUDA thread blocks, we optimize our GPU kernel such that each thread block either performs a partial dot product for a row with many non-zero entries or computes the full dot products for multiple rows with fewer non-zero entries. Another challenge in optimizing SpMM using TCs is increasing the utilization of TCs by minimizing data movement. Hence, in order to increase the arithmetic intensity of our parallelization approach, we judiciously utilize shared memory resources and memory coalescing techniques to ensure the optimal use of global memory bandwidth. Experimental results on a large number of highly sparse matrices from the Deep Learning Matrix Collection (DLMC) dataset show that our TCA-SpMM achieves up to 29.58× average speedup over various state-of-the-art SpMM implementations optimized with TCs.

In this paper, we explore the use of TCs for SpMM. To the best of our knowledge, this is the first work to optimize SpMM using TCs without requiring any preprocessing, such as sparse matrix reordering or compression techniques. The key contributions of this work are as follows:

- We present a novel Tensor Core-Adapted SpMM kernel that eliminates the need for sparse matrix reordering and utilizes the most memory-efficient CSR format.
- We optimized our TCA-SpMM to improve load balancing and reduce data movement costs for SpMM by increasing the utilization of TCs.
- We systemically analyzed the computational complexity to show the effectiveness of our TCA-SpMM that fully utilizes the high throughput of TCs.
- We conducted a comparative evaluation using various sparse matrices from the DLMC dataset to demonstrate that our TCA-SpMM outperforms existing state-of-the-art SpMM implementations.

This paper is organized as follows. Section 2 provides background on sparse matrix representation, SpMM, and TCs. Section 3 presents prior studies related to optimizing SpMM with TCs. Section 4 provides an overview of our TCA-SpMM and details the parallelization strategies. In Section 5, we compare our TCA-SpMM with existing state-of-the-art SpMM implementations using TCs.

2. Background

2.1. Sparse Matrix Representation

In order to efficiently store the non-zero elements in a sparse matrix, various sparse matrix formats can be used [14,15]. The compressed sparse row (CSR) is one the most commonly used sparse matrix formats to store sparse matrices [16–18]. The CSR format maintains three data structures, *row_ptr*, *col_idx*, and *value*, for storing the indices and values of non-zero elements in a sparse matrix S of size M × K, as shown in Figure 1. For example, *row_ptr*[*i*] consists of the index of the first non-zero element of the *i*-th row in sparse matrix S. *col_idx* and *value* store the corresponding actual column indices and the actual values for each non-zero element, row by row, respectively. The CSC format represents the sparse matrix in a similar way to CSR, but stores the indices and values of the non-zero elements in a column-wise fashion. Unlike CSR, the COO stores the corresponding actual row indices of non-zero elements. The *col_idx* and *value* arrays remain the same as those in the CSR format.

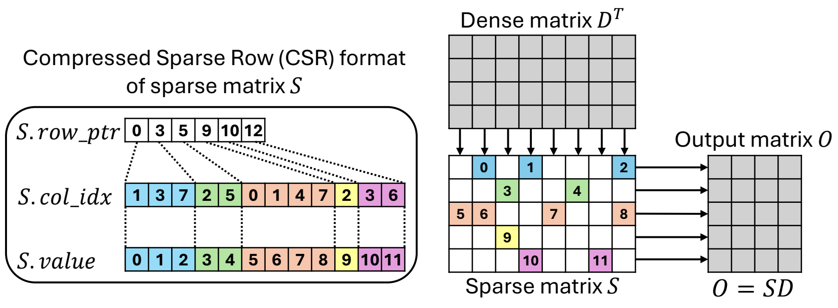

Figure 1. Illustration of SpMM using the compressed sparse row (CSR) format. The sparse matrix S is represented in CSR format, where the number of rows (M) is set to 5, and the number of columns (K) is set to 8. The number of columns (N) on dense matrix D is set to 4; thus, the size of the resulting output matrix O is $M \times N = 5 \times 4$.

Alternatively, sparse matrices can also be represented with a block-based sparse format, such as compressed sparse blocks (CSBs), Blocked-Ellpack (Blocked-ELL), and Variable Block Rows (VBRs) [19–22]. Given a fixed block size of R $\times$ C, the CSB format partitions a sparse matrix S into (M/R) $\times$ (K/C) blocks, where R and C must be factors of M and K, respectively. The CSB format can be viewed as a blocked representation of the CSR format. Similar to the CSR format, the CSB maintains *blk_row_ptr*, *blk_col_idx*, and *blk_value* arrays to store the indices and values of the non-zero elements in a block-wise fashion. The difference between CSR and CSB is that in CSR format, the non-zero elements in each row are stored contiguously, whereas in CSB format, the non-zero blocks, containing non-zero elements in each row panel (row block), are stored contiguously. Compared to the CSR format, the CSB significantly reduces the overhead of metadata, except for *blk_value*. This is because the CSB stores all values in each non-zero block, including both zero and non-zero values. In other words, the size of the *value* array held in CSR format is determined by the number of non-zero elements (*nnz*), whereas the size of *blk_value* in CSB format is dictated by the number of non-zero blocks multiplied by R and C.

The Blocked-ELL format is a variation of the ELLPACK storage format that exploits block structures present in sparse matrices. It partitions the sparse matrix into (M/B) $\times$ (K/B) blocks, with a fixed square block size B $\times$ B. In Blocked-ELL format, the number of column blocks in the row panel (group of rows) is set to Q, where Q represents the maximum number of non-zero column blocks across all row panels. The Blocked-ELL format is composed of two data structures, *ellColInd* and *ellValue*. *ellColInd* stores the indices of non-zero column blocks in each row-panel, and therefore, the size of *ellColInd* is (M/B) $\times$ Q. Furthermore, the size of *ellValue* is M $\times$ Q $\times$ B, and *ellValue* holds all the values of the elements, including both non-zero and zero elements in non-zero blocks. The Blocked-ELL format is memory-inefficient when a particular row panel contains no non-zero column blocks. In this case, the Blocked-ELL format still requires filling zero values in the *ellValue* for an empty row panel. In contrast to the fixed block sizes used in Blocked-ELL formats, the VBR format stores variable sizes of non-zero blocks to enable flexibility in block size.

2.2. Sparse Matrix–Dense Matrix Multiplication (SpMM)

This work optimizes SpMM, which is shorthand for sparse matrix–dense matrix multiplication. Given a sparse matrix S of size M $\times$ K and a dense matrix D of size K $\times$ N, the SpMM is defined as the matrix product $O = SD$. Algorithm 1 shows the SpMM when a sparse matrix S is represented in the CSR format using *row_ptr*, *col_idx*, and *value* data structures. As described in Algorithm 1, the outermost loop in line 1 iterates over all rows of S with i, and the algorithm then iterates over all columns of D or O to access every columns via j in line 2. Finally, the non-zero elements in the i-th row of S are accessed by the k loop in line 3. The non-zero elements of S are multiplied by the corresponding elements in D (selected based on the column indices of the non-zero elements in S). The results are then accumulated to produce the elements of the i-th row of output matrix O.

Algorithm 1: Sequential SpMM

```
  Input: $S.row\_ptr$, $S.col\_idx$, $S.value$, $D$[K][N]
  Output: $O$[M][N]
1 for $i = 0$ to $M - 1$ do
2     for $j = 0$ to $N - 1$ do
3         for $k = S.row\_ptr[i]$ to $S.row\_ptr[i+1] - 1$ do
4             $O[i][j] \mathrel{+}= S.value[k] \times D[S.col\_idx[k]][j]$;
```

2.3. Tensor Cores on GPUs

Tensor Cores (TCs) are specialized computational units first introduced with NVIDIA's Volta GPU architecture [23]. These units perform arithmetic operations in the form

$D = A \times B + C$, where A, B, C, and D are matrices [3]. Whereas other units in NVIDIA GPUs operate at the thread level, TCs function at the warp level—or at the warp group level in the Hopper architecture [24]. In the early stage of TC adoption in the Volta architecture, there were significant limitations on supported data types. Initially, matrices A and B, which serve as the operands for matrix multiplication, could only be in FP16 format, while matrices C and D, which act as accumulators, could be in either FP16 or FP32 formats. However, these restrictions have been progressively relaxed with the evolution of GPU architectures, including Turing, Ampere, and Hopper architectures. Table 1 represents the data types available on the TCs of each architecture. Additionally, TCs impose restrictions on matrix shapes for multiplication, although these constraints have also been gradually relaxed across different architectures, similar to the data type limitations. Table 2 details the matrix shapes supported by each PTX ISA version.

Table 1. Datatypes supported on Tensor Cores of each architecture; ✓: full-support; ●: support with reduced performance; -: not supported [24].

Generation	Architecture	Product Name	Specification			Precision Support					
			#SM [1]	#CC (FP32) [2]	#TC [3]	FP64	TF32	FP16	FP64	FP32	INT8
1	Volta [3]	V100S	80	5120	640	-	✓	✓	-	●	✓
2	Turing [25]	RTX 6000	72	4608	576	-	✓	●	-	●	✓
3	Ampere [4]	A100	108	6912	432	✓	✓	✓	✓	✓	✓
4	Hopper [5]	H100	132	16,896	528	✓	✓	✓	✓	✓	✓
5	Ada [26]	L40S	142	18,176	568	-	✓	●	●	✓	✓

[1] Number of streaming multiprocessors; [2] Number of shading units; [3] Number of Tensor Cores

There are two primary approaches to utilizing TCs. The first approach involves leveraging NVIDIA's libraries, such as cuBLAS, cuSPARSE, and cuDNN, which automatically apply TCs to optimize computational performance. The second approach entails directly employing the CUDA WMMA (Warp Matrix Multiply and Accumulate) API [27], as shown in Listing 1, which provides more granular control over TC utilization. Since Tensor Cores require specific matrix configurations depending on data types such as FP16, FP32, and FP64, it is necessary to partition matrices into corresponding fragments, as illustrated in Table 2.

Table 2. Matrix shapes that could be allowed on each PTX ISA version [28] (†: preview feature).

Instruction	Sparsity	Multiplicand Data Type	Shape	PTX ISA Version
wmma	Dense	.f16	.m16n16k16, .m8n32k16, and .m32n8k16	6.0
		.bf16	.m16n16k16, .m8n32k16, and .m32n8k16	7.0
		.tf32	.m16n16k8	7.0
		.u8/.s8	.m16n16k16, .m8n32k16, and .m32n8k16	6.3
		.u4/.s4	.m8n8k32	6.3 †
		.b1	.m8n8k128	6.3 †

Table 2. *Cont.*

Instruction	Sparsity	Multiplicand Data Type	Shape	PTX ISA Version
mma	Dense	.f64	.m8n8k4	7.0
			.m16n8k4, .m16n8k8, and .m8n8k16	7.8
		.f16	.m8n8k4	6.4
			.m16n8k8	6.5
			.m16n8k16	7.0
		.bf16	.m16n8k8 and .m16n8k16	7.0
		.tf32	.m16n8k4 and .m16n8k8	7.0
		.u8/.s8	.m8n8k16	6.5
			.m8n8k16 and .m16n8k32	7.0
		.u4/.s4	.m8n8k32	6.5
			.m16n8k32 and .m16n8k64	7.0
		.b1	.m16n8k128, .m16n8k128, and .m16n8k256	7.0
		.e4m3/.e5m2	.m16n8k32	8.4
mma	Sparse	.f16	.m16n8k16 and .m16n8k32	7.1
		.bf16	.m16n8k16 and .m16n8k32	7.1
		.tf32	.m16n8k8 and .m16n8k16	7.1
		.u8/.s8	.m16n8k32 and .m16n8k64	7.1
		.u4/.s4	.m16n8k64 and .m16n8k128	7.1
		.e4m3/.e5m2	.m16n8k64	8.5
mma	Sparse with ordered metadata	.f16	.m16n8k16 and .m16n8k32	8.5
		.bf16	.m16n8k16 and .m16n8k32	8.5
		.tf32	.m16n8k8 and .m16n8k16	8.5
		.u8/.s8	.m16n8k32 and .m16n8k64	8.5
		.u4/.s4	.m16n8k64 and .m16n8k128	8.5
		.e4m3/.e5m2	.m16n8k64	8.5

Listing 1. CUDA WMMA API to leverage Tensor Cores.

```
template<typename Use, int m, int n,int k, typename T, typename Layout=void> class fragment;
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const
    fragment<...> &c, bool satf=false);
```

In Listing 1, the `fragment` is a crucial data structure used to store sub-matrices (tiles) that are loaded into Tensor Cores for matrix operations. Once the input matrices are loaded into WMMA-specific data structures using `load_matrix_sync()`, the `mma_sync()` function performs the matrix multiply-and-accumulate (MMA) operation with TCs. After computation, the result stored in the accumulator fragment must be written back to the output matrix in global memory using `store_matrix_sync()`. However, the use of TCs does not inherently guarantee improved overall performance. To achieve accelerated operations through the direct utilization of TCs via the CUDA WMMA API, it is imperative to optimize the implementation. This optimization can be achieved by employing strategies such as shared memory utilization and pipelining, while taking into account the specific characteristics of GPUs.

3. Related Work on SpMM Using Tensor Cores

Several sparse matrix reordering and compression techniques have been proposed to maximize the utilization of TCs for SpMM by enhancing the data locality of sparse matrices. Alternatively, several previous studies and NVIDIA's libraries have optimized SpMM using TCs without reordering the sparse matrix. Therefore, previous work on optimizing SpMM using Tensor Cores can be categorized based on whether the reordering/compression process is performed prior to the actual SpMM computation, or if only matrix blocking is applied to the original sparse matrix to perform block matrix–matrix multiplication without any reordering/compression process.

Using Tensor Cores for SpMM with Sparse Matrix Reordering/Compression

1-SA, proposed by Labini et al. [12], reorders the rows of a sparse matrix based on their sparsity patterns, clustering similar rows to extract high-density blocks in the reordered matrix. The columns of each row are partitioned into multiple column blocks, and a set data structure is generated using binary values, where 1 indicates that the corresponding column block contains at least one non-zero element, and 0 indicates that all elements in the corresponding column block are zero. Using the sets representing the sparsity patterns of rows based on non-zero column blocks, 1-SA reorders the rows according to the Jaccard similarity between different rows by comparing their sparsity patterns. Subsequently, because the number of rows grouped into different clusters can vary, 1-SA uses the VBR format to store variable-sized blocks that include non-zero elements. These variable-sized blocks are then multiplied with the dense RHS matrix using NVIDIA's cublasGemmEx(), leveraging TCs on GPUs. However, 1-SA exhibits several limitations. First, the row-reordering process can be considered a significant preprocessing overhead. Second, 1-SA utilizes TCs by sequentially executing NVIDIA's cuBLAS library on all variable-sized blocks to obtain the final results. This approach is not effective for maximizing Tensor Core occupancy because it results in low parallelism when performing matrix multiplication with small-sized blocks. Furthermore, 1-SA processes all non-zero blocks in the reordered matrix, even if a block contains only a single non-zero element. Computing these sparse blocks on TCs is inefficient because each block contains a high proportion of zero elements.

TC-GNN, proposed by Yuke et al. [29], utilizes TCs for SpMM, which constitutes the largest portion of operations in the training of graph neural networks. They developed a sparse graph translation technique that compresses the sparse matrix, enabling the efficient extraction of dense tiles, which are then processed on TCs.

NVIDIA provides both software (e.g., cuSPARSELt APIs) and hardware (e.g., TCs) support for leveraging TCs in sparse matrix multiplication [30]. For example, by assuming a fine-grained 2:4 sparsity pattern, the original dense matrix is first converted into a structured sparse matrix by pruning half of the lower values in each row using the cusparseLtSpMMPrune() API. Thereafter, the structured sparse matrix is transformed into a compressed dense matrix by removing all zero values using the cusparseLtSpMMCompress() API. Then, the compressed matrix is multiplied with the RHS dense matrix on TCs using the cusparseLtMatmul() API, achieving double the peak performance compared to a GEMM operation on the original uncompressed dense matrix. For accurate SpMM computation, it is essential to use all non-zero elements in the sparse matrix. However, since the cuSPARSELt API was originally developed to accelerate deep neural networks by transitioning from dense to sparse training through the elimination of unnecessary computations, its use for traditional SpMM operations is not appropriate when the number of non-zero elements in any row exceeds half the number of columns in the input sparse matrix. Furthermore, similar to sparse matrix reordering, compressing the sparse matrix is a time-consuming task that must be completed before performing the actual matrix multiplication.

Using Tensor Cores for SpMM without Sparse Matrix Reordering/Compression

The NVIDIA cuSPARSE API supports the use of TCs when the Blocked-ELL format is provided to the cusparseSpMM() library [13]. With the Blocked-ELL format, the non-zero blocks (sub-matrices) in the sparse matrix S are simultaneously used to perform block matrix multiplication by leveraging TCs. However, compared to the CSR format, representing a sparse matrix in the Blocked-ELL format is ineffective in terms of both memory usage and preprocessing time. Since the zero values within each non-zero block (zero fillings) must also be stored in the Blocked-ELL format, inefficiency increases as the sparsity and irregularity of the matrix grow. Processing these zero fillings on TCs results in wasteful computations. Therefore, determining an optimal fixed block size for each sparse matrix is crucial for improving the utilization of TCs with the Blocked-ELL format.

4. GPU Implementation of TCA-SpMM

In this section, we present an in-depth exploration of the GPU-based implementation of TCA-SpMM algorithm. We first outline the architectural considerations and optimization strategies employed in adapting the algorithm to the GPU environment, especially Tensor Cores. Then, we delve into the techniques utilized to maximize Tensor Core utilization and achieve load balancing. Finally, we provide a comprehensive evaluation of the algorithm's computational and memory complexity on the GPU.

4.1. Design Overview of TCA-SpMM

In Algorithm 1, the innermost loop computes the dot product between the row vector $S[i,:]$ from the sparse matrix S and the column vector $D[:,j]$ from the dense matrix D, resulting in the value $O[i,j]$ in the output matrix O. In using the non-zero element values, $S.value$, and column indices, $S.col\_idx$, from the CSR format, only the non-zero elements in S are multiplied by the corresponding elements in D. However, since TCs operate on sub-matrices (2D blocks) and perform GEMM operations, it is not possible to directly utilize TCs with the original SpMM algorithm in the CSR format.

Figure 2 illustrates our fundamental concept of adapting the SpMM algorithm to utilize TCs in the TCA-SpMM approach. This concept transforms the dot product problem, with vector–vector operations (BLAS-1), into a matrix–matrix multiplication problem, with matrix-matrix operations (BLAS-3). Initially, we convert the two vectors involved in the innermost loop in Algorithm 1, $S[i,:]$ and $D[:,j]$, into matrices. Specifically, we reformulate each dot product with two vectors as a smaller blocked matrix–matrix multiplication. Once the vectors are transformed into matrices, these transformed matrices are passed to the TCs to execute the accelerated GEMM operation. However, as shown in Figure 2a, the output matrix has intermediate partial results so far. Thus, following the GEMM operation, the diagonal elements of the output matrix must be accumulated to yield the final result $O[i,j]$, corresponding to the dot product of $S[i,:]$ and $D[:,j]$.

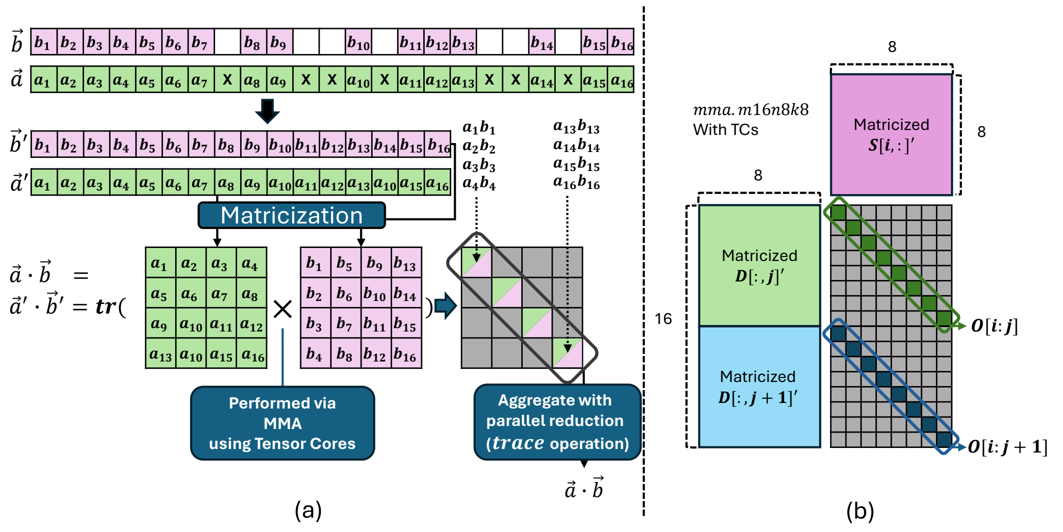

Figure 2. The left side of this figure (**a**) illustrates fundamental concept of our approach, the transformation of vector–vector dot product into matrix–matrix multiplication via matricization. The right side of this figure (**b**) describes the transformation using the practical MMA instruction, the $16 \times 8 \times 8$ MMA.

For example, let us assume that we have two vectors $\vec{a} = [a_1, a_2, \ldots, a_{16}]$ and $\vec{b} = [b_1, b_2, \ldots, b_{16}]$. Then, we convert them into two 4×4 matrices D and S as follows:

$$D = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix} \quad S = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \\ b_9 & b_{10} & b_{11} & b_{12} \\ b_{13} & b_{14} & b_{15} & b_{16} \end{bmatrix}.$$

To compute the dot product, $\vec{a} \cdot \vec{b}$, using transformed matrices, D and S, we multiply D by the transpose of S, yielding $O = D \times S^T$. The element-wise multiplication of corresponding elements in D and S produces intermediate results, similar to performing element-wise multiplication of the vectors. Each element $O[i, j]$ of the resulting matrix O is composed as $O[i, j] = \sum_{k=1}^{4} D[i, k] \times S[k, j]$. In this case, however, since we are trying to capture the dot product (which is a scalar), we are interested in summing the element-wise products of D and S. This effectively translates to computing the sum of element-wise products of corresponding elements in A and B, which can be written as $\vec{a} \cdot \vec{b} = \sum_{i=1}^{4} \sum_{j=1}^{4} D[i, j] \times S[i, j]$. These values represent the diagonal elements of O. The final sum yields the same result as the original dot product of the vectors.

As shown in Figure 2a, $\vec{b}$ and $\vec{a}$ represent a sparse row vector from matrix S and a dense column vector from matrix D, respectively. Since the element-wise product is zero when either element is zero, the dot product of $\vec{b}$ and $\vec{a}$ is identical to the dot product of the compressed vectors $\vec{b}'$ and $\vec{a}'$, where $\vec{b}'$ contains only non-zero elements from $\vec{b}$, and $\vec{a}'$ retains the corresponding elements from $\vec{a}$ with matching indices. With the compressed vectors $\vec{b}'$ and $\vec{a}'$, a single dot product operation can be reformulated as a matrix–matrix multiplication, with an additional step for aggregating the diagonal elements. While converting the dot product into a matrix–matrix multiplication, we reshape the vectors into operand matrices, with the LHS operand in a row-wise fashion, and the RHS operand in a column-wise fashion. Hereafter, we refer to this transformation process as the matricization of vectors. The transformed matrix–matrix multiplication is performed using the MMA operation from TCs, while the aggregation of diagonal elements, also known as the *trace* operation, is optimized through parallel reduction using warp shuffle. More specifically, because the most fine-grained shape for the half-precision MMA instruction on the TCs in the Ampere architecture is $16 \times 8 \times 8$ [4], we leverage this by matricizing the sparse row vector into a column-wise RHS operand matrix and converting multiple dense column vectors into the LHS operand matrix, as described in Figure 2b. This approach enables the simultaneous computation of multiple output elements using the MMA instruction.

Figure 3 describes the overall design of our parallelized SpMM implementation. After matricizing the vectors, as illustrated in Figure 2, each CUDA thread block performs the computations required for either an entire row vector or a part of it, as described in Figure 3. Since the number of non-zero elements varies across different rows, as illustrated in Figure 3, we optimize our kernel so that each thread block processes multiple row vectors with fewer non-zero elements, thereby mitigating the load imbalance problem. As all elements in a single row vector of the output matrix are derived from the same sparse row vector, the thread block can share this sparse row vector as an operand for the dot product operation. Moreover, fetching the sparse row vector from global memory is straightforward when using the CSR format. Therefore, we store the sparse row vector in shared memory, allowing all warps within the thread block to access it with minimal memory transactions. Furthermore, since the same non-zero pattern of a sparse row vector is used to compute the corresponding elements across all column vectors in D, we compress the column vectors from D by referencing the same indices in $S.col\_idx$. The set of compressed column vectors is managed in shared memory and referred to as the *compressed RHS buffer*. Once theses compressed column vectors are stored in shared memory, all warps within the thread block participate in computing different output elements. For example, when an $8 \times 8 \times 8$ MMA operation is available, two such operations are required to compute an output element if the corresponding sparse row vector contains 128 non-zero elements. In this case, the accumulator fragment of the MMA operation, acc, is used to accumulate the results of two MMA operations, allowing the final output element to be obtained with a single *trace* operation. Thus, the main challenges and considerations for achieving high performance in SpMM using TCs are as follows.

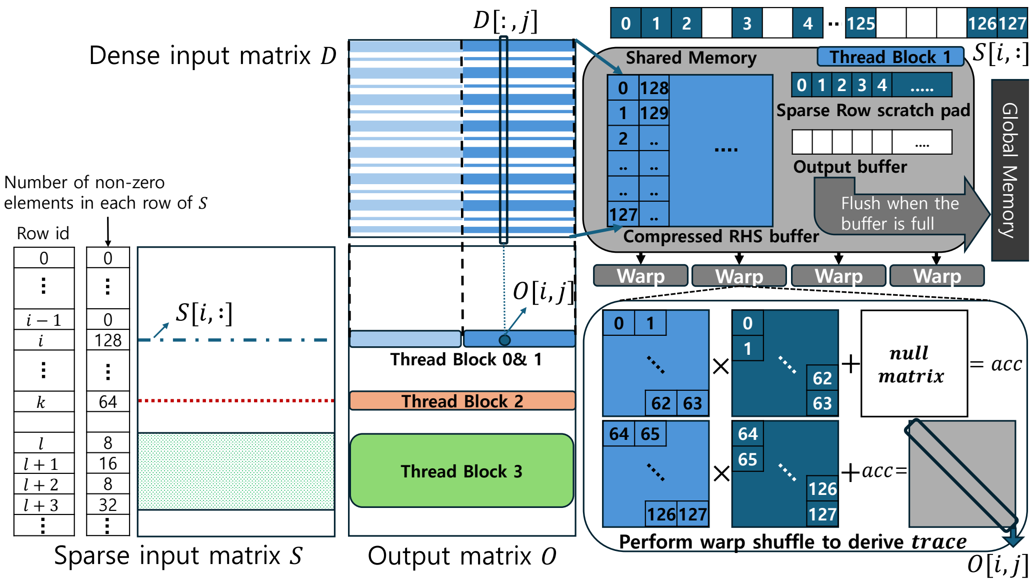

Figure 3. Overview of parallelization strategies for TCA-SpMM.

Challenge 1: Underutilization of Tensor Cores for SpMM

Since the total number of MMA operations required for SpMM in our approach is deterministic, maximizing the utilization of TCs is crucial for achieving high performance. Increasing arithmetic intensity is a key optimization strategy for improving TC utilization. In other words, maximizing the ratio of MMA operations to data movement is essential. However, minimizing data movement in SpMM poses significant challenges due to the irregular distribution of non-zero elements in the LHS sparse matrix, which results in irregular access patterns in the RHS dense matrix and complicates memory coalescing. To mitigate this issue, we leverage shared memory to optimize global memory access, thereby enhancing the efficiency of TCs.

Challenge 2: Load Imbalance in SpMM with an Irregular Sparse Matrix

The sparsity pattern of the sparse matrix can lead to significant variation in the number of non-zero elements across different rows, causing a load imbalance problem. For instance, as shown in Figure 3, if the number of non-zero elements in the $i-1$-th row and the i-th row of the sparse matrix S are 0 and 128, respectively, the computation for the $i-1$-th row in the output matrix O can be skipped, while the i-th row must undergo multiplication. In this scenario, parallelizing computations across rows, where each thread block performs a vector–vector multiplication, can result in unused thread blocks for rows containing only zero values. Launching such unused thread blocks leads to performance degradation, as other active thread blocks may be delayed, waiting to be scheduled on Streaming Multiprocessors (SMs). Additionally, if the number of non-zero elements in the k-th row is 64, the thread block assigned to process the k-th row performs fewer computations than the thread block processing the i-th row. To address this load imbalance, we dynamically allocate a variable number of thread blocks to different rows, enabling finer-grained control and more efficient resource utilization.

4.2. Parallelization of TCA-SpMM

Algorithm 2 presents the pseudo-code for the global kernel in the TCA-SpMM, while Algorithms 3 and 4 provide the pseudo-codes for the device kernels, which are used for executing the MMA operations based on the number of non-zero elements, invoked within Algorithm 2. To achieve a high degree of parallelism, TCA-SpMM parallelizes SpMM computations across the rows of the sparse matrix S, taking into account both the distribution of non-zero elements and the warp-level matrix multiplication (MMA) structure of the Tensor Cores. TCA-SpMM is designed around the $16 \times 8 \times 8$ MMA shape, which requires a 16×8 LHS operand matrix for the dense matrix D and an 8×8 RHS operand matrix for the sparse matrix S. Then, TCA-SpMM determines how many rows of S process each thread block with 64 (i.e., 8×8) non-zero elements, which can be mapped onto the RHS operand matrix. If a row contains more than 64 non-zero elements, multiple thread blocks will process the row as described in Algorithm 3. However, if a row contains 64 or fewer non-zero elements, a single thread block will process multiple rows or a single row, as described in Algorithm 4.

We assume that each CUDA thread block contains eight warps. In order to distribute SpMM computations appropriately, TCA-SpMM uses three preprocessed information sources for different warps, as shown in Algorithm 2: *row_id_for_each_warp*, *num_warps_per_row*, and *warpid_within_row*. First of all, the array *row_id_for_each_warp* specifies the row indices of the output matrix O that each warp contributes to computing. For example, as shown in Figure 3, since the i-th row of the sparse input matrix contains 128 non-zero elements, two thread blocks are launched for the i-th row to distribute the workload evenly across the blocks, thereby mitigating load imbalance. All warps in Thread Block 0 and Thread Block 1, as illustrated in Figure 3, maintain the same row index i for the variable *target_row*, as defined in line 5 of Algorithm 2. Secondly, the array *num_warps_per_row* specifies the number of warps assigned to compute each row. For example, as shown in Figure 3, there are 16 warps (16 warps = 8 warps $\times$ 2 thread blocks) for the i-th row because both Thread Block 0 and 1 handle the same row (line 6 in Algorithm 2). Finally, the array *warpid_within_row* provides identifiers (IDs) for each warp to manage the output elements that it is responsible for computing (line 7 in Algorithm 2). For example, using the array *warpid_within_row*, the 16 distinct warps in both Thread Block 0 and Thread Block 1 can be identified by unique ID values ranging from 0 to 15. Based on the unique warp ID values, we divide the workload of computing the output values in the i-th row of matrix O between Thread Block 0 and Thread Block 1 by distributing the output elements assigned to each thread block (lines 8–14 in Algorithm 2).

Algorithm 2: Parallel implementation of TCA-SpMM on GPUs

```
Input: $S.row\_ptr$, $S.col\_idx$, $S.value$, $D$[K][N], $tilewidth$,
$row\_id\_for\_each\_warp$ : Array indicating the row index of $O$ that each warp participates in computing,
$num\_warps\_per\_row$ : Array indicating the number of warps participating for computing the same row to each warp,
$warpid\_within\_row$ : Array indicating the ID of each warp within the set of warps that performs computation for the specific row
Output: $O$[M][N]
1  extern __shared__ shm;
2  $block\_num\_warps \leftarrow$ blockDim.x / warpSize;
3  $local\_warpid \leftarrow$ threadIdx.x / warpSize;
4  $global\_warpid \leftarrow block\_num\_warps \times$ blockIdx.x $+ local\_warpid$;
5  $target\_row \leftarrow row\_id\_for\_each\_warp[global\_warpid]$;
6  $num\_warps\_current\_row \leftarrow num\_warps\_per\_row[target\_row]$;
7  $warpid\_current\_row \leftarrow warpid\_within\_row[global\_warpid]$;
8  $blockIdx\_current\_row \leftarrow warpid\_current\_row\ /\ block\_num\_warps$;
9  $num\_threadblock\_current\_row \leftarrow num\_warps\_current\_row\ /\ block\_num\_warps$;
10 $stride \leftarrow \lceil N\ /\ num\_threadblock\_current\_row\ \rceil$;
11 $output\_col\_start \leftarrow stride \times blockIdx\_current\_row$;
12 $output\_col\_end \leftarrow stride \times (blockIdx\_current\_row + 1)$
13 if $blockIdx\_current\_row == num\_threadblock\_current\_row - 1$ then
14     $output\_col\_end \leftarrow N$;
15 $head\_warpid \leftarrow local\_warpid - warpid\_current\_row + block\_num\_warps \times blockIdx\_current\_row$;
16 output_buffer $\leftarrow$ &shm[$head\_warpid \times tilewidth$];
17 LHSval_scratch $\leftarrow$ &shm[$block\_num\_warps \times 8$];
18 RHS_buffer $\leftarrow$ &shm[$block\_num\_warps \times 8 + block\_num\_warps \times tilewidth$];
19 LHSval_scratch $\leftarrow$ &LHSval_scratch[$head\_warpid \times 8$];
20 RHS_buffer $\leftarrow$ &RHS_buffer[ $head\_warpid \times 8 \times tilewidth$];
21 if $S.row\_ptr[target\_row + 1] - S.row\_ptr[target\_row] > 64$ then
22     multiblock_per_row($S.row\_ptr$, $S.col\_idx$, $S.value$, $D$, $tilewidth$, $target\_row$,
         $warpid\_current\_row$, $num\_warps\_current\_row$, $block\_num\_warps$, $local\_warpid$,
         output_buffer, LHSval_scratch, RHS_buffer);
23 else
24     multirow_per_block($S.row\_ptr$, $S.col\_idx$, $S.value$, $D$, $tilewidth$, $target\_row$,
         $warpid\_current\_row$, $num\_warps\_current\_row$, $block\_num\_warps$, $local\_warpid$,
         $output\_col\_start$, $output\_col\_end$, output_buffer, LHSval_scratch, RHS_buffer);
```

As shown in the top right of Figure 3, TCA-SpMM exploits shared memory to reuse and share data within a thread block. After allocating the shared memory resources for the three components—the (1) sparse row scratch pad, (2) compressed RHS buffer, and (3) output buffer (lines 16–18 in Algorithm 2)—the shared memory is not further divided among different warps because *head_warpid* is always 0 in this case (lines 16 and 19–20), as shown in Figure 3. Thereafter, Thread Block 0 and Thread Block 1 are directed to the `__device__ function multiblock_per_row()` to perform accelerated matrix multiplication for the *i*-th row, which requires multiple thread blocks to handle the large workload (Algorithm 3).

After distributing the workloads required to obtain the output elements in $O[target\_row, :]$ across multiple thread blocks, Algorithm 3 details how the output results of $O[target\_row, output\_col\_start : output\_col\_end - 1]$ are computed using multiple thread blocks. Since the total number of operand elements processed at once by each MMA operation is set to $16 \times 8 \times 8$, our implementation pads the remaining portions of the row vectors of S and the collections of compressed column vectors of D with zero values in the MMA operand when the number of non-zero elements in these vectors is not divisible by a fragment size of 64.

Algorithm 3: multiblock_per_row() device kernel on GPUs

```
Input: $S.row\_ptr$, $S.col\_idx$, $S.value$, $D$[K][N], $tilewidth$, $target\_row$, $warpid\_current\_row$,
       $num\_warps\_current\_row$, $block\_num\_warps$, $local\_warpid$, $output\_col\_start$,
       $output\_col\_end$, output_buffer, LHSval_scratch, RHS_buffer
Output: $O[target\_row][output\_col\_start : output\_col\_end - 1]$
1  $row\_nnz \leftarrow S.row\_ptr[target\_row + 1] - S.row\_ptr[target\_row]$;
2  $num\_fragments \leftarrow tilewidth\ /\ (block\_num\_warps \times 2)$;
3  $buffer\_offset \leftarrow 0$;
4  $output\_offset \leftarrow output\_col\_start$;
5  for $tile \leftarrow output\_col\_start$ to $output\_col\_end - 1$ by $tilewidth$ do
6      $fragments\_c[0 : num\_fragments - 1] \leftarrow 0$;
7      for $nonzero\_offset \leftarrow 0$ to $row\_nnz - 1$ by 64 do
8          $offset \leftarrow S.row\_ptr[target\_row] + nonzero\_offset$;
9          LHSval_scratch[0 : 63] $\leftarrow S.value[offset : offset + 63]$;
10         $col\_indices[0 : 63] \leftarrow S.col\_idx[offset : offset + 63]$;
11         RHS_buffer[0 : $tilewidth - 1$][0 : 63] $\leftarrow D[col\_indices][tile : tile + tilewidth - 1]^T$;
12         __syncthreads();
13         $fragment\_b \leftarrow$ LHSval_scratch.$reshape\_k8n8^T$;
14         for $frag \leftarrow 0$ to $num\_fragments - 1$ do
15             $idx \leftarrow frag \times num\_fragments \times 2 + local\_warpid \times 2$;
16             $fragment\_a \leftarrow$ RHS_buffer[$idx : idx + 1$][0:63].$reshape\_m16k8$;
17             $fragments\_c[frag] \leftarrow$ mma.sync.m16n8k8($fragment\_a$, $fragment\_b$)
18     for $frag \leftarrow 0$ to $num\_fragments - 1$ do
19         output_buffer[$buffer\_offset + 2 \times local\_warpid$] $\leftarrow$
             $trace(fragments\_c[frag][0{:}7][0{:}7])$;
20         output_buffer[$buffer\_offset + 2 \times local\_warpid + 1$] $\leftarrow$
             $trace(fragments\_c[frag][8{:}15][0{:}7])$;
21     __syncthreads();
22     $buffer\_offset \leftarrow buffer\_offset + tilewidth$;
23     if $buffer\_offset == tilewidth \times block\_num\_warps$ then
24         $O[target\_row][output\_offset : output\_offset + buffer\_offset - 1] \leftarrow$
             output_buffer[];
25         output_buffer[] $\leftarrow 0$;
26         $output\_offset \leftarrow output\_offset + buffer\_offset$;
27         $buffer\_offset \leftarrow 0$;
```

From lines 5 to 27 in Algorithm 3, each thread block performs tiled matrix multiplication to compute output in parallel. The *tilewidth* specifies how many operands are computed in the outermost tiled loop (line 5). To minimize the number of aggregations required for the *trace* operation, we declare multiple output fragments, denoted as $fragments\_c$, which serve as different accumulators for the MMA operations. Through distributing the output tiles across warps, the number of output fragments in $fragments\_c$ is determined (line 2). Using multiple output fragments, lines 6 to 17 describe how our parallel implementation efficiently utilizes MMA operations for the reformulated computation. Specifically, we first initialize all output fragments to null (line 6) and then apply the sequential computation, dividing the number of non-zero elements in the sparse row vector into fragment size of 64 (lines 7–8). In order to fill the RHS_buffer of the shared memory with the corresponding compressed column vectors of D, we use $S.col\_idx$ to identify the set of column vectors corresponding to the current output tile (lines 10–12).

Moreover, the transpose is applied to the set of compressed column vectors to store their elements in a row-wise fashion (line 11). After successfully fetching the elements for the reformulated computation, we matricize the sparse row vector in a column-wise fashion (line 13), and the subset of compressed column vectors in RHS_buffer in a row-wise fashion (line 16), storing these matrices in the fragments $fragment\_b$ and $fragment\_a$, respectively. Then, we perform the MMA operation (line 17), with all compressed column vectors used as operands during the iteration (lines 14–17). Lastly, we perform two *trace* operations on each output fragment using warp shuffle, since it is possible to obtain two different output values from each 16 × 8 × 8 MMA operation (lines 18–20).

Algorithm 4: multirow_per_block() device kernel on GPUs

```
Input: $S.row\_ptr$, $S.col\_idx$, $S.value$, $D$[K][N], $tilewidth$, $target\_row$, $warpid\_current\_row$,
       $num\_warps\_current\_row$, $block\_num\_warps$, $local\_warpid$, output_buffer,
       LHSval_scratch, RHS_buffer
Output: $O[target\_row]$
1  $row\_start \leftarrow S.row\_ptr[target\_row]$;
2  $row\_nnz \leftarrow S.row\_ptr[target\_row + 1] - row\_start$;
3  $buffer\_offset \leftarrow 0$;
4  $output\_offset \leftarrow 0$;
5  LHSval_scratch$[0 : row\_nnz - 1] \leftarrow S.value[row\_start : row\_start + row\_nnz - 1]$;
6  $col\_indices[0 : row\_nnz - 1] \leftarrow S.col\_idx[row\_start : row\_start + row\_nnz - 1]$;
7  LHSval_scratch$[row\_nnz : 8 \times num\_warps\_current\_row - 1] \leftarrow 0$;
8  $col\_indices[row\_nnz : 8 \times num\_warps\_current\_row - 1] \leftarrow None$;
9  __syncthreads();
10 $fragment\_b \leftarrow$ LHSval_scratch$[0 : 8 \times num\_warps\_current\_row - 1].reshape\_k8n8^T$;
11 $output\_elem\_per\_mma \leftarrow 128\ /\ 2^{\lfloor \log_2 row\_nnz \rfloor}$;
12 for $tile \leftarrow 0$ to $N - 1$ by $tilewidth$ do
13     $fragment\_c \leftarrow 0$;
14     RHS_buffer$[0 : tilewidth - 1][0 : 8 \times num\_warps\_current\_row - 1] \leftarrow D[col\_indices][tile$
        $: tile + tilewidth - 1]^T$;
15     __syncthreads();
16     $fragment\_a \leftarrow$ RHS_buffer$[output\_elem\_per\_mma \times warpid\_current\_row :$
        $output\_elem\_per\_mma \times (\ warpid\_current\_row + 1\ ) - 1][0 : 8 \times$
        $num\_warps\_current\_row - 1].reshape\_m16n8$
17     $fragment\_c \leftarrow$ mma.sync.m16n8k8($fragment\_a$, $fragment\_b$)
18     do in parallel
19         for $elem \leftarrow 0$ to $output\_elem\_per\_mma - 1$ do
20             output_buffer$[buffer\_offset + warpid\_current\_row \times output\_elem\_per\_mma$
                $+ elem] \leftarrow trace(fragment\_c[elem \times num\_warps\_current\_row : (\ elem + 1\ ) \times$
                $num\_warps\_current\_row][0 : num\_warps\_current\_row - 1])$;
21     $buffer\_offset \leftarrow buffer\_offset + output\_elem\_per\_mma \times num\_warps\_current\_row$;
22     __syncthreads();
23     if $buffer\_offset == tilewidth \times num\_warps\_current\_row$ then
24         $O[target\_row][output\_offset : output\_offset + buffer\_offset - 1] \leftarrow$
            output_buffer[];
25         output_buffer[] $\leftarrow 0$;
26         $output\_offset \leftarrow output\_offset + buffer\_offset$;
27         $buffer\_offset \leftarrow 0$;
```

4.2.1. Maximizing Tensor Core Utilization

One of the most significant factors affecting the performance of our TCA-SpMM implementation is the method used for fetching compressed column vectors of D, particularly due to strided memory access patterns. Fetching a compressed column vector directly from global memory using a warp results in uncoalesced memory access, leading to an increased number of memory transactions. It is evident that the efficiency of TCs is closely tied to the cost of data movement. Therefore, to maximize TC utilization, we employ the RHS_buffer located in shared memory, corresponding to the *compressed RHS buffer* depicted in Figure 3. This approach enables cooperative memory fetches of compressed column vectors using the warps within a thread block (line 11 in Algorithm 3).

Since the compressed column vectors pass through shared memory, a warp is able to access global memory contiguously by fetching certain parts of different compressed column vectors, while other warps access the remaining parts. Furthermore, since the multiple threads within a warp can only obtain a few output values, directly storing the output results in global memory limits write instruction efficiency. Hence, we use the output_buffer residing in shared memory to temporarily collect multiple output tiles and then flush them into global memory to achieve a higher global memory write efficiency (lines 19–27 in Algorithm 3). Moreover, to increase the number of elements accessed by load/store memory instruction, we explicitly align the address of every memory access by controlling *output_col_start* and *tilewidth* using PTX instruction such as ld.global.b128 [28].

4.2.2. Achieving Load Balancing

In SpMM computation, workload imbalance occurs due to the uneven distribution of non-zero elements across rows. To address this, the TCA-SpMM method parallelizes the computation across rows to improve load balancing. Specifically, the number of rows assigned to each thread block is determined by the number of non-zero values in those rows.

Figure 4 presents implementation details for TCA-SpMM. The top section of Figure 4 shows how to distribute six rows to two thread blocks. Let us assume that Thread Block 0 handles five rows from row 0 to row 4 and Thread Block 1 handles one row— row 5. As shown in Figure 4, row 5 has 64 non-zero elements (64 = 124 − 60 = *row_ptr[6]* − *row_ptr[5]*) based on the values in *row_ptr*. In this case, the load imbalance problem does not occur when executing Algorithm 3, because it is enough to generate two different output values from a single output fragment, as described in the rightmost MMA fragment layout in Figure 4, by employing the 16 × 8 × 8 MMA operation from Thread Block 1. However, since there are eight non-zero elements in row index 0, obtaining only two output values from a single output fragment is inefficient. In our approach, as the 16 compressed column vectors can be maintained in a single fragment, one MMA operation is able to compute 16 output elements for row 0 in parallel, as illustrated in the bottom-left MMA fragment layout in Figure 4. In addition, for row 4, which has 32 non-zero elements, it is possible to obtain four different output values from the MMA operation by applying the *trace* operation to the finer-grained sub-matrices of the output fragment. Likewise, eight output values can be computed from the MMA operation for row 1. The number of non-zero elements is not necessarily divisible by 8, 16, 32, or 64, as padding zero values to the remaining part is possible, as shown in the cases of row 1 and row 2.

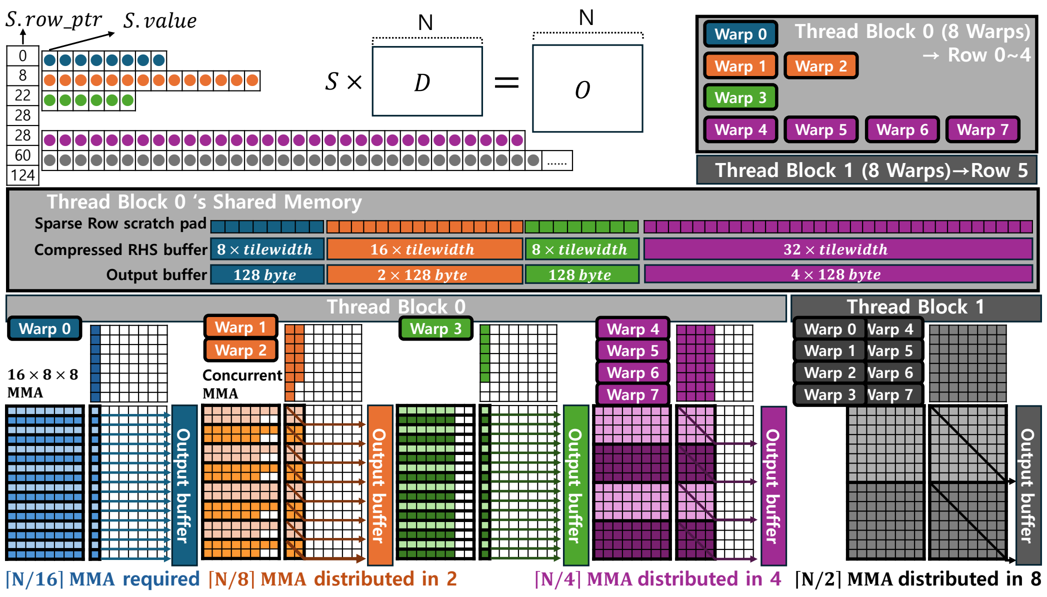

Figure 4. Implementation details for TCA-SpMM. The number of warps, proportional to the number of non-zero elements, is assigned to perform the computation of multiple row vectors within a single CUDA thread block by distributing its shared memory.

However, applying different optimization schemes for different rows using the same number of warps may cause an additional workload imbalance problem. For example, as described in Figure 4, we can yield sixteen different output values from a single output fragment in row 0, while we can yield only four different output values from a single output fragment in row 4. Therefore, the total number of MMA operations required to complete the computations for every output values in row 0 is different from that of row 4. Assuming that we assign only one warp for each output row's computation, the warp assigned for row 0 requires $\lceil N/16 \rceil$ iterative MMA operations, while the warp assigned for row 4 requires $\lceil N/4 \rceil$, where N stands for the number of column vectors of the output matrix, which is equivalent to the number of output values in a single row. Our fundamental solution to address this situation is to enable a thread block to compute multiple output rows by assigning a number of warps proportional to the number of non-zero elements. After balancing the workloads, we change the configuration of fragments to perform the dot product by placing additional compressed column vectors of D to maximize the number of output element values produced. Our TCA-SpMM assigns a single warp to process rows with 8 or fewer non-zero elements, doubles the number of warps for rows with 9 to 16 non-zero elements, and doubles it again for rows with 17 to 32 non-zero elements. Finally, eight warps are needed for rows with 33 to 64 non-zero elements. To assign a variable number of warps to different rows, we distribute shared memory resources to regularize the behavior of each warp, as shown in the middle of Figure 4. Since we set the number of warps based on the number of non-zero elements in the rows, the scratch pad (*sparse row*) is distributed in proportion to the number of warps. For example, the memory space required to store all non-zero elements in row index 4 is four times larger than that of row 0; therefore, we assign four times as many warps to row 4 compared to row 0. Similarly, the memory space required to store all elements from the compressed column vectors corresponding to the tile is proportional to the number of assigned warps. This warp assignment scheme also applies to the output buffer, which is designed for efficient writing to global memory.

Algorithm 4 shows the pseudo-code for the device kernel that processes rows with fewer than 64 non-zero elements, where a single thread block performs computations for multiple rows to achieve better load balancing. In Algorithm 4, only the non-zero elements are stored in shared memory (line 5), while zero values are used to fill the positions beyond the range of the non-zero elements in the given sparse row (line 7). For *col_indices*, we store *None* instead of 0 to minimize data movement when fetching the set of compressed column vectors from D. Since the number of non-zero elements does not always exceed $8 \times num\_warps\_current\_row$, we use $fragment\_b$ as the invariant MMA operand to hold the non-zero elements in the sparse row (line 10) and maintain *col_indices* for the corresponding column indices. After computing the number of output values for a single MMA operation (line 11), the tiled multiplication for the output row is performed in a manner similar to Algorithm 3. The major change in Algorithm 4 is that the matricization performed in line 16 of Algorithm 3 is now carried out in line 16 of Algorithm 4, allowing $fragment\_a$ to contain the total number of $output\_elem\_per\_row$ and different compressed column vectors from D. As the number of output values is identical to $output\_elem\_per\_row$, the same number of *trace* operations is performed for the corresponding sub-matrices represented in $fragment\_c$. However, it is possible to perform these *trace* operations simultaneously using shared warp shuffle instructions, provided that the communication offsets in warp shuffle are judiciously controlled (line 20). Finally, output_buffer is used to temporarily store the resulting output values and then flush them into global memory to reduce the number of global memory write operations.

4.3. Detailed Complexity Analysis of TCA-SpMM

A single MMA instruction executes multiple $4 \times 4 \times 4$ matrix multiplications to leverage the high degree of parallelism and throughput of TCs. In this subsection, an MMA instruction is defined as a single instruction, denoted as μ, where M_{mma}, N_{mma}, and K_{mma} denote the size of the row in the left fragment, the size of the column in the right fragment, and the size of the column/row in the left/right fragment, respectively. For an incremental analysis, we first assume that every sparse row vector contains more than ($K_{mma} \times N_{mma}$) non-zero elements. The value $K_{mma} \times N_{mma}$ represents the size of the right fragment used in a single MMA instruction, which corresponds to the number of non-zero elements from the sparse row vector that are processed simultaneously via MMA instruction in our approach. As the number of MMA instructions required for our TCA-SpMM is associated with the number of non-zero elements in the sparse matrix, the total number of MMA instructions required for our TCA-SpMM with TCs is

$$\begin{aligned}\sum_{i=0}^{M-1} \left\lceil \frac{nnz_i}{K_{mma} \times N_{mma}} \times \mu \right\rceil \times \left\lceil N \times \frac{N_{mma}}{M_{mma}} \right\rceil &\approx \frac{NNZ}{K_{mma} \times N_{mma}} \times N \times \frac{N_{mma}}{M_{mma}} \times \mu \\ = M \times K \times (1-\alpha) \times N \times \frac{1}{M_{mma} \times K_{mma}} \times \mu &= (1-\alpha) \times \frac{MNK}{M_{mma} \times K_{mma}} \times \mu\end{aligned} \tag{1}$$

where M, N, and K denote the number of rows of the LHS matrix, the number of columns of the RHS matrix and the number of columns/rows of the LHS/RHS matrix, respectively. In addition, nnz_i, NNZ, and α denote the number of non-zero elements in the i-th row, the total number of non-zero elements in a matrix (i.e., $NNZ = \sum_{i=0}^{M-1} nnz_i$), and the sparsity of the matrix, respectively. As the non-zero elements in a specific row are divided into the blocks of $K_{mma} \times N_{mma}$ and then the blocks applied to MMA instructions (μ) in our approach, $\lceil nnz_i/(K_{mma} \times N_{mma}) \rceil$ MMA instructions are required to complete the computation for a single output value at the i-th row, which is maintained in a single output fragment. On the other hand, if the number of rows in the output fragment, M_{mma}, is divisible by the number of columns, N_{mma}, more than one output value can be obtained using a single output fragment by maintaining multiple column vectors of the dense input matrix in the left fragment. For example, in assuming that $M_{mma} = 16$ and $N_{mma} = 8$, two output values are obtainable from a single output fragment by maintaining the specific column vector input in the top-half of the left fragment, while maintaining another column vector in the bottom-half of the left fragment. This implies that the number of output values that we can obtain from a single output fragment is M_{mma}/N_{mma}. Thus, to compute all the output values in a specific row, $\lceil N \times (N_{mma}/M_{mma}) \rceil$ output fragments are need to be maintained. In multiplying the number of MMA instructions required to complete the computation for a single output fragment by the number of output fragments needed to maintain all output values in a specific row, the total number of MMA instructions required to compute all output values in a specific row can be derived. Finally, the total number of MMA instructions required to compute every entry of output matrix can be found by summing the number of MMA instructions needed to compute all output values in each row, as organized in Equation (1). Since α represents the sparsity of the LHS matrix, the matrix density, defined as the ratio of the non-zero elements to the total number of elements, can be expressed as $(1 - \alpha)$. Therefore, with the total number of elements in the LHS matrix being $M \times K$, we can utilize the relationship between M, K, and α as $M \times K \times (1 - \alpha) = NNZ$ in Equation (1). Equation (1) demonstrates that our TCA-SpMM can effectively reduce the total number of operations required for the computation as the sparsity grows. For example, when the sparsity value α changes from 0.5 to 0.9, the total number of $4 \times 4 \times 4$ matrix multiplications is reduced by a factor of 5, because $(1 - 0.5) = 5 \times (1 - 0.9)$.

To achieve load balancing across rows, when the $M_{mma} = 16$, $N_{mma} = K_{mma} = 8$, and $nnz_i < 64$, the number of MMA instructions (the number of μ) required to compute the i-th row in the resulting output matrix O, as a function of the number of non-zero elements in the i-th row ($f(nnz_i)$), is as follows:

$$f(nnz_i) = \begin{cases} \lceil N/16 \rceil \mu, & \text{if } 0 < nnz_i \leq 8, \\ \lceil N/8 \rceil \mu, & \text{if } 8 < nnz_i \leq 16, \\ \lceil N/4 \rceil \mu, & \text{if } 16 < nnz_i \leq 32, \\ \lceil N/2 \rceil \mu, & \text{if } 32 < nnz_i \leq 64. \end{cases} \tag{2}$$

Since the number of non-zero elements varies across different rows, the number of output values produced by a single MMA instruction can differ in our TCA-SpMM. Therefore, the number of MMA instructions required to compute each output row varies depending on the number non-zero elements in the corresponding row of the input sparse matrix S. By assigning a number of warps proportional to the number of non-zero elements, our TCA-SpMM is able to evenly distribute the workload required for computing multiple rows across the warps in a thread block.

5. Experimental Evaluation

5.1. Experimental Setup

Datasets

For the experimental evaluations, since the key contribution of our TCA-SpMM is leveraging TCs for highly sparse matrices in deep neural networks, we used sparse matrices from the publicly available Deep Learning Matrix Collection (DLMC) [8], which were collected from both the training and inference phases of deep neural networks to benchmark sparse computational kernels. More specifically, the DLMC dataset contains sparse matrices collected from deep neural networks that employ various sparsification and pruning techniques, such as variational dropout, magnitude-based weight pruning, random weight pruning, and l_0 regularization, applied to transformer models [31]. Several of the aforementioned SpMM implementations using TCs that first reorder the sparse matrix and then perform SpMM on the reordered matrix can benefit from improved data locality with the reordered sparse matrix. Therefore, comparing our TCA-SpMM with approaches that use reordered matrices is not a fair comparison, as our TCA-SpMM does not perform any reordering and operates on the original sparse matrix, even if the data locality is very low. To fairly compare the performance of our TCA-SpMM implementation with well-optimized CUDA kernels from other SpMM implementations that use reordered matrices, we select highly sparse matrices from the DLMC dataset. In practice, since the pruned and sparsified weight matrices used in sparse deep neural networks exhibit an average sparsity of 95% [29,32], we selected 698 sparse matrices with sparsity ranging from 90% to 98% from the DLMC dataset. This choice ensures a fair evaluation because the irregularity of sparse matrices generally increases with their sparsity, making it more challenging to capture sparsity patterns and improve data locality when reordering matrices with higher sparsity.

Benchmarking Machines

We evaluated the all experiments on an NVIDIA RTX 3080 GPU. Table 3 shows the details of the benchmarking machines.

Table 3. Machine configuration.

Machine	Details
CPU	12th Gen Intel(R) Core(TM) i7-12700 (12 CPU cores, 20 threads per core) GCC version 9.4.0
GPU	NVIDIA RTX 3080 (10 GB Global Memory, 68 Ampere SMs, 5 MB L2 Cache 760 GB/second Bandwidth) CUDA version 12.1

SpMM Implementations Compared

We compared the performance of our TCA-SpMM with those of state-of-the-art parallel SpMM implementations that reorder the sparse matrix to improve data locality and then perform matrix–matrix multiplication using the reordered matrix, efficiently utilizing TCs. Our implementation and the two state-of-the-art SpMM implementations using TCs that we evaluated in our experiments were the following:

- **1-SA** (https://github.com/HicrestLaboratory/SPARTA (accessed on 12 September 2024)): 1-SA reorders the rows of the original sparse matrix based on its sparsity pattern and performs SpMM by processing the non-zero blocks of the reordered matrix on TCs [12];
- **TC-GNN** (https://github.com/YukeWang96/TC-GNN_ATC23 (accessed on 12 September 2024)): TC-GNN compresses the original sparse matrix and performs SpMM by processing the dense tiles in each row panel of the compressed matrix on TCs [29];
- **TCA-SpMM** (https://github.com/mlsys-lab-sogang/TCA-SpMM (accessed on 12 September 2024)): Our TCA-SpMM uses the original sparse matrix without reordering or compression, performing SpMM by processing only the non-zero elements in the sparse matrix on TCs.

5.2. Performance Evaluation

5.2.1. Speedup

Figure 5 shows a performance comparison of various SpMM parallel implementations optimized with TCs. The top and bottom graphs in Figure 5 show the execution time and achieved peak performance (GFLOPs) of the SpMM implementations, respectively. Note that for a fair evaluation, only the actual execution time of sparse matrix–matrix multiplication was measured for both 1-SA and TC-GNN, excluding the time required for reordering and compression. All experimental results presented in this section are the average of 10 distinct executions. Furthermore, the GFLOPs for each compared implementation were measured using $\frac{\text{total floating points}}{\text{execution time (s)}} \times 10^{-9} = \frac{2 \times \delta \times N}{\text{average execution time (s)}} \times 10^{-9}$, where δ is the average number of non-zero elements in the sparse matrices with the same sparsity. The black solid line within each box at the top of Figure 5 represents the median runtime, while the $\star$ symbols indicate the average runtime of different implementations across matrices with varying sparsity. The 698 sparse matrices we used in our evaluation are categorized based on four different pruning techniques applied to the transformer model. In our TCA-SpMM implementation, each thread block is configured with 256 threads, resulting in 8 warps per thread block. Compared to 1-SA and TC-GNN, our TCA-SpMM achieved an average speedup of 29.58× and 6.58×, respectively, across all sparse matrices with sparsity ranging from 90% to 98%. For example, for the sparse matrices pruned with the variational dropout technique, TCA-SpMM achieved a 25.43× and 8.11× average speedup compared to that of 1-SA and TC-GNN, respectively. Furthermore, as the sparsity of the matrices increases, our TCA-SpMM achieved higher speedup over 1-SA and TC-GNN. For instance, with sparsity ranging from 90% to 98%, TCA-SpMM is 13.75×, 19.46×, and 40.81× faster than 1-SA, and 4.07×, 5.47×, and 7.85× faster than TC-GNN on sparse matrices pruned using the magnitude technique. This result implies that the computational complexity of TCA-SpMM is associated with the number of non-zero elements and the operations required for SpMM. It can be observed that our TCA-SpMM shows a greater degree of performance improvement compared to other implementations as the sparsity of the matrices increases. As the sparsity increases, it is evident that the number of floating-point operations decreases. Consequently, the GFLOPs of all compared implementations decrease as the number of non-zero elements decreases across all experiments. However, the GFLOPs of our TCA-SpMM consistently outperformed those of other implementations as shown at the bottom of Figure 5. As 1-SA operates TCs on non-zero blocks in the reordered matrix and TC-GNN operates TCs on dense blocks after the compressing the matrix, both approaches are unable to completely ignore the zero elements within the non-zero blocks. Moreover, as the 1-SA implementation executes cuBLAS serially for both dense and sparse tiles in the reordered matrix, this approach hinders achieving a high degree of parallelism on TCs. However, since our TCA-SpMM processes only non-zero elements using TCs, the number of operations greatly decreases with highly sparse matrices. As a result, the performance of our TCA-SpMM improves more significantly with increasing sparsity than other SpMM implementations.

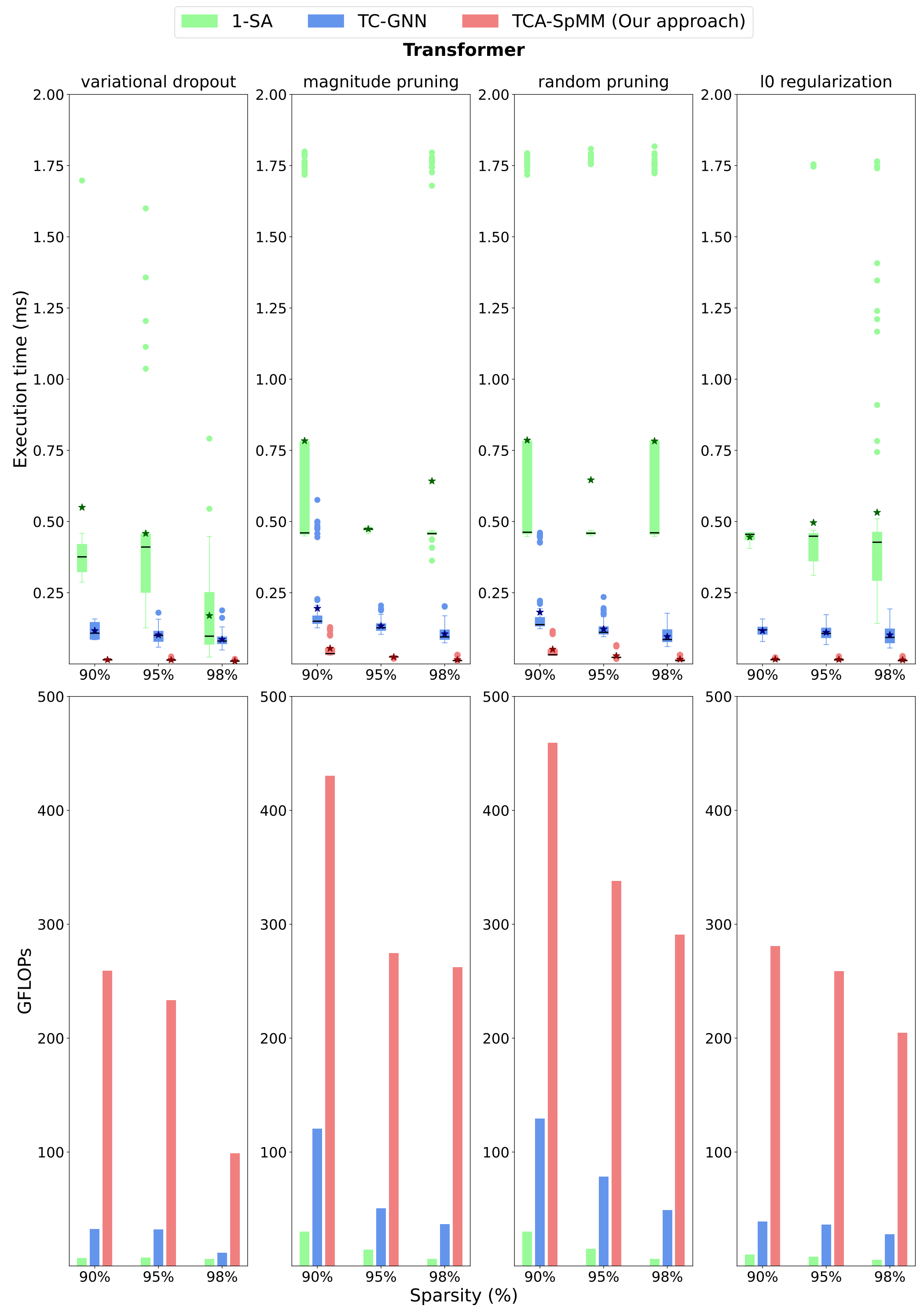

Figure 5. Comparison of the SpMM performance on sparse matrices with a different sparsity on the DLMC dataset. The dimensions of the sparse matrices (M and K) in the DLMC dataset vary for each matrix, whereas we set a fixed size of $N = 256$ for all experiments. If we assume that the sparse matrices from the DLMC are pruned weight matrices, the value of N can be interpreted as the mini-batch size, i.e., the number of data points in each mini-batch.

5.2.2. Effectiveness of Load Balancing Scheme in TCA-SpMM

To demonstrate the effectiveness of our load balancing scheme, we compare the number of MMA operations executed across different CUDA thread blocks, with and without load balancing in our TCA-SpMM implementation. Figure 6 shows the distribution of MMA operations across different thread blocks for two versions of the TCA-SpMM implementation: one that considers load balancing (bottom) and another that does not (top). The x-axis indicates the indices of different CUDA thread blocks, and the y-axis indicates the number of MMA operations computed in each thread block. Note that we assigned a single thread block to compute each output row in order to evaluate the performance of our TCA-SpMM implementation without applying the load balancing scheme. Since we used sparse matrices of size 512 × 512 for this experiment, a sparse matrix with 90% sparsity can be assumed to contain an average of 512 × 0.1 = 51.2 non-zero elements per row. However, depending on the matrix's sparsity pattern, the number of non-zero elements in a row may either exceed or be less than 64, which is the number of elements required for a single MMA operation. Therefore, a matrix with 90% sparsity can lead to a workload imbalance across thread blocks, as each thread block processes a different row of the matrix. As an example, the results from the upper-left graph with 90% sparsity show several outliers in certain thread blocks that process a significantly larger number of MMA operations compared to others. These outliers occur when certain rows have noticeably more non-zero elements compared to others, leading to a significantly higher peak number of MMA operations compared to the majority of thread blocks. Since GPU kernels require all thread blocks to complete before proceeding to the next, the overall performance of the GPUs depends on the completion time of the thread block with the highest workload. As described in Section 4.2.2, we addressed the load imbalance issue caused by exceptionally high workloads by launching additional CUDA thread blocks for rows with a larger number of non-zero elements.

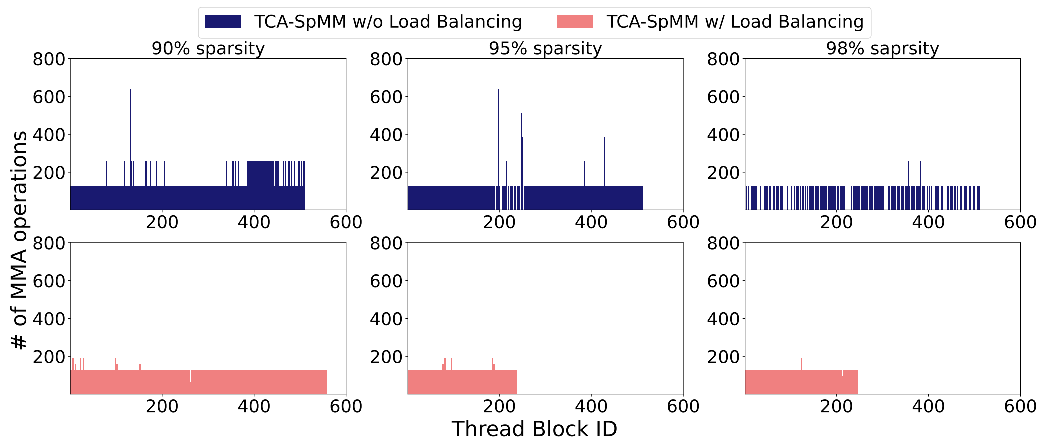

Figure 6. Distribution of MMA operations across different thread blocks in our TCA-SpMM, with and without load balancing. For this experiment, we used DLMC sparse matrices of size 512 × 512 with sparsity ranging from 90% to 98%.

As shown at the bottom of Figure 6, our load balancing scheme effectively balances the distribution of executed MMA operations by distributing the high workloads across multiple CUDA thread blocks. In launching additional thread blocks to handle high workloads, the total number of launched thread blocks increased compared to the case without considering load balancing, as clearly shown in the two leftmost graphs of Figure 6. Although the number of thread blocks used has increased, the total number of MMA operations required for computation is uniformly distributed across the thread blocks. In contrast to the 90% sparsity case, the distribution patterns for sparse matrices with 95% and 98% sparsity, as shown in the middle and the right bar graphs of Figure 6, differ from those of the 90% sparsity matrix. Since the average number of non-zero elements per row in the 95% and 98% sparsity matrices is less than in the 90% sparsity matrix and fewer than 64, our load balancing scheme efficiently reduces both the total number of thread blocks and the total number of MMA operations by assigning a single thread block to compute multiple highly sparse rows. Furthermore, our TCA-SpMM reformulates the matrix layout for thread blocks assigned to computations across multiple rows, enabling each MMA operation to be performed more efficiently. As a result, it is clear that with high sparsity and irregularity in the sparse matrix, our TCA-SpMM implementation, in considering load balancing, dramatically reduces the total number of thread blocks while achieving effective load balancing across thread blocks.

6. Conclusions

In this paper, we present a new parallelization approach for efficiently utilizing Tensor Cores in the SpMM kernel. Since Tensor Cores were originally developed to accelerate GEMM computations using matrix fragments, performing SpMM with the memory-efficient CSR sparse matrix format on Tensor Cores is infeasible. Therefore, to exploit Tensor Cores for SpMM using the CSR format, the vector–vector dot product operation is transformed into blocked matrix–matrix multiplication. Our TCA-SpMM reduces data movement overhead by strategically enhancing data reuse through the efficient mapping of data into shared memory. Furthermore, to address the load imbalance caused by the irregularity of sparse matrices, TCA-SpMM dynamically allocates thread blocks based on the sparsity pattern of rows, enabling finer-grained control and more efficient resource utilization. We systemically analyzed the computational complexity for TCA-SpMM to demonstrate the efficiency of our approach. Experimental results show that, compared to state-of-the-art SpMM parallel implementations, our TCA-SpMM achieves up to a 29.58$\times$ average speedup with highly sparse matrices from the DLMC dataset.

Our Tensor Core-adapted optimization method presented in this paper is primarily developed to accelerate the SpMM kernel for highly sparse matrices. Therefore, we plan to further optimize TCA-SpMM for sparse matrices with lower levels of sparsity, aiming to apply our optimization technique to a broader range of scientific computations involving SpMM kernels, extending beyond deep learning models. In addition, we plan to extend this work by applying our optimization technique to the SDDMM (Sampled Dense–Dense Matrix Multiplication) kernel, which constitutes a large fraction of the multi-head attention operation in sparse transformer models.

Author Contributions: Writing—original draft, Y.H.; methodology, conceptualization, resources, and writing—review and editing, Y.H., I.K. and G.E.M.; project administration, supervision, and writing—review and editing, J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the MSIT (Ministry of Science and ICT), Korea, under the Graduate School of Metaverse Convergence support program (IITP-2023-RS-2022-00156318) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), and in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-RS-2024-00416666). This research was supported by the Chung-Ang University Research Scholarship Grants in 2023.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Ben-Nun, T.; Hoefler, T. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–43. [CrossRef]
2. Moon, G.E.; Kwon, H.; Jeong, G.; Chatarasi, P.; Rajamanickam, S.; Krishna, T. Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *33*, 1002–1014. [CrossRef]
3. NVIDIA. NVIDIA Volta Architecture. Available online: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (accessed on 12 September 2024).
4. NVIDIA. NVIDIA Ampere Architecture. Available online: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (accessed on 12 September 2024).
5. NVIDIA. NVIDIA Hopper Architecture. Available online: https://resources.nvidia.com/en-us-tensor-core (accessed on 12 September 2024).

6. Hoefler, T.; Alistarh, D.; Ben-Nun, T.; Dryden, N.; Peste, A. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.* **2021**, *22*, 10882–11005.
7. Yoon, B.; Han, Y.; Moon, G.E. Layer-Wise Sparse Training of Transformer via Convolutional Flood Filling. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, Taipei, Taiwan, 7–10 May 2024 ; Springer: Berlin/Heidelberg, Germany, 2024; pp. 158–170.
8. Gale, T.; Zaharia, M.; Young, C.; Elsen, E. Sparse gpu kernels for deep learning. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–19 November 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–14.
9. Huang, G.; Dai, G.; Wang, Y.; Yang, H. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–19 November 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–12.
10. Peng, H.; Xie, X.; Shivdikar, K.; Hasan, M.A.; Zhao, J.; Huang, S.; Khan, O.; Kaeli, D.; Ding, C. Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, La Jolla, CA, USA, 27 April–1 May 2024; Volume 2, pp. 683–698.
11. Lee, E.; Han, Y.; Moon, G.E. Accelerated block-sparsity-aware matrix reordering for leveraging tensor cores in sparse matrix-multivector multiplication. In Proceedings of the European Conference on Parallel Processing, Madrid, Spain, 26–30 August 2024; Springer: Berlin/Heidelberg, Germany, 2024; pp. 3–16.
12. Labini, P.S.; Bernaschi, M.; Nutt, W.; Silvestri, F.; Vella, F. Blocking Sparse Matrices to Leverage Dense-Specific Multiplication. In Proceedings of the 2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3), Dallas, TX, USA, 13–18 November 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 19–24.
13. NVIDIA. Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores. Available online: https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/ (accessed on 12 September 2024).
14. AlAhmadi, S.; Mohammed, T.; Albeshri, A.; Katib, I.; Mehmood, R. Performance analysis of sparse matrix-vector multiplication (SpMV) on graphics processing units (GPUs). *Electronics* **2020**, *9*, 1675. [CrossRef]
15. Filippone, S.; Cardellini, V.; Barbieri, D.; Fanfarillo, A. Sparse matrix-vector multiplication on GPGPUs. *Acm Trans. Math. Softw. (TOMS)* **2017**, *43*, 1–49. [CrossRef]
16. Steinberger, M.; Zayer, R.; Seidel, H.P. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU. In Proceedings of the International Conference on Supercomputing, Chicago, IL, USA, 14–16 June 2017; pp. 1–11.
17. Hong, C.; Sukumaran-Rajam, A.; Nisa, I.; Singh, K.; Sadayappan, P. Adaptive sparse tiling for sparse matrix multiplication. In Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, Washington , DC, USA, 16–20 February 2019; pp. 300–314.
18. Jiang, P.; Hong, C.; Agrawal, G. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, USA, 22–26 February 2020; pp. 376–388.
19. Buluç, A.; Fineman, J.T.; Frigo, M.; Gilbert, J.R.; Leiserson, C.E. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, Calgary, AB, Canada, 11–13 August 2009; pp. 233–244.
20. Aktulga, H.M.; Buluç, A.; Williams, S.; Yang, C. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 19–23 May 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 1213–1222.
21. Choi, J.W.; Singh, A.; Vuduc, R.W. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM Sigplan Not.* **2010**, *45*, 115–126. [CrossRef]
22. Ahrens, P.; Boman, E.G. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. *arXiv* **2020**, arXiv:2005.12414.
23. Markidis, S.; Der Chien, S.W.; Laure, E.; Peng, I.B.; Vetter, J.S. Nvidia tensor core programmability, performance & precision. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, Canada, 21–25 May 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 522–531.
24. Hanindhito, B.; John, L.K. Accelerating ml workloads using gpu tensor cores: The good, the bad, and the ugly. In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, London, UK, 7–11 May 2024; pp. 178–189.
25. NVIDIA. NVIDIA Turing GPU Architecture. Available online: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (accessed on 12 September 2024).
26. NVIDIA. NVIDIA Ada Lovelace Architecture. Available online: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf (accessed on 12 September 2024).
27. NVIDIA. CUDA C++ Programming Guide. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (accessed on 12 September 2024).
28. NVIDIA. Parallel Thread Execution ISA Version 8.5. Available online: https://docs.nvidia.com/cuda/parallel-thread-execution/ (accessed on 12 September 2024).

29. Wang, Y.; Feng, B.; Wang, Z.; Huang, G.; Ding, Y. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC 23), Boston, MA, USA, 10–12 July 2023; pp. 149–164.
30. NVIDIA. Exploiting NVIDIA Ampere Structured Sparsity with cuSPARSELt. Available online: https://developer.nvidia.com/blog/exploiting-ampere-structured-sparsity-with-cusparselt/ (accessed on 12 September 2024).
31. Gale, T.; Elsen, E.; Hooker, S. The state of sparsity in deep neural networks. *arXiv* **2019**, arXiv:1902.09574.
32. Lee, N.; Ajanthan, T.; Torr, P.H. Snip: Single-shot network pruning based on connection sensitivity. In Proceedings of the International Conference on Learning Representations (ICLR), New Orleans, LA, USA, 6–9 May 2019.