SSpMM: Efficiently Scalable SpMM Kernels Across Multiple Generations of Tensor Cores

Zeyu Xue, Mei Wen, Jianchao Yang, Minjin Tang, Zhongdi Luo, Jing Feng, Yang Shi, Zhaoyun Chen, Junzhong Shen, and Johannes Langguth

***Abstract*—Sparse-Dense Matrix-Matrix Multiplication (SpMM) has emerged as a foundational primitive in HPC and AI. Recent advancements have aimed to accelerate SpMM by harnessing the powerful Tensor Cores found in modern GPUs. However, despite these efforts, existing methods frequently encounter performance degradation when ported across different Tensor Core architectures. Recognizing that scalable SpMM across multiple generations of Tensor Cores relies on the effective use of general-purpose instructions, we have meticulously developed a SpMM library named *SSpMM*. However, a significant conflict exists between granularity and performance in current Tensor Core instructions. To resolve this, we introduce the innovative *Transpose Mapping Scheme*, which elegantly implements fine-grained kernels using coarse-grained instructions. Additionally, we propose the *Register Shuffle Method* to further enhance performance. Finally, we introduce *Sparse Vector Compression*, a technique that ensures our kernels are scalable with both structured and unstructured sparsity. Our experimental results, conducted on four generations of Tensor Core GPUs using over 3,000 sparse matrices from well-established matrix collections, demonstrate that *SSpMM* achieves an average speedup of 2.04 ×, 2.81 ×, 2.07 ×, and 1.87 ×, respectively, over the state-of-the-art SpMM solution. Furthermore, we have integrated *SSpMM* into PyTorch, achieving a 1.81 × speedup in end-to-end Transformer inference compared to *cuDNN*.**

***Index Terms*—GPGPU, tensor core, CUDA, SpMM.**

I. Introduction

SPARSE-DENSE Matrix-Matrix Multiplication (SpMM) has emerged as a pivotal computational primitive, significantly influencing the total execution time in a wide array of foundational applications, spanning from Artificial Intelligence [1], [2], [3], [4], [5], [6], [7] to Scientific Computing [8], [9], [10], [11], [12]. The multi-level memory hierarchy and pipeline parallelism scheduling of NVIDIA GPUs bring more optimization opportunities for SpMM acceleration. Due to the diversity of CUDA programming and the fast upgrade rate of GPU hardware, it is challenging to maintain the efficient scalability of the SpMM kernels.

TABLE I
FP16 Performance Comparison of Four Generations of NVIDIA Tensor Core GPUs [19], [20], [21], [22]

GPU (Architecture)	CUDA Core (FP16@TFLOPS)	Tensor Core (FP16@TFLOPS)
V100 (Volta)	31.4	125
2080Ti (Turing)	28.5	108
A100 (Ampere)	78.0	312
4090 (Ada)	82.6	330

Tensor Cores, the computing power backbone within GPUs, provide significantly superior performance, delivering up to four times the throughput of CUDA Cores, as demonstrated in Table I. The input of Tensor Cores is fixed-size matrices. And Tensor Cores require massive parallelism to unlock their immense computing power, but parallelism of unstructured sparse matrices is constrained. Consequently, prior approaches to unstructured SpMM acceleration predominantly utilize CUDA Cores [13], [14], [15], [16], [17], [18]. Due to the fine granularity and randomness inherent in sparse matrices, achieving high parallelism while maintaining data compression is challenging.

With the evolution of Tensor Core hardware, many kernels exhibit a lack of scalability when migrated across different architectures. Some works [2], [23], [24], [25], [26] are even unable to run on all Tensor Core architectures. To evaluate the performance of state-of-the-art (SOTA) SpMM solutions across various architectures and sparsity[1] levels, we selected two leading solutions: *Sputnik*, which accelerates unstructured SpMM on CUDA Cores [14], and *vectorSparse*, which accelerates structured (vector-wise) SpMM on Tensor Cores [27]. We used 2703 matrices with 7 different sparsity levels as benchmarks. GPU performance varies with architectural upgrades, as illustrated in Table I. Fig. 1(d) shows that the performance trend of *Sputnik* is nearly consistent with the performance variation of CUDA Cores, meaning it maintains similar performance percentages across different architectures and sparsity levels. In contrast, the performance percentage of *vectorSparse* (Fig. 1(b)) decreases with architectural upgrades. Notably, its performance percentage is particularly low on A100 and RTX 4090. In this paper, we propose a Tensor Core-based SpMM acceleration solution called *SSpMM*. It performs both structured SpMM (Fig. 1(a))

Received 9 December 2024; revised 30 June 2025; accepted 28 September 2025. Date of publication 1 October 2025; date of current version 20 October 2025. This work was supported by the Natural Science Foundation of Hunan Province, China under Grant 2023JJ40679 and Grant 2024JJ6470. Recommended for acceptance by F.M.M. Ciorba. *(Corresponding author: Mei Wen.)*

Zeyu Xue, Mei Wen, Jianchao Yang, Minjin Tang, Zhongdi Luo, Jing Feng, Yang Shi, Zhaoyun Chen, and Junzhong Shen are with the College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China, and also with the Key Laboratory of Advanced Microprocessor Chips and Systems, Changsha 410000, China (e-mail: xuezeyu18@nudt.edu.cn; meiwen@nudt.edu.cn).

Johannes Langguth is with Simula Research Laboratory, 0164 Oslo, Norway, and also with University of Bergen, 5020 Bergen, Norway.

Digital Object Identifier 10.1109/TPDS.2025.3616981

[1]In this paper, sparsity refers to the proportion of zero elements.

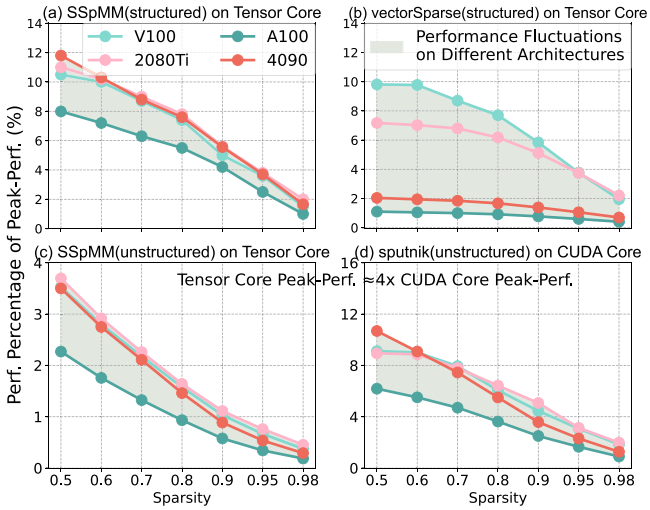

Fig. 1. Performance percentages- and fluctuations- of SpMM solutions on four GPU architectures.

and unstructured SpMM (Fig. 1(c)) with stable performance results on different architectures, consistently outperforming the corresponding SOTA solutions.

As hardware iterates at an increasingly rapid pace, the significance of hardware upgrades is greatly diminished if the accompanying software cannot achieve effective and scalable performance [28]. We aim to use Tensor Cores to develop high-performance SpMM kernels with efficient scalability and supporting for both structured and unstructured sparsity. We identify three primary challenges: *(1) Conflict between instruction granularity and efficiency.* The performance degradation of *vectorSparse* across Tensor Core architectures can be attributed to its reliance on the `mma.m8n8k4` instruction, which is optimized for the Volta architecture and may exhibit substantially reduced performance on other GPU versions [29]. To mitigate performance degradation caused by architecture-specific instructions during migration, it is essential to utilize general-purpose instructions as the cornerstone for achieving architectural scalability. However, current Tensor Core instructions often suffer from either too coarse or inefficient instruction granularity, making it challenging to achieve both fine granularity and operational efficiency [23], [25], [26], [27]. *(2) Complex data layout for MMA instruction and inefficient data movement.* Scalability of SpMM kernels necessitates the use of coarse-grained general-purpose instructions, which inevitably leads to complex data layouts when programming. In SpMM, memory access latency constitutes a significant portion of the kernel execution time, making it crucial to employ an efficient coalesced access pattern. However, coalesced data loads often struggle to adapt to non-contiguous data layouts inherent in these instructions. Consequently, the development of efficient SpMM kernels is confronted with the dual challenges of complex data layout for MMA instruction and inefficient data movement. *(3) Lack of unstructured sparsity support in programming interface.* The instructions of Tensor Cores necessitate input matrices with fixed dimensions, which aligns well with structured matrix multiplication tasks. However, unstructured sparse matrices are commonly stored in linear compression formats such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) [30], which complicates the exploitation of parallelism on Tensor Cores. Consequently, to further enhance the scalability of SpMM kernels in terms of computational granularity, it is essential to address the conflict between parallelism and utilization. This involves finding ways to effectively leverage the parallelism while accommodating the irregular data structures of sparse matrices.

To address the aforementioned challenges, we propose *SSpMM*.[2] We make the following contributions:

- To mitigate performance degradation, proper utilization of general-purpose instructions is crucial. We introduce *Transpose Mapping Scheme* to tackle *Challenge (1)*. This scheme leverages the transpose property of matrix multiplication to shift the N-dimension of instructions to the M-dimension of SpMM. This allows SpMM to operate at a finer granularity than typically permitted by coarse-grained general-purpose instructions. (Section III)
- To address *Challenge (2)* and further enhance the performance of SpMM, we propose *Register Shuffle Method*. This method reorganizes the input and output data of Tensor Core, resolving the issues of inefficient data movement and mismatches in data layout for MMA instructions. (Section IV)
- To address *Challenge (3)*, we introduce *Sparse Vector Compression*. This strategy dynamically selects vector lengths to compress unstructured sparse matrices using the Column Vector Sparse Encoding (CVSE) format. It significantly enhances scalability in terms of computational granularity, making it suitable for AI applications and scientific computing. Combined with the previous optimization strategies, we develop a high-performance SpMM library, *SSpMM*. (Section V)

We select over 3,000 sparse matrices from Deep Learning Matrix Collection [31] and SuiteSparse Matrix Collection [32] as benchmarks, with sizes ranging from 512 × 512 to 1.38M × 1.38M. And experiments are conducted on four generations of GPUs, including Data-Center-GPUs and Consumer-GPUs. Compared to the SOTA method for unstructured SpMM, *SSpMM* achieves speedups of 2.04 ×, 2.81 ×, 2.07 ×, and 1.87 × on four generations of GPUs. Against the SOTA structured SpMM, *SSpMM* shows improvements of 1.12 ×, 1.15 ×, 6.92 ×, and 5.75 ×. For end-to-end inference, integrating *SSpMM* into an inference framework results in a 1.33 × speedup in latency over *vectorSparse* and 1.81 × over *cuDNN*.

II. Background

A. Programming Tensor Cores

The inputs and outputs of Tensor Cores are matrices of fixed size, leading to relatively coarse programming granularity [33], [34], [35]. Tensor Cores offer three types of instructions. The first type, Warp Matrix Multiply-Accumulate (WMMA), is accessible through both the CUDA C++ API and the Parallel Thread

[2]https://github.com/xuezy-mmi/SSpMM

TABLE II
SHAPES OF MMA INSTRUCTIONS WITH DIFFERENT PRECISION

Precision	Shape
FP64	m8**n8**k4 m16**n8**k4, m16**n8**k8, m16**n8**k16
TF32	m16**n8**k4, m16**n8**k8
FP16	m8**n8**k4 m16**n8**k8, m16**n8**k16
BF16	m16**n8**k8, m16**n8**k16
FP8	m16**n8**k32
INT8	m8**n8**k16, m16**n8**k16, m16**n8**k32
INT4	m8**n8**k32, m16**n8**k32, m16**n8**k64
INT1	m8**n8**k128, m16**n8**k128, m16**n8**k256

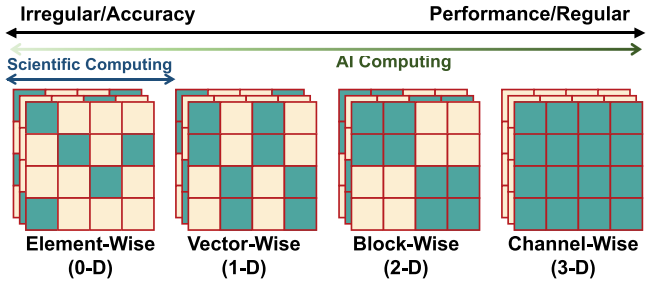

Fig. 2. Four levels of sparse patterns, each created by different algorithms, with the green blocks indicating non-zero elements.

Execution (PTX) API. This dual availability offers a high level of abstraction, effectively obviating the necessity for intricate data arrangement [29], [36]. However, WMMA instructions exhibit fixed granularity and a predetermined memory access pattern. Consequently, they lack flexible operational modes, thereby constraining the optimization landscape available to developers. The second type, Warp Group Matrix Multiply-Accumulate (WGMMA), operates at a coarser granularity than WMMA. Despite their enhanced performance capabilities, WGMMA instructions are not readily adaptable across diverse architectural platforms. This limitation renders them unsuitable for implementing fine-grained SpMM, thereby restricting their scalability in scenarios demanding architectural versatility.

The third instruction type, Matrix Multiply-Accumulate (MMA), is available solely through the PTX API. As shown in Table II, MMA instructions can invoke Tensor Cores in various configurations. Unlike WMMA, MMA instructions do not necessitate specific registers (i.e., Fragments), nor do they require dedicated Load/Store instructions. Instead, they operate directly on general-purpose registers, albeit with stringent requirements on input layout. This complexity, while presenting programming difficulties, offers greater flexibility and a broader potential optimization space, allowing for more nuanced performance tuning.

By observing all Tensor Core computational instructions [29], it can be found that the minimum M-dimension of the instruction shape is 8. Therefore, to fully utilize the computing power of Tensor Core, the minimum Block_M of SpMM kernels should be 8.

B. Sparsity Patterns in SpMM

Prior research has eliminated the inherent redundancy in DNN models [37], [38], [39]. These algorithms aim to reduce the model based on different objectives [40], [41], [42], [43], [44], resulting in multi-dimension sparse patterns (i.e., 0-D, 1-D, 2-D, or 3-D), as shown in Fig. 2. The increased latency is a trade-off for achieving higher accuracy, as irregular AI models can capture nuanced patterns and relationships within the data. Conversely, structured model features reduce latency but sacrifice accuracy.

In scientific computing applications, data often adheres to a power-law distribution [45], [46], manifesting in patterns like block structures or diagonal distributions. While this data typically exhibits structured characteristics, the presence of unavoidable noise prevents it from being stored in a strictly structured format. This noise introduces variability that can disrupt the structured sparse patterns, making it challenging to strictly adhere to the high precision- and accuracy- requirements of scientific computing. Therefore sparse data in scientific computing applications must be stored in unstructured pattern (i.e., 0-D).

Algorithm 1: Pseudocode for Parallel CVSE SpMM on GPU.

Input: vec_M, K, N, V,// V = Block_M, M = vec_M × V
vecPtr[vec_M+1], colInd[NNZV],
// NNZV means Number of Non-Zero-Vector
LHS[NNZV × V], RHS[K × N]
Output: Output[M × N]

```
1: for bm ← 0 to M step Block_M in parallel do
2:   for bn ← 0 to N step Block_N in parallel do
3:     //Loops in Thread Block
4:     for p ← vecPtr[m]to vecPtr[m+1] do
5:       for v ← 0 to V-1 step 1 do
6:         for n ← 0 to Block_N-1 step 1 do
7:           Output[(bm × V+v) ×N+bn × Block_N+n]
             +=
             LHS[p × V+v] ×
             RHS[colInd[p] ×N+bn × Block_N+n];
8:         end for
9:       end for
10:    end for
11:  end for
12: end for
```

Sparsity is a key factor in reducing both storage overhead and computational workload. Structured sparsity, while not achieving the same level of parallelism as dense computations, allows coarse-grained data to effectively utilize the power of accelerators. However, unstructured sparsity presents challenges for these dense computation-oriented accelerators due to the irregularity. This irregularity can hinder efficient parallel processing and optimization.

Wang et al. [27] found that vector-wise sparsity offers the same reusability as block-wise sparsity in SpMM. This means vector-wise can achieve similar acceleration efficiency as block-wise on Tensor Cores. They introduced a vector-wise oriented CVSE format, which builds on the coding rules of the CSR format, as shown in Fig. 3. In CVSE format, all elements within

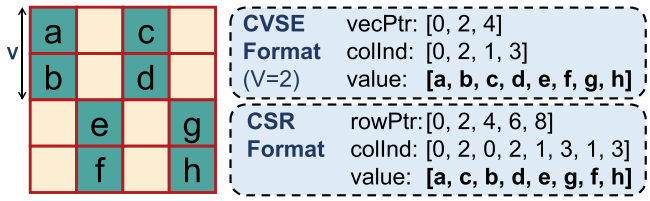

Fig. 3. Vector-wise sparse matrix and its CSR- and CVSE- representations.

a vector share one column index. This simplifies both encoding and decoding processes, making them computationally efficient. CVSE-based SpMM leverages this simplicity to enhance performance, and its parallel pseudocode is shown in Algorithm 1. Owing to the mapping scheme leveraging the transpose property of matrix multiplication designed in this paper, matrix A is not on the left and matrix B is not on the right of the multiplication operator according to conventional practice. To disambiguate between the input matrices of the SpMM algorithm and those of the MMA instruction, we adopt the following nomenclature throughout this paper: the sparse input matrix is denoted as the Left-Hand-Side (LHS) matrix, the dense input matrix is designated as the Right-Hand-Side (RHS) matrix, while the input operands for MMA instructions are explicitly identified as `Input_A` and `Input_B`, respectively.

In GPUs, SpMM is divided into multiple Thread Blocks, each of which performs a fixed-size (Block_M $\times$ Block_N) computation. Vector-wise sparsity provides the smallest granularity among structured sparse patterns, offering increased flexibility.

III. Transpose Mapping Scheme for Coarse-Grained Instructions

A. Conflict Between Instruction Granularity and Efficiency

As delineated in Section II-B, there exists a direct correlation between the structural granularity and the resultant model accuracy. Specifically, finer structural granularity enhances model accuracy. Consequently, to optimize the execution of models after fine-grained pruning, it is imperative that the SpMM operation be conducted at the smallest feasible granularity, with particular emphasis on minimizing the height (i.e., Block_M) of each sparse matrix tile.

In accordance with the exposition in Section II-A, Tensor Cores provide three distinct programming interfaces: WMMA APIs, WGMMA APIs, and MMA APIs. Notably, the WGMMA instructions are characterized by a granularity that is excessively coarse, rendering them unsuitable for executing fine-grained SpMM. The WMMA API necessitates that input data be encapsulated within Fragments, which are loaded via the `wmma::load` instruction. A critical limitation of this instruction is its inability to fetch non-contiguous data based on indices. This necessitates that the RHS matrix, which requires indexed access, be initially transferred from Global Memory to Shared Memory before being loaded into Fragments using `wmma::load`. Given the inherently low reuse of the RHS matrix in SpMM contexts, employing Shared Memory as an intermediate storage medium proves inefficient [27]. Hence, the application of WGMMA and WMMA instructions for fine-grained SpMM is suboptimal.

MMA instructions offer the flexibility to utilize general-purpose registers as inputs, allowing computations to be performed through efficient adaptive access modes. As indicated in Table II, MMA computation instructions are categorized into two types: m=8 and m=16. Some mma instructions with m=8 are optimized for specific architectures, providing easier programming interfaces and performance benefits. However, these optimizations result in a performance reduction for the corresponding instructions on non-specific architectures, and this architectural specificity leads to scalability challenges for SpMM acceleration solutions (e.g., vectorSparse) on modern Tensor Core architectures. To effectively integrate with the pruning algorithm and maintain the fine-grained nature of SpMM, controlling the Block_M size to 8 is crucial. The mma instruction with m=16, although efficient across various architectures, struggles to implement fine-grained SpMM due to its larger M-dimension. Specifically, when performing an SpMM with Block_M = 8 using the m=16 instruction, 50% of the arithmetic capacity is wasted, as only half of the computational power is utilized. This inefficiency underscores the need for strategies that better align with the architecture's capabilities to optimize performance.

The development of efficiently scalable SpMM kernels fundamentally hinges on the strategic use of general-purpose instructions. However, it has been determined that WMMA, WGMMA, and MMA instructions all face significant challenges in executing SpMM with optimal efficiency. A detailed analysis reveals that these Tensor Core instructions are hindered by a conflict between computational granularity and computational efficiency. Specifically, MMA instructions with m=16 and WGMMA instructions exhibit high computational efficiency but suffer from coarse computational granularity. Conversely, MMA instructions with m=8 and WMMA instructions offer finer computational granularity but at the cost of reduced computational efficiency. This dichotomy underscores the necessity for innovative strategies to harness the full potential of modern Tensor Core architectures, particularly for fine-grained computations.

Despite the coarse granularity associated with MMA instructions having m=16, a critical observation can be made regarding their dimensional characteristics: all possess an N-dimension of 8, as delineated in Table II. Leveraging the transpose property of matrix multiplication, it becomes feasible to transpose the N-dimension of MMA instructions to the M-dimension of SpMM. This enables the effective execution of SpMM with Block_M = 8 by employing general-purpose MMA instructions with n=8, thereby addressing the aforementioned *challenge(1)*: Conflict between instruction granularity and efficiency. And this solution is specified in Section III-B.

B. Transpose Mapping Scheme

In parallel computing, the efficient mapping of SpMM is paramount. However, the inherent conflict between instruction granularity and computational efficiency often poses significant challenges. To mitigate this issue, we propose a novel *Transpose*

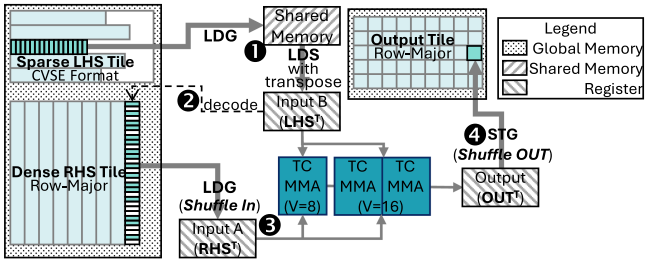

Fig. 4. Transpose Mapping Scheme. Utilizing different memory access pattern, the LHS Tile is loaded directly into Shared Memory and then loaded with transpose into `Input_A`, while the RHS Tile is loaded directly into `Input_B`.

Mapping Scheme based on coarse-grained MMA instructions, as illustrated in Fig. 4. According to the transpose property of matrix multiplication (i.e., $(C_{M\times N})^T = (A_{M\times K}B_{K\times N})^T = (B^T)_{N\times K}(A^T)_{K\times M}$), the N-dimension of the matrix multiplication can be transposed to the M-dimension. This scheme utilizes coarse-grained efficient instructions to achieve fine-grained SpMM, optimizing the memory access and the overall efficiency of SpMM on Tensor Cores.

The LHS matrix is strategically stored in Global Memory using compression format like CVSE, while the RHS matrix is preserved in Global Memory utilizing a conventional Row-Major format. The reuse characteristics of these data significantly influence our approach to memory access and computational efficiency. Due to the limitation of the compression format parameter, usually, the `Block_M` of SpMM is smaller than `Block_N`, so the reusability of the RHS Tile is lower than that of the LHS Tile. As shown in Fig. 4, ❶ given the higher reusability of the LHS Tile, it is loaded directly into Shared Memory in its original format for reuse and decoding indices. Given the low latency for transfers between shared memory and registers, the LHS Tile can be efficiently loaded from shared memory into registers. Within the registers, it is stored in transposed form as `Input_B` to facilitate subsequent computations on Tensor Cores. ❷ Subsequently, the decoded indices retrieved from registers are used to access corresponding rows within the RHS matrix. From an instruction pipeline perspective, a significant latency exists between the issuance of the LHS Tile load instruction and the availability of the decoded indices. During this interval, memory access instructions for the RHS Tile cannot be issued. Consequently, the memory access unit is forced to remain idle, resulting in stall-induced bandwidth underutilization that is fundamentally unavoidable.

❸ The RHS Tile has low reusability and does not require inter-thread data sharing, so loading it to Shared Memory is inefficient. Following this feature, the RHS Tile is directly loaded into registers without format conversion. To optimize the GPU's high bandwidth capabilities, both RHS Tiles and LHS Tiles are transferred from Global Memory to on-chip storage employing coalesced memory access patterns (e.g., `LDG.128` (CUDA instruction: load 128-bit data from Global Memory to Register)). This method not only enhances data transfer rates but also mitigates potential bandwidth bottlenecks. However, the transposition of the RHS Tile into the `Input_A` format for Tensor Core operations introduces additional complexity. To address the layout discrepancy, we implement a novel method (i.e., Register Shuffle Method) as elucidated in Section IV. This method is essential in optimizing memory access during the multiplication process and ensuring that the input and output matrices are aligned correctly for efficient computation.

❹ Upon the successful transposition of both the LHS Tiles and RHS Tiles into the input registers, Tensor Cores execute the MMA instructions, producing the transposed output matrix, denoted as $Output^T$. Subsequently, *Register Shuffle Method* is employed again to transpose the $Output^T$ matrix before it is written back to Global Memory, ensuring the correct data layout and employing coalesced memory access patterns.

Importantly, the proposed *Transpose Mapping Scheme* fully supports all MMA instructions with N = 8, thereby facilitating the comprehensive utilization of available Tensor Core instructions. This capability is pivotal in achieving efficient SpMM across a diverse range of Tensor Core architectures. By systematically addressing the idiosyncrasies of matrix data formats and leveraging advanced memory management techniques, our approach not only enhances computational efficiency but also contributes to the broader discourse on optimizing high-performance computing workflows in contemporary architectures. This alignment with architectural capabilities underscores the practicality and relevance of the proposed scheme in modern computational applications.

IV. Register Shuffle Method

A. Complex Data Layout and Inefficient Data Movement

The integration of MMA instructions presents significant challenges regarding the alignment of data layouts and efficient data movement, particularly following the implementation of the *Transpose Mapping Scheme*. This section elucidates the complexities associated with these requirements and their implications for performance in computing environments.

A notable aspect of the MMA instruction set is its intricate data layout requirements, which diverge from the traditional linear data structures typically employed in matrix operations. Specifically, the inputs and outputs necessitated by each thread exhibit non-consecutive access patterns, as depicted in Fig. 5. Specifically, the `Input_A` necessitates that each thread fetch two groups of two contiguous data in row-major order, while the `Input_B` requires each thread to fetch one group of two contiguous data in column-major order. This characteristic complicates the process of optimizing data retrieval and storage, exacerbating the overhead of data movement between off-chip Global Memory and on-chip resources. The necessity for efficient coalesced memory access (e.g., `LDG.128`) becomes paramount to mitigate performance degradation. However, it is crucial to note that coalesced accessed data cannot be seamlessly mapped into the data layout of the MMA instructions. To address these challenges, numerous approaches have emerged, including the utilization of Shared Memory for staging data [33] or employing `__shfl` instructions for inter-thread data exchange [23], [27], [47]. However, as discussed in Section III, the efficacy of Shared Memory is compromised in SpMM, due to the varying degrees of

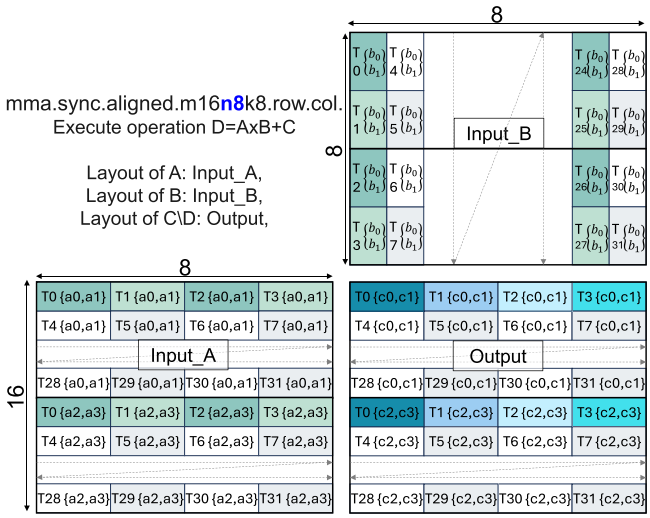

Fig. 5. Input- and Output- data layout of `mma.m16n8k8` instruction.

Tile reusability. Furthermore, while `__shfl` instructions facilitate data swapping among threads, they introduce the burden of thread synchronization and incur substantial register overhead, thereby rendering the data movement process more inefficient. Consequently, the complex data layout prerequisites inherent in MMA instructions exacerbate the inefficiencies associated with data movement operations.

Additionally, the contrasting requirements for data representation in MMA instructions further compound these challenges: `Input_A` necessitates a Row-Major format, while `Input_B` requires a Col-Major format. To optimize the time overhead associated with data preprocessing, we endeavor to maintain the data in its raw format. Specifically, the LHS matrix is represented in a vector compression format analogous to the CVSE format, which will be addressed in Section V, whereas the RHS matrix is maintained in an element-by-element Row-Major format. Following the application of the *Transpose Mapping Scheme*, the LHS Tile transposes to a Row-Major arrangement, while the RHS Tile adopts a Col-Major format. However, the requirement of the *Transpose Mapping Scheme* for loading the RHS Tile into `Input_A` and the LHS Tile into `Input_B` introduces a significant mismatch between the anticipated data formats and their actual mappings. To reconcile this disparity and achieve appropriate data layouts, a substantial allocation of on-chip registers is necessitated as intermediate storage to reorganize the data prior to its input into the Tensor Core. However, this excessive reliance on registers — which serve as private resources for individual threads — results in a compromise of thread-level parallelism. Thus, the outcomes of employing the *Transpose Mapping Scheme* in conjunction with MMA instructions manifest a profound challenge in achieving congruence between the data layout requirements and the operational constraints imposed by the hardware architecture.

In summary of *Challenge (2)*, the use of MMA instructions induces the intricate data layout requirements, compounded by the performance limitations of existing memory management method, engendering significant inefficiencies in data movement within high-performance computing systems. Recognizing and addressing these challenges are paramount for optimizing the utilization of MMA instructions and enhancing computational performance in increasingly demanding applications.

B. Shuffle in and Shuffle Out

To address the *Challenge (2)*, we have devised *Register Shuffle Method* based on the register renaming technique. This method consists of two main steps: *Shuffle In* and *Shuffle Out*. These steps respectively handle the mismatch between the input data and the required input layout, as well as the mismatch between the output data and the write-back position. *Register Shuffle Method* is not specific to any particular precision and operates with 128-bit data as the granularity for data reorganization, providing the necessary inputs and outputs for Tensor Core operations. In order to facilitate the concrete elaboration of the execution process of this method, this section selects FP16 precision and 128 b operation granularity as an example to explain in detail.

During the execution of *Transpose Mapping Scheme*, the RHS Tile is loaded directly from Global Memory into registers as `Input_A` of Tensor Core using coalesced access instruction. However, this direct loading can lead to layout matching difficulties. To address this, `Shuffle In` is employed. This method reorganizes the data of two 128-bit vectors and converts them into eight 32-bit vectors, each suitable for the input layout requirements of a single thread. As shown on the left side of Fig. 6, each thread loads two 128 b vectors (2x8 FP16 data, i.e., a-b-c-d-e-f-g-h and i-j-k-l-m-n-o-p) into registers, and converts these two sets of 128 b register data into eight sets of 32 b register data (i.e., a-i, b-j, c-k, d -l, e-m, f-n, g-o, h-p) by means of register renaming. This 8 sets of data just match the `Input_A` layout after executing *Transpose Mapping Scheme*. This process reorganizes the data without additional register overhead and also utilizes the GPU's efficient coalesced access instructions.

After executing the MMA instruction, the output data from the Tensor Core is transposed and discontinuous. This means that data with adjacent addresses in the output are also discontinuous at the write-back location. Therefore, `Shuffle Out` is utilized to interleave the output data of each thread. This operation is the opposite of `Shuffle In`, which converts eight 32 b output vectors back into two 128 b vectors. As shown on the right side of Fig. 6, the eight sets of 32 b output data (i.e., 0-1, 2-3, 4-5, 6-7, 8-9, 10-11, 12-13, and 14-15) are convert into two sets of 128 b data (i.e., 0-2-4-6-8-10-12-14 and 1-3-5-7-9-11-13-15) and utilized with the coalesced access instruction `STG.128` (GPU instruction: store 128-bit data to Global Memory) to write these two 128 b vectors back to Global Memory. In the end, the data is not only in the correct order, but also conforms to the 128 b coalesced transaction and the long vector memory operation, thus efficiently utilizing the bandwidth [27].

This process is very similar to the shuffling operation in poker, where neat cards (input data) are interleaved (into the Tensor Core) and regrouped into a new set of cards (output data), hence the name *Register Shuffle Method*. It is worth noting that *Register Shuffle Method* works with any precision, and usually operates at a granularity of 128 bits. The number of groups to be shuffled can

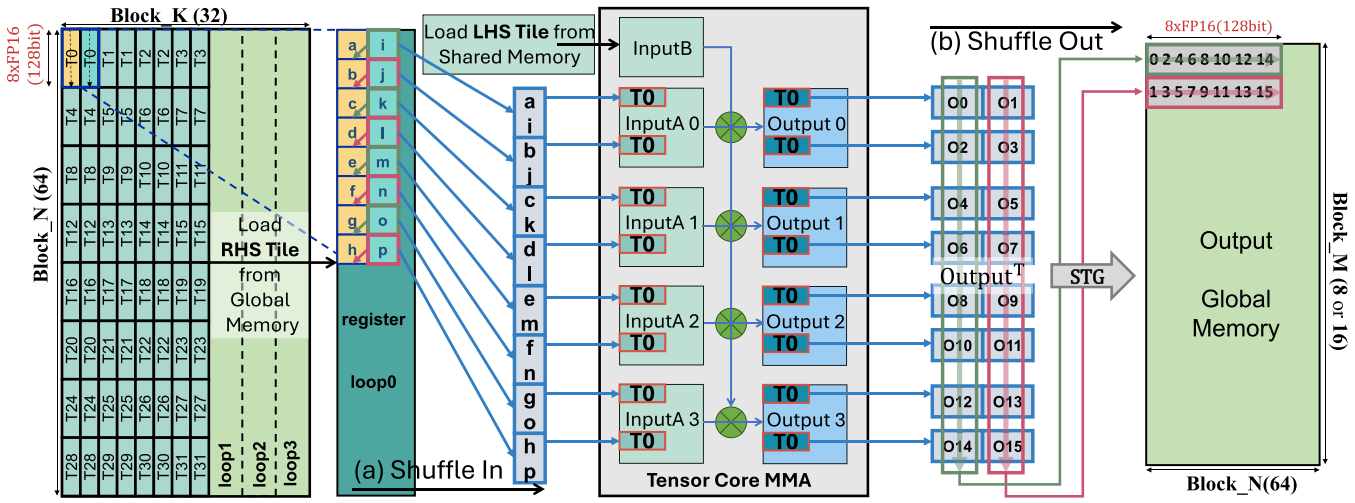

Fig. 6. *Register Shuffle Method*. (a) Shuffle In. By register renaming, the data stored in its original form is transposed and matched with the Tensor Core input layout. (b) Shuffle Out. A similar method is exploited to restore the transposed output matrix and write it back to Global Memory.

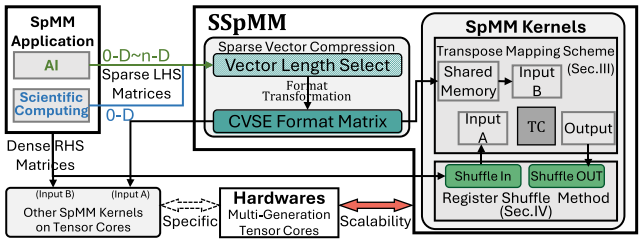

Fig. 7. Overview of *SSpMM*.

also be changed according to the data layout of the instruction. Two vectors are used for shuffling in this section because the MMA instruction of FP16 requires 2 consecutive elements as input and output for each thread, as shown in Fig. 5. *Register Shuffle Method* provides effective insight into MMA instruction programming while utilizing efficient coalesced memory access instructions and highly efficient MMA instructions, solving the problem of inefficient data movement and the mismatch between data layout and MMA instruction requirement.

V. Implementation and Optimization for SSpMM

Combining two optimization strategies, *Transpose Mapping Scheme* and *Register Shuffle Method*, we are able to achieve performance scalable structured SpMM kernels. However, structured sparsity is more common in AI applications, and the granularity scalability of SpMM needs to be extended if we want to accelerate high-accuracy scientific computing applications. Therefore, we propose *Sparse Vector Compression*, integrating both element-wise and vector-wise sparsity while offering configurable vector length, allowing for balance between computational utilization and parallelism. Ultimately, we have implemented an efficient and scalable SpMM library, *SSpMM*, on multiple generations of Tensor Core in this section. And its overview is shown in Fig. 7.

A. Sparse Vector Compression

There have been numerous efforts to accelerate unstructured SpMM on GPUs [13], [14], [15], [16], [17], [18]. However, they have largely relied on CUDA Cores rather than taking advantage of Tensor Cores, while the latter provides significant performance advantages. Our work focuses on accelerating fine-grained SpMM on Tensor Cores. And the storage format is the key to distinguish the granularity of the computation.

To conserve storage space and transmission latency, sparse data is commonly stored in a compression format. This compression format assigns specific indices to non-zero elements to denote their locations in the original matrix. However, the compressed sparse matrix loses the high degree of parallelism inherent in its dense counterpart. In matrix multiplication, columns of the LHS matrix correspond one-to-one with rows of the RHS matrix. multiple rows or columns of the result matrix are typically computed in parallel. However, unstructured sparse matrices, when compressed into specific formats, are usually stored by individual rows (e.g., CSR) or individual columns (e.g., CSC). Consequently, SpMM requires extra overhead for index retrieval. Furthermore, the unique indices of the non-zero elements reduce the potential parallelism between multiple rows [48], [49].

As described in Section II-A, Tensor Cores impose stringent requirements on the layout of the input data. All Tensor Core compute instructions necessitate fixed-size matrices as input. However, compressed sparse matrices do not display structured characteristics and are difficult to directly enter into Tensor Cores for computation. Forcing an unstructured compressed format onto Tensor Cores can only exert its computing power of $1/M$ (M is the M-dimension of Tensor Core instructions).

In summary, unstructured sparse matrices have low parallelism in SpMM and are difficult to directly enter Tensor Cores for computation using traditional compression formats. Therefore, we propose *Sparse Vector Compression* to solve this problem.

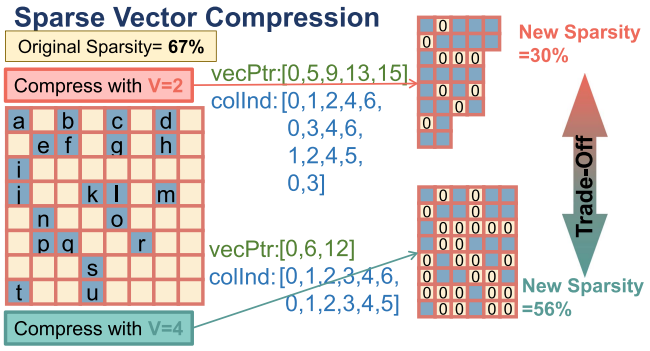

Fig. 8. *Sparse Vector Compression* and Trade-Off between Computational Utilization and Computational Parallelism.

From the analysis of the sparse patterns in Fig. 2, with the exception of element-wise sparsity, all other structured sparse patterns can achieve parallelism between multiple rows. Furthermore, vector-wise sparsity, being the finest granularity in all structured sparse patterns, not only retains the same high parallelism as block-wise [27], but also easily combines with unstructured data. Consequently, we select vector-wise sparsity as the basis for *Sparse Vector Compression*.

We introduce sparsity into the CVSE format, as depicted in Fig. 8. CVSE format can store structured vectors, but *Sparse Vector Compression* allows zero elements within these vectors. If elements in a vector are all zero, the vector will be deleted. Therefore, the CVSE format can compress multiple rows of unstructured data into structured vectors. Compared to dense matrices, we compress the zero-vectors in the sparse matrix to achieve storage advantages. Although the data compression rate is not as high as that of fully compressed formats like CSR format, unstructured sparse data presented in CVSE format acquires higher parallelism for SpMM.

B. Trade-Off Between Utilization and Parallelism

After *Transpose Mapping Scheme*, the Block_M of SpMM is minimized to 8. Therefore, in *Sparse Vector Compression*, the V (vector length) can be set as a multiple of 8. As illustrated in Fig. 8, for unstructured sparse matrices, a larger V results in fewer all-zero vectors, retaining more redundant zero elements (i.e., higher new sparsity). This leads to computing power being wasted on redundant computation. However, a larger V can achieve higher computational parallelism. Hence, selecting different V values becomes a trade-off between utilization and parallelism.

To strike a balance between computational parallelism and computational utilization, we have also designed multiple kernels with varying values of V. This approach mirrors the previous mapping method, extending only the Block_M of SpMM. This allows for more effective use of on-chip resources and achieves higher computational parallelism.

We set V in *Sparse Vector Compression* to equal M in Tensor Core instructions, enabling the compressed unstructured LHS matrix to be computed on Tensor Cores across multiple rows simultaneously. By retrieving the corresponding rows in the RHS matrix according to `colInd`, we can efficiently implement unstructured SpMM on Tensor Cores.

Algorithm 2: Pseudocode for *SSpMM* on Tensor Core.

Input: M, K, N, V,
vecPtr[ ], colInd[ ]
LHS[ ], RHS[ ], Output[ ]
vec_M = blockIdx.x; //row_id
NNZV = vecPtr[vec_M+1] - vecPtr[vec_M];
if vec_M $\times$ V >M **then**
  Return;
end if
__shared__ $SMEM_A$[2][Block_M $\times$ Block_K];
__shared__ $Index_A$[2][Block_K];
reg Reg_A[Block_M $\times$ Block_K/ThreadNum];
reg Reg_B[Block_K $\times$ Block_N/ThreadNum];
reg Reg_C[Block_M $\times$ Block_N/ThreadNum];
Load LHS_0 to $SMEM_A$[0];//pre-load LHS Tile
Load $colInd_0$ to $Index_A$[0];//pre-load index
# pragma unroll //main loop
for $k \leftarrow 1$ *to* $NNZV/Block\_K - 1$ *step* 1 **do**
  Load LHS_k to $SMEM_A$[k% 2];
  Load $colInd_k$ to $Index_A$[k% 2];
  Load $SMEM_A[(k-1)\%2]$ to Reg_A;
  Load RHS_{k-1} to Reg_B via $Index_A$[(k-1)% 2];
  MMA(Reg_C, Reg_A, Reg_B, Reg_C);
end for
Load $SMEM_A[(NNZV/Block\_K - 1)\%2]$ to Reg_A;
Load $RHS_{NNZV/Block\_K-1}$ to Reg_B
via $Index_A$[(NNZV/Block_K-1)% 2];
MMA(Reg_C, Reg_A, Reg_B, Reg_C);
Store Reg_C to **Output**;

CVSE format is tailored for structured (vector) pruning, and the pruning algorithm determines its vector length. Different from initial design intention of CVSE format, *Sparse Vector Compression* can adaptively choose different V values based on matrix parameters, which can help SpMM kernels obtain higher performance. For sparse matrices without structured features (e.g., matrices in DL), when the sparsity is less than 90%, a larger V is more appropriate. For sparse matrix with spatial locality (e.g., matrices in scientific computing), smaller V can play its local performance advantage. And this opinion has been verified in Section VI-F.

C. CUDA Code Implementation

Finally, integrating all the aforementioned optimization strategies, we have developed a high-performance SpMM library, *SSpMM*, utilizing Tensor Cores through CUDA codes. This library initially adaptively selects V to perform *Sparse Vector Compression* based on the sparsity of the input sparse matrix. Subsequently, the GPU executes parallel SpMM kernels, which incorporate the *Transpose Mapping Scheme* and the *Register Shuffle Method*.

As shown in Algorithm 2, this is pseudocode for a Thread Block in *SSpMM*. To increase the grid size to hide the latency through thread-level-parallelism [27], the thread block size should not be too large. Therefore, we do not consider the case of V=32. In concrete implementation, each thread block comprises 32 threads, computing an output matrix of 8×64 or 16×64.

It is noteworthy that we have also incorporated several general optimizations from prior work [14], [23], [24] to address access conflicts and load imbalance issues, such as data prefetching, padding, and row reordering, within the concrete CUDA code implementation.

VI. Evaluation

A. Experimental Setup

Environments: We elaborate the experimental evaluation of SpMM kernels on all Tensor Core architectures: the V100 GPU (1st Gen Tensor Core) [19], the RTX 2080 Ti GPU (2nd Gen Tensor Core) [20], the A100 GPU (3 rd Gen Tensor Core) [21], and the RTX 4090 GPU (4th Gen Tensor Core) [22]. We conduct our experiments using NVCC version `11.8` and NVIDIA driver version `535.113.01`. The performance improvement achieved by *SSpMM* remains valid in other CUDA versions. The purpose of selecting this version in this paper is to facilitate the construction of a stable experimental environment for comparison objects.

Benchmark matrices: To comprehensively evaluate our contributions, we construct benchmarks using two well-established datasets: the Deep Learning Matrix Collection (DLMC) [31] and SuiteSparse Matrix Collection [32]. These datasets offer diverse test cases to rigorously assess the performance and scalability of our SpMM kernels.

DLMC is a real AI model dataset, consisting of weight matrices pruned by four algorithms: l0_regularization, magnitude_pruning, random_pruning, and variational_dropout. And their distributions are relatively random. They are pruned to 7 sparsity levels (0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.98). The most common sizes are 512×512, 512×2048, 2048×512, and up to $33K \times 33K$. And we selected all matrices in DLMC for experiments.

The SuiteSparse dataset is a widely recognized collection of sparse matrices that encompasses a diverse range of application areas, including scientific computing, machine learning, and more. We test over 1000 of these sparse matrices specifically tailored for scientific computing, utilizing them to illustrate the scalability of our *SSpMM* library in terms of computational granularity. Sparse matrices in scientific computing are typically large and exhibit extreme sparsity with varying properties, leading to highly variable performance outcomes for SpMM operations in this domain.

In this section, we select all matrices in DLMC to verify the efficiency of *SSpMM* in AI computing. For scientific computing, we select a representative set of SuiteSparse matrices with varying sizes and different distribution characteristics—which are summarized in Table III—as benchmark matrices for scientific computing.

TABLE III
Information of the 15 Representative Matrices

ID	Matrix	Plot	Size	*nnz*
1	circuit204		1K×1K	588
2	psmigr_1		3K×3K	543.1K
3	poisson3Da		13K×13K	352.7k
4	mosfet2		46K×46K	149.9k
5	mip1		66K×66K	10.3M
6	2cubes_sphere		101K×101K	1.6M
7	filter3D		106K×106K	2.7M
8	cop20k_A		121K×121K	2.6M
9	scircuit		170K×170K	958.9K
10	m133-b3		200K×200K	800.8K
11	mario002		389K×389K	2.1M
12	ASIC_680k		682K×682K	3.8M
13	eu-2005		862K×862K	19.2M
14	web-Google		916K×916K	5.1M
15	in-2004		1382K×1382K	16.9M

B. Performance Comparison

Given the extensive body of research on GPU-accelerated SpMM, we select four of the most representative libraries—*cuBLAS* (with dense format), *cuSPARSE* (with CSR format), *Sputnik* (with CSR format), and *vectorSparse* (with CVSE format)—to compare their FP16 performance with our work. Considering that *Magicube* (with SR-BCSR format) is only applicable for execution on the Ampere architecture and only supports low-precision formats (INT8 and INT16), we have compared its performance specifically for experiments on the A100 GPU. For libraries that do not support unstructured sparse matrix multiplication, we have extensively modified their input interfaces to accommodate such matrices.

For the DLMC benchmark, we use *cuBLAS* as the baseline to measure the speedup of each library against *cuBLAS*. For the SuiteSparse benchmark, due to the large matrix sizes, most SpMM operations cannot be directly computed using *cuBLAS*. Therefore, we directly compare *SSpMM* with *cuSPARSE*, and *Sputnik*. We choose different V values to execute *Sparse Vector Compression*. Thus, the *SSpMM* results shown in Fig. 9 represent the best performance outcomes among kernels with different V values in *SSpMM*.

As shown in Fig. 9, *SSpMM* consistently outperforms *cuSPARSE*, *vectorSparse*, and *Sputnik* in all cases. Since *Sputnik* is currently the SOTA method for accelerating unstructured SpMM, its speedup serves as the primary indicator of *SSpMM*'s superiority. The final experimental results demonstrate that *SSpMM* achieves significant performance improvements over

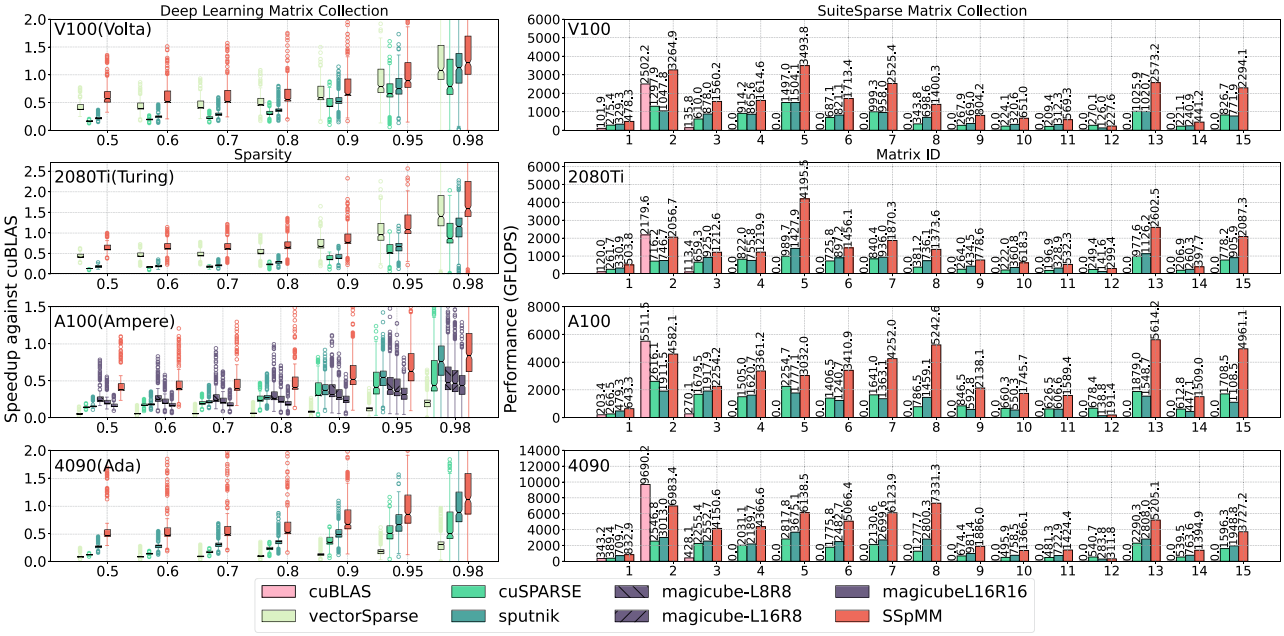

Fig. 9. Performance comparison of different libraries across four generations of NVIDIA Tensor Core GPUs.

Sputnik. Specifically, on the Volta architecture, *SSpMM* delivers an average performance increase of 2.04× compared to *Sputnik*; whereas on the Turing architecture, the improvement is 2.81 × over *Sputnik*. Furthermore, for the Ampere architecture, the performance gain is 2.07 × compared to *Sputnik*; and finally, on the Ada architecture, *SSpMM* achieves an average performance improvement of 1.87 × over *Sputnik*. *SSpMM* also achieves performance improvements of 1.35 ×, 1.37 ×, 7.24 ×, and 5.96 ×, respectively, when compared to SOTA structured SpMM on Tensor Cores (i.e., *vectorSparse*). Notably, *SSpMM* has consistently achieved a stable percentage (7.5%\,11.7%) of the Tensor Core peak performance across four architectures when executing structured kernels.

Similarly, in the SuiteSparse benchmark, *SSpMM* consistently outperforms *cuSPARSE* and *Sputnik* across different matrix sizes. Moreover, across all four architectures, it achieves average performance improvements of 2.30 ×, 2.06 ×, 2.66 ×, and 1.98 ×, respectively, when compared to *Sputnik*. All above experiments can demonstrate the efficient scalability of *SSpMM* on different Tensor Core architectures.

C. AI Application: End-to-End Sparse Transformer Inference

Currently, Transformers are dominant among the large-scale AI models [50]. The architecture of the network and the size of the model are primarily determined by the head dimension, the number of heads, and the number of layers. Furthermore, the self-attention workload increases quadratically with the input sequence length. By adjusting these parameters, our evaluation covers a variety of architectures and workloads relevant to Transformers.

We evaluate the performance advantages of *SSpMM* in end-to-end inference using the Long-Range Arena (LRA) [51], with mixed precision of FP16/FP32. We keep the number of encoder layers to 4 since the total workload is proportional to the number of layers, and head dimension to 64, which is commonly used in Transformer models. We use input sequence lengths L of 4096 and 8192.

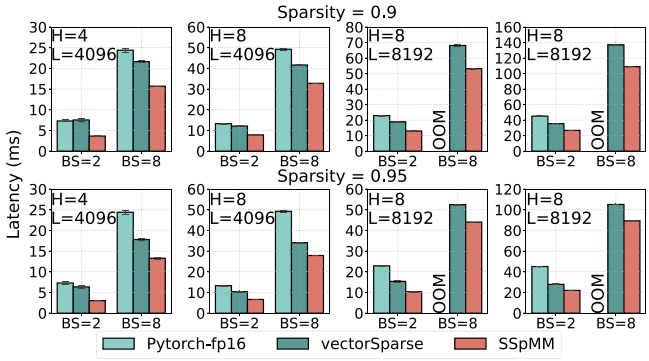

Fig. 10. Latency of end-to-end inference of Transformers with different sparsity, H, L and BS. (Note: H (number of heads), L (sequence length), BS (batchsize), OOM (out of memory)).

Fig. 10 presents latency results for the end-to-end inference performance of sparse Transformer models with varying sparsity levels, numbers of heads, and batch sizes. Note that the latency essentially measures compute time, including overheads. We integrate *SSpMM* into Pytorch [52] to compare inference performance against *cuDNN* and *vectorSparse*. *SSpMM* achieve up to a 1.81 × speedup over *cuDNN* and a 1.33 × speedup over *vectorSparse*.

D. Ablation Study

We conduct a series of ablation experiments to evaluate the efficiency of *Sparse Vector Compression*, *Transpose Mapping Scheme*, and *Register Shuffle Method* in Fig. 11. Additionally,

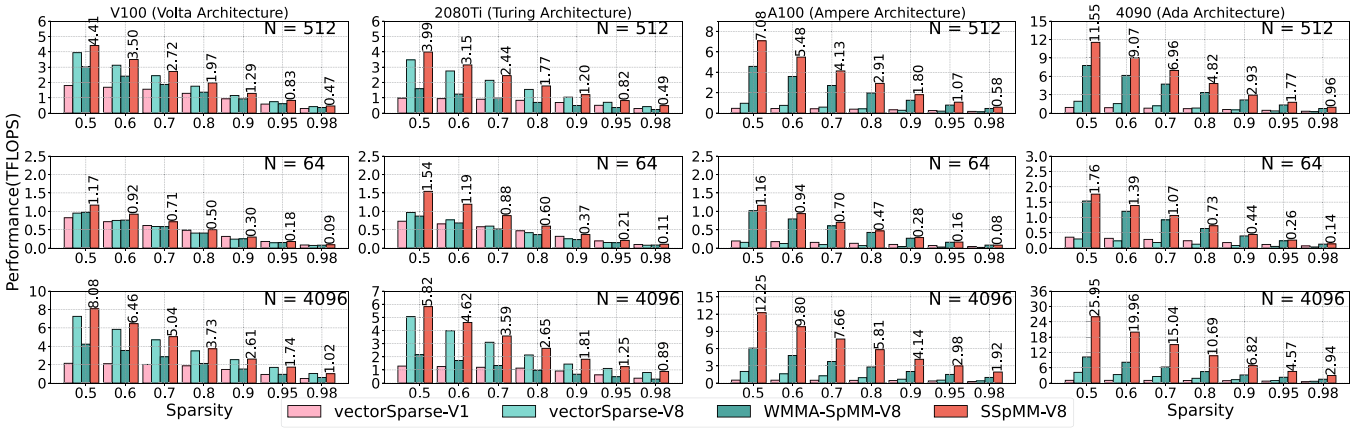

Fig. 11. Ablation Study for *Sparse Vector Compression* (V=8), *Transpose Mapping Scheme* and *Register Shuffle Method*. *vectorSparse*-V8 means the combination of *vectorSparse* and *Sparse Vector Compression*. WMMA-SpMM-V8 means the combination of *Sparse Vector Compression* and *Transpose Mapping Scheme*. *SSpMM*-V8 means the combination of *Sparse Vector Compression*, *Transpose Mapping Scheme* and *Register Shuffle Method*.

we set the number of columns in the RHS matrix (i.e., the N-dimension) to 64, 512, and 4096, respectively, to verify that the optimization strategies are effective at different scales.

Sparse Vector Compression: If unstructured SpMM is directly mapped to *vectorSparse* (i.e., directly mapping a row of CSR data onto Tensor Cores), it results in a $7/8$ performance waste. When MMA instructions are fully utilized, a theoretical $8\times$ performance gain is achievable. However, residual zero elements persist after *Sparse Vector Compression*, limiting actual improvements to $1\times\backslash,8\times$. The specific acceleration factor is determined by the new sparsity ratio analyzed in Section VI-E. When N = 512, *Sparse Vector Compression* brings average performance improvements of 1.67 ×, 2.34 ×, 1.35 ×, and 1.40 ×, respectively; whereas when N = 64, *Sparse Vector Compression* brings average performance improvements of 0.98 ×, 1.06 ×, 0.61 ×, and 0.65 ×, respectively. Furthermore, when N = 4096, *Sparse Vector Compression* brings average performance improvements of 2.40 ×, 2.59 ×, 2.43 ×, and 2.32 ×, respectively.

Transpose Mapping Scheme: The performance degradation of *vectorSparse* and similar works across different architectures is attributed to the lack of general-purpose instructions. We also design a set of SpMM kernels, referred to as WMMA-SpMM, using general-purpose instructions——specifically, the WMMA API of Tensor Cores——as a baseline. These kernels utilize *Transpose Mapping Scheme*, loading matrices from Global Memory to Shared Memory, and then transpose them to Fragments for computation. Through these kernels, the performance improvement achieved by using general-purpose instructions and *Transpose Mapping Scheme* can be analyzed.

WMMA-SpMM underperforms compared to *vectorSparse* on Volta and Turing architectures, because the `mma.m8n8k4` instruction in *vectorSparse* maintains its performance advantage. Due to the inability of WMMA API to efficiently allocate on-chip resources, *Transpose Mapping Scheme* cannot provide performance advantages on Volta and Turing architectures. However, with newer Tensor Core architectures, the performance advantages brought by general-purpose instructions have compensated for the losses caused by resource allocation.

In addition, the transpose overhead is subsumed within the kernel execution time and does not constitute offline processing. We conduct tests on NVIDIA's V100 GPU using the mma.m8n8k4 instruction, noting that its symmetrical M-dimension and N-dimension ensure the transpose strategy does not alter the Thread Block configuration. Critically, the execution times for operations utilizing the transpose strategy were virtually identical to those without it. Consequently, we irrefutably demonstrate that the temporal cost associated with matrix transpose is virtually zero.

Register Shuffle Method: Transpose Mapping Scheme and *Register Shuffle Method* are tightly coupled, proposed to efficiently utilize the on-chip registers and leverage the capabilities of PTX instructions for Tensor Cores. Compared to WMMA-SpMM, *SSpMM* can achieve performance improvements brought by rational resource allocation and coalesced memory access. Across four architectures, it achieves average speedups of (1.21, 1.69, 1.13, 1.14) × for N=64, (1.43, 2.48, 1.50, 1.45) × for N=512, and (1.81, 2.70, 2.03, 2.38) × for N=4096.

Compared to *vectorSparse*+SVC (*Sparse Vector Compression*), *SSpMM* leverages the synergistic integration of *Transpose Mapping Scheme* and *Register Shuffle Method* to deliver performance gains that reflect intrinsic kernel scalability across architectures. *SSpMM* achieves average performance improvements of 1.12 ×, 1.15 ×, 6.92 ×, and 5.75 × when N = 512, respectively; whereas when N = 64, *SSpMM* achieves average performance improvements of 1.23 ×, 1.51 ×, 7.14 ×, and 5.69 ×, respectively. Furthermore, when N = 4096, *SSpMM* achieves average performance improvements of 1.08 ×, 1.17 ×, 6.03 ×, and 5.75 ×, respectively.

Given our prior discussion that the *Transpose Mapping Scheme* incurs virtually negligible temporal overhead, we attribute the entirety of the speedup achieved by *SSpMM* over the *vectorSparse*+SVC implementation on the V100 solely to the *Register Shuffle Method*. Specifically, this technique yields speedup of 1.23 ×, 1.12 ×, and 1.08 × across different N-dimensions.

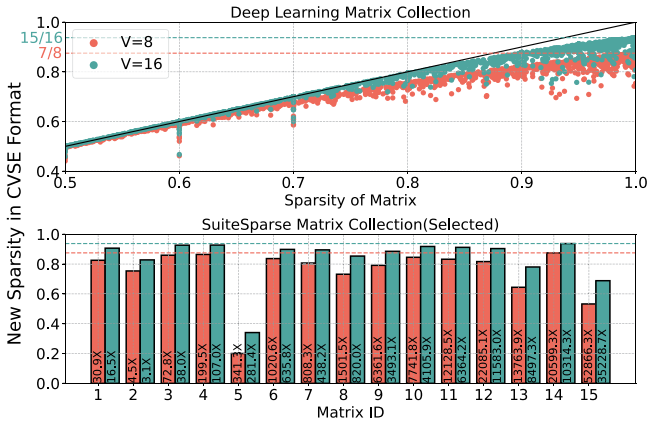

Fig. 12. The new sparsity after executing *Sparse Vector Compression* (V= 8 and 16). The bar numbers show the density ratio post-compression to pre-compression.

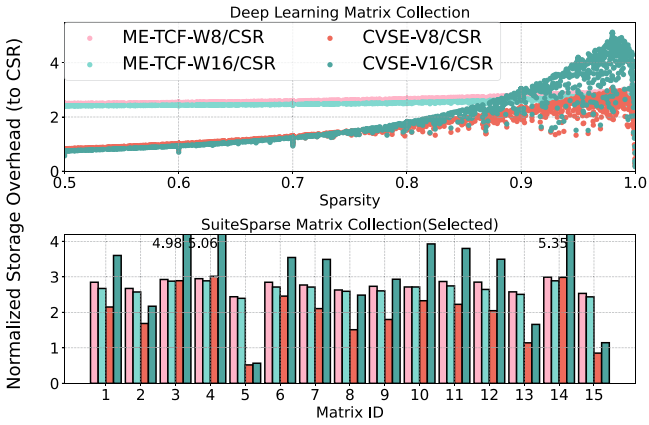

Fig. 13. The storage overhead of CVSE after executing *Sparse Vector Compression* (V= 8 and 16), and that of ME-TCF, compared to CSR.

E. Effectiveness of Data Structure

Sparse Vector Compression eliminates a significant number of redundant all-zero vectors, thereby forming a new sparsity $(1 - nnz/(V \times vec\_num))$. As the density of each tile reflects computational utilization of Tensor Cores, the decrease in sparsity leads to improved computational utilization of the Tensor Cores. Fig. 12 illustrates the new sparsity achieved by *Sparse Vector Compression* (V=8 & 16) across the DLMC and SuiteSparse. The improvement in Tensor Core utilization can be expressed as:

$$Compute - Density\ Improvement = \frac{New\ Density}{Original\ Density}$$
$$= \frac{NNZ}{vec\_num \times V} \div \frac{NNZ}{Rows \times Cols} = \frac{Rows \times Cols}{vec\_num \times V} \quad (1)$$

As shown in DLMC benchmark of Fig. 12, the greater the original sparsity of the matrices, the more significant the reduction in new sparsity. It is visually apparent that when V=8, the reduction in sparsity is more pronounced, leading to a greater improvement in utilization, which aligns with the results discussed in Section V-B. Since all-zero vectors are removed, each vector has at least one non-zero element, ensuring that the new sparsity of the compressed matrix will always be less than $1 - 1/V$. Consequently, all points in the Fig. 12 are below the corresponding colored dashed lines. For DLMC, *Sparse Vector Compression* achieves an average utilization improvement of 11.50 $\times$ and 6.48 $\times$ for V=8 and V=16, respectively. By observing results of DLMC, the original sparsity rather than matrix dimensions may influence the result of *Sparse Vector Compression*.

Consider the 15 SuiteSparse matrices depicted in Fig. 12: while all exhibit original sparsity exceeding 99%, post-compression sparsity predominantly remains between 80% – 90%. Strikingly, Matrix 5 achieves a new sparsity as low as 20% (effectively rendering it a dense matrix). Among the selected matrices, the average *utilization improvements* are 9301.74 $\times$ and 5461.77 $\times$, respectively. For Suiteparse, their sparsity is extremely high and their scale varies greatly, but the impact of sparse patterns is the greatest(e.g., Matrix 5).

Although the CVSE format cannot compress redundant zero elements as effectively as other compression formats, it still benefits from compression. And it operates at the granularity of column vectors, which results in fewer indices being required. As illustrated in Fig. 13, we compare CVSE against two established formats: the fully compressed ME-TCF [47] and CSR. Experiments use FP16 for data and INT32 for indices to quantify storage overhead. Although ME-TCF achieves full compression similar to CSR, it incurs substantial indices overhead. Consequently, CVSE with Sparse Vector Compression requires less storage than ME-TCF in most scenarios, particularly when sparsity levels are below 0.9. Our results show that CVSE reduces storage relative to ME-TCF by factors of 0.65 $\times$ (V = 8) and 0.86 $\times$ (V = 16). When compared to CSR, CVSE incurs 1.75 $\times$ and 2.23 $\times$ overhead for V = 8 and V = 16, respectively.

F. Evaluating the Trade-Off Between Utilization and Parallelism

In Section V-B, we discuss the trade-off between computational utilization and parallelism based on the characteristics of SpMM and vector-wise operations. By extending the variable V in *Sparse Vector Compression*, the M-dimension of SpMM is increased, thereby enhancing computational parallelism. However, this simultaneously reduces computational density, which means computational utilization.

To analyze this trade-off, we designed the experiment shown in the left part of Fig. 14. As illustrated in Fig. 12, when sparsity is relatively low, the data density in vectors with different values of V is comparable, which also results in similar levels of computational utilization.Therefore, *SSpMM*-V16, which exhibits higher parallelism, performs better under these conditions. However, when sparsity reaches 0.9, the computational utilization advantage of V=8 gradually surpasses the computational parallelism benefit of V=16.

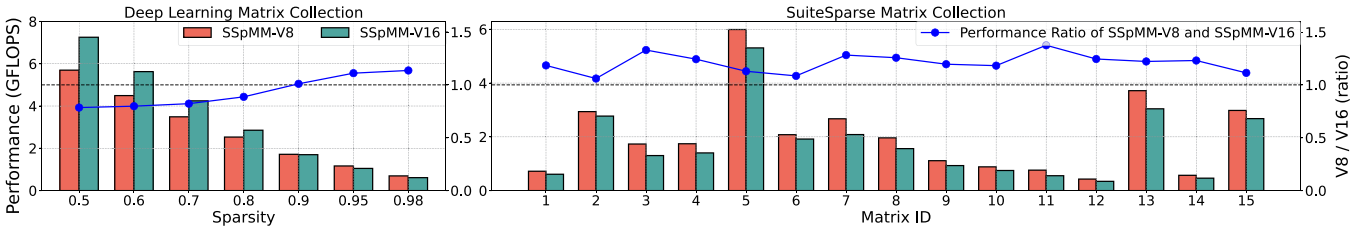

Fig. 14. Performance evaluation of *SSpMM* when executing *Sparse Vector Compression* with different V (8 and 16) on A100. The bars represent the performance of *SSpMM*-V*x*. The blue lines represent the performance ratio of *SSpMM*-V8 and *SSpMM*-V16.

In SuiteSparse, the matrices exhibit very high sparsity ($\geq$0.99). Consequently, the benefit of computational utilization generally outweighs that of computational parallelism. However, because the data in scientific computing have their own distribution scales and are not as random as in DLMC, even at high sparsity, the performance advantage of computational utilization varies across different matrices.

VII. Related Work and Discussion

There are numerous works dedicated to accelerating sparse matrix multiplication (such as SpMM, SpGEMM, SpMV, etc.) on GPUs. In terms of hardware platforms, we can categorize them into CUDA Core-based and Tensor Core-based.

CUDA Core-based: Several studies [14], [15], [18], [45], [53], [54], [55], [56] have identified and addressed memory access and load imbalance issues in unstructured SpMM. These works optimize memory access and load balancing within the kernel through CUDA coding, without preprocessing the data. In contrast, other studies [16], [17], [57], [58], [59] employ preprocessing techniques such as partitioning and reordering to enhance the locality of sparse matrices or make them more structured, subsequently feeding them into corresponding kernels to achieve higher performance. To tackle specific performance issues, some research [9], [10], [11], [60] focuses on converting data into desired formats and designing high-performance kernels tailored to these formats. Additionally, other works [8], [61] primarily explore the combination of sparse compression formats with kernels, demonstrating the capability to automatically generate format-kernel implementations based on input sparse data and hardware architecture. Our work is orthogonal to the CUDA Core-based studies mentioned previously, as we are dedicated to integrating optimization methods into our work on Tensor Cores. Consequently, *SSpMM* draws valuable insights from the techniques and optimization principles established in the aforementioned works.

Tensor Core-based: tSparse [62] employs blocking sparse matrix storage formats to group elements into tiles for SpGEMM on Tensor Cores. To align with the programming interface of Tensor Cores, Guo et al. [63] and Huang et al. [64] prune models into structured patterns, such as tile-wise sparsity. Similarly, *vectorSparse* [27] and *Magicube* [23] design their own sparse compression formats and corresponding kernels based on vector-wise sparsity. To further enhance the computational utilization of Tensor Cores, numerous studies [2], [24], [47], [64], [65] reorder data in advance, thereby improving spatial locality. Ampere GPUs have extended their Tensor Cores to handle row-wise 2:4 sparsity [21]. Leveraging this feature, several works [25], [26], [66] design fine-grained SpMM kernels, achieving significant performance gains. Additionally, *TC-GNN* [33] and *DTC-SpMM* [47] utilize TF32 Tensor Core to accelerate irregular GNN computations.

Among the aforementioned works, the performance of *vectorSparse* [27] diminishes on the latest generations of Tensor Cores. Meanwhile, some designs [23], [24], [25], [26], [33], [47], [66] are tailored exclusively for Tensor Cores post-Ampere architecture. Other works [63], [64] are overly dependent on the sparse patterns of models, lacking universality in SpMM acceleration. Consequently, there are few solutions that support efficient SpMM across all architectures. With the rapid pace of hardware advancements, kernel scalability is becoming increasingly critical.

For strategies proposed in this paper, we have the following opinions. *Transpose Mapping Scheme* is broadly applicable to most matrix multiplication types. By exploiting the transpose property, it exchanges M-dimension and N-dimension to enable flexible use of MMA instructions. However, our SpMM implementation leverages unique operand reuse patterns, necessitating distinct GPU memory hierarchy selections for intermediate storage. Thus, kernels like SpGEMM, SDDMM, and SpMV require tailored intermediate storage analysis rather than direct adoption. *Register Shuffle Method* targets low-reusability RHS matrices. As this approach assumes dense operands, direct application to sparse kernels like SpGEMM and SpMV is constrained.

VIII. Conclusion

In this paper, we present *SSpMM*, an efficiently scalable SpMM library adept at handling both unstructured and structured (vector-wise) sparsity. *SSpMM* comprises *Sparse Vector Compression* and SpMM kernels. *Sparse Vector Compression* enables the compression of multiple rows of unstructured sparse matrices into the CVSE format, thereby addressing the challenge of leveraging Tensor Cores to accelerate unstructured SpMM. For scalable SpMM kernel implementation, we introduce the *Transpose Mapping Scheme*, which facilitates fine-grained SpMM using coarse-grained general-purpose instructions. Building upon this scheme, the *Register Shuffle Method* further enhances SpMM performance by combining coalesced memory access with data reorganization.

Whether in scientific computing or AI computing, *SSpMM* demonstrates its efficient scalability at any sparsity level. Compared to the current SOTA SpMM solutions, *SSpMM* consistently demonstrates significant speedups across all Tensor Core architectures, achieving efficient kernel scalability. When integrated with existing frameworks, it enables effective end-to-end inference acceleration.

REFERENCES

[1] Z. Wang, J. Wohlwend, and T. Lei, "Structured pruning of large language models," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2020, pp. 6151–6162.

[2] H. Xia et al., "Flash-LLM: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity," in *Proc. VLDB Endowment*, vol. 17, no. 2, pp. 211–224, Oct. 2023.

[3] E. Frantar and D. Alistarh, "SparseGPT: Massive language models can be accurately pruned in one-shot," in *Proc. 40th Int. Conf. Mach. Learn.*, 2023, pp. 10323–10337.

[4] Y. Zhao, D. Wu, and J. Wang, "ALISA: Accelerating large language model inference via sparsity-aware KV caching," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit.*, 2024, pp. 1005–1017.

[5] S. Dong et al., "Spartan: A sparsity-adaptive framework to accelerate deep neural network training on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 10, pp. 2448–2463, Oct. 2021.

[6] Z. Zhang and C. Wang, "MIPD: An adaptive gradient sparsification framework for distributed DNNs training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 3053–3066, Nov. 2022.

[7] Z. Fan et al., "Accelerating convolutional neural networks by exploiting the sparsity of output activation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 12, pp. 3253–3265, Dec. 2023.

[8] Z. Xie, G. Tan, W. Liu, and N. Sun, "A pattern-based SpGEMM library for multi-core and many-core architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 159–175, Jan. 2022.

[9] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, 2022, pp. 90–106.

[10] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "TileSpMV: A tiled algorithm for sparse matrix-vector multiplication on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 68–78.

[11] H. Ji, H. Song, S. Lu, Z. Jin, G. Tan, and W. Liu, "TileSpMSpV: A tiled algorithm for sparse matrix-sparse vector multiplication on GPUs," in *Proc. 51st Int. Conf. Parallel Process.*, New York, NY, USA, 2023, pp. 1–11.

[12] Y. Lu and W. Liu, "DASP: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA, 2023, pp. 1–14.

[13] cuSPARSE Library, NVIDIA, Dec. 2022. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf

[14] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse GPU Kernels for deep learning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–14.

[15] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–12.

[16] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, 2019, pp. 300–314.

[17] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, 2020, pp. 376–388.

[18] G. Dai et al., "Heuristic adaptability to input dynamics for SpMM on GPUs," in *Proc. 59th ACM/IEEE Des. Automat. Conf.*, New York, NY, USA, 2022, pp. 595–600.

[19] NVIDIA TESLA V100 GPU ARCHITECTURE, NVIDIA, Aug. 2017. [Online]. Available: https://images.nvidia.cn/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[20] NVIDIA TURING GPU ARCHITECTURE, NVIDIA, 2018. [Online]. Available: https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[21] NVIDIA A100 Tensor Core GPU Architecture, NVIDIA, 2020. [Online]. Available: https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[22] NVIDIAADA GPU ARCHITECTURE, NVIDIA, 2023. [Online]. Available: https://images.nvidia.cn/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[23] S. Li, K. Osawa, and T. Hoefler, "Efficient quantized sparse matrix operations on tensor cores," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2022, pp. 1–15.

[24] Z. Xue et al., "Releasing the potential of tensor core for unstructured SpMM using tiled-CSR format," in *Proc. IEEE 41st Int. Conf. Comput. Des.*, 2023, pp. 457–464.

[25] K. Zhang et al., "Jigsaw: Accelerating SpMM with vector sparsity on sparse tensor core," in *Proc. 53rd Int. Conf. Parallel Process.*, New York, NY, USA, 2024, pp. 1124–1134.

[26] R. L. Castro, A. Ivanov, D. Andrade, T. Ben-Nun, B. B. Fraguela, and T. Hoefler, "VENOM: A vectorized N:M format for unleashing the power of sparse tensor cores," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA, 2023, Art. no. 72.

[27] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, "Efficient tensor core-based GPU Kernels for structured sparsity under reduced precision," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2021, pp. 1–13.

[28] K. Liegeois, S. Rajamanickam, and L. Berger-Vergiat, "Performance portable batched sparse linear solvers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1524–1535, May 2023.

[29] Parallel thread execution ISA version 8.3, NVIDIA, Nov. 2023. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[30] E. Karimi, N. B. Agostini, S. Dong, and D. Kaeli, "VCSR: An efficient GPU memory-aware sparse format," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3977–3989, Dec. 2022.

[31] Google, "Deep learning matrix collection (DLMC)," 2020. [Online]. Available: https://storage.googleapis.com/sgk-sc2020/dlmc.tar.gz

[32] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Dec. 2011.

[33] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding, "TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs," in *Proc. USENIX Annu. Tech. Conf.*. Boston, MA, USENIX Association, Jul. 2023, pp. 149–164.

[34] R. Hu, H. Wang, W. Yang, R. Ouyang, K. Li, and K. Li, "BCB-SpTC: An efficient sparse high-dimensional tensor contraction employing tensor core acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 12, pp. 2435–2448, Dec. 2024.

[35] H. Wang, W. Yang, R. Hu, R. Ouyang, K. Li, and K. Li, "A novel parallel algorithm for sparse tensor matrix chain multiplication via tcu-acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 8, pp. 2419–2432, Aug. 2023.

[36] CUDA C programming guide, NVIDIA, Aug. 2024, [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[37] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 1398–1406.

[38] X. Sun, X. Ren, S. Ma, and H. Wang, "meProp: Sparsified back propagation for accelerated deep learning with reduced overfitting," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 3299–3308.

[39] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, New York, NY, USA, 2019, pp. 359–371.

[40] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights," in *Proc. IEEE*, vol. 109, no. 10, pp. 1706–1752, Oct. 2021.

[41] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," in *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[42] H. Mao et al., "Exploring the regularity of sparse structure in convolutional neural networks," 2017, *arXiv:1705.08922*.

[43] H. Fan et al., "Adaptable butterfly accelerator for attention-based NNs via hardware and algorithm co-design," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture*, 2022, pp. 599–615.

[44] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, NY, USA, Curran Associates Inc., 2016, pp. 1387–1395.
[45] T. Park, S. Kang, M.-H. Jang, S.-W. Kim, and Y. Park, "Orchestrating large-scale SpGEMMs using dynamic block distribution and data transfer minimization on heterogeneous systems," in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 2456–2459.
[46] M. Li et al., "Accelerating sparse cholesky factorization on sunway many-core architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 7, pp. 1636–1650, Jul. 2020.
[47] R. Fan, W. Wang, and X. Chu, "DTC-SpMM: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2024, pp. 253–267.
[48] K. Hegde et al., "Extensor: An accelerator for sparse tensor algebra," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, New York, NY, USA, 2019, pp. 319–333.
[49] M. Zhu and Y. Xie, "Taming unstructured sparsity on GPUs via latency-aware optimization," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.
[50] A. Vaswani et al., "Attention is all you need," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA, Curran Associates Inc., 2017, pp. 6000–6010.
[51] Y. Tay et al., "Long range arena: A benchmark for efficient transformers," 2020, *arXiv: 2011.04006*.
[52] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA, Curran Associates Inc., 2019, pp. 8026–8037.
[53] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the GPU," in *Proc. Parallel Processing: 24th Int. Conf. Parallel Distrib. Comput.*, Turin, Italy, Berlin, Heidelberg, Springer-Verlag, 2018, pp. 672–687.
[54] M. Li, Y. Ao, and C. Yang, "Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1842–1853, Jul. 2021.
[55] H. Huang and E. Chow, "Exploring the design space of distributed parallel sparse matrix-multiple vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 11, pp. 1977–1988, Nov. 2024.
[56] T. Xia et al., "A comprehensive performance model of sparse matrix-vector multiplication to guide kernel optimization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 2, pp. 519–534, Feb. 2023.
[57] C. Hong et al., "Efficient sparse-matrix multi-vector product on GPUs," in *Proc. 27th Int. Symp. High- Perform. Parallel Distrib. Comput.*, New York, NY, USA, 2018, pp. 66–79.
[58] J. Lee, S. Kang, Y. Yu, Y.-Y. Jo, S.-W. Kim, and Y. Park, "Optimization of GPU-based sparse matrix multiplication for large sparse networks," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 925–936.
[59] R. Fan, W. Wang, and X. Chu, "Fast sparse GPU kernels for accelerated training of graph neural networks," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2023, pp. 501–511.
[60] L. Xiang, P. Sadayappan, and A. Sukumaran-Rajam, "High-performance architecture aware sparse convolutional neural networks for GPUs," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, New York, NY, USA, 2023, pp. 265–278.
[61] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "AlphaSparse: Generating high performance SpMV codes directly from sparse matrices," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2022, pp. 1–15.
[62] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix–matrix multiplication with GPU tensor cores," *Comput. Elect. Eng.*, vol. 88, 2020, Art. no. 106848.
[63] C. Guo et al., "Accelerating sparse DNN models without hardware-support via tile-wise sparsity," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–15.
[64] G. Huang, H. Li, M. Qin, F. Sun, Y. Ding, and Y. Xie, "Shfl-BW: Accelerating deep neural network inference with tensor-core aware weight pruning," in *Proc. 59th ACM/IEEE Des. Automat. Conf.*, New York, NY, USA, 2022, pp. 1153–1158.
[65] P. S. Labini, M. Bernaschi, W. Nutt, F. Silvestri, and F. Vella, "Blocking sparse matrices to leverage dense-specific multiplication," in *Proc. IEEE/ACM Workshop Irregular Appl.: Architectures Algorithms*, 2022, pp. 19–24.
[66] B. Lin et al., "Efficient GPU kernels for N:M-SPARSE weights in deep learning," in *Proc. Mach. Learn. Syst.*, 2023, pp. 513–525.

Zeyu Xue received the BS and MS degrees from the National University of Defense Technology in Changsha, China, in 2022 and 2024, respectively. He is currently working toward the PhD degree with the National University of Defense Technology in Changsha. His current research is focused on supporting sparsity in AI/DL on GPU and he is also interested in computer architecture and high performance computing.

Mei Wen received the BS, MS, and PhD degrees in computer science and technology from the National University of Defense Technology, in 1995, 1999 and 2006, respectively. She is currently a professor with Computer College, National University of Defense Technology, China. She is a long-term researcher specializing in stream processing, microprocessor architecture, high-performance computing, and AI accelerator chip architecture. As the principal investigator, she led multiple research projects funded by the National Natural Science Foundation of China, encompassing youth, general, and key projects focused on stream processing.

Jianchao Yang received the BS and MS degrees from the National University of Defense Technology, Changsha, China, in 2020 and 2022, respectively. He is currently working toward the PhD degree with the National University of Defense Technology. His research interests include computer architecture and compilers.

Minjin Tang received the BS, MS and PhD degrees from the National University of Defense Technology in Changsha, China, in 2018, 2020 and 2025, respectively. His research interests include computer architecture and sparse processing.

Zhongdi Luo received the B.S. degrees from the National University of Defense Technology in Changsha, China, in 2025. He is currently toward the PhD degree the National University of Defense Technology. His research interests include computer architecture and compilers.

Jing Feng is currently working toward the PhD degree in electronic science and technology with the National University of Defense Technology in Changsha, China. Her research interests include computer architecture and AI hardware accelerators.

Yang Shi received the BS and PhD degrees from the Tsinghua University and the National University of Defense Technology respectively, in 2014 and 2021, respectively. He is currently an assistant researcher with the National University of Defense Technology. His research interests focus on parallel computing, heterogeneous programming frameworks, and task scheduling.

Zhaoyun Chen received the BS, MS, and PhD degrees from the College of Computer, National University of Defense Technology, China, in 2013, 2015, and 2019, respectively. He joined the Computer College, National University of Defense Technology, China, as a research assistant, in 2019. His research interests include computer architecture, compiler and embedded system.

Junzhong Shen received the BS, MS, and PhD degree in computer science and technology from the National University of Defense Technology, in 2012, 2015 and 2020, respectively. He is currently working with the College of Computer, National University of Defense Technology. He does research in computer architecture, parallel computing, reconfigurable computing and programming languages.

Johannes Langguth is a senior research scientist with Simula Research Laboratory, and an associate professor with the University of Bergen, Norway. Prior to that, he worked with ENS Lyon, France. His research is centred around architectures, algorithms, and applications of parallel graph algorithms and sparse linear algebra. His previous projects include interdisciplinary work on social network analysis using combinations of network science, GNNs, and NLP, as well as parallel matching algorithms for combinatorial scientific computing and performance optimization for irregular applications the European High-Performance Computing project SparCity. Recent work has focussed on graph algorithms on tile-centric accelerators such as Graphcore IPUs and Cerebras WSEs, providing the first implementations of several algorithms on these platforms.