HC-SpMM: Accelerating Sparse Matrix-Matrix Multiplication for Graphs with Hybrid GPU Cores

Zhonggen Li*, Xiangyu Ke*, Yifan Zhu*, Yunjun Gao*, Yaofeng Tu†
*Zhejiang University, Hangzhou, China †ZTE Corporation, Nanjing, China
*{zgli, xiangyu.ke, xtf_z, gaoyj}@zju.edu.cn, †tu.yaofeng@zte.com.cn

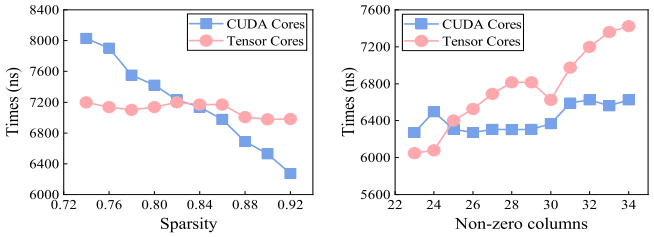
***Abstract*—Sparse Matrix-Matrix Multiplication (SpMM) is a fundamental operation in graph computing and analytics. However, the irregularity of real-world graphs poses significant challenges to achieving efficient SpMM operation for graph data on GPUs. Recently, significant advancements in GPU computing power and the introduction of new efficient computing cores within GPUs offer new opportunities for acceleration.**

In this paper, we present HC-SpMM, a pioneering algorithm that leverages hybrid GPU cores (Tensor cores and CUDA cores) to accelerate SpMM for graphs. To adapt to the computing characteristics of different GPU cores, we investigate the impact of sparse graph features on the performance of different cores, develop a data partitioning technique for the graph adjacency matrix, and devise a novel strategy for intelligently selecting the most efficient cores for processing each submatrix. Additionally, we optimize it by considering memory access and thread utilization, to utilize the computational resources to their fullest potential. To support complex graph computing workloads, we integrate HC-SpMM into the GNN training pipeline. Furthermore, we propose a kernel fusion strategy to enhance data reuse, as well as a cost-effective graph layout reorganization method to mitigate the irregular and sparse issues of real-world graphs, better fitting the computational models of hybrid GPU cores. Extensive experiments on 13 real-world graph datasets demonstrate that HC-SpMM achieves an average speedup of 1.33× and 1.23× over state-of-the-art SpMM kernels and GNN frameworks.

***Index Terms*—GPU cores, SpMM, Graph Computing**

I. Introduction

Sparse Matrix-Matrix Multiplication (SpMM) is a fundamental operation that multiplies a sparse matrix and a dense matrix, which has been widely used in various graph computing tasks such as Graph Neural Networks (GNNs) training & inference [8], [25], [59], PageRank [18], [26], [33] and graph clustering [3], [50], [56]. Recent studies employ matrix multiplication operations on GPUs to accelerate commonly used graph algorithms such as triangle counting and shortest path computing, with SpMM emerging as one of the core operations and becoming the efficiency bottleneck [36], [54]. SpMM is also a fundamental operation in GNNs, accounting for more than 80% of the GNN training time [43]. However, characterized by *irregular memory access patterns* and *low computational intensity*, the real-world graphs pose significant challenges for achieving efficient SpMM on GPU [43]. Especially for GNNs, as the scale of graph data proceeds to grow and the complexity of GNN architectures escalates with an increasing number of layers, the efficiency issue becomes more serious, constraining the large-scale applicability[1], even though it achieves state-of-the-art performance on a myriad of problems such as node classification [34], [44], [58], link prediction [15], [31], [61], [65], recommendation [46], [48], [57] and beyond [4], [23], [35].



(a) Varying sparsity. (b) Varying non-zero columns.

Fig. 1. SpMM execution time with varying sparsity and non-zero columns.

To enhance the efficiency of SpMM, some efforts have been made to employ CUDA cores in GPU for acceleration [13], [17], [20], [28]. As the performance ceil is constrained by the hardware structure of CUDA cores, recent attempts have turned to explore harnessing the Tensor cores, which are specialized for efficient matrix multiplication, to accelerate SpMM [5], [24], [55]. However, despite significantly improving the efficiency of dense matrix multiplication [24], the irregularity and sparsity of real-world graphs impeding Tensor cores from achieving satisfactory performance [11], [43], [55]. Various preprocessing techniques have been investigated to enhance the density of sparse matrices, yet significant sparse portions persist, hindering performance improvement[2].

Figure 1 illustrates the time overhead incurred by CUDA cores and Tensor cores across matrices with varying sparsity and the number of non-zero columns, which highlights their distinct applicabilities[3]. CUDA cores demonstrate superior performance when handling sparser matrices with a higher number of non-zero columns, whereas Tensor cores exhibit greater efficacy in denser matrices characterized by fewer non-zero columns. Thereafter, *relying solely on CUDA cores or Tensor cores for SpMM acceleration fails to fully exploit their respective computational strengths*, primarily due to the irregularity of real-world graphs, which typically comprise a mixture of dense and sparse regions [52].

[1]Training a GNN-based recommendation system over a dataset comprising 7.5 billion items consumes approximately 3 days on a 16-GPU cluster [9].

[2]For instance, the average sparsity of the matrices output by the preprocessing method proposed in TC-GNN [43] is still 90.9% on 10 tested datasets.

[3]An in-depth analyzation is provided in § IV-C.

In this paper, we aim to devise a graph-friendly SpMM kernel using hybrid CUDA cores and Tensor cores, tailored to the varying computational demands of different graph regions, to support high-performance graph computing workloads. Furthermore, as a DB technique for AI acceleration, we integrate the hybrid SpMM kernel into the GNN training pipeline to enhance the training efficiency and demonstrate its effectiveness in complex graph computing scenarios.

However, crafting a hybrid SpMM kernel using CUDA cores and Tensor cores based on the characteristics of graph data and systematically integrating it into the GNN training pipeline encounters non-trivial challenges.

Challenges of designing hybrid SpMM kernel. (1) Effectively partitioning the adjacency matrix into sub-regions with distinct sparsity characteristics to leverage the different cores for collaborative computation presents a considerable challenge. **(2)** The computational characteristics of CUDA and Tensor cores vary significantly, which is also influenced by the graph features, making the selection of the optimal core for matrices with different sparsity levels crucial for enhancing efficiency. **(3)** Accurately modeling the computational performance of CUDA and Tensor cores for SpMM to facilitate precise core selection is another major difficulty. **(4)** Last but not least, inefficiencies inherent in the SpMM kernel, such as suboptimal memory access patterns and underutilized threads, limit the overall performance. To address these issues, we **firstly** propose a fine-grained partition strategy, which divides the adjacency matrix into equal-sized submatrices along the horizontal axis, allocating each submatrix to the appropriate GPU cores for efficient computation (§ IV-A). This allows CUDA and Tensor cores to perform calculations independently, eliminating the need to merge results between cores. **Secondly**, comprehensive quantitative experiments reveal that CUDA cores are *memory-efficient* while Tensor cores are *computing-efficient* (§ IV-B). These experiments identify two pivotal factors for submatrix characterization: sparsity and the number of non-zero columns, which dominate the most expensive parts for CUDA and Tensor cores, *computation* and *memory access*, respectively. **Thirdly**, leveraging these factors, we develop an algorithm tailored for the selection of appropriate GPU cores for submatrices, optimizing computational capability (§ IV-B). **Finally**, we conduct in-depth optimizations of the SpMM kernel, considering thread collaboration mode (§ IV-D1) and memory access patterns (§ IV-D2).

Challenges of integrating the SpMM kernel into GNN. When integrating our proposed hybrid SpMM kernel into the GNN training pipeline, there arise new challenges. **(1)** The isolation among GPU kernels within a GNN layer in prevalent GNN training frameworks [39], [43] impedes data reuse, leads to additional memory access overhead, and introduces significant kernel launch overhead. **(2)** Tensor cores incur significantly higher throughput than CUDA cores, potentially offering substantial efficiency gain. However, real-world graph layouts inherently exhibit irregularity and sparsity, resulting in a majority of segments partitioned from the adjacency matrix being sparse and less amenable to processing via Tensor cores. Consequently, the performance gains achievable with Tensor cores are often negligible [43]. To tackle the **first** problem, we discover opportunities to reuse data in the shared memory of GPU and present a kernel fusion strategy to mitigate kernel launch costs and global memory access (§ V-A). To address the **second** problem, we first introduce a metric termed *computating intensity* to estimate the calculation workload of the submatrices multiplication (§ V-B), which is calculated by the quotient of the number of non-zero elements and the number of non-zero columns. Higher computational intensity is achieved with more non-zero elements and fewer non-zero columns. Subsequently, we propose an efficient algorithm to reconstruct submatrices, adjusting the graph layouts to obtain more dense segments suitable for processing by Tensor cores, gaining significant efficiency with Tensor cores (§ V-B). This adjustment has a relatively small cost compared with GNN training but renders the graph data more compatible with hybrid GPU cores, unlocking the full computational potential of Tensor cores and thereby significantly enhancing the efficiency of GNN training (§ VI-C3).

Contributions. In this work, we propose HC-SpMM, a novel approach for accelerating SpMM using hybrid GPU cores. Our key contributions are summarized as follows:

- We quantify the difference between CUDA and Tensor cores in SpMM, and propose a hybrid SpMM kernel, which partitions the graph adjacency matrix and intelligently selects appropriate GPU cores for the computation of each submatrix based on its characteristics. We further optimize the SpMM kernel considering thread collaboration mode and memory access patterns (§ IV).
- We propose a kernel fusion strategy for integrating HC-SpMM into the GNN training pipeline, eliminating kernel launch time and enabling efficient data reuse, and present a lightweight graph layout optimization algorithm to enhance irregular graph layouts and better align with the characteristics of both GPU cores (§ V).
- We conduct comprehensive evaluations demonstrating that HC-SpMM outperforms existing methods, achieving 1.33× speedup in SpMM and 1.23× speedup in GNN training on average (§ VI).

The rest of this paper is organized as follows. Section II reviews the related work. Section III gives the preliminaries. Section IV presents the design of the hybrid SpMM kernel. Section V describes the optimization of integrating the SpMM kernel into GNNs. Section VI exhibits the experimental results. We conclude the paper in Section VII.

II. Related Work

SpMM Using CUDA Cores. Optimization of SpMM has been a subject of extensive study [2], [17], [47], [53], [62]. In recent years, Yang et al. [53] leveraged merge-based load balancing and row-major coalesced memory access strategies to accelerate SpMM. Hong et al. [17] designed an adaptive tiling strategy to enhance the performance of SpMM. cuSPARSE [28] offers high-performance SpMM kernels and has been integrated into numerous GNN training frameworks

such as DGL [39]. Gale et al. point out that cuSPARSE is efficient only for matrices with sparsity exceeding 98%. Consequently, they proposed Sputnik [13] to accelerate the unstructured SpMM in deep neural networks and achieve state-of-the-art performance. Dai et al. [7] introduced an approach capable of heuristically selecting suitable kernels based on input matrices. However, it demonstrates superior performance only when the matrix dimension is less than 32. Furthermore, Fan et al. [10] proposed an SpMM kernel employing a unified hybrid parallel strategy of mixing nodes and edges, achieving efficient performance. To the best of our knowledge, there are few SpMM kernels specifically designed for graph computing workloads. Huang et al. [20] introduced a GNN-specified SpMM kernel, utilizing coalesced row caching and coarse-grained warp merging to optimize memory access. However, it did not fully consider the characteristics of the graph adjacency matrix. Despite highly optimized algorithms, the computational capabilities of CUDA cores are inherently restricted by the hardware structures and are less efficient than Tensor cores, hindering the efficiency improvement in dense matrix multiplication.

SpMM Using Tensor Cores. Numerous recent works have been shifted to accelerate SpMM with the assistance of Tensor cores [5], [11], [24], [41], [43]. However, most of them require structured input matrices, which is often impractical for adjacency matrices of real-world graphs. Alternatively, other works focus on unstructured SpMM and employ preprocessing methods to reduce the number of tiles needing traversal and to avoid unnecessary computation [11], [43], [55]. For example, TC-GNN [43] compresses all zero columns within a row window and DTC-SpMM [11] transforms the matrix into memory-efficient format named ME-TCF. Although TC-GNN implements a CUDA and Tensor cores collaboration design, it just employs CUDA cores for data loading, not for computing. Different from it, our proposed HC-SpMM employs both cores for computing according to their characteristics. Besides, Xue et al. [51] propose an unstructured SpMM kernel using Tensor cores, introducing a format named Tile-CSR to reduce the zero elements in submatrices traversed by Tensor cores. However, this kernel only supports half precision. As mentioned, Tensor cores are not natively suitable for unstructured SpMM. While preprocessing can densify submatrices, they still involve some computational waste and lead to suboptimal performance.

GNN Training Frameworks. Motivated by the remarkable performance but inefficient training of GNN, plenty of GNN training frameworks have been proposed, focusing on the graph locality [42], [59], the GPU memory utilization [25], etc. Besides, various frameworks for distributed GNN training have emerged recently [8], [27], [30], [37], [38], [40], [60], [64], which represent an orthogonal research direction. Our work focuses on optimizing SpMM and employing hybrid GPU cores for accelerating basic matrix multiplication, which can be seamlessly integrated into any aforementioned frameworks. This integration allows the *Aggregation* phase to directly call the optimized SpMM kernel, significantly reducing the execution time of *Aggregation*. Works addressing similar problems include GE-SpMM [20] and TC-GNN [43] mentioned above, both of which have integrated the kernels into GNN training frameworks. However, they solely employ CUDA or Tensor cores to perform SpMM operations in GNNs, failing to fully exploit the respective computational strengths of GPU cores.

Heterogeneous processing in data computing. Some efforts are dedicated to employing heterogeneous systems for queries in databases [1], [6], [32], general computing tasks [19] and deep learning [22], [63]. Other research employs heterogeneous processing elements for SpMM [14] and employs CUDA & Tensor cores for general matrix multiplication [16]. Our work focuses on accelerating SpMM by heterogeneous GPU cores, a topic that has barely been explored.

III. Preliminaries

In this section, we provide a concise description of GPU architecture, including the characteristics of CUDA and Tensor cores, followed by an overview of GNNs.

A. GPU Architecture

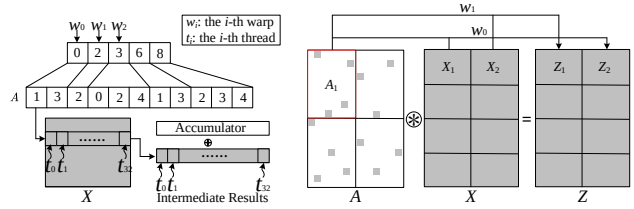
GPU is a highly parallel hardware comprising tens of Streaming Multiprocessors (SMs). Each SM features dedicated local memory, registers, and processing cores. These SMs individually schedule the execution of threads in warps, each consisting of 32 threads. Threads in a warp run simultaneously in a Single Instruction Multiple Threads (SIMT) fashion. A block consisting of multiple warps is allocated to an SM.

The GPU's memory hierarchy includes global memory, shared memory, and registers. Global memory, shared by all SMs, offers the largest capacity but lower I/O bandwidth. Each SM contains fast but limited shared memory, typically ranging from 16 KB to 64 KB. Additionally, each SM includes registers, which serve as the fastest storage structure.

Access to global memory within a warp is granular at 128 bytes when the L1 cache is enabled or 32 bytes otherwise. Consequently, when 32 threads within a warp request consecutive data within 128 bytes, only one memory access transaction is necessary, resulting in coalesced memory access. Shared memory is partitioned into multiple independent storage areas known as banks. Each bank can independently serve a thread in a single clock cycle. Concurrent access by multiple threads to the same bank results in a conflict, diminishing memory access efficiency. In shared memory, data is allocated across 32 consecutive banks, with each bank having a granularity of 4 bytes. For instance, when storing 64 numbers of float type in shared memory, the first 32 numbers and the last 32 numbers will be mapped to the 32 banks respectively. The 1st and 33rd numbers are assigned to the same bank, with the remaining numbers following a similar pattern.

B. Computing Cores in GPU

CUDA Cores. CUDA cores are the primary computing cores used in GPUs. Each thread is assigned to a CUDA core for performing various calculations. Furthermore, CUDA cores



(a) Computing procedure of CUDA cores (b) Computing procedure of Tensor cores

Fig. 2. SpMM computing procedure of CUDA cores and Tensor cores.

can execute only one operation per clock cycle. Multiplying two 4×4 matrices involves 64 multiplications and 48 additions.

Tensor Cores. Tensor cores, specifically designed for deep learning computation, have been integrated into advanced GPUs since 2017. Unlike CUDA cores, Tensor cores operate at the warp level, where threads within a warp collaboratively multiply two fixed-size matrices (e.g. 4× 4) in a single clock cycle. In addition to utilizing the matrix multiplication APIs provided by cuBLAS [29], Tensor cores can also be leveraged through the Warp Matrix Multiply-Accumulate (WMMA) API for more flexible programming. The WMMA API requires a fixed-size input matrix, with the required size varying depending on the data type. In this paper, we use TF-32 as the input data type of Tensor cores, following previous work [43], which requires $16 \times 8 \times 16$ as the input size. Although more efficient than CUDA cores in matrix multiplication, the fix-sized input requirement restricts the flexibility to avoid computing numerous zero elements in sparse adjacency matrices of graphs, limiting its efficiency in performing SpMM.

Figure 2 illustrates the disparity between CUDA and Tensor cores during SpMM. w_i represents the i-th warp and t_j represents the j-th thread. As depicted in Figure 2(a), each thread computes a single element in the result matrix, i.e., Z in Equation 1. CUDA cores possess the flexibility to efficiently skip zero elements in the sparse matrix and perform multiplication operations based on the CSR format. Conversely, Tensor cores conduct computations at the warp level as shown in Figure 2(b). Each warp retrieves a submatrix from both the sparse matrix A and the dense matrix X, subsequently computing the multiplication products. Despite the presence of numerous zeros in the sparse matrix, Tensor cores are unable to skip them, resulting in the waste of computational resources.

Based on the characteristics outlined above, we can observe that while Tensor cores offer high efficiency in matrix multiplication, they lack flexibility compared to CUDA cores, particularly in handling sparse matrices. Tensor cores process all elements indiscriminately due to their strict input criteria. Therefore, determining which cores are more efficient in the context of SpMM is challenging. This paper aims to devise a strategy for dynamically selecting the appropriate cores for submatrices within a sparse matrix for enhanced performance.

C. Graph Neural Networks

In a graph $G = (V, E)$, where V and E represent the node set and edge set, respectively, the adjacency matrix of G is denoted by $A_{|V|\times|V|}$. Each node has an embedding, which is a continuous vector representation with dimension D. The

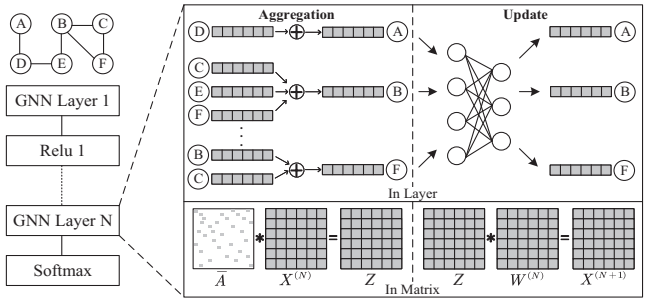


Fig. 3. An example of GNN.

embedding encodes various properties of a node according to the downstream tasks. In GNNs, node embeddings are represented as matrix $X_{|V|\times D}$. Typically, a GNN layer consists of two fundamental operations: *Aggregation* and *Update*. An example of GNN is depicted in Figure 3.

In the *Aggregation* operation, each node aggregates its feature vector and those of its neighbors from X to form the new one. This process at layer k can be formalized as a matrix multiplication operation, as depicted in Equation 1, where $\bar{A}$ is calculated from A, and Z is the aggregated result. The matrix $\bar{A}$ is always sparse, making the *Aggregation* operation analogous to an SpMM-like operation.

$$Z = \bar{A}X^{(k)} \tag{1}$$

Equation 2 formalizes the *Update* operation, where $W^{(k)}$ denotes the network parameters at layer k and $X^{(k+1)}$ is the updated feature matrix. This operation can be efficiently executed using the *gemm* kernel in cuBLAS.

$$X^{(k+1)} = ZW^{(k)} \tag{2}$$

Suppose the gradient is $X'^{(k+1)}$. Backward propagation also involves *Aggregation* and *Update*. The *Update* phase in layer k is formalized as Equation 3, containing two *gemm* operations.

$$W'^{(k)} = Z^T X'^{(k+1)}; Z'^{(k)} = X'^{(k+1)} W^{(k)T} \tag{3}$$

The *Aggregation* operation can be abstracted as an SpMM operation as Equation 4.

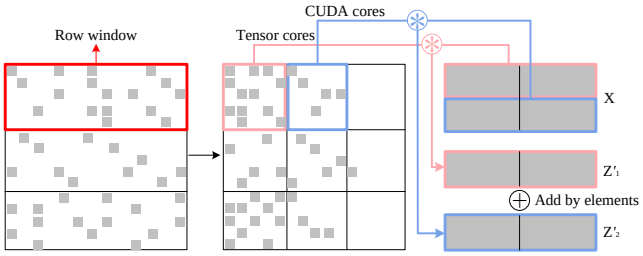
$$X'^{(k)} = \bar{A}Z'^{(k)} \tag{4}$$

IV. Hybrid SpMM Kernel

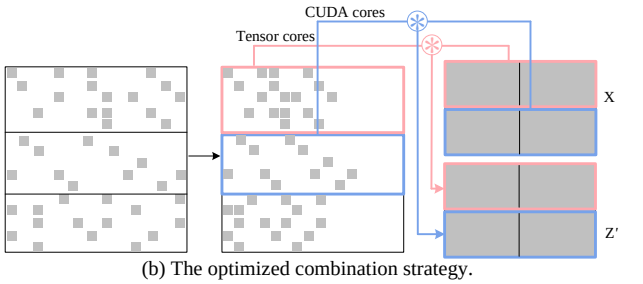
In this section, we justify our selection of row window as the fundamental hybrid unit, contrasting it with the straightforward choice of submatrices. Additionally, we delve into the essential characteristics that influence the efficiency of GPU cores, serving as the driving force behind our subsequent designs.

A. Combination Strategy

Initially, we need to consider how to combine the two GPU cores in a sparse matrix computation, i.e., the partitioning of the sparse matrix into submatrices and the processing strategy of each submatrix using different GPU cores. A straightforward strategy is illustrated in Figure 4(a). Due to the fixed-size input requirement of the WMMA API, it divides the input matrix into submatrices sized $16 \times \#nodes$, named *row windows*. Within each row window, columns are rearranged based on the count of non-zero elements. Intuitively, the first few columns within a row window tend to be denser, while



(a) A straightforward combination strategy.



(b) The optimized combination strategy.

Fig. 4. Combination strategies for SpMM on CUDA and Tensor cores.

subsequent ones are sparser. It further splits each row window into 16×8 submatrices and traverses the matrix in units of 16×8 which is the minimum granularity required by WMMA API for input. Each 16×8 submatrix is assigned to appropriate GPU cores based on our identified characteristics (§ IV-C).

While the straightforward strategy offers fine-grained traversal, upon closer examination, it reveals several limitations. **(1)** Merging the results from CUDA and Tensor cores introduces extra overhead. Results computed by Tensor cores are initially buffered in registers before being transferred to shared or global memory, while those computed by CUDA cores are directly stored in shared or global memory. Within each row window, the merging of results from Tensor and CUDA cores necessitates additional I/O and adds to the overhead of addition operations[4]. **(2)** Hybrid computation within a row window necessitates the separate storage of edges for Tensor cores and CUDA cores, resulting in increased preprocessing overhead and poor access locality. **(3)** The execution time in the granularity of $16 \times 8 \times 16$ matrix multiplication complicates accurate measurement[5]. Furthermore, sparsity is the only characteristic that can be leveraged for core selection in 16×8 submatrices. These constraints hinder a comprehensive analysis of the relationship between GPU cores and matrix characteristics.

To tackle the aforementioned issues while maintaining fine granularity, instead of using a 16×8 submatrix, we adopt the row window as the minimum hybrid unit. Following a strategy similar to TC-GNN [43], we position non-zero columns at the forefront of row windows, thereby density the submatrices, i.e., the row windows. As depicted in Figure 4(b), within each row window, Tensor cores execute across all 16×8 submatrices, while CUDA cores compute directly using the CSR format. This approach eliminates the need for merging the results from CUDA and Tensor cores, reducing the extra

[4]The overhead is up to 31%, which is unacceptable.

[5]Execution time remains in microseconds in this granularity, whose tendency regarding sparsity is not easily visible.

Algorithm 1: SpMM on CUDA cores

Input: rowPtr, colInd, val and dense matrix X.
Output: The result of SpMM Z.

```
for i = 0 to m − 1 in parallel do
    for j = 0 to n − 1 in parallel do
        res ← 0;
        for k = rowPtr[i] to rowPtr[i + 1] do
            res+ = val[k] ∗ X[colInd[k], j];
        Z[i, j] = res;
```

Algorithm 2: SpMM on Tensor cores

Input: rowPtr, colInd, val and dense matrix X.
Output: The result of SpMM Z.

```
__shared__ ASh[16 ∗ 8], XSh[WarpSize][8 ∗ embDim];
foreach row window in parallel do
    foreach 16 × 8 block in row window do
        Convert CSR into ASh in matrix format;
        Load submatrices of X into XSh;
        Each warp calculates ASh × XSh using Tensor
          cores;
    Store the result into Z;
```

computation overhead. Edges within each row window can be stored consecutively in this granularity, which ensures the access locality of edges. The sufficient duration of execution in the scale of row windows aids in the analysis of appropriate GPU core selection. Another pivotal characteristic in this granularity, the number of non-zero columns, stands out as a reference for better core allocation (§ IV-C).

B. Computing Characteristics of GPU Cores

In this subsection, we identify key features of row windows that impact the GPU cores' performance and quantify the difference in computing characteristics between different cores.

The algorithms presented in Algorithm 1 and 2 highlight the distinct computational strategies employed by CUDA and Tensor cores in SpMM. CUDA cores operate in parallel to compute elements in the result matrix Z, utilizing the CSR format to efficiently skip zeros in A. On the other hand, the process for Tensor cores, adhering to the specifications of the WMMA API, necessitates retrieving a 16×8 submatrix from A and an 8×16 submatrix from X, followed by caching these submatrices in shared memory for subsequent loading and computation using the WMMA API. The computational cost of SpMM on CUDA cores is primarily influenced by the *number of non-zero elements* in the sparse matrix. In contrast, for Tensor cores, the computational cost is tied to the *quantity of* 16×8 *blocks*. To validate these observations, we conducted a series of experiments to evaluate the impact of these two metrics on the performance of GPU cores.

Sparsity of the Input Matrix. The sparsity of a sparse matrix is reflected in the number of zero elements it contains. To explore the relationship between the performance of GPU cores and sparsity, we conducted a series of evaluations following the strategy outlined in § IV-A. We specify the size

TABLE I
COMPUTING AND MEMORY ACCESS COSTS OF GPU CORES IN SPMM[6].

Datasets	C-m	C-c	m/c(C)	T-m	T-c	m/c(T)
DD	5.04	7.13	**0.71**	15.15	11.13	**1.36**
YS	25.09	31.77	**0.79**	48.41	21.14	**2.29**
RD	71.16	82.84	**0.86**	130.44	55.10	**2.37**

of a row window as 16×32 and the dense matrix dimension as 32. Sparse matrices with varying degrees of sparsity were generated for evaluation. The execution times of different GPU cores are visualized in Figure 1(a). Interestingly, the execution times of Tensor cores remained stable as sparsity increased. This can be attributed to the fixed number of 16×8 submatrices, resulting in relatively consistent execution times for lines 4-6 in Algorithm 2. Conversely, the execution times of CUDA cores decrease with higher sparsity, surpassing Tensor cores when sparsity exceeds 83%. This observation suggests a positive correlation between the computational costs of CUDA cores and the number of non-zero elements in sparse matrices.

Non-zero Columns of the Input Matrix. The number of non-zero columns is also a pivotal characteristic of the performance of both GPU cores. To evaluate this characteristic, we maintain a constant level of sparsity while varying the number of non-zero columns in a row window. Figure 1(b) depicts the execution times of different GPU cores under these conditions. We observe distinct behaviors: CUDA cores demonstrate relatively consistent computational costs as the number of non-zero columns rises, while the computational costs of Tensor cores increase. This discrepancy arises because, for CUDA cores, the bottleneck lies in computation. With the constant sparsity, the increase in the number of non-zero columns will not bring about a large increase in calculations, leading to relatively consistent computational costs of CUDA cores. For Tensor cores, they are unable to skip zero elements in adjacent matrices due to the input requirements, thereby triggering more data loading as the number of non-zero columns increases. However, the bottleneck of Tensor cores exactly lies in the loading of X. Experimental results indicate that the loading time for X is about $2\times$ longer than the time spent on multiplication, constituting more than 60% of the total execution time. As a result, the significant increase in Tensor cores' computational costs is attributed to the memory access overhead. As the number of non-zero columns increases, more memory access is required to load X for Tensor cores, resulting in low efficiency.

Other factors such as the distribution of non-zero elements within sparse matrices may also influence the performance of GPU cores. However, their impact is considerably insignificant[7] compared to the aforementioned two characteristics, therefore we choose to disregard them. To further discover the difference between CUDA cores and Tensor cores, we evaluate the computing and memory access costs of both types of GPU cores in SpMM operation, which is reported in Table I. The data loading of CUDA cores is faster than the calculation, while it is exactly the opposite for Tensor cores. As a result, CUDA cores are optimal for tasks that are ***memory access-intensive***, while Tensor cores excel in ***computation-intensive*** tasks. Our identified key characteristics, sparsity and the number of non-zero columns, exactly govern *computation* and *memory access*, respectively. These two characteristics offer sufficient information to select the appropriate cores, resulting in a high accuracy reported in the next paragraph. The row windows with low sparsity and a small number of non-zero columns require more computation overhead and less memory access overhead, which are suitable for Tensor cores. Otherwise, we opt for CUDA cores.

C. Adaptive Core Selection

We train a logistic regression model to determine the appropriate GPU cores for matrix multiplication of a row window, based on the performance of both GPU cores on synthetic sparse matrices with diverse sparsity and non-zero column counts. This model takes the sparsity and the number of non-zero columns as inputs and predicts the appropriate GPU cores. The logistic regression model is lightweight and efficient, allowing for rapid computation. It can complete the selection of GPU cores for a row window in a few nanoseconds, with an accuracy greater than 90%.

The training pipeline of the model consists of 4 procedures: (1) sparse matrices generating, (2) execution results collecting, (3) model training, and (4) model encoding.

Sparse matrices generating. We begin by generating a set of matrices for evaluating GPU cores. The logistics regression model is used to identify the appropriate GPU cores for each row window with 16 rows. Therefore, the number of rows in the generated matrices is set to 16. The number of columns ranges from 1 to 130[8]. Each column contains at least one non-zero element. The matrix sparsity ranges from $\frac{1}{16}$ to $\frac{15}{16}$[9], with the corresponding number of non-zero elements ranging from $\#columns$ to $\#columns \times 15$. To ensure each column contains at least one non-zero element, we first generate an element per column, with the row position determined by a uniform random number function. Subsequently, the row and column positions of the remaining elements are generated randomly using the same function.

Execution results collecting. Using the matrices generated in the previous step, we execute the SpMM kernel on Tensor cores and CUDA cores, respectively. For each matrix, we perform 100 executions to evaluate the average runtime. It's important to note that the kernels used are identical to the deployed SpMM kernel, with the same parameter settings.

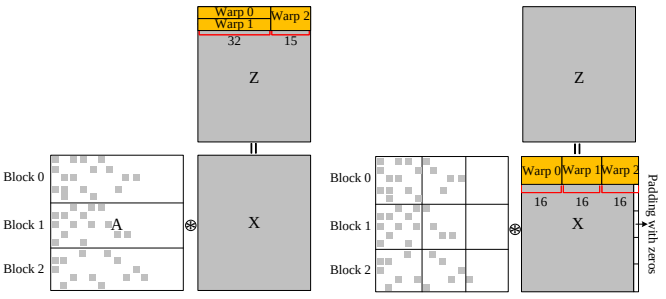
Model training. We use the logistics regression model API from Sklearn. Each matrix in the previous step serves as a training sample characterized by two features: sparsity and

[6]C: CUDA core, T: Tensor core, m: memory access cost, c: computing cost, m/c: the quotient of memory access and computing cost, Units: 10^{-2} ms.

[7]The variation of the execution time is less than 10%.

[8]The maximum number of non-zero columns is set to 130 because the average degree of most graphs is less than 8. If all nodes within a row window have no common neighbors, the number of non-zero columns is $16\times8 = 128$. Setting the maximum at 130 accommodates most cases.

[9]The maximum sparsity is set to $\frac{15}{16}$ to ensure each column has at least one non-zero element. The minimum sparsity is $\frac{1}{16}$ because the observed sparsity of most row windows is greater than 0.06. Lower minimum sparsity is optional for higher accuracy according to user needs.



(a) Warp allocation for CUDA cores (b) Warp allocation for Tensor cores

Fig. 5. Warp allocation strategies on different GPU cores.

the number of non-zero columns. For each sample, we have measured the execution times on Tensor and CUDA cores, respectively. If the execution time of CUDA cores is shorter than that of Tensor cores, the sample is labeled as 1; otherwise, it's labeled as 0. Then we train the model until it converges.

Model encoding. Finally, we extract the coefficients from the logistics regression model and hard-code them into the cores selection function. The classification is performed during the preprocessing step, with the classification results for each row window stored in a boolean array. In this array, 0 represents the selection of CUDA cores and 1 denotes Tensor cores. During SpMM computation, the kernel refers to the boolean array to assign the appropriate GPU cores for calculation.

The model is trained offline using synthetic data. Once trained, it can be reused for inference without retraining, provided the GPU architecture and precision remain unchanged. For a new dataset (sparse matrix), the classification overhead is minimal, as it only involves computing $w_1 * x_1 + w_2 * x_2 + b$, which can be completed within a few nanoseconds.

D. Kernel Optimizations

In this subsection, we further optimize the SpMM kernel considering the thread collaboration mode and memory access pattern to achieve higher efficiency.

1) Optimizations on CUDA Cores

In the SpMM kernel designed for CUDA cores, we utilize multiple warps to concurrently compute the results of a row window. For simplicity, let's assume a dense matrix dimension of 32, which we will generalize in the subsequent discussion. Each warp within a thread block is allocated a row from the sparse matrix. Within each warp, each thread is responsible for computing one element in the result matrix Z. This approach guarantees coalesced memory access when retrieving elements from the dense matrix stored in global memory.

Generalization. In scenarios where the dense matrix dimension is not a multiple of 32, using a warp to compute one row may lead to inefficient thread utilization. For instance, with a dimension of 47, a warp requires two iterations over a row, leaving 17 idle threads during the second iteration. To address this inefficiency, we develop an adaptive kernel that can handle varying dimensions. This kernel utilizes different numbers of threads to calculate a row, such as 16 or 8. When employing 16 threads, for example, a warp is utilized to compute 2 rows of the result matrix, as depicted in Figure 5(a).

Algorithm 3: Optimized SpMM on CUDA cores

Input: rowPtr, colInd, val and dense matrix X.
Output: The result of SpMM Z.

```
// Memory Management
1  __shared__ TmpEdges[MaxSize],TmpVals[MaxSize];
2  for i = 0 to m − 1 in parallel do
3      for j = rowPtr[i] to rowPtr[i + 1] in parallel do
4          TmpEdges[j] ← colIdx[j];
5          TmpVals[j] ← val[j];
   // Generalization
6  AlignN ← dim/32;
7  for r = 0 to AlignN do
8      for i = 0 to m − 1 in parallel do
9          for j = 0 to n − 1 in parallel do
10             res ← 0;
11             for k = rowPtr[i] to rowPtr[i + 1] do
12                 res+ = TmpVals[k]∗X[TmpEdges[k], j];
13             Z[i, j] = res;
   // Process the unaligned dimension of X
14 for (i = 0; i < m − 1; i+ = 2) in parallel do
15     for j = AlignN ∗ 32 to dim in parallel do
16         res ← 0;
17         for k = rowPtr[i] to rowPtr[i + 1] do
18             res+ = TmpVals[k] ∗ X[TmpEdges[k], j];
19         Z[i, j] = res;
20 return Z;
```

Memory Management. Optimizing memory access in the SpMM kernel on CUDA cores commonly involves storing the dense submatrix in shared memory due to its frequent usage. However, our observations indicate that loading either all data or only the frequently accessed ("hot") data into shared memory is not as efficient as directly accessing it from global memory for GNN-tailored SpMM[10], even with vectorized data loading techniques. This inefficiency stems from modern GPUs' efficient cache management mechanisms, which can effectively handle such scenarios. Therefore, we opt to load only the edge indices in CSR into shared memory. Threads within a warp accessing identical addresses when calculating results can trigger a broadcast mechanism in global memory, leading to time-consuming operations. By loading this data into shared memory, we circumvent this overhead.

In Algorithm 3, we provide the implementation details of the optimized SpMM kernel on CUDA cores. The memory management strategy is implemented in lines 1-5. We employ all threads to load the $colIdx$ and val to shared memory in parallel. Lines 6-19 implement the strategy to handle general dense matrix dimensions besides 32. dim in line 6 denotes the dimension of the dense matrix X. In lines 7-13, we first calculate the parts that can align with 32 like the processing parts of warp 0 and warp 1 in Figure 5. The remaining parts are calculated in lines 14-19, with each warp processing 2 rows like warp 2 in Figure 5.

[10]Loading all the data including the sparse and dense matrices will result in a performance loss of up to 9%, while only loading hot data has the similar performance to directly accessing it from the global memory.

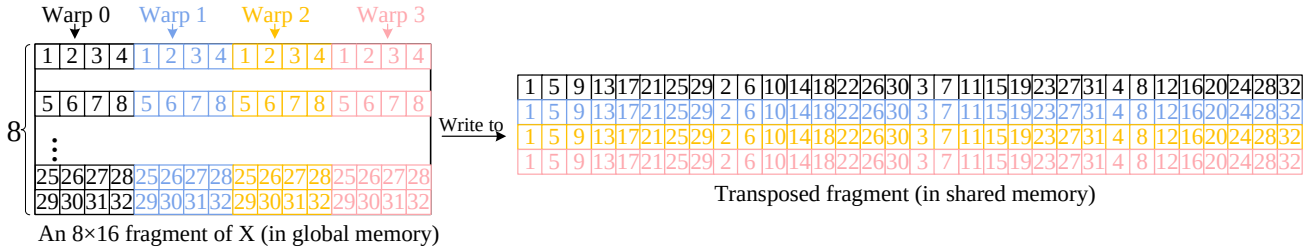


Fig. 6. Data loading strategy for the dense matrix X. The 8×16 submatrix is transposed during loaded into shared memory.

2) Optimizations on Tensor Cores

To enhance data loading efficiency for SpMM on Tensor cores, we address the bottleneck of loading data from the dense matrix by employing all warps within a block to cooperatively load data into shared memory. This strategy allows data from the dense matrix to be loaded in units of 8×16 matrices, aligning with the input constraints of the WMMA API.

Specifically, as illustrated in Figure 6, rather than retrieving 2 or 1 row, a warp is utilized to fetch 8 rows of an 8×16 matrix[11], with each row containing 4 elements for a warp. This approach ensures that threads within the warp can write these retrieved elements into distinct banks of shared memory, thereby preventing bank conflicts and eliminating extra overhead. In shared memory, the data is organized in a one-dimension format. However, for visualization purposes and space limitations, we display the data in a 4×32 layout.

Additionally, we adopt the parallel strategy for Tensor cores presented in [43], which is illustrated in Figure 5(b). Following collaboratively loading the required data from X by warps, each warp is assigned an 8×16 submatrix to independently compute the results using Tensor cores concurrently. With each block responsible for calculating a row window, this method effectively harnesses the parallel processing capabilities of GPU and enhances computational efficiency.

In lines 6-10 of Algorithm 4, we present the implementation of data loading strategy in Figure 6. It's worth noting that the parallel data loading strategy is designed for the retrieval of the dense matrix. For loading CSR entries, we follow the strategy in TC-GNN. Line 6 adopts the parallel strategy depicted in Figure 6, with each warp performing the data loading of 4 elements within each row. It calculates the index of data to load in X, denoted as $srcIdx$ and the index to write, denoted by $dstIdx$ in lines 8-9. Data is loaded to XSh according to the calculated indices in line 10. More details can be found in our source code.

Discussion. The distinction between HC-SpMM and existing SpMM methods is twofold. First, existing techniques solely rely on CUDA or Tensor cores for calculation, which leads to suboptimal performance. In contrast, HC-SpMM is an SpMM method specifically designed for the irregular distribution of graphs, adaptively selecting the proper cores according to the characteristics of submatrices, which achieves better performance. Second, for both CUDA and Tensor cores, data loading is one of the bottlenecks. HC-SpMM optimizes the data loading for CUDA and Tensor cores by reasonably utilizing shared memory and multi-thread, which are not considered in existing methods for Tensor cores.

[11]For Tensor cores, we split the matrix into 8×16 submatrices according to the input requirement of WMMA API.

Algorithm 4: Optimized SpMM on Tensor cores

Input: rowPtr, colIdx, rowIdx, val and dense matrix X.
Output: The result of SpMM Z.

1 $\_\_shared\_\_$ $ASh[16*8], XSh[WarpSize][8*embDim]$;
2 **foreach** *row window in parallel* **do**
3 **foreach** 16×8 *block in row window* **do**
4 **for** $i = rowPtr[block*16]$ *to* $rowPtr[(block+1)*16]$ *in parallel* **do**
5 $ASh[colIdx[i]\%8 + (rowIdx[i]\%16)*8] \leftarrow val[i]$;
6 **for** $idx = warpId; idx < dim/4; idx += warpNum$ **do**
7 $denseRowId \leftarrow$calculate the row idx;
8 $srcIdx \leftarrow denseRowId*dim + threadIdx\%4 + idx*4$;
9 $dstIdx \leftarrow warpId*32 + (threadIdx\%4)*8 + (threadIdx/4)$;
10 $XSh[dstIdx] \leftarrow X[srcIdx]$;
11 Each warp calculates $ASh \times XSh$ using Tensor cores;
12 Store the result into Z;
13 **return** Z;

V. SpMM Integration into GNN Training

In this section, we integrate HC-SpMM into the GNN training pipeline and propose a series of optimization techniques from system and graph data management perspectives to enhance the GNN training efficiency.

A. Kernel Fusion

To integrate HC-SpMM into the GNN training pipeline and achieve higher efficiency, we rethink the training pipeline and propose a novel kernel fusion strategy. Existing implementations, such as DGL [39] and TC-GNN [43], typically treat the *Aggregation* and *Update* phases independently, leading to increased kernel launch overhead and additional memory access overhead. During the *Aggregation* phase, results are written to global memory; then, during the *Update* phase, they are read from the same address of global memory. Moreover, the separate launching of the *Update* and *Aggregation* kernels contributes to significant time overhead[12].

[12]The launch time of a single matrix multiplication kernel is measured to be around 0.03ms.

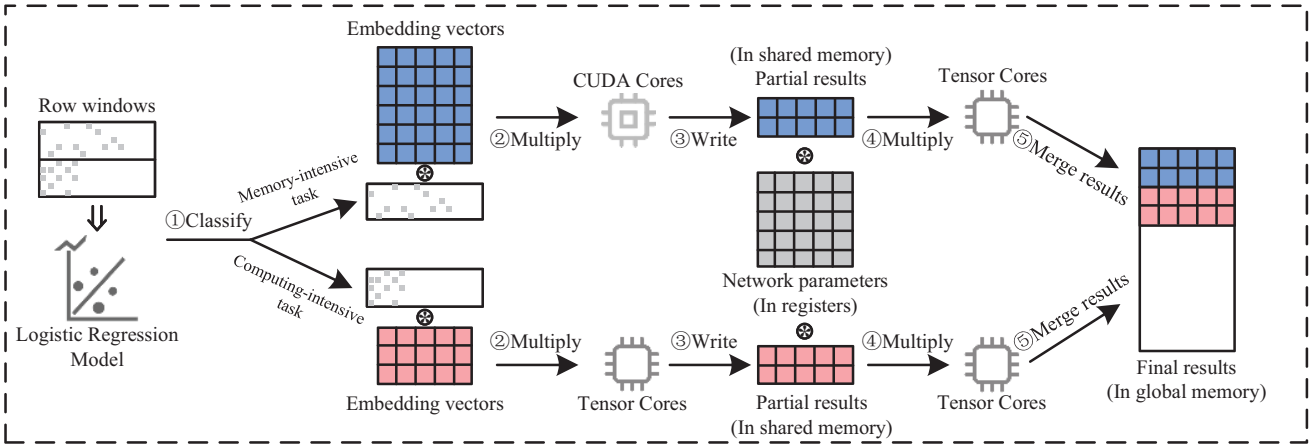


Fig. 7. Process in a GNN layer with kernel fusion.

Upon analyzing the allocation of row windows to thread blocks, we recognize an opportunity to optimize performance by fusing the *Update* and *Aggregation* kernels. Kernel fusion proves particularly effective when the *Update* phase directly follows the *Aggregation* phase, as seen in the backward propagation of GCN [21] and the forward propagation of GIN [49]. In such scenarios, the data required for the *Aggregation* phase is readily available and consistent, facilitating efficient fusion. Otherwise, implementing kernel fusion becomes complex due to the varying and partially overlapping embedding vectors needed for each row window, making it challenging to determine and calculate the precise embedding vectors required for the *Aggregation* phase during the *Update* phase. Fortunately, during the forward and backward propagation stages, there is always one stage where the *Update* phase follows the *Aggregation* phase, enabling effective kernel fusion.

As depicted in Figure 7, in forward propagation with kernel fusion, the row windows within the matrix are ①classified by the logistic regression model and distributed to the corresponding GPU cores. The GPU cores ②multiply the row windows with the corresponding embedding vectors and ③write the intermediate results into shared memory. Following this, Tensor cores ④multiply the results stored in shared memory with the network parameters used during *Update* phase, and ⑤store the final results in the global memory. During backward propagation, the process is similar, with the addition of computing $W'^{(k)}$ in Equation 3. To calculate $W'^{(k)} = Z^T X'^{(k+1)}$, the submatrix result in shared memory is transposed first, as $X'^{(k+1)}$ is the right-hand side of the equation. Subsequently, Tensor cores compute the final results. The kernel fusion approach consolidates the number of kernels from two to one, thereby reducing the kernel launch overhead. Additionally, storing results in shared memory rather than global memory significantly alleviates memory access overhead.

B. Layout Optimization

As discussed in § IV-C, CUDA cores are ideally suited for memory access-intensive row windows, while Tensor cores excel in computing-intensive row ones. Real-world graphs typically exhibit row windows with numerous non-zero columns

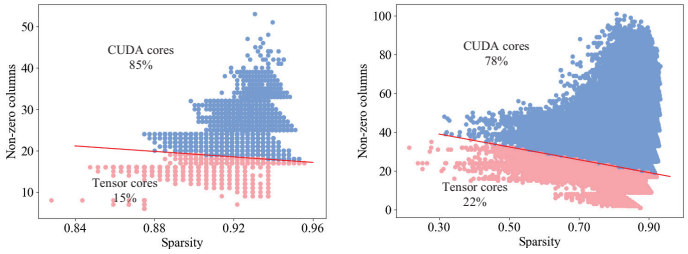


Fig. 8. Sparsity of row windows in representative graphs.

but few non-zero elements in each of them. To quantify this, we compute the sparsity and number of non-zero columns for row windows within two representative datasets, as depicted in Figure 8. The red line denotes the classification boundary learned by the logistic regression model. Only 15% and 22% of row windows are deemed appropriate for Tensor cores in the two datasets, respectively. Given that Tensor cores are introduced to efficiently handle computations, which CUDA cores may struggle with, a low proportion of suitable row windows restricts the potential for acceleration. In response, we propose a novel algorithm named LOA in this section to reformat the graph layout, increasing density and obtaining more row windows conducive to Tensor cores.

Our objective is to reconstruct each row window to enhance its computing intensity. For each row window, we define the *computing intensity* in Equation 5.

$$computing\ intensity = \frac{\#nonzero\ elements}{\#nonzero\ columns} \tag{5}$$

A higher *computing intensity* indicates a denser layout of row windows, making them more suitable for Tensor cores, as illustrated in Figure 1 and § IV-C. Based on this objective, we devise an efficient greedy strategy as below.

Our basic algorithm is to iteratively select an initial vertex for each row window and incrementally expand it by adding the vertex that maximizes the *computing intensity* in conjunction with the existing vertices in the row window. To mitigate the computational overhead of identifying the vertex with the maximum *computing intensity* among all vertices, we introduce a vertices window VW to confine the search scope. As outlined in Algorithm 5, we sort all vertices based on their neighbors' smallest index (line 1). $N(v)$ in line 1

Algorithm 5: Layout reformat

Input: Graph $G = (V, E)$, vertices window VW.
Output: Graph G' after layout optimization.

1 $NRW \leftarrow \emptyset$;
2 $soList \leftarrow$ sort $v \in V$ by $min(N(v))$;
3 **foreach** *row window* **do**
4 $\quad RW \leftarrow$ the first vertex in $soList$ hasn't been visited;
5 $\quad$ **for** $i = 1$ *to* 15 **do**
6 $\quad\quad CanVtxs \leftarrow$ vertices with the highest *computing intensity* within VW;
7 $\quad\quad v \leftarrow$ vertex with the highest degree in $CanVtxs$;
8 $\quad\quad RW \leftarrow RW \cup \{v\}$;
9 $\quad NRW \leftarrow NRW.append(RW)$;
10 Reorder G using NRW;

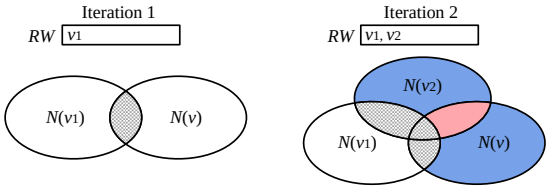


Fig. 9. An example of avoiding redundant computing.

represents the set of neighbors for vertex v. For each row window, we select the first unvisited vertex from the sorted list as the initial vertex and add it to the set row window RW (line 4). Subsequently, we compute the *computing intensity* when considering adding each vertex to the row window according to Equation 5. The loop in line 5 is executed 15 times because the row window is fixed to 16. Then, we add the vertex that maximizes the *computing intensity* to RW and proceed to the next iteration. Multiple vertices may yield the highest *computing intensity*. To prioritize low sparsity, we select the vertex with the highest degree to add to RW (lines 7-8).

Efficiency Optimization. To address the computational bottleneck in line 6, where VW vertices are added to the current row window and their corresponding *computing intensity* is calculated, we need to optimize the computation of $\#nonzero\ columns$. The most time-consuming aspect of this process is the frequent set union operations. Suppose RW denotes the vertices in a row window. The number of non-zero columns within this row window can be computed as $\#nonzero\ columns = |\cup_{v \in RW}\ N(v)|$. However, this direct method results in significant redundancy, especially in the 15 iterations in lines 5-8 of Algorithm 5. For instance, as depicted in Figure 9, the union of $N(v_1)$ and $N(v)$ in the first iteration has been calculated, where $v_1 \in RW$ and v is a vertex from the range window. In the second iteration, we proceed to compute $N(v_1) \cup N(v_2) \cup N(v)$. However, we have already computed $N(v_1) \cup N(v)$ and $N(v_2) \cup N(v)$, resulting in redundant computation if calculated using a brute-force approach. Instead, we only need to calculate the union of the blue parts, namely $(N(v_2) - N(v_1)) \cup (N(v) - N(v_1))$. Similarly, we can also avoid redundant computations in subsequent iterations. In general, in the i-th iteration, assuming the appended vertex is v_i from the previous iteration, the calculation can be formulated as $N(v) \cup N(v_i) - \cup_{u \in RW \setminus \{v_i\}} N(u)$.

Algorithm 6: LOA

Input: Graph $G = (V, E)$, vertices window VW.
Output: Reconstructed row windows NRW.

1 $NRW \leftarrow \emptyset$;
2 $soList \leftarrow$ sort $v \in V$ by $min(N(v))$;
3 **foreach** *row window* **do**
4 $\quad RW \leftarrow$ the first vertex v_0 in $soList$ hasn't been visited;
5 $\quad Resi \leftarrow N(v_0), allCols \leftarrow N(v_0)$;
6 $\quad$ **for** $i = 1$ *to* 15 **do**
7 $\quad\quad$ **foreach** $u \in Resi$ **do**
8 $\quad\quad\quad$ **foreach** $w \in N(u)$ **do**
9 $\quad\quad\quad\quad w.cns$++;
10 $\quad\quad$ **for** $j = v_0.id$ *to* $(v_0.id + VW)$ **do**
11 $\quad\quad\quad v \leftarrow soList[j]$;
12 $\quad\quad\quad P = \frac{curEles + |N(v)|}{curCols + |N(v)| - v.cns}$;
13 $\quad\quad\quad$ **if** $P > maxP$ **then**
14 $\quad\quad\quad\quad maxP \leftarrow P, v_{max} \leftarrow v$;
15 $\quad\quad RW \leftarrow RW \cup \{v_{max}\}$;
16 $\quad\quad Resi \leftarrow N(v_{max}) - allCols$;
17 $\quad\quad allCols \leftarrow allCols \cup N(v_{max})$;
18 $\quad\quad curEles \leftarrow curEles + |N(v_{max})|$;
19 $\quad\quad curCols \leftarrow |allCols|$;
20 $\quad NRW \leftarrow NRW.append(RW)$;
21 **return** NRW

In addition, we calculate the intersection instead of the union result. This enables efficient calculation by traversing the vertices in $N(N(v_i))$. Algorithm 6 illustrates the optimized layout reformatting algorithm. In each iteration, LOA calculates the intersection of each vertex with the vertices in the current row window in lines 7-9, i.e., $|N(v) \cap (\cup_{u \in RW} N(u))|$. In lines 10-14, LOA traverses the vertices in the range window and calculates the *computation intensity* by Equation 6.

$$\frac{|N(v)| + \Sigma_{u \in RW} |N(u)|}{|N(v)| + |\cup_{u \in RW}\ N(u))| - |N(v) \cap (\cup_{u \in RW} N(u))|} \tag{6}$$

Subsequently, the vertex with the highest *computation intensity* is appended to row window (line 15). At the end of each iteration, the intermediate results are updated in lines 16-19.

VI. Experiments

In this section, we evaluate HC-SpMM and conduct a comparative evaluation with state-of-the-art SpMM kernels as well as GNN training frameworks. More information on the experiments is provided in Appendix A.

A. Experimental Setup

Datasets. In order to fully evaluate the proposed SpMM kernel, we run experiments on 13 datasets, some of which have been used in previous related work [11], [12], [39], [43]. All the datasets are accessible at SNAP[13], TUDataset[14] and KONECT[15]. Table II summarizes the properties of the 13

[13] snap.stanford.edu/data/index.html
[14] chrsmrrs.github.io/datasets/docs/datasets
[15] konect.cc/networks

TABLE II
DETAILS OF DATASETS.

Datasets	#Vertex	#Edges	Dim
Citeseer (*CS*)	3,327	9,464	3,703
Cora (*CR*)	2,708	10,858	1,433
Pubmed (*PM*)	19,717	88,676	500
PROTEINS (*PT*)	43,471	162,088	29
DD (*DD*)	334,925	1,686,092	89
Amazon (*AZ*)	410,236	3,356,824	96
Yeast (*YS*)	1,710,902	3,636,546	74
OVCAR (*OC*)	1,889,542	3,946,402	66
Github (*GH*)	1,448,038	5,971,562	64
YeastH (*YH*)	3,138,114	6,487,230	75
Reddit (*RD*)	4,859,280	10,149,830	96
Twitch (*TT*)	3,771,081	22,011,034	96
Depedia (*DP*)	18,268,981	172,183,984	96

datasets. Most datasets do not have information on the number of classes, so we uniformly use 22.

Baselines. We compare HC-SpMM with existing methods in two aspects: SpMM and GNN forward/backward propagation.

1) SpMM kernels: Sputnik [13] is a library developed by Google, which provides the state-of-the-art unstructured SpMM kernel for full precision on CUDA cores. GE-SpMM [20] is another efficient SpMM kernel on CUDA cores specifically designed for SpMM operation in GNN. cuSPARSE [28] is a CUDA sparse matrix library developed by Nvidia, which contains a set of high-performance SpMM kernels. In our experiments, we employ the SpMM kernel taking the CSR format as its input. TC-GNN [43] is a method to use Tensor cores to accelerate the full-precision unstructured SpMM operation in GNNs. DTC-SpMM [11] is the state-of-the-art SpMM kernel using Tensor cores. Notably, using CUDA cores or Tensor cores alone for SpMM cannot achieve comparable efficiency to the methods in the experiment.

2) GNN training: We conduct end-to-end experiments and compare the frameworks targeting the optimization of SpMM to demonstrate the ability of HC-SpMM to handle complex graph computing tasks. Due to the GNN algorithm remaining unchanged, the training results of these frameworks are identical. Other frameworks, such as [42], [59], are not considered in experiments, which are orthogonal to our work. GE-SpMM and TC-GNN integrate the optimized SpMM kernels into PyTorch, which are the state-of-the-art works for accelerating GNN by optimizing SpMM on CUDA cores and Tensor cores, respectively. As the efficiency of GE-SpMM and TC-GNN is much more significant[16] than PyG and DGL, we do not evaluate these two GNN training frameworks in experiments.

Platforms. All experiments are conducted on a CentOS 7 server, featuring an Intel Core i9-10900K CPU and an Nvidia RTX 3090 GPU. The GPU has 82 SMs, 10,496 CUDA cores, and 328 Tensor cores. We implement HC-SpMM[17] in C++ under Nvidia CUDA 12.2 and integrate it into PyTorch 1.8.

B. Evaluations on SpMM kernel

In this subsection, we compare HC-SpMM with 4 state-of-the-art SpMM kernels, and conduct comprehensive experiments to evaluate our proposed optimization techniques.

1) Comparison with SpMM kernels

Figure 10 presents the overall performance of various SpMM kernels across 13 datasets, using cuSPARSE as the baseline. Execution durations are measured employing nvprof. We report the SpMM kernel time and exclude the preprocessing and PCIe transfers overhead. The data transferred through PCIe includes only the dense matrix and the CSR format of the sparse matrix. The preprocessing step prepares the data used for Tensor cores and classifies row windows to appropriate cores using GPU. It is 13.0× of a single SpMM execution in HC-SpMM on average, which is negligible in many real-world scenarios that require thousands of SpMM operations such as GNN. Further discussions about the preprocessing are detailed in Appendix F. The caches are cold when performing SpMM kernels. The absolute numbers of the execution time are detailed in Appendix A. Our results show that HC-SpMM consistently outperforms all compared methods across all datasets. HC-SpMM achieves speed improvements of 1.85-18.89× over cuSPARSE, 1.07-1.57× over Sputnik, 1.05-1.57× over GE-SpMM, 1.30-6.76× over TC-GNN and 0.99-3.03× over DTC-SpMM. Notably, in the best cases, HC-SpMM achieves a remarkable speedup of 19.56× over cuSPARSE on *DP*, 1.57× speedup over Sputnik on *TT*, 1.57× speedup over GE-SpMM on *RD*, 6.76× and 3.00× speedup over TC-GNN and DTC-SpMM on *PM*. We also compare HC-SpMM with the SpMM kernel in PyTorch executed on CPU, which achieves an average speedup of 183.77×. These results underscore the effectiveness of leveraging hybrid cores to perform SpMM. HC-SpMM demonstrates its capability to intelligently identify submatrices suitable for CUDA and Tensor cores to fully utilize the characteristics of different hardware structures and maximize the performance of the GPU. Please note that HC-SpMM don't need to merge the results because of the combination strategy proposed in § IV-A even if we employ two cores for simultaneous calculation.

The speedup ratios of HC-SpMM vary across different datasets when contrasted against SOTA methods. Compared with Sputnik and GE-SpMM which employ CUDA cores for computation, the improvements of HC-SpMM on *CS*, *CR* and *PM* are less pronounced than on other datasets. As detailed in Table II, this discrepancy can be attributed to the density of the adjacency matrices. Datasets such as *CS*, *CR* and *PM*, characterized by relatively few vertices and edges, possess limited dense parts in the adjacency matrices. Consequently, the efficiency of Tensor cores, which excel at dense matrix multiplication, is not fully leveraged. This conclusion is further corroborated by the performance of TC-GNN and DTC-SpMM, which use Tensor cores for SpMM on *CS*, *CR*, and *PM*. On these three datasets, the speedup ratio of TC-GNN and DTC-SpMM is considerably behind Sputnik and GE-SpMM, which is because the potential of Tensor cores in dense parts cannot be fully realized on small-scale matrices. Conversely, HC-SpMM, Sputnik and GE-SpMM demonstrate remarkable speedups on datasets like *AZ* and *DP*. Analysis of these datasets reveals that the adjacency lists of vertices exhibit poor

[16]TC-GNN achieves 1.7× speedup on average over DGL, while GE-SpMM brings up to 3.7× speedup over PyG.

[17]https://github.com/ZJU-DAILY/HC- SpMM

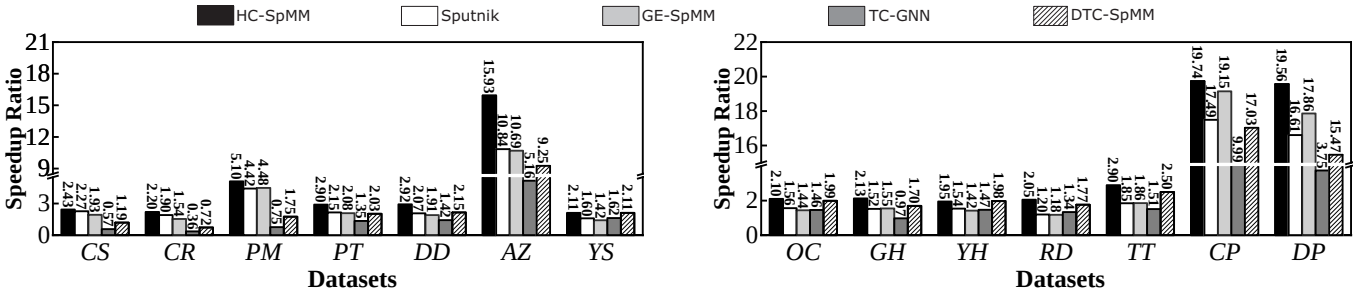


Fig. 10. Overall performance of SpMM kernels.

TABLE III
EFFECTIVENESS OF GENERALIZATION.

Datasets	Generalization	No optimization	Speedup
DD	0.398ms	0.498ms	25.1%
YS	1.456ms	1.593ms	9.4%
OC	1.576ms	1.869ms	18.6%
YH	3.205ms	3.912ms	22.1%

locality, with neighbor IDs scattered rather than compactly distributed as in other datasets. This scattered distribution leads to inefficient memory access for cuSPARSE.

We also evaluate the performance of the five kernels in various sparsity to further demonstrate the necessity and effectiveness of using hybrid GPU cores for SpMM. The results are reported in Appendix D. We also evaluate the adaptability of the regression model, the sensitivity of performance to specific parameters, and the utilization metrics of GPU cores, which are discussed in Appendix A&B, E, and H, respectively.

2) Effectiveness of the optimization techniques

Next, we evaluate the three optimization techniques respectively to demonstrate their effectiveness.

Optimizations on CUDA cores. **(i)** *Generalization.* We evaluate the execution time of kernels using the generalization technique as well as without using this optimization. The datasets used for this evaluation are those featuring an unaligned embedding dimension (not a multiple of 32) in Table II. Table III illustrates the results. On average, the generalization technique achieves 18.8% time savings across tested datasets, which demonstrates that the generalization of SpMM on CUDA cores saves threads in each warp when the dimension is not a multiple of 32. **(ii)** *Memory Management.* In § IV-D1, shared memory is used to store the column indices of CSR. We evaluate the effectiveness of this technique and report the results in Table IV. The usage of shared memory achieves an average speedup of 2.85%. On *YS*, this strategy has the most significant improvement. This is because *YS* has more row windows suitable for CUDA cores before performing LOA. In addition, *YS* has a low average degree which leads to less shared memory usage, thus increasing the number of warps that can be concurrently scheduled by GPU.

Optimizations on Tensor Cores. Finally, we conduct experiments to validate the effectiveness of data loading. In this evaluation, we only record the calculation time of Tensor cores. Table V presents the results. Our proposed data loading strategy leads to an average speedup of 17.50%. This improvement is attributed to the increased participation of thread warps in data loading and reduced bank conflicts in shared memory.

TABLE IV
EFFECTIVENESS OF SHARED MEMORY USING STRATEGY.

Datasets	Shared memory	No optimization	Speedup
YS	0.581ms	0.603ms	3.79%
OC	0.625ms	0.639ms	2.24%
YH	1.046ms	1.072ms	2.49%
RD	1.575ms	1.614ms	2.48%
TT	1.384ms	1.429ms	3.25%

TABLE V
EFFECTIVENESS OF DATA LOADING STRATEGY.

Datasets	Opt. data loading	No optimization	Speedup
YS	0.387ms	0.456ms	17.83%
OC	0.330ms	0.386ms	16.97%
YH	0.552ms	0.663ms	20.11%
RD	0.880ms	1.006ms	14.32%
TT	0.678ms	0.802ms	18.29%

It is worth noting that although our data loading strategy enhances performance, data loading remains the bottleneck of SpMM on Tensor cores and requires further exploration.

C. Evaluations on GNN training

In this subsection, with the support of HC-SpMM, we compare the GNN training efficiency with 2 methods, which also focus on accelerating the SpMM operation in GNN training. Subsequently, we evaluate the performance of kernel fusion strategy and LOA algorithm.

1) Comparison with GNN training frameworks

Although our proposed SpMM kernel accelerated by hybrid GPU cores can be used for general sparse matrix-matrix multiplication, our primary objective is to utilize this kernel to enhance the efficiency of graph computing. To demonstrate the efficiency, we integrate our SpMM kernel into PyTorch and compare its performance with two efficient GNN training frameworks. Specifically, We implement GCN [21] and GIN [49] as benchmarking models. It's worth noting that the backward propagation of GCN and forward propagation of GIN employs a unique kernel fusion technique distinct from the forward(backward) propagation. Consequently, we separately report the performance of these two stages within end-to-end evaluations. Due to the large memory requirements of the dataset *DP*, an out-of-memory error occurs when training GNN with all three frameworks. So we omit to report the results of *DP*. Additional experiments in Appendix G suggest that the memory usage of HC-SpMM is only up to 2% and 6% more than GE-SpMM and TC-GNN, respectively. All the reported time below is the average execution time of one epoch unless noted otherwise.

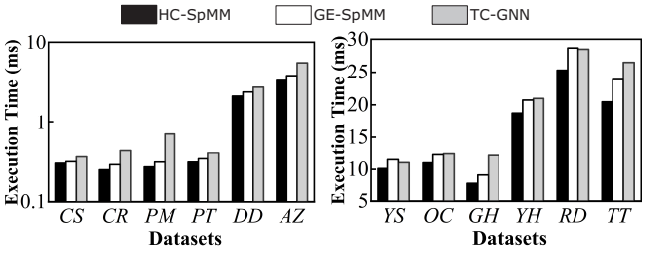


Fig. 11. Comparison of GCN forward propagation.

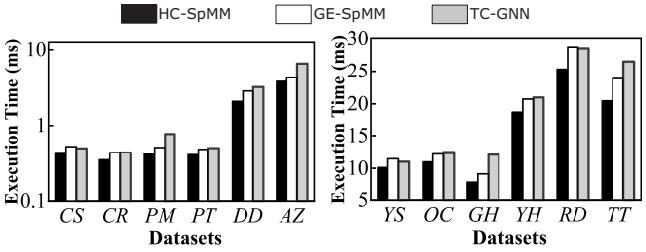
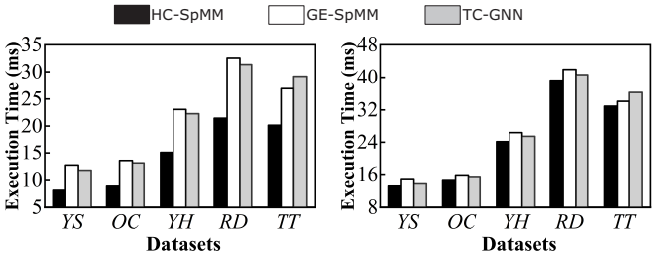


Fig. 12. Comparison of GCN backward propagation.

Figure 11 illustrates the performance of GNN frameworks in forward propagation of GCN. Overall, HC-SpMM outperforms GE-SpMM and TC-GNN across all datasets. Specifically, HC-SpMM achieves an average speedup of 1.42× over TC-GNN and 1.12× over GE-SpMM. Forward propagation comprises two operations: sparse matrix-matrix multiplication and dense matrix-matrix multiplication, corresponding to the *Aggregation* and *Update* phases respectively. Since our optimization primarily targets the *Aggregation* phase, the performance of HC-SpMM in forward propagation is similar to that of SpMM in § VI-B. The acceleration of HC-SpMM becomes more pronounced for large graphs, such as *GH*, *RD* and *TT*. As previously discussed, there are more dense parts in large graphs and Tensor cores are efficient on the dense parts.

Figure 12 illustrates the performance of backward propagation of GCN among the three methods. It is evident that HC-SpMM demonstrates the best performance in all instances. In general, HC-SpMM achieves a 1.48× speedup over TC-GNN and a 1.33× speedup over GE-SpMM on average. Notably, HC-SpMM exhibits a higher speedup ratio during backward propagation compared with forward propagation. The enhanced acceleration is attributed to both the hybrid SpMM kernel and the sophisticated kernel fusion technique. HC-SpMM showcases superior performance on large datasets such as *YS*, *OC*, *GH*, *YH*, *RD* and *TT*. On these 6 datasets, HC-SpMM consistently surpasses GE-SpMM by at least 1.4×. In addition, the most significant speedup cases of HC-SpMM compared to TC-GNN occur on *AZ* and *GH*, where the speedup ratio reaches at least 1.7×. The reason is that *AZ* and *GH* possess less dense row windows. The utilization of Tensor cores alone for SpMM calculation leads to suboptimal performance in such scenarios. These results further demonstrate the necessity and effectiveness of our hybrid strategy.

Regarding GIN, Figure 13 depicts the results. On average, HC-SpMM achieves a speedup of 1.49× and 1.08× over GE-SpMM in forward and backward propagation, respectively. Compared with TC-GNN, it achieves 1.46× and 1.06×



(a) Forward propagation of GIN (b) Backward propagation of GIN

Fig. 13. Comparison of GIN propagation.

TABLE VI
EFFECTIVENESS OF KERNEL FUSION METHOD.

Datasets	Fusing kernel	No optimization	Speedup
YS	9.24ms	12.20ms	32.03%
OC	10.12ms	13.36ms	32.02%
YH	16.82ms	22.05ms	31.09%
RD	26.46ms	34.76ms	31.37%
TT	21.94ms	27.74ms	26.44%

speedup in the forward and backward propagation on average. The performance is similar to GCN, hence warranting no further in-depth elucidation.

The state-of-the-art performance of HC-SpMM demonstrates its ability to handle complex graph computing tasks.

2) Evaluation of kernel fusion strategy

We assess the execution time of a single GNN layer during backward propagation with and without kernel fusion. The results are illustrated in Table VI. The kernel fusion technique achieves an average speedup of 30.6% over the 5 representative datasets. The performance of kernel fusion is stable on different datasets, which demonstrates the efficacy of the technique in mitigating the overhead associated with memory access and kernel launch. As mentioned in § IV-D2, the bottleneck in matrix multiplication using Tensor cores is global memory access. By directly storing the results of *Aggregation* phase into shared memory and fetching them in *Update* phase, the memory access overhead can be significantly reduced.

3) Layout optimization

The improvements brought by LOA are depicted in Figure 14. LOA achieves an average 8.40% performance improvement across the 11 datasets, with the exception of *GH* and *DP*. LOA can achieve a maximum performance improvement of 36.3%. The remarkable improvement on *AZ* is attributed to two factors. Firstly, the original layout of *AZ* is suboptimal, with vertex IDs in the adjacency list being scattered, leading to poor data locality. Secondly, each row window in the original *AZ* is relatively sparse, with only one row window deemed suitable for Tensor cores according to the logistic regression model. LOA adjusts the original layout heuristically, densifying it to allow for the efficient computation of more row windows by Tensor cores. In contrast, the impact of LOA is sometimes not pronounced on some datasets, such as *GH* and *DP*, as these datasets already have favorable original layouts.

We then conduct a thorough analysis of the effectiveness of LOA. We quantify the number of row windows calculated by Tensor cores and CUDA cores before and after employing LOA. An increase in the count of row windows suitable

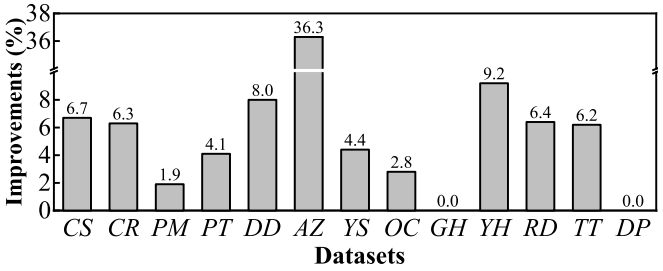
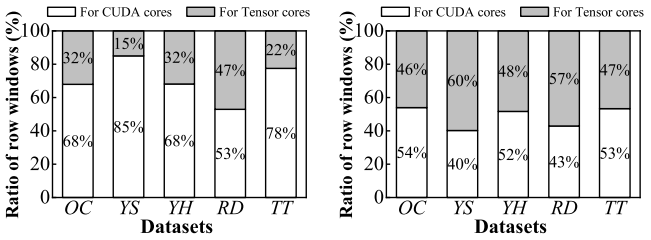


Fig. 14. Improvements of layout optimization.



(a) Before performing LOA (b) After performing LOA

Fig. 15. Effectiveness of LOA.

for Tensor cores suggests that the calculation time using Tensor cores is shorter than using CUDA cores for more row windows, potentially resulting in more improvements relative to solely using CUDA cores. The results are presented in Figure 15. It is evident that across most datasets before using LOA, there are significantly fewer row windows suitable for Tensor cores compared to CUDA cores. This phenomenon arises from the inherently sparse nature of the original graph layout. We can improve the graph layout with little overhead by leveraging LOA. As shown in Figure 15(b), LOA increases the number of row windows suitable for Tensor cores, leading to more efficient calculations.

As a preprocessing algorithm, LOA necessitates that its time cost be maintained within a tolerable threshold. Figure 16 compares the overhead of LOA with the training time (200 epochs). The overhead consistently remains low, accounting for only 6.58% of the training time on average, which is already lower than the benefit of LOA (8.40%) in 200 epochs. The execution of LOA is offline, which has constant overhead regardless of the number of epochs and layers. Conversely, the benefit of LOA will accumulate as the epochs increase. With an increase in the number of epochs, particularly for larger datasets and deeper models that require more epochs to converge, this preprocessing overhead becomes more negligible[18]. In such scenarios, we advocate for utilizing LOA.

VII. Conclusion

We present HC-SpMM, a novel algorithm for accelerating SpMM on graphs. Being the first GPU accelerator to utilize hybrid GPU cores, i.e., CUDA and Tensor cores, HC-SpMM offers pioneering ideas that lay the foundation for future research. It intelligently selects hardware structures suitable for submatrices, fully leveraging the computational characteristics

[18]For example, the overhead of LOA accounts for only about 3% of the GNN training time in the case of 400 epochs, while the speedup ratio of LOA remains unchanged, which is 8.4% on average.

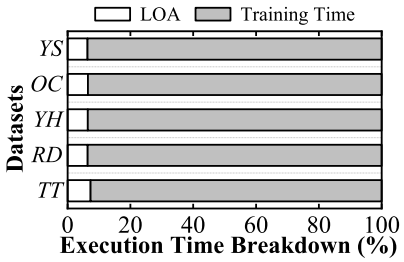


Fig. 16. Overhead of LOA in the entire training.

of different GPU cores. Meanwhile, we propose a thread utilization and data loading strategy to optimize the efficiency. Furthermore, to support graph computing applications, we integrate HC-SpMM into PyTorch and employ it to accelerate GNN training. To better improve the efficiency of GNN training, we devise a kernel fusion strategy and an algorithm LOA to reuse data, improve the graph layout, and better fit the computational models of hybrid cores. Experimental results consistently demonstrate the superiority of HC-SpMM over existing SpMM kernels and GNN training frameworks.

References

[1] K. S. Bøgh, S. Chester, D. Šidlauskas, and I. Assent. Template skycube algorithms for heterogeneous parallelism on multicore and gpu architectures. In *SIGMOD*, pages 447–462, 2017.

[2] A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP*, pages 503–510, 2008.

[3] L. Chang, W. Li, L. Qin, W. Zhang, and S. Yang. pscan: fast and exact structural graph clustering. *TKDE*, 29(2):387–401, 2017.

[4] F. Chen, Y. Zhang, L. Chen, X. Meng, Y. Qi, and J. Wang. Dynamic traveling time forecasting based on spatial-temporal graph convolutional networks. *FCS*, 17(6):176615, 2023.

[5] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *SC*, pages 1–14, 2021.

[6] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. *PVLDB*, 12(5):544–556, 2019.

[7] G. Dai, G. Huang, S. Yang, Z. Yu, H. Zhang, Y. Ding, Y. Xie, H. Yang, and Y. Wang. Heuristic adaptability to input dynamics for spmm on gpus. In *DAC*, pages 595–600, 2022.

[8] G. V. Demirci, A. Haldar, and H. Ferhatosmanoglu. Scalable graph convolutional network training on distributed-memory systems. *PVLDB*, 16(4):711–724, 2023.

[9] K. Duan, Z. Liu, P. Wang, W. Zheng, K. Zhou, T. Chen, X. Hu, and Z. Wang. A comprehensive study on large-scale graph training: Benchmarking and rethinking. *NeurIPS*, 35:5376–5389, 2022.

[10] R. Fan, W. Wang, and X. Chu. Fast sparse gpu kernels for accelerated training of graph neural networks. In *IPDPS*, pages 501–511, 2023.

[11] R. Fan, W. Wang, and X. Chu. Dtc-spmm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores. In *ASPLOS*, pages 253–267, 2024.

[12] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop*, 2019.

[13] T. Gale, M. Zaharia, C. Young, and E. Elsen. Sparse gpu kernels for deep learning. In *SC*, pages 1–14, 2020.

[14] G. Gerogiannis, S. Aananthakrishnan, J. Torrellas, and I. Hur. Hottiles: Accelerating spmm with heterogeneous accelerator architectures. In *HPCA*, pages 1012–1028, 2024.

[15] T. Gu, C. Wang, C. Wu, Y. Lou, J. Xu, C. Wang, K. Xu, C. Ye, and Y. Song. Hybridgnn: Learning hybrid representation for recommendation in multiplex heterogeneous networks. In *ICDE*, pages 1355–1367, 2022.

[16] K. Ho, H. Zhao, A. Jog, and S. Mohanty. Improving gpu throughput through parallel execution using tensor cores and cuda cores. In *ISVLSI*, pages 223–228, 2022.

[17] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP*, pages 300–314, 2019.

[18] G. Hou, X. Chen, S. Wang, and Z. Wei. Massively parallel algorithms for personalized pagerank. *PVLDB*, 14(9):1668–1680, 2021.
[19] K.-C. Hsu and H.-W. Tseng. Simultaneous and heterogenous multithreading. In *MICRO*, pages 137–152, 2023.
[20] G. Huang, G. Dai, Y. Wang, and H. Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC*, pages 1–12, 2020.
[21] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.
[22] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *HPCA*, pages 71–83, 2021.
[23] C.-T. Li, Y.-C. Tsai, and J. C. Liao. Graph neural networks for tabular data learning. In *ICDE*, pages 3589–3592, 2023.
[24] S. Li, K. Osawa, and T. Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC*, pages 1–15, 2022.
[25] Z. Li, X. Jian, Y. Wang, Y. Shao, and L. Chen. Daha: Accelerating gnn training with data and hardware aware execution planning. *PVLDB*, 17(6):1364–1376, 2024.
[26] S. Luo, X. Xiao, W. Lin, and B. Kao. Efficient batch one-hop personalized pageranks. In *ICDE*, pages 1562–1565, 2019.
[27] S. W. Min, K. Wu, S. Huang, M. Hidayetoglu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *PVLDB*, 14(11):2087–2100, 2021.
[28] Nvidia. Cuda sparse matrix library. 2023. https://developer.nvidia.com/cusparse.
[29] Nvidia. Dense linear algebra on gpus. 2023. https://developer.nvidia.com/cublas.
[30] J. Peng, Z. Chen, Y. Shao, Y. Shen, L. Chen, and J. Cao. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *PVLDB*, 15(9):1937–1950, 2022.
[31] X. Ran, Q. Ye, H. Hu, X. Huang, J. Xu, and J. Fu. Differentially private graph neural networks for link prediction. In *ICDE*, pages 1632–1644, 2024.
[32] V. Rosenfeld, S. Breß, and V. Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Computing Surveys*, 55(1):1–38, 2022.
[33] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang. Realtime top-k personalized pagerank over large graphs on gpus. *PVLDB*, 13(1):15–28, 2019.
[34] Z. Song, Y. Zhang, and I. King. Towards an optimal asymmetric graph structure for robust semi-supervised node classification. In *SIGKDD*, pages 1656–1665, 2022.
[35] C. Sun, Y. Ning, D. Shen, and T. Nie. Graph neural network-based short-term load forecasting with temporal convolution. *DSE*, pages 1–20, 2023.
[36] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey. Graphmat: High performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.
[37] D. Tang, J. Wang, R. Chen, L. Wang, W. Yu, J. Zhou, and K. Li. Xgnn: Boosting multi-gpu gnn training via global gnn memory store. *PVLDB*, 17(5):1105–1118, 2024.
[38] X. Wan, K. Xu, X. Liao, Y. Jin, K. Chen, and X. Jin. Scalable and efficient full-graph gnn training for large graphs. In *SIGMOD*, pages 1–23, 2023.
[39] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv:1909.01315*, 2019.
[40] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu. Neutronstar: distributed gnn training with hybrid dependency management. In *SIGMOD*, pages 1301–1315, 2022.
[41] Y. Wang, B. Feng, and Y. Ding. Qgtc: accelerating quantized graph neural networks via gpu tensor core. In *PPoPP*, pages 107–119, 2022.
[42] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus. In *OSDI*, pages 515–531, 2021.
[43] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding. Tc-gnn: Bridging sparse gnn computation and dense tensor cores on gpus. In *USENIX ATC*, pages 149–164, 2023.
[44] Z. Wen, Y. Fang, and Z. Liu. Meta-inductive node classification across graphs. In *SIGIR*, pages 1219–1228, 2021.
[45] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen. Thundersvm: A fast svm library on gpus and cpus. *Journal of Machine Learning Research*, 19(21):1–5, 2018.
[46] J. Wu, X. He, X. Wang, Q. Wang, W. Chen, J. Lian, and X. Xie. Graph convolution machine for context-aware recommender system. *FCS*, 16(6):166614, 2022.
[47] H. Xia, Z. Zheng, Y. Li, D. Zhuang, Z. Zhou, X. Qiu, Y. Li, W. Lin, and S. L. Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *PVLDB*, 17(2):211–224, 2023.
[48] L. Xia, C. Huang, Y. Xu, P. Dai, M. Lu, and L. Bo. Multi-behavior enhanced recommendation with cross-interaction collaborative relation modeling. In *ICDE*, pages 1931–1936, 2021.
[49] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.
[50] Z. Xu, Y. Ke, Y. Wang, H. Cheng, and J. Cheng. A model-based approach to attributed graph clustering. In *SIGMOD*, pages 505–516, 2012.
[51] Z. Xue, M. Wen, Z. Chen, Y. Shi, M. Tang, J. Yang, and Z. Luo. Releasing the potential of tensor core for unstructured spmm using tiled-csr format. In *ICCD*, pages 457–464, 2023.
[52] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.
[53] C. Yang, A. Buluç, and J. D. Owens. Design principles for sparse matrix multiplication on the gpu. In *Euro-Par*, pages 672–687, 2018.
[54] C. Yang, A. Buluç, and J. D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *TOMS*, 48(1):1–51, 2022.
[55] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares. Accelerating sparse matrix–matrix multiplication with gpu tensor cores. *Computers & Electrical Engineering*, 88:106848, 2020.
[56] F. Zhang and S. Wang. Effective indexing for dynamic structural graph clustering. *PVLDB*, 15(11):2908–2920, 2022.
[57] J. Zhang, C. Gao, D. Jin, and Y. Li. Group-buying recommendation for social e-commerce. In *ICDE*, pages 1536–1547, 2021.
[58] L. Zhang, S. Wang, J. Liu, X. Chang, Q. Lin, Y. Wu, and Q. Zheng. Mulgrn: Multi-level graph relation network for few-shot node classification. *TKDE*, 35(6):6085–6098, 2022.
[59] X. Zhang, Y. Shen, Y. Shao, and L. Chen. Ducati: A dual-cache training system for graph neural networks on giant graphs with the gpu. In *SIGMOD*, pages 1–24, 2023.
[60] Y. Zhang and A. Kumar. Lotan: Bridging the gap between gnns and scalable graph analytics engines. *PVLDB*, 16(11):2728–2741, 2023.
[61] Y. Zhang, W. Wang, H. Yin, P. Zhao, W. Chen, and L. Zhao. Disconnected emerging knowledge graph oriented inductive link prediction. In *ICDE*, pages 381–393, 2023.
[62] Z. Zhang, H. Wang, S. Han, and W. J. Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *HPCA*, pages 261–274, 2020.
[63] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *ASPLOS*, pages 859–873, 2020.
[64] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. Aligraph: A comprehensive graph neural network platform. *PVLDB*, 12(12):2094–2105, 2019.
[65] Y. Zou, Z. Ding, J. Shi, S. Guo, C. Su, and Y. Zhang. Embedx: A versatile, efficient and scalable platform to embed both graphs and high-dimensional sparse data. *PVLDB*, 16(12):3543–3556, 2023.

APPENDIX

A. Evaluation on various GPU architectures

We report the absolute numbers related to the execution times of SpMM kernels and compare the SpMM kernels on Nvidia A100 and RTX 4090 GPUs to evaluate the generality of the regression model, which are depicted in Table XVI.

Our proposed SpMM kernel is superior to other SpMM kernels in most cases, and the performance of the logistic regression model is stable on different types of GPUs.

TABLE VII
OVERHEAD OF SPMM ON DIFFERENT FP TYPES.($\times 10^{-6}$S)

	Sputnik	TC-GNN	HC-SpMM(half)	HC-SpMM(bfloat)
CS	4.99	23.29	**4.06**	4.06
CR	6.26	41.95	5.54	**5.41**
PM	9.09	94.2	**10.43**	10.56
PT	**12.77**	37.63	15.42	15.39
DD	117.82	248.64	**105.28**	105.47
AZ	**159.87**	618.24	223.14	223.10
YS	575.90	683.55	**477.31**	477.34
OC	585.79	813.49	515.74	**515.10**
GH	591.04	1226.78	476.93	**476.51**
YH	975.03	1319.19	851.84	**851.16**
RD	1652.57	1916.91	1304.47	**1301.69**
TT	1484.54	2207.99	**1200.03**	1201.24
DP	**9096.03**	149166.60	15138.63	15119.01

B. Evaluation on various FP types

We evaluate 3 commonly used FP types: TF32, half, and bfloat16. In this evaluation, we exclude GE-SpMM and DTC-SpMM from the comparison, as they do not support other FP types. Sputnik supports both float and half precision. We modify TC-GNN to enable half precision. The results are reported in Table VII. For HC-SpMM, the performance of half and bfloat16 is similar. Sputnik, having been specifically optimized for half precision, achieves up to more than 2× speedup over its own full precision implementation. Although HC-SpMM doesn't include optimizations for half precision, it outperforms Sputnik by up to 1.26× and exhibits superior performance in most datasets. This demonstrates the adaptability and scalability of our proposed method across different FP types. Due to the 16×16×16 matrix input requirement of the WMMA API, TC-GNN exhibits worse performance than itself with TF32. This is because TF32 has an input requirement of 16×8×16, which has a smaller granularity than 16×16×16, resulting in fewer unnecessary calculations involving zeros.

C. Absolute Numbers of End-to-end Training

We report the absolute numbers of the training overhead of GCN in Table VIII (corresponding to Figure 12) and GIN in Table IX (corresponding to Figure 13).

D. Evaluations on various sparsity

To verify the adaptability of HC-SpMM to different densities and demonstrate the necessity of using hybrid GPU cores, we conduct experiments on synthetic matrices with different sparsity. We vary the number of non-zero elements in 16×8 non-zero blocks to generate various synthetic matrices.

Execution times of different SpMM kernels are reported in Table X. HC-SpMM achieves the best performance on all synthetic matrices. When the sparsity is less than 85%, DTC-SpMM exhibits less execution time than Sputnik, while Sputnik performs better than DTC-SpMM when the sparsity is greater than 90%. This phenomenon is consistent with our experimental results in Figure 1, which demonstrate that when there are more dense parts in a matrix, Tensor cores (represented by DTC-SpMM) have higher efficiency than CUDA cores (represented by Sputnik).

TABLE VIII
AVERAGE EPOCH TIME OF GCN TRAINING. ($\times 10^{-3}$ S)

		GE-SpMM	TC-GNN	HC-SpMM
CS	Forward	0.33	0.37	**0.31**
	Backward	0.53	0.51	**0.45**
CR	Forward	0.30	0.45	**0.26**
	Backward	0.45	0.45	**0.36**
PM	Forward	0.32	0.72	**0.28**
	Backward	0.43	0.78	**0.43**
PT	Forward	0.35	0.42	**0.32**
	Backward	0.49	0.51	**0.42**
DD	Forward	2.45	2.81	**2.17**
	Backward	2.85	3.22	**2.09**
AZ	Forward	3.94	5.63	**3.41**
	Backward	4.36	6.59	**3.82**
YS	Forward	11.46	11.01	**10.12**
	Backward	13.44	13.02	**9.24**
OC	Forward	12.19	12.32	**10.98**
	Backward	14.56	14.75	**10.12**
GH	Forward	9.15	12.10	**7.88**
	Backward	11.76	14.67	**8.30**
YH	Forward	20.73	20.98	**18.74**
	Backward	23.90	24.20	**16.82**
RD	Forward	28.67	28.48	**25.30**
	Backward	38.03	37.77	**26.46**
TT	Forward	23.86	26.49	**20.46**
	Backward	31.06	33.40	**21.94**

TABLE IX
AVERAGE EPOCH TIME OF GIN TRAINING. ($\times 10^{-3}$ S)

		GE-SpMM	TC-GNN	HC-SpMM
YS	Forward	12.70	11.75	**8.16**
	Backward	14.89	13.82	**13.26**
OC	Forward	13.55	13.10	**8.92**
	Backward	15.81	15.44	**14.65**
YH	Forward	23.09	23.32	**15.11**
	Backward	26.38	25.46	**24.14**
RD	Forward	32.55	31.36	**21.49**
	Backward	41.92	40.65	**39.27**
TT	Forward	26.88	29.19	**20.15**
	Backward	34.09	36.26	**32.92**

E. Sensity Evaluation

We evaluate the sensitivity of performance to the specific logistic regression model parameters on two datasets *YH* and *RD*. The model includes three parameters: w_1, corresponding to the number of non-zero columns; w_2 representing the coefficient of the sparsity; and b, the intercept. Figure 17 depicts the results. The performance sensitivity to w_1 and b is similar, where a 50% change in parameter values leads to a performance variation of around 14%. The changes in parameters w_1 and b significantly affect the number of row windows classified into CUDA cores and Tensor cores. In contrast, a 50% change in the value of w_2 results in a performance difference of up to 3%. Overall, the performance is more sensitive to variations in w_1 and b than in w_2.

F. Evaluation of the Preprocessing Overhead

We adopt the preprocessing method from DTC-SpMM,

TABLE X
RUNTIME OF SPMM KERNELS WITH VARIOUS SPARSITY. ($\times 10^{-6}$ S)

Methods \ Sparsity	80%	85%	90%	95%
Sputnik	9.28	8.58	6.67	6.10
GE-SpMM	9.34	8.93	8.77	7.90
TC-GNN	14.85	14.56	13.41	10.75
DTC-SpMM	8.21	8.35	7.94	6.45
HC-SpMM	**7.49**	**6.62**	**5.73**	**5.31**

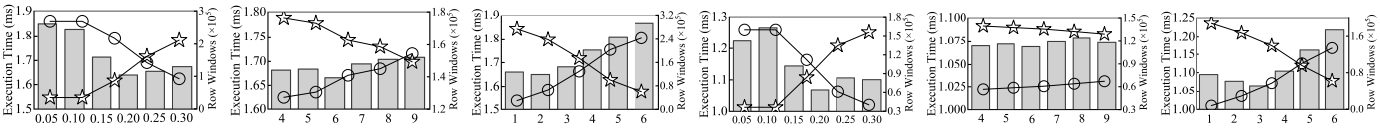


Fig. 17. Sensity of regression model parameters to SpMM performance.

TABLE XI
COMPARISON OF PREPROCESSING OVERHEAD (MS).

Datasets	DTC-SpMM	TC-GNN	HC-SpMM
YS	11.48	241.50	**8.72**
OC	11.56	284.81	**9.38**
YH	15.03	457.70	**11.82**
RD	20.44	671.76	**15.72**
TT	33.94	966.86	**24.02**

which performs preprocessing directly on the GPU, eliminating additional PCIe transfers. The preprocessing overhead consists of two parts: preparing data for Tensor cores and selecting appropriate GPU cores for row windows.

The preprocessing overhead related to Tensor cores is unavoidable due to the sparsity of input matrices. This step involves densifying each row window by moving non-zero columns to the front of the row windows. Existing SpMM algorithms on Tensor cores such as TC-GNN and DTC-SpMM have the same preprocessing process. We adopt the preprocessing kernel from DTC-SpMM while removing the unnecessary process in HC-SpMM. The comparison of preprocessing overhead is presented in Table XI. Specifically, the preprocessing of HC-SpMM outperforms that of DTC-SpMM and TC-GNN by factors of 1.3× and 36.0×, respectively. On average, the preprocessing overhead in HC-SpMM is about 13.0× that of a single SpMM execution.

The selection overhead is negligible because the inference of the logistic regression model simply involves calculating $w_1 * x_1 + w_2 * x_2 + b$, where x_1 and x_2 are computed during the preprocessing of Tensor cores, and w_1, w_2, and b are parameters retrieved from the regression model trained offline. Consequently, this selection step takes only a few nanoseconds to complete. There is no preprocessing overhead for SpMM on CUDA cores. The transferred data through PCIe consists only of the CSR format of graphs.

We didn't include the preprocessing overhead in Figure 10 because, in many real-world applications such as GNNs, thousands of SpMM operations are performed on unchanged sparse matrices [11], [45]. The preprocessing overhead is negligible in these cases. For scenarios where sparse matrices are constantly changing, SpMM methods optimized for CUDA cores such as Sputnik are more suitable.

G. Evaluation of Memory Usage

Note that an out-of-memory error occurs during training GNNs on *DP* for HC-SpMM, GE-SpMM, and TC-GNN. To demonstrate the comparable memory usage of HC-SpMM, we evaluate the memory usage of the three methods on 5 datasets. Table XII depicts the results. The memory usage of HC-SpMM is only up to 2% more than GE-SpMM and up to 6% more than TC-GNN due to the additional data structure for efficient SpMM with hybrid GPU cores.

TABLE XII
COMPARISON OF MEMORY USAGE (MB).

Datasets	GE-SpMM	TC-GNN	HC-SpMM
YS	4736	**4542**	4752
OC	5096	**4870**	5122
YH	7558	**7210**	7608
RD	10976	**10432**	11046
TT	8938	**8570**	9050

TABLE XIII
COMPARISON OF TENSOR CORES' UTILIZATION (%).

Datasets	DTC-SpMM	TC-GNN	HC-SpMM
YS	3.69	2.77	**3.85**
OC	**3.82**	2.84	2.82
YH	**3.79**	2.82	2.94
RD	**3.51**	2.69	2.72
TT	**4.07**	2.96	2.36

H. Utilization Metrics of CUDA and Tensor Cores

We evaluate several utilization metrics of the GPU cores. Accurately measuring the CUDA cores' utilization during SpMM is challenging, as they not only handle SpMM calculations but also manage data loading and preparation before the Tensor cores' computations. Therefore, we report only the utilization of Tensor cores in Table XIII. In HC-SpMM, CUDA and Tensor cores don't execute concurrently, resulting in lower overall utilization because one type of GPU cores remains idle while the other is active.

We also evaluate the execution time for CUDA and Tensor cores, as shown in Table XIV. The execution time is proportional to the number of row windows processed by CUDA and Tensor cores in Figure 15(b).

Additionally, we measure and compare the computing and memory throughput using Nsight Compute. The results, presented in Table XV, demonstrate that HC-SpMM achieves the highest computing and memory throughput compared to the other methods evaluated.

TABLE XIV
COMPARISON OF GPU CORES' EXECUTION TIME (MS).

GPU cores	*YS*	*OC*	*YH*	*RD*	*TT*
CUDA cores	563.00	512.87	803.04	990.30	992.99
Tensor cores	110.68	202.73	327.17	786.15	245.35

TABLE XV
COMPARISON OF COMPUTING AND MEMORY THROUGHPUT (%).

Types	Methods	*YS*	*OC*	*YH*	*RD*	*TT*
Computing	TC-GNN	31.35	31.66	31.63	30.68	34.35
	Sputnik	18.36	18.51	18.53	20.04	26.31
	GE-SpMM	40.61	39.74	39.52	38.28	57.05
	DTC-SpMM	44.08	45.24	45.09	42.17	47.61
	HC-SpMM	**53.77**	**52.22**	**51.94**	**51.62**	**75.97**
Memory	TC-GNN	31.24	31.46	31.45	31.38	33.11
	Sputnik	64.22	64.71	64.90	56.84	54.04
	GE-SpMM	55.49	58.72	58.84	55.51	57.05
	DTC-SpMM	78.70	75.06	76.58	79.48	67.16
	HC-SpMM	**84.58**	**87.11**	**87.51**	**89.76**	**82.77**

TABLE XVI
OVERHEAD OF SPMM ON DIFFERENT GPUS. ($\times 10^{-6}$ S)

		Sputnik	GE-SpMM	TC-GNN	DTC-SpMM	cuSPAESE	HC-SpMM
CS	3090	5.63	6.62	22.40	10.73	12.77	**5.25**
	4090	4.06	4.19	18.08	8.45	9.89	**3.93**
	A100	14.30	15.43	56.07	27.07	23.07	**10.66**
CR	3090	6.98	8.64	6.98	18.33	13.28	**6.05**
	4090	5.44	5.98	29.92	12.80	9.76	**4.83**
	A100	17.44	20.83	76.39	42.11	25.02	**13.76**
PM	3090	13.41	13.22	78.53	33.79	59.23	**11.62**
	4090	7.46	**7.20**	72.48	26.72	47.52	7.87
	A100	25.02	21.54	211.27	84.26	90.91	**15.43**
PT	3090	23.90	24.70	38.21	25.40	51.49	**17.76**
	4090	**7.84**	8.90	25.22	12.00	27.71	11.23
	A100	23.46	27.87	59.11	32.48	61.25	**17.82**
DD	3090	171.42	185.98	249.98	164.76	354.85	**121.57**
	4090	**67.68**	106.50	138.27	112.90	249.54	73.54
	A100	**155.78**	267.30	433.26	202.24	509.10	178.79
AZ	3090	353.57	358.53	743.17	414.25	3833.33	**240.67**
	4090	118.37	584.84	293.83	171.62	856.68	**112.61**
	A100	292.26	384.36	1051.22	471.43	1081.33	**271.97**
YS	3090	769.09	866.75	756.80	**581.36**	1226.88	581.41
	4090	479.24	585.10	548.04	498.15	1048.91	**461.51**
	A100	790.67	1181.46	1211.83	**669.47**	1286.10	743.82
OC	3090	841.21	909.57	899.20	660.56	1313.76	**624.58**
	4090	533.29	634.89	627.75	577.55	1178.73	**505.99**
	A100	851.28	1000.53	1451.10	**786.56**	1471.58	825.17
GH	3090	850.59	836.10	1330.56	764.60	1296.15	**568.41**
	4090	475.97	549.74	806.22	534.95	1104.04	**441.32**
	A100	1046.29	1102.03	2221.16	811.76	1863.42	**761.20**
YH	3090	1395.39	1510.75	1461.47	1085.29	2145.85	**1045.92**
	4090	917.48	1038.70	1045.17	954.03	1901.68	**869.32**
	A100	1431.06	2011.01	2073.99	1129.12	2502.35	**1032.05**
RD	3090	2441.08	2469.04	2182.26	1651.71	2917.27	**1574.69**
	4090	1484.53	1639.77	1597.62	1474.62	2945.31	**1436.98**
	A100	2617.16	2870.19	3167.70	**1846.44**	2856.08	2013.03
TT	3090	2164.82	2153.66	2426.94	1601.94	4003.09	**1382.53**
	4090	1276.56	1347.54	1541.43	1231.86	3109.70	**1126.61**
	A100	2245.99	2918.77	2073.99	1963.72	5752.35	**1880.32**
DP	3090	19696.12	18312.22	87216.13	21148.42	327079.96	**16718.30**
	4090	12061.82	**11206.81**	95456.71	14240.56	169181.09	11350.72
	A100	16926.84	15532.93	178816.99	15808.13	145479.16	**13778.36**