HR-SpMM: Adaptive Row Partitioning and Hybrid Kernel Design for Sparse Matrix Multiplication

Qi Wang
School of Computer Science and Technology
Southwest University of Science and Technology
Mianyang, Sichuan, China
qiwang0617@foxmail.com

Yaobin Wang
School of Computer Science and Technology
Southwest University of Science and Technology
Mianyang, Sichuan, China
wangyaobin@foxmail.com

Yi Luo
School of Computer Science and Technology
Southwest University of Science and Technology
Mianyang, Sichuan, China
yiluose@gmail.com

Rong Luo
School of Computer Science and Technology
Southwest University of Science and Technology
Mianyang, Sichuan, China
luorong@swust.edu.cn

Pingping Tang
School of Computer Science and Technology
Southwest University of Science and Technology
Mianyang, Sichuan, China
tangpingping@swust.edu.cn

Abstract

Sparse Matrix-Matrix Multiplication (SpMM) plays a critical role in high-performance computing and applications like Graph Neural Networks (GNNs). However, due to the sparsity and irregularity of real-world data, optimizing SpMM performance on modern GPUs has remained a significant challenge. Existing methods often involve trade-offs between load balancing and hardware utilization, making it difficult to efficiently handle both long and short rows in sparse matrices.

To address these issues, we propose HR-SpMM, a lightweight framework based on adaptive row partitioning and hybrid kernel design. HR-SpMM divides sparse matrix rows into two categories: long rows and short rows, leveraging Tensor Cores and CUDA Cores, respectively, to optimize computational efficiency. Long rows are further partitioned into fixed-size blocks to fully align with the hardware characteristics of Tensor Cores, while short rows adopt a flexible tiling strategy to maximize thread-level parallelism. Additionally, the framework introduces auxiliary arrays for efficient indexing and offset management, significantly reducing preprocessing overhead. We used the DLMC(Deep Learning Matrix Collection) and SuiteSparse datasets, and compared HR-SpMM with SOTA. Our method achieves an average speedup of 2.05 times over cuSPARSE, 2 times over the latest Tensor Core-based TC-GNN, and 1.25 times over Sputnik. Additionally, while ensuring acceleration, we keep the preprocessing overhead extremely low, averaging only 12% of Sputnik's preprocessing cost.

CCS Concepts

• **Computer systems organization → Single instruction, multiple data**.

Keywords

CUDA Cores, Tensor Cores, SpMM, Load Balancing

ACM Reference Format:
Qi Wang, Yaobin Wang, Yi Luo, Rong Luo, and Pingping Tang. 2025. HR-SpMM: Adaptive Row Partitioning and Hybrid Kernel Design for Sparse Matrix Multiplication. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3721145.3725770

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725770

1 Introduction

Sparse matrix-matrix multiplication (SpMM) [13] is a critical operation in various high-performance computing and machine learning applications, particularly in graph neural

networks (GNNs) [22]. In GNNs, SpMM dominates computational workloads during training and inference, especially in the aggregation phase, where node features are propagated and combined across graph structures [28]. Despite its importance, SpMM remains challenging to optimize on modern GPUs due to the inherent irregularity and sparsity of real-world data. The uneven distribution of non-zero elements across rows and columns often leads to imbalanced workloads, inefficient GPU resource utilization, and significant performance bottlenecks [12, 24]. Optimizing SpMM is thus key to improving computational throughput and memory efficiency, enabling faster data processing in resource-constrained environments.

Modern GPUs provide two main computational resources: CUDA cores [11, 14, 18], which are versatile and better suited for sparse matrix computations, and Tensor Cores [27], which deliver high throughput for dense matrix multiplication. While Tensor Cores have shown immense promise in accelerating dense computations, their efficiency in sparse workloads is significantly hindered by the need for uniform, structured input data. This limitation creates a computational gap when dealing with sparse matrices, where most of the matrix elements are zeros, rendering Tensor Cores underutilized. On the other hand, CUDA cores, while more flexible for handling sparsity, often lack the computational density to fully exploit the hardware throughput offered by GPUs for larger, denser regions of data. Consequently, designing an efficient SpMM framework capable of leveraging both computational resources remains a pressing challenge.

In existing SpMM kernels, one row of a sparse matrix is typically allocated to a warp (a group of 32 threads on NVIDIA GPUs), with each non-zero element processed by a single thread [4]. However, this approach encounters significant inefficiencies due to the irregular distribution of non-zero elements across rows.

We divide rows into two categories: short rows, which contain fewer than 64 nonzero elements and are processed by CUDA cores, and long rows, which contain 64 or more nonzero elements and are processed by tensor cores.

We chose the threshold of 64 nonzero elements based on the architectural characteristics of tensor cores and the thread-level parallelism of GPUs. Tensor cores operate on fixed-size input blocks, typically multiples of $8 \times 8 \times 8$, which makes 64 (i.e., 8×8) and its multiples a natural fit for splitting sparse rows. This choice ensures that long rows can be efficiently processed in aligned blocks, fully utilizing the computational density of tensor cores while minimizing padding and fragmentation overhead. Additionally, the CUDA kernels we use perform well on rows with fewer than 64 nonzero elements.

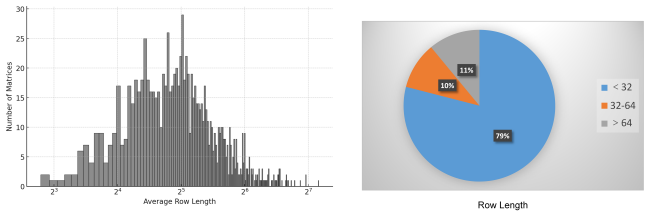

Figure 1: Distribution of average Row Lengths and Row Length in Sparse Matrices from DLMC and SuiteSparse Datasets.

As shown in Figure 1 (left), we analyzed 1,000 randomly selected matrices from the DLMC(Deep Learning Matrix Collection) and SuiteSparse datasets. The results indicate that most matrices have an average row length of fewer than 64, with only a small portion exceeding this threshold. Figure 1 (right) shows that 89% of rows have fewer than 64 nonzero elements, while the remaining rows exceed 64. Moreover, rows with fewer than 32 nonzero elements account for 79%. A large number of short rows reduce computational resource utilization because hardware threads may remain idle for extended periods when processing these rows, leading to significant performance losses. Additionally, the small fraction of long rows is another critical factor affecting performance. Some of these rows even contain thousands of nonzero elements, causing severe load imbalance among warps, as a single warp is forced to process too many nonzero elements, reducing parallel efficiency [4].

This analysis highlights that the uneven distribution of nonzero elements across rows (characterized by the coexistence of extremely short and extremely long rows) is a major factor limiting SpMM performance. Therefore, designing more efficient SpMM kernels must prioritize addressing the performance impact of both short and long rows.

To address these shortcomings, we propose HR-SpMM, a hybrid SpMM framework that combines fine-grained row partitioning with efficient GPU resource utilization. Sparse rows are divided into two categories: long rows and short rows. Long rows with 64 or more non-zero elements are further subdivided into fixed-size segments of 64 non-zero elements (and their multiples), ensuring an efficient match with Tensor Core requirements. The remaining portion of long rows is processed together with short rows using CUDA Cores, employing a flexible tiling strategy to maximize warp-level parallelism. This approach not only avoids fragmentation but also ensures a balanced workload between Tensor Cores and CUDA Cores. Furthermore, HR-SpMM introduces a lightweight preprocessing step using one auxiliary array for indexing and offset management, significantly reducing overhead compared to prior methods. By efficiently addressing both short and long rows, HR-SpMM achieves superior

load balancing, minimizes resource underutilization, and delivers enhanced performance for a wide range of sparse matrix patterns.

2 Background

In this section, we present background information on sparse matrices in Graph neural Networks (GNNS), computational kernels, sparse matrix multiplication (SpMM), the Compressed Sparse Row (CSR) format for sparse matrix representation, and recent advances in dealing with sparse matrices.

2.1 Sparse Matrix-Matrix Multiplication (SpMM)

In deep learning and computer science, SpMM typically refers to Sparse Matrix Multiplication [13]. Sparse Matrix Multiplication is a crucial problem in computer science and numerical computation, especially when dealing with large-scale data and machine learning applications. SpMM algorithms aim to improve the efficiency of sparse matrix multiplication by reducing unnecessary computations and focusing only on non-zero elements, thereby optimizing the use of computing resources. With the rapid development of big data and artificial intelligence, there has been an increasing demand across various fields for efficient sparse matrix multiplication. Therefore, the research and optimization of SpMM algorithms have become important research directions.

Sparse matrix-matrix multiplication (SpMM) is a critical operation in many high-performance computing and machine learning applications. It involves the multiplication of a sparse matrix A with a dense matrix B, resulting in a dense matrix C. While efficient on dense matrices, GPU acceleration of SpMM is challenging due to the irregular distribution of non-zero elements in sparse matrices, leading to imbalanced workloads and inefficient memory access patterns. These issues significantly impact performance, especially in graph-related tasks such as graph neural networks (GNNs), where SpMM is a core operation.

Many researchers are dedicated to proposing more efficient SpMM algorithms to address the computational challenges posed by large-scale sparse matrices. These algorithms typically integrate hardware features, parallel computing, and optimization techniques to enhance computational speed and efficiency. Many independent SpMM studies have achieved better performance than cuSPARSE [18](NVIDIA's official math library), but they are not applicable to mainstream GNN computing frameworks. ASpT [14] utilizes dense matrix data and requires a special sparse format composed of CSR and additional arrays to explicitly mark the positions of locally-dense blocks explicitly. TLPGNN [10] divides GNN computations into two levels, adopting a hybrid dynamic load balancing method to address load imbalance issues. GE-SpMM [15], based on GPU hardware, optimizes the reuse of sparse row data in the row-partitioning algorithm through shared memory, thereby improving parallel performance of SpMM by reducing memory transactions [30].

2.2 Graph Neural Networks (GNNs)

Graph neural networks (GNNs) have emerged as a powerful tool for learning from graph-structured data, with applications spanning recommendation systems, molecular property prediction, and social network analysis [19, 25, 26]. A key operation in GNNs is the aggregation of node features, where information is propagated across graph edges to update node embeddings [9, 16]. This process involves sparse matrix-matrix multiplication (SpMM), where a sparse adjacency matrix A is multiplied by a dense feature matrix B to generate the updated feature matrix C.

The efficiency of SpMM is crucial for GNNs, as it directly impacts the computational cost of both training and inference [20, 27, 29]. However, the irregular sparsity patterns of real-world graphs, such as power-law distributions of node degrees, pose significant challenges for achieving high parallel efficiency on modern GPUs.

2.3 GPU Computation Cores

CUDA Cores are the general-purpose compute units in NVIDIA GPUs, designed to handle a wide range of arithmetic and logical operations [1]. Each CUDA Core is a simple scalar processor capable of executing individual threads in parallel. On GPUs, threads are organized into warps, with each warp consisting of 32 threads that execute instructions in lockstep. This warp-based execution model allows CUDA Cores to provide high parallelism and efficient utilization of GPU resources for a variety of computational workloads.

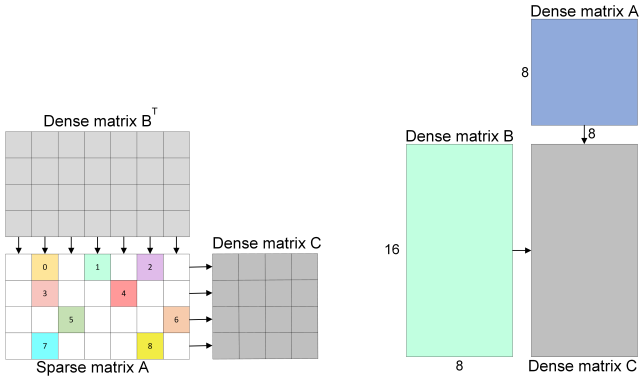

Figure 2: Operations of SpMM in CUDA Cores and GEMM in Tensor Cores.

Figure 2 (left) shows the operations of SpMM in CUDA Core. In sparse matrix-matrix multiplication (SpMM), CUDA Cores play a critical role due to their flexibility in handling irregular sparsity patterns. Typically, one row of a sparse matrix is assigned to a warp, with each thread processing a single non-zero element. This thread-to-element mapping ensures that irregular data distributions in sparse matrices can be processed efficiently, even if rows have varying numbers of non-zero elements. For rows with fewer non-zero elements, CUDA Cores avoid the padding overhead that Tensor Cores would require, making them better suited for handling short rows or sparse regions of the matrix.

The strength of CUDA Cores lies in their ability to dynamically adapt to varying sparsity levels and irregular data structures, which are common in real-world applications such as graph neural networks (GNNs). CUDA Cores efficiently manage the irregular computations that arise in SpMM operations, ensuring balanced utilization of GPU hardware.

Tensor Cores are specialized computational units developed by NVIDIA, first introduced in the Volta GPU architecture, and subsequently included in Turing, Ampere, and Hopper architectures [5, 17, 21]. These cores are designed to accelerate matrix computations, which are critical for applications in machine learning, deep learning, and scientific computing. Figure 2 (right) illustrates the operations of GEMM in Tensor Core. Tensor Cores operate on fixed-size matrix fragments, such as 16×8×8 in mixed-precision formats, performing fused multiply-add operations in the form D = A × B + C [23]. They achieve high computational throughput by combining multiple lower-precision arithmetic operations into a single instruction cycle, allowing for efficient execution of matrix multiplication and accumulation tasks. Supported data types include FP16, BF16, TF32, and INT formats, which balance precision and speed for tasks such as training and inference in deep neural networks [8]. By working at the warp level (32 threads), Tensor Cores deliver substantial performance improvements over traditional CUDA cores, particularly in dense matrix multiplication. However, their efficiency relies on structured and aligned data, which makes their use in irregular workloads like sparse matrix operations challenging. Nonetheless, their ability to handle structured fragments with minimal overhead has made them a critical hardware feature for accelerating large-scale computations.

2.4 CSR

The sparse matrix representation is in COO format, but COO format has certain limitations in terms of storage and computational efficiency in SpMM operations. Therefore, it is converted to the CSR (Compressed Sparse Row) format, which is widely adopted due to its compactness and efficiency [13].

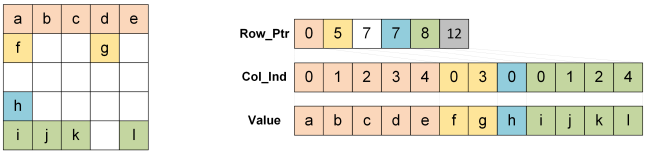

Figure 3: The sparse matrix (left) and its CSR representation (right).

As shown in Figure 3, in the CSR format, three one-dimensional arrays are used to represent a sparse matrix: 1. Row_Ptr: This array contains the starting offset positions of the first element of each row. 2. Col_Ind: This array contains the column indices corresponding to the elements in the values array. 3. Values array: This array contains all the non-zero elements of the matrix.

3 Related Works And Motivation

Sparse matrix-matrix multiplication (SpMM) has been the focus of extensive research due to its critical role in graph neural networks (GNNs), scientific computing, and other domains. Existing methods primarily address the challenges of irregular sparsity, load imbalance, and efficient hardware utilization by optimizing data representation, workload scheduling, and GPU kernel designs.

One line of work focuses on designing efficient data structures for sparse matrices, such as the Compressed Sparse Row (CSR) format and its variants. While CSR is highly memory efficient and widely used, it struggles to balance workloads on GPUs due to the irregular distribution of non-zero elements across rows. To address this, formats like Compressed Sparse Blocks (CSB) and Hybrid CSR-COO have been proposed, which improve memory coalescing and hardware efficiency [2, 3, 6, 7]. However, these methods often incur additional preprocessing costs and lack flexibility in handling both short and long rows effectively.

Another line of research emphasizes workload scheduling to reduce load imbalance during SpMM execution [11]. For example, Sputnik introduces the Row Offset Modification Algorithm (ROMA) to redistribute computation by reordering sparse matrix rows, while dynamic row tiling assigns rows with fewer non-zero elements to smaller thread groups. Although these approaches effectively balance workloads and improve CUDA core utilization, they often introduce significant overhead for matrix reordering and fail to fully leverage Tensor Cores for acceleration.

Recent work has also leveraged Tensor Cores to implement SpMM. For instance, TC-GNN is the first and most recent framework to adopt a Tensor Core strategy for SpMM [27]. It converts sparse matrices into dense submatrices to take advantage of Tensor Core acceleration, achieving significant

performance improvements in GNN training and inference. However, the preprocessing overhead required to generate dense submatrices and the padding needed to align rows with fixed tile sizes remain major challenges.

While existing methods have made substantial progress, they often fail to simultaneously address the combined challenges of irregular sparsity and load imbalance introduced by short and long rows. To overcome these limitations, we propose HR-SpMM, a lightweight hybrid framework that divides sparse rows into short and long categories based on their non-zero counts. By assigning long rows to Tensor Cores and short rows to CUDA Cores, HR-SpMM achieves a balanced workload distribution, minimizes preprocessing overhead, and maximizes hardware utilization. Unlike previous approaches that rely on computationally expensive preprocessing to densely pack non-zero elements from multiple rows, HR-SpMM transforms a single long row into a matrix and directly assigns it to Tensor Cores, further reducing overhead and improving efficiency.

Our contribution is as follows:

1. Accelerated Performance: HR-SpMM achieves significant performance improvements by utilizing a hybrid framework that combines Tensor Cores for long rows (with 64 or more non-zero elements) and CUDA Cores for short rows (with fewer than 64 elements). This results in an average speedup of 2.05x over cuSPARSE, 2x over TC-GNN, and 1.25x over Sputnik.

2. Hybrid Framework for Efficient GPU Utilization: The framework efficiently divides sparse rows into two categories (long and short rows) and assigns them to different computational units. Long rows are processed using Tensor Cores, while short rows are processed by CUDA Cores. This improves resource utilization and enhances computational efficiency.

3. Optimized Memory and Computational Resource Utilization: HR-SpMM also optimizes memory and computational resources by reducing unnecessary memory transfers and latency. Through effective memory management techniques such as shared memory and efficient data loading, the framework reduces overhead and ensures better memory bandwidth utilization.

4 HR-SpMM Design

HR-SpMM is a lightweight hybrid framework designed to address the challenges of irregular sparsity and load imbalance in sparse matrix-matrix multiplication (SpMM). The key idea is to partition sparse rows into short rows and long rows based on their non-zero counts, with long rows assigned to Tensor Cores and short rows processed by CUDA Cores. This design ensures efficient utilization of GPU hardware resources while minimizing preprocessing overhead.

4.1 Row Partitioning

The first step in HR-SpMM is partitioning the sparse matrix rows into two categories:

Figure 4: Sparse Matrix Row Partitioning: Long Rows and Short Rows.

As shown in Figure 4, the original sparse matrix is divided into two parts during preprocessing:

Long Rows: Rows containing 64 or more non-zero elements (aligned to multiples of 64). For example, in the figure, rows with indices 0, 1, and 3 are classified as long rows, where their non-zero elements are grouped into chunks of 64 or more. Notably, to achieve load balancing, we set the maximum length of a long row to 256 non-zero elements. In other words, if a long row contains 320 non-zero elements, we divide it into two long rows: one with 256 non-zero elements and the other with 64 non-zero elements, and handle them using atomic addition.

Short Rows: Rows containing fewer than 64 non-zero elements, such as the row with index 2, as well as the remaining elements from long rows that do not form complete 64-element chunks, like the leftover elements from the row with index 3.

After partitioning the sparse matrix into long rows and short rows, an auxiliary array, New Row Index, is introduced to efficiently manage the computation. This array maps the original row indices to their corresponding positions in the partitioned structure. For example:

In the figure 4, original row index 0 is mapped to new row index 0 in the long rows. Row index 3 is split into two parts: the long row part maps to new row index 3 in the long rows, while the short row part maps to new row index 3 in the short rows. The offsets for processing the non-zero elements are not explicitly stored, as they can be inferred during runtime based on the structure of the partitioned matrix.

As illustrated in Algorithm 1, the preprocessing is straightforward, with a time complexity of O(N), where N is the

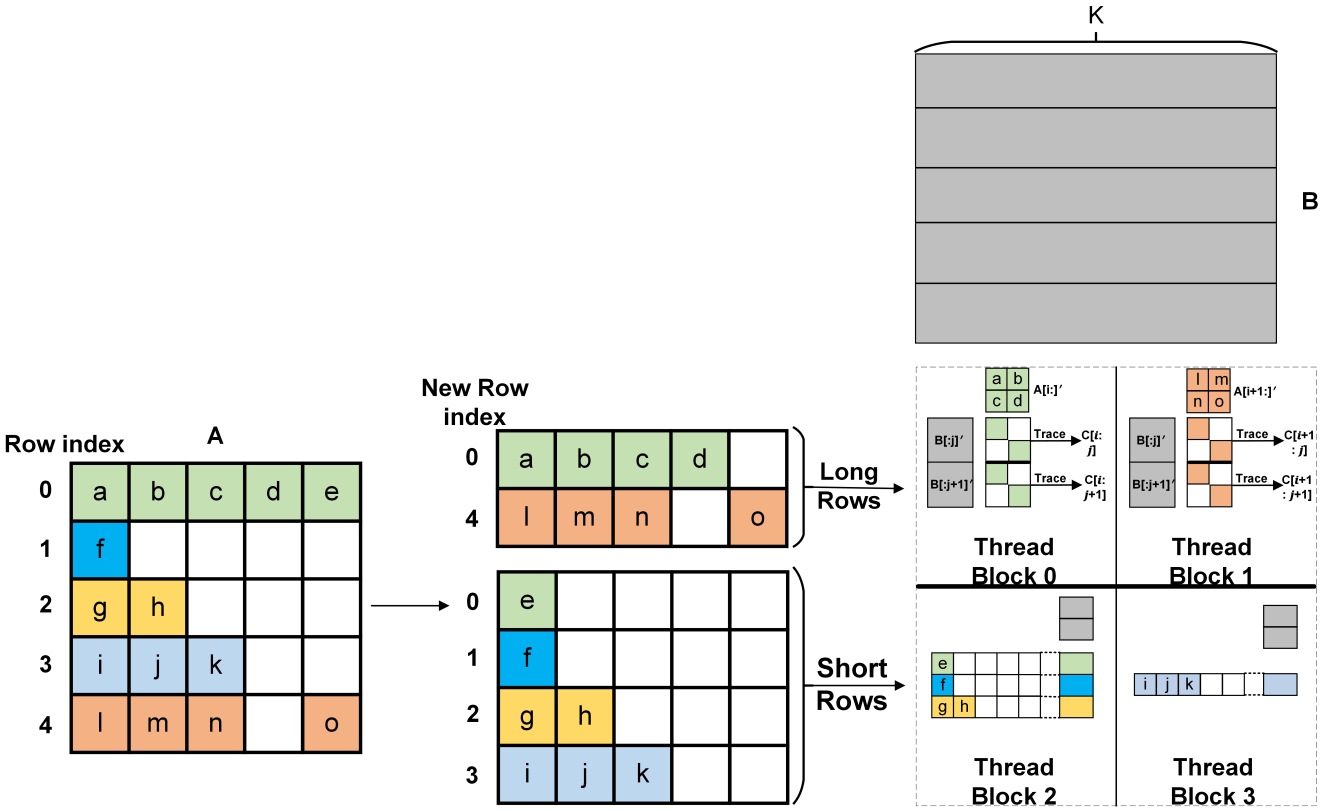

Figure 5: HR-SpMM Overview: Flowchart of Adaptive Row Partitioning and Hybrid Kernel Design for Sparse Matrix Optimization.

number of rows in the matrix. This process involves extracting the necessary metadata and iterating through the Row_Ptr array of the CSR format. During the calculation of the New Row Index, the offsets for the partitioned rows are also computed simultaneously. This lightweight preprocessing ensures minimal overhead while effectively preparing the matrix for efficient computation.

4.2 Design Overview of HR-SpMM

As shown in Figure 5, this is the overall design of our framework. The sparse matrix A is preprocessed and partitioned into two parts: long rows and short rows, which are handled by different kernels.

4.2.1 Design for Long Rows. We first introduce the processing of long rows. Due to image size constraints, 64 non-zero elements are assigned to a single thread block for processing, forming a matrix with 4 elements. Each sparse row is transformed into a smaller dense matrix. Once the vectors are transformed into matrices, these transformed matrices are passed to the Tensor Core to perform accelerated GEMM (General Matrix Multiplication) operations.

In our implementation, we utilize the 8 × 8 × 8 block size as a logical unit of computation for Tensor Core operations. While Tensor Cores on Ampere GPUs officially support a minimum MAC(Matrix Multiply and Accumulate) instruction size of 16 × 8 × 8, the use of 8 × 8 × 8 arises from the need to better align with the irregular sparsity patterns of the input matrix. Specifically, when handling sparse rows with uneven non-zero distributions, smaller block sizes such as 8 × 8 × 8 allow for more fine-grained mapping of non-zero elements to Tensor Core operations, minimizing the computational waste caused by padding. At the same time, we use shared memory to optimize global memory access and reduce data movement overhead. This design choice ensures efficient utilization of Tensor Core resources while preserving compatibility with the underlying hardware's MAC instructions. Furthermore, the subdivision of larger operations into smaller blocks facilitates parallelism within thread blocks, enabling warps to collaborate effectively and accumulate intermediate results for sparse matrix computations.

Algorithm 1: Parallel Preprocessing

Input: Row_Ptr
Output: New_Row_Index
target_row;
$nnz =$
$Row\_Ptr[target\_row + 1] - Row\_Ptr[target\_row]$;
$Short\_Row\_Flag = (nnz\%64) > 0$;
$Long\_Row\_Flag = (nnz > 64)$;
$Short\_Row\_Prefix\_Sum =$
$prefix\_sum(Short\_Row\_Flag)$;
$Long\_Row\_Prefix\_Sum =$
$prefix\_sum(Long\_Row\_Flag)$;
if $Short\_Row\_Flag > 0$ **then**
$New\_Row\_Index\_short[Short\_Row\_Prefix\_Sum] =$
$target\_row$;
end
if $Long\_Row\_Flag > 0$ **then**
$New\_Row\_Index\_long[Long\_Row\_Prefix\_Sum] =$
$target\_row$;
end

The SpMM algorithm processes a sparse row vector A[i:] and a dense column vector B[:j]. These vectors are reshaped into operand matrices: the sparse row becomes a row-major matrix, and the dense column becomes a column-major matrix. This transformation reformulates the dot product operation into blocked matrix-matrix multiplication. The Tensor Core performs MAC (Matrix Multiply-Accumulate) instructions on these matrices, generating intermediate results in the form of partial output matrices. The final result is obtained by aggregating the diagonal elements of the output matrices through a trace operation, optimized with warp shuffle for parallel reduction.

To efficiently process rows, each thread block handles either a full sparse row or a segment of it, depending on the row length. Since we set the row length to 64 and its multiples, long rows are split into segments of 64 non-zero elements, and each segment is assigned to a thread block. All warps within the block collaborate to execute MAC operations. The sparse row segments are loaded from global memory in CSR format and stored in shared memory, allowing fast access by all warps in the block. Similarly, the dense column vectors are compressed and stored in shared memory to reduce memory access overhead.

As shown in Figure 5, the sparse row 0 contains 5 non-zero elements. To better illustrate, we have assumed that a block contains only 4 threads, so during the preprocessing phase, we divide it into a long-row part and a short-row part. The long-row part requires MAC (Matrix Multiply and Accumulate) operations. The intermediate results are stored in accumulator fragments and aggregated through tracking operations to compute the final result. Then, the results are merged in shared memory using Atomic Add with the corresponding results from the short-row part. The final result is accumulated into global memory to ensure correctness when merging outputs from multiple thread blocks. Sparse row 4 already contains 4 non-zero elements, which can form a matrix. Therefore, we directly perform the MAC operation on it, and then use the same method to obtain the final result. Of course, this operation is performed in parallel with row 0, and it does not require Atomic Add.

Algorithm 2: Tensor Core Utilization in SpMM Kernel

Input: *Row_Ptr*, *New_Row_Index*
Output: *Dense Output Matrix C*
$target\_row$ = New_Row_Index[row_idx];
row_start = Row_Ptr[$target\_row$];
$total\_nnz$ = Row_Ptr[$target\_row + 1$] − row_start;
$part\_nnz = \lfloor total\_nnz/64 \rfloor \times 64$;
$nnz = \min(part\_nnz, 256)$;
$row\_start = row\_start$ + (blockIdx.x
mod $(total\_nnz/64)) \times 64$;
$buffer\_offset = 0$;
$output\_offset$ = output_col_start;
for $tile$ = output_col_start **to** output_col_end − 1 **by** tilewidth **do**
$c\_tile = 0$;
// Load sparse row and dense matrix column into shared memory
load.memory(S, D, shared_memory);
$b\_tile = B$.reshape_k8n8T;
for $frag = 0$ **to** num_fragments − 1 **do**
$a\_tile = A$[frag : frag + 1][0 : Block_Size − 1].reshape_m16k8;
$c\_tile = c\_tile + a\_tile \times b\_tile$;
end for
output_buffer[$buffer\_offset$] = trace(c_tile);
$buffer\_offset = buffer\_offset$ + tilewidth;
end for
AtomicAdd(C[target_row][output_offset],
warp_reduce_sum(output_buffer));

4.2.2 Design for Short Rows. Our CUDA kernel design draws inspiration from Sputnik's subwarp tiling approach [11], enabling each warp to collaboratively process short rows or multiple rows of the sparse matrix. By treating the warp as

the basic computation unit, threads within the warp cooperate to load data, perform calculations, and optimize memory access patterns, reducing synchronization overhead and maintaining load balance while fully utilizing GPU resources. For short rows, we leverage a combination of subwarp tiling and residue unroll techniques to achieve efficient computation and load balancing. Subwarp tiling divides a warp into smaller subsets of threads, or subwarps, each responsible for processing one or more short rows, ensuring that sparse rows with fewer non-zero elements are efficiently distributed across threads to avoid underutilization. For cases where non-zero elements cannot be evenly divided among subwarps, the residue unroll technique dynamically allocates threads to process the remaining elements, minimizing branching overhead and improving thread utilization. Intermediate results are stored in shared memory to reduce latency, and the final results for short rows are written directly to global memory. For the remaining parts of long rows, Atomic Add is used to accumulate intermediate results into global memory.

As shown in Figure 5, the remaining part of the long row at index 0 is marked as a short row, and indices 0, 1, and 2 are assigned to a single thread block for processing. Since we assume that a thread block contains only four threads, index 3 is assigned to another thread block. The result of the short row at index 0 is then accumulated with the result of the long row at index 0 using an atomic addition. Although the atomic addition introduces a certain amount of overhead, our method effectively achieves load balancing.

4.3 HR-SpMM Kernel

As shown in Algorithm 2, the Long Row Algorithm is an optimized sparse matrix-dense matrix multiplication algorithm, primarily designed for efficiently computing the matrix product of long rows. First, the target row to be processed is selected, and the number of non-zero elements in the row is calculated using the row pointers and row indices from the input sparse matrix. Then, the row is divided into multiple blocks, with each block processed by a thread block to ensure appropriate workload distribution. In lines 13 and 14, each thread block loads the sparse matrix row data and the dense matrix column data into shared memory, utilizing Tensor Cores for efficient matrix multiplication. Finally, the computation results are stored in the local buffer in shared memory, and after completion, the results are accumulated into the global output matrix C at the corresponding positions using atomic add operations, ensuring data consistency during parallel computations. The overhead of transforming a row into a matrix is very small and can be neglected.

As shown in algorithm 3, since Sputnik's method is particularly suitable for our method, we use its method to deal with our rows part less than 64, the first 5 lines, we use ROMA algorithm to compute the index and nnz, and then, in lines 6 through 12, using 1D Tiling technique and residue unroll, Each thread block loads the relevant data of sparse matrix A and dense matrix B into shared memory and computes it to optimize memory access and improve computational efficiency. Finally, for the remaining elements of the long line, we use AtomicAdd in shared memory, and for the short line, we write it directly into the output matrix C. Through these techniques, the algorithm can significantly improve the computational efficiency when dealing with sparse matrices, especially in deep learning applications, and can greatly improve the computational performance of sparse matrices.

Algorithm 3: Cuda Core Utilization in SpMM Kernel

1 **Input**: *New_Row_Index*, *m_idx*, *Row_Ptr*
2 **Output**: *Dense Output Matrix C*
3 $target\_row = New\_Row\_Index[m\_idx]$;
4 $nnz = (Row\_Ptr[target\_row + 1] - ROMA(Row\_Ptr[target\_row]))\%64$;
5 $row\_start = Row\_Ptr[target\_row + 1] - nnz$;
6 c_tile = Tile1D(init_to=0);
7 **while** $nnz > 0$ **do**
8 a_tile = LoadTileShared(A);
9 b_tile = LoadTileShared(B);
10 $c\_tile = c\_tile + a\_tile \times b\_tile$;
11 $nnz = nnz - kUnroll$;
12 **end while**
13 **if** $Row\_Ptr[target\_row+1] - Row\_Ptr[target\_row] \geq 64$ **then**
14 AtomicAdd($C[target\_row], c\_tile$);
15 **else**
16 Add($C[target\_row], c\_tile$);
17 **end**

5 Evaluation

5.1 Experiment Setup

Datasets: To comprehensively evaluate the proposed SpMM kernel, we selected 388 datasets from the publicly available Deep Learning Matrix Collection (DLMC) and SuiteSparse datasets. These datasets exhibit significant differences in size, sparsity, and structural characteristics, covering various real-world application scenarios and effectively testing the algorithm's performance under different conditions.

The DLMC dataset includes sparse matrices collected from deep neural networks, such as Transformer models. These networks utilize various sparsification and pruning techniques during training and inference to improve efficiency

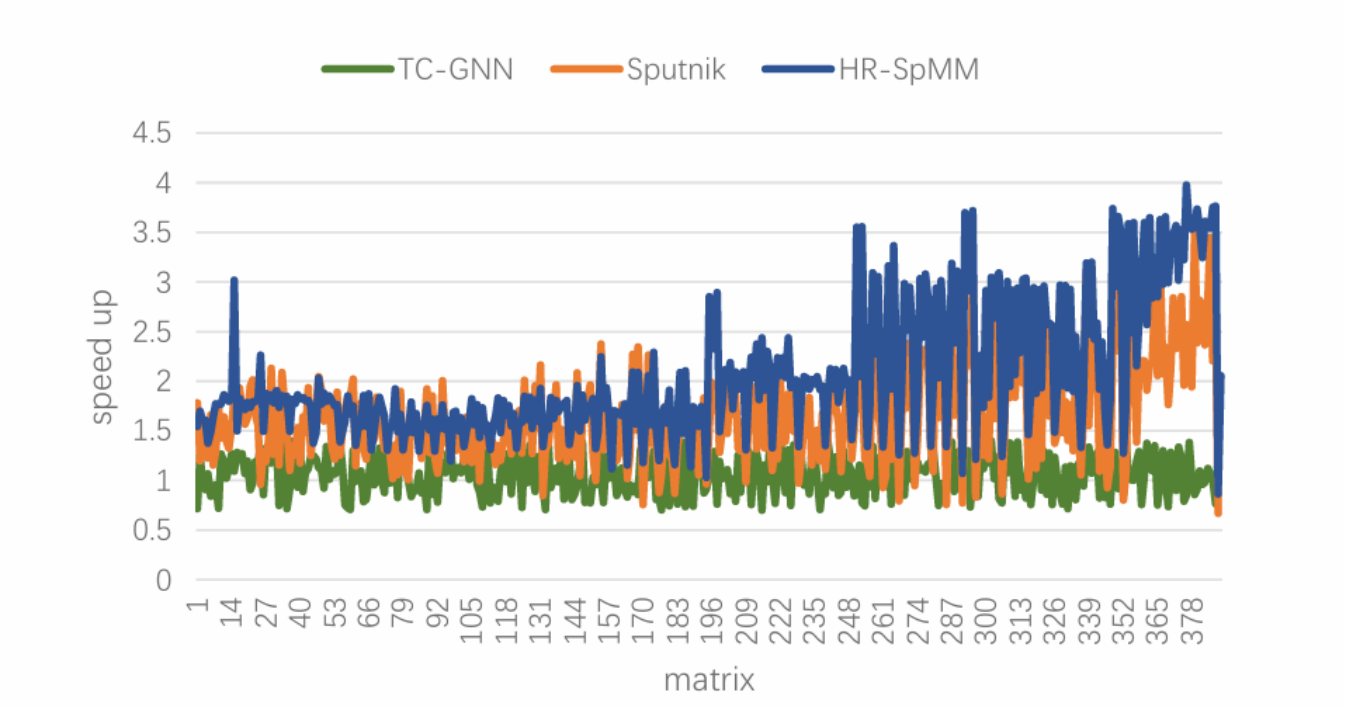

Figure 6: Comparison of HR-SpMM's Overall Performance with cuSPARSE, TC-GNN, and Sputnik

and reduce computational overhead. The sparsification techniques include: L0_regularization, Magnitude_pruning, Random_pruning, Variational_dropout.

SuiteSparse is a widely-used collection of sparse matrices from real-world applications, designed for benchmarking and evaluating sparse matrix algorithms. It includes matrices from diverse fields such as structural engineering, circuit simulation, image processing, optimization, and graph analysis. The matrices vary significantly in size, sparsity, and structural complexity, ranging from small-scale problems to large-scale ones with millions of rows. SuiteSparse serves as a valuable resource for testing the performance and scalability of algorithms like Sparse Matrix-Matrix Multiplication (SpMM), sparse linear solvers, and graph-based computations. Its diversity and real-world relevance make it an essential benchmark for researchers and developers working on sparse matrix algorithms.

By conducting experiments on the DLMC and SuiteSparse datasets, we effectively validated the performance and adaptability of the SpMM kernel for handling sparse matrices and graph data, providing critical insights for optimizing graph neural networks and addressing sparsity challenges.

Baselines: We compared the performance of HR-SpMM with state-of-the-art parallel SpMM implementations, including Sputnik, cuSPARSE, and TC-GNN. Below is an introduction to these methods:

Sputnik (Sparse GPU Kernels for Deep Learning) [11]: Sputnik focuses on the characteristics of sparse matrices in deep learning and proposes an efficient method for sparse matrix computations to accelerate deep neural network operations on GPUs. The main contribution of Sputnik is the development of two high-performance GPU kernels (SpMM and SDDMM) that operate directly on the standard compressed sparse row (CSR) format without imposing specific constraints on the locations of non-zero values.

cuSPARSE [18]: cuSPARSE is a library provided by NVIDIA as part of the CUDA toolkit. It offers a suite of highly optimized sparse matrix operations, including SpMM. As one of the most widely used GPU-accelerated libraries, cuSPARSE serves as a benchmark for sparse matrix computations, providing efficient implementations for a wide range of sparsity patterns.

TC-GNN [27]: TC-GNN is a framework designed to accelerate Graph Neural Network (GNN) computations on GPU Tensor Cores (TCUs). The core idea is to leverage a "Sparse Graph Translation" technique, which efficiently maps sparse GNN computations onto the dense computation capabilities

of Tensor Cores. Additionally, TC-GNN employs collaborative optimization between CUDA cores and Tensor Cores to further enhance performance.

By comparing HR-SpMM with these implementations, we aim to demonstrate its competitive performance and highlight its advantages across diverse sparsity levels, hardware platforms, and application scenarios.

Platform: All experiments were conducted on Ubuntu 20.04, using an Intel Core i9-10900F CPU and an Nvidia RTX 3070 GPU, with CUDA version 11.6.

5.2 Performance Evaluation

In this subsection, we compare HR-SpMM with state-of-the-art SpMM kernels and conduct comprehensive experiments to evaluate the proposed optimization techniques.

Figure 6 presents the performance comparison of HR-SpMM optimization against various parallel SpMM implementations, using cuSPARSE as the baseline. The execution time was measured with nvprof, and we reported only the SpMM kernel time, excluding preprocessing and PCIe transfer overhead. The data transferred via PCIe included only the dense matrix and the sparse matrix in CSR format. Due to the significant difference in matrix sizes between the DLMC and SuiteSparse datasets, we use the speedup ratio to represent our performance comparison. Our results show that HR-SpMM achieves an average speedup of 2.05× compared to cuSPARSE, 2× compared to TC-GNN, and 1.25× compared to Sputnik. Additionally, to highlight the effectiveness of the hybrid approach using Tensor Core and CUDA Core, we also compared HR-SpMM with the PyTorch SpMM kernel executed on a CPU, achieving an average speedup of 167.59×.

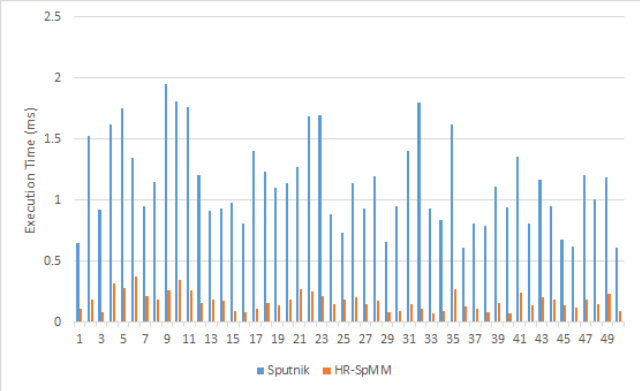

Figure 7: Preprocessing Overhead Comparison: HR-SpMM vs. Sputnik

For TC-GNN, while preprocessing improves SpMM performance by making the matrix denser, it introduces significant overheads and it doesn't work particularly well. In contrast, our approach demonstrates that using a hybrid kernel is effective, even though HR-SpMM does not always outperform Sputnik in speedup. However, as shown in Figure 7, Sputnik involves much higher preprocessing overhead, whereas the preprocessing time of HR-SpMM is only about 12% of that of Sputnik. Overall, HR-SpMM achieves better speedups with significantly lower preprocessing overhead. To illustrate the validity of our method, we analyze their performance using an unevenly distributed matrix with both long and short rows, which represents the bulk of the sparse matrix. In Table 1, we present the performance metrics of various kernels, highlighting the differences between the cuSPARSE, TC-GNN, Sputnik, and our HR-SpMM implementations, which utilize both Tensor Cores and CUDA Cores.

Execution Time: Our HR-SpMM implementation using Tensor Cores and CUDA Cores significantly reduces execution time to just 585 microseconds, much faster than cuSPARSE (1.8 milliseconds), TC-GNN (1.2 milliseconds), and Sputnik (4.1 milliseconds). Memory Throughput (GB/s): Memory throughput measures the amount of data transferred per second, which is crucial for GPU performance. The HR-SpMM (Tensor Core) method achieves a memory throughput of 196 GB/s, and the HR-SpMM (CUDA Core) implementation reaches 283 GB/s, highlighting its ability to efficiently handle large data volumes. These values are much higher than TC-GNN (186 GB/s), Sputnik (32 GB/s), and cuSPARSE (73 GB/s), showcasing the effective memory management in our design. Warp Cycles Per Instruction: This metric measures the number of cycles required to complete one instruction, with lower values indicating better efficiency. The number of warp cycles of the Tensor core in our method is relatively smaller than that of the cuda core because irregular short rows are handled by the cuda core.

In summary, HR-SpMM (Tensor Core) excels in both memory throughput and warp cycle efficiency, making it ideal for handling long rows in sparse matrices. On the other hand, HR-SpMM (CUDA Core) executes faster in terms of execution time, but it requires more warp cycles due to the need to handle more irregular matrix rows. This trade-off highlights how our hybrid approach optimizes performance depending on the matrix's characteristics, making HR-SpMM highly adaptable and efficient for a range of sparse matrix computations.

6 Conclusion

This paper introduces HR-SpMM, a method that enhances Sparse Matrix-Matrix Multiplication (SpMM) performance through adaptive row partitioning and hybrid kernel design. Our approach partitions sparse matrix rows into long and short categories, leveraging Tensor Cores and CUDA Cores

Kernel	Execution Time	Memory Throughput (GB/s)	Warp Cycles Per Instr
cuSPARSE	1.8 ms	73 GB/s	13.61
TC-GNN	1.2ms	186 GB/s	23.6
Sputnik	4.1 ms	32 GB/s	26.93
HR-SpMM (Tensor core)	512 us	196 GB/s	12.4
HR-SpMM (Cuda core)	73 us	283 GB/s	28.6

Table 1: Performance Metrics of Different Kernels

for efficient computation, thus maximizing hardware utilization and achieving high computational efficiency.

Although the atomic addition overhead between the tensor kernel and CUDA kernel is unavoidable, and the conversion of long rows into matrices has some overhead, the overall performance is still strong. In the future, we plan to integrate HR-SpMM into the graph Neural network (GNN) framework to further optimize sparse matrix operations in large-scale graph data processing and promote efficient graph computation applications.

Acknowledgments

This work is financially supported by the Sichuan Natural Science Foundation for Distinguished Young Scholar (2023NSFSC1966), the National Natural Science Foundation of China (61672438), and the Postgraduate Innovation Fund Project by Southwest University of Science and Technology (24ycx1137). Yaobin Wang is the corresponding author.

References

[1] [n. d.]. *Nvidia cuda toolkit.* https://docs.nvidia.com/cuda/index.html.

[2] Willow Ahrens and Erik G. Boman. 2020. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. *ArXiv* abs/2005.12414 (2020). https://api.semanticscholar.org/CorpusID:218889474

[3] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1213–1222. https://doi.org/10.1109/IPDPS.2014.125

[4] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

[5] Nvidia blocked-sparse api. [n. d.]. *Nvidia.* https://docs.nvidia.com/cuda/cusparse/index.html..

[6] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 115–126. https://doi.org/10.1145/1693453.1693471

[7] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Not.* 45, 5 (Jan. 2010), 115–126. https://doi.org/10.1145/1837853.1693471

[8] Nvshmem communication library. [n. d.]. *NVIDIA.* https://developer.nvidia.com/nvshmem..

[9] Gunduz Vehbi Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. 2022. Scalable Graph Convolutional Network Training on Distributed-Memory Systems. *Proceedings of the VLDB Endowment* 16, 4 (1 Dec. 2022), 711–724. https://doi.org/10.14778/3574245.3574256 The 49th International Conference on Very Large Data Bases, 2023, VLDB 2023 ; Conference date: 28-08-2023 Through 01-09-2023.

[10] Qiang Fu, Yuede Ji, and H Howie Huang. 2022. TLPGNN: A lightweight two-level parallelism paradigm for graph neural network computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 122–134.

[11] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00021

[12] Mingfeng Guo, Yaobin Wang, Yajun Gu, Yufang Chen, Huan Liu, Huarong Chen, Dongxuan Han, Hengyang Xu, Chunhua Deng, Pingping Tang, et al. 2023. Bs-SpMM: Accelerate Sparse Matrix-Matrix Multiplication by Balanced Split Strategy on the GPU. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–6.

[13] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. 2018. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 66–79.

[14] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 300–314.

[15] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Gespmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[16] Zhiyuan Li, Xun Jian 0001, Yue Wang, Yingxia Shao, and Lei Chen. 2024. DAHA: Accelerating GNN Training with Data and Hardware Aware Execution Planning. *PVLDB* 17, 6 (February 2024), 1364–1376. https://www.vldb.org/pvldb/vol17/p1364-li.pdf

[17] Warp matrix multiply accumulate(wmma). [n. d.]. *Nvidia.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html..

[18] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*, Vol. 12.

[19] Yingchen Song, Yaobin Wang, Chaoyu Xiong, Tianhai Wang, and Pingping Tang. 2024. An Efficient Sampling-Based SpMM Kernel for Balancing Accuracy and Speed in GNN Inference. In *2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. 468–475. https://doi.org/10.1109/ISPA63168.2024.00066

[20] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi,

Dipankar Das, and Pradeep K. Dubey. 2015. GraphMat: High performance graph analytics made productive. *ArXiv* abs/1503.07241 (2015). https://api.semanticscholar.org/CorpusID:8312489

[21] Improved tensor core operations. [n. d.]. *NVIDIA*. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html.

[22] Kiran Koshy Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based Graph Neural Network for Semi-supervised Learning. *ArXiv* abs/1803.03735 (2018). https://api.semanticscholar.org/CorpusID:3847272

[23] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–11. https://doi.org/10.1109/SC.2008.5214359

[24] Aristidis G. Vrahatis, Konstantinos Lazaros, and Sotiris Kotsiantis. 2024. Graph Attention Networks: A Comprehensive Review of Methods and Applications. *Future Internet* 16, 9 (2024). https://doi.org/10.3390/fi16090318

[25] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. (2019).

[26] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[27] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 149–164. https://www.usenix.org/conference/atc23/presentation/wang-yuke

[28] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? *ArXiv* abs/1810.00826 (2018). https://api.semanticscholar.org/CorpusID:52895589

[29] Carl Yang, Aydın Buluç, and John Douglas Owens. 2019. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48 (2019), 1 – 51. https://api.semanticscholar.org/CorpusID:198167536

[30] Yi Yang, Ping Xiang, Jingfei Kong, Mike Mantor, and Huiyang Zhou. 2012. A unified optimizing compiler framework for different GPGPU architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 2 (2012), 1–33.