HyGNN: Accelerating GNN Training with Hybrid GPU Cores on Sparse Graphs

Zhonggen Li*, Xiangyu Ke*, Yifan Zhu*, Yunjun Gao*, Yaofeng Tu†

*Zhejiang University, Hangzhou, China   †ZTE Corporation, Nanjing, China

Email: *{zgli, xiangyu.ke, xtf_z, gaoyj}@zju.edu.cn, †tu.yaofeng@zte.com.cn

Abstract—Graph Neural Networks (GNNs) have demonstrated remarkable performance in various tasks, including link prediction, recommendation, and classification. However, training GNNs, particularly the sparse matrix-matrix multiplication (SpMM) component, is often time-consuming, thus constraining their large-scale applicability. Fortunately, the 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 **HyGNN**, a pioneering algorithm that leverages hybrid GPU cores (Tensor cores and CUDA cores) to accelerate SpMM and GNN training from a data management perspective. To adapt to the computing characteristics of different GPU cores, we investigate the impact of sparse graph features on GPU performance, 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 propose a cost-effective data layout reorganization method to mitigate the irregular and sparse issues of real-world graphs, better fitting the computational models of hybrid cores. Furthermore, we systematically integrate **HyGNN** into **PyTorch**, and optimize it by considering kernel fusion, memory access, thread utilization, etc., to utilize the computational resources to their fullest potential. Extensive experiments on 13 real-world datasets demonstrate that **HyGNN** achieves an average speedup of 1.33× over state-of-the-art SpMM kernels and a 1.23× speedup over leading GNN training frameworks.

Index Terms—GNNs, Hybrid GPU cores, SpMM

I. INTRODUCTION

In recent years, Graph Neural Networks (GNNs) have demonstrated versatility in addressing a myriad of problems such as node classification [25], [34], [44], link prediction [11], [24], [47], recommendation [35], [37], [43], [52] and beyond [2], [15], [26]. As the scale of graph data proceeds to grow and the complexity of GNN architectures escalates with an increasing number of layers[1], a pressing demand arises in both academia and industry for faster GNN training [50]. Enhanced training speed can not only diminish the cost of computing, thereby freeing up additional resources and time for other endeavors, but also accelerate iterative model assessment and foster the model evolution [18], [48].

During both forward and backward propagations in GNNs, two fundamental operations are performed: *Aggregation* and *Update*. The *Update* operation can be efficiently computed using CUDA library cuBLAS [21]. In contrast, *Aggregation* involves sparse matrix-matrix multiplication (SpMM), which

is characterized by *irregular memory access patterns* and *low computational intensity*, especially on real-world graphs, posing significant challenges for GPU acceleration [33]. A recent study has indicated that *Aggregation* accounts for more than 80% of the execution time [33], emerging as the primary bottleneck of GNNs.

CUDA cores and Tensor cores represent distinct computing units in GPUs. CUDA cores are versatile units capable of executing general computing tasks, while Tensor cores are specialized for efficient matrix multiplication operations [42]. Tensor cores exhibit a remarkable ability to perform fixed-size (e.g., 4 × 4) matrix multiplications in a single clock cycle [22], which is more efficient than CUDA cores. However, despite significantly improving the efficiency of dense matrix multiplication and deep learning [16], this fix-sized input requirement restricts the flexibility to avoid computing numerous zero elements in sparse matrices, thus limiting the efficiency of Tensor cores in performing SpMM.

Leading GNN training frameworks like Deep Graph Library (DGL) [29] employ the SpMM kernel in cuSPARSE [20] to implement *Aggregation* using CUDA cores. Subsequently, several efficient SpMM kernels have been proposed, achieving a notable speedup of up to 14.2× over cuSPARSE [10], [12], [13]. Recent attempts have explored harnessing Tensor cores to accelerate SpMM by employing custom data structures to avoid unnecessary computations involving zero elements in structured sparse matrices [3], [16]. However, the real-world graphs are always irregular and sparse, impeding them from achieving satisfactory performance [8], [33], [42]. 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 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 or Tensor cores for SpMM acceleration fails to fully exploit their respective computational strengths*, primarily due to the irregularity of

---

[1]Training a GNN-based recommendation system over a dataset comprising 7.5 billion items consumes approximately 3 days on a 16-GPU cluster [6].

[2]For example, the average sparsity of the matrices output by the preprocessing method proposed in TC-GNN [33] is still 90.9% on 10 tested datasets.

[3]An in-depth analyzation is provided in § IV-B.
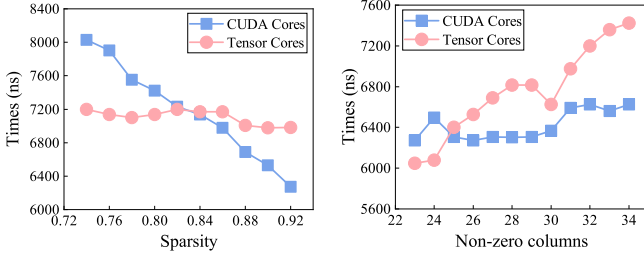
(a) Varying sparsity. (b) Varying non-zero columns.

Fig. 1. Execution time of different GPU cores.

real-world graphs, which typically comprise a mixture of dense and sparse regions [40]. Consequently, we explore the distribution characteristics of graph adjacency matrices and devise a hybrid strategy tailored to the varying computational demands of different graph regions, enabling more effective utilization of the capabilities of different GPU cores.

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

Challenge I*: How to leverage CUDA and Tensor cores jointly for enhanced computational capability?* The computational characteristics of CUDA and Tensor cores vary significantly, making the selection of the optimal core for matrices with different sparsity levels crucial for enhancing computational efficiency. Effectively partitioning a sparse adjacency matrix into sub-regions with distinct sparsity characteristics to leverage the different cores for collaborative computation presents a considerable challenge. Additionally, accurately modeling the computational performance of CUDA and Tensor cores for SpMM to facilitate precise core selection is another major difficulty. To address these issues, we design a fine-grained graph data partition strategy combined with a core selection method to efficiently utilize CUDA and Tensor cores jointly. Firstly, the fine-grained partition strategy 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 and avoiding establishing separate data structures for each core, thus preserving the sequential memory access of edges. Next, comprehensive quantitative experiments (§ IV-B) reveal that CUDA cores are *memory-efficient* while Tensor cores are *computing-efficient*. 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. Leveraging these factors, we develop an algorithm tailored for the selection of appropriate GPU cores for submatrices, optimizing computational capability (§ IV-B).

Challenge II*: How to fully harness the computational potential of Tensor cores on real-world graph datasets?* For dense matrix multiplication, Tensor cores incur significantly higher throughput than CUDA cores, potentially offering sub-

stantial 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 more amenable to processing via CUDA cores. Consequently, the performance gains achievable with Tensor cores are often negligible [33]. To unlock the full computational potential of Tensor cores and thereby significantly enhance the efficiency of GNN training, we first introduce a metric termed *computating intensity* to estimate the calculation workload of the submatrices multiplication (§ V), 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 denser segments suitable for processing by Tensor cores, gaining significant efficiency with Tensor cores (§ V). This adjustment has a relatively small cost but renders the graph data more compatible with hybrid GPU cores (§ VII-D).

Challenge III*: How to optimize the training pipeline from a global perspective?* Firstly, including multiple kernel functions within each GNN layer introduces significant overhead due to kernel launch. Secondly, the isolation among kernels within a layer in prevalent GNN training frameworks [29], [33] impedes data reuse, leading to additional memory access overhead. Lastly, inefficiencies inherent in the SpMM kernel, such as suboptimal memory access patterns and underutilized threads, limit the overall performance. To address these issues, we present a kernel fusion strategy to mitigate kernel launch costs and global memory access (§ VI-A). Additionally, we conduct in-depth optimizations of the SpMM kernel, considering thread collaboration mode (§ VI-B) and memory access patterns (§ VI-C), to further enhance efficiency.

Contributions. In this work, we propose HyGNN, a novel approach for accelerating SpMM and GNNs 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 hardware-aware 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 (§ IV).
- We present a lightweight graph layout optimization algorithm to enhance irregular graph layouts and better align with the characteristics of both GPU cores (§ V).
- We identify and optimize bottlenecks in GNN training from a global perspective, employing kernel fusion to tightly couple each module, thereby eliminating kernel launch time and enabling efficient data reuse (§ VI).
- We conduct comprehensive evaluation demonstrating that HyGNN outperforms existing methods and achieves $1.23\times$ speedup in GNN training on average (§ VII).

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 elaborates the graph layout reformat algorithm. In Section VI, we describe the global optimization techniques of GNN training. Section VII exhibits the experimental results. We conclude the paper in Section VIII.

II. RELATED WORK

SpMM Using CUDA Cores. Optimization of SpMM has been a subject of extensive study [1], [12], [36], [41], [49]. In recent years, Yang et al. [41] proposed two high-performance algorithms leveraging merge-based load balancing and row-major coalesced memory access strategies for SpMM. Hong et al. [12] design an adaptive tiling strategy to enhance the performance of SpMM and SDDMM. Huang et al. [13] introduced a GNN-specified SpMM kernel, utilizing coalesced row caching and coarse-grained warp merging methods to optimize memory access. cuSPARSE [20], a closed-source linear algebra library developed by Nvidia, offers GPU-accelerated linear algebra subroutines for sparse matrices and has been integrated into numerous GNN training frameworks such as DGL [29]. Gale et al. point out that cuSPARSE is efficient only for matrices with sparsity exceeding 98%, which may not be suitable for deep learning scenarios. To address this limitation, they propose a SpMM library Sputnik [10] to accelerate the unstructured SpMM in deep neural networks and achieve state-of-the-art performance. Dai et al. [4] introduced an adaptive approach capable of heuristically selecting suitable kernels based on input matrices. However, this approach demonstrates superior performance only when the matrix dimension is less than 32. Furthermore, Fan et al. [7] proposed an SpMM kernel employing a unified hybrid parallel strategy of mixing nodes and edges, but it is limited to Ampere (A100, A800) or Hopper (H100, H800) GPUs. 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 [3], [8], [16], [31], [33]. 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 [8], [33], [42]. For example, TC-GNN [33] compresses all zero columns within a row window and DTC-SpMM [8] 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 HyGNN employs both cores for computing according to their characteristics. Besides, Xue et al. [39] 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
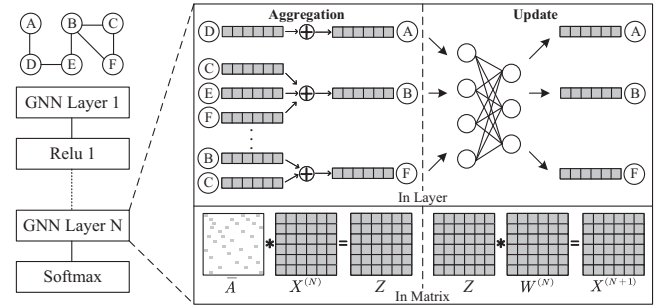

Fig. 2. An example of GNN.

preprocessing can densify submatrices, they still involve some computational waste and lead to suboptimal performance.

GNN Training Frameworks. Deep Graph Library (DGL) [29] and PyTorch Geometric (PyG) [9] stand out as comprehensive GNN training frameworks. Additionally, GNNAdvisor [32] identifies the graph locality and leverages 2D workload management to accelerate the training. DUCATI [45] designs a novel dual-cache strategy to better utilize the locality of the adjacency matrix and the GPU memory. DAHA [17] presents a GNN training framework with data and hardware-aware execution planning. Besides, various frameworks for distributed GNN training have emerged recently [5], [19], [23], [27], [28], [30], [46], [51], 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* to directly call the optimized SpMM kernel, significantly reducing the execution time of *Aggregation*. Works addressing similar problems include GE-SpMM [13] and TC-GNN [33] mentioned above, both of which have integrated the kernels into GNN training frameworks. However, they solely employ CUDA or Tensor cores to perform SpMM in GNNs, failing to fully exploit the respective computational strengths of these GPU cores.

III. PRELIMINARIES

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

A. 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 $A_{|V| \times |V|}$. Additionally, the node embeddings are represented as matrix $X_{|V| \times D}$, where D signifies the embedding dimension. Typically, a GNN layer consists of two fundamental operations: *Aggregation* and *Update*. An example of GNN is depicted in Figure 2.

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.

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

The *Update* operation involves neural processing, typically employing a neural network like Multilayer Perceptron which usually contains a simple matrix multiplication operation, to transform the aggregated vector of each node to an updated vector, serving as the new embedding vector for each node. 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* operation in layer k can be formalized as Equation 3, which contains 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.

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

B. 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 — the threads in a warp run simultaneously in a Single Instruction Multiple Threads (SIMT) fashion. The Compute Unified Device Architecture (CUDA) provides an abstraction of the GPU's architecture and serves as a programming model. In CUDA, a block consists of multiple warps and 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.
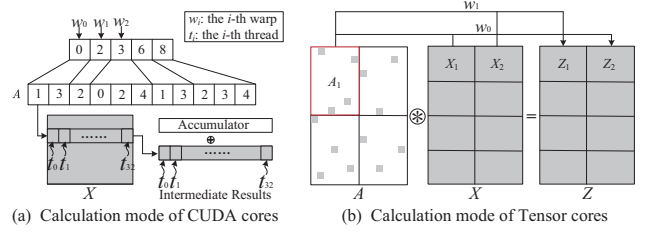

(a) Calculation mode of CUDA cores   (b) Calculation mode of Tensor cores

Fig. 3. Comparison of CUDA cores and Tensor cores.

C. 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 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 and cuSPARSE, 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 [33], which requires $16 \times 8 \times 16$ as the input size.

Figure 3 illustrates the disparity between CUDA cores and Tensor cores during sparse matrix-matrix multiplication. Here, w_i represents the i-th warp and t_j represents the j-th thread. As depicted in Figure 3(a), each thread is tasked with computing 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 3(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.

IV. HYBRID SpMM KERNEL

In this section, we justify our selection of the row window as the fundamental hybrid unit, contrasting it with the straightforward choice of using submatrices. Additionally, we delve

(a) A straightforward combination strategy.

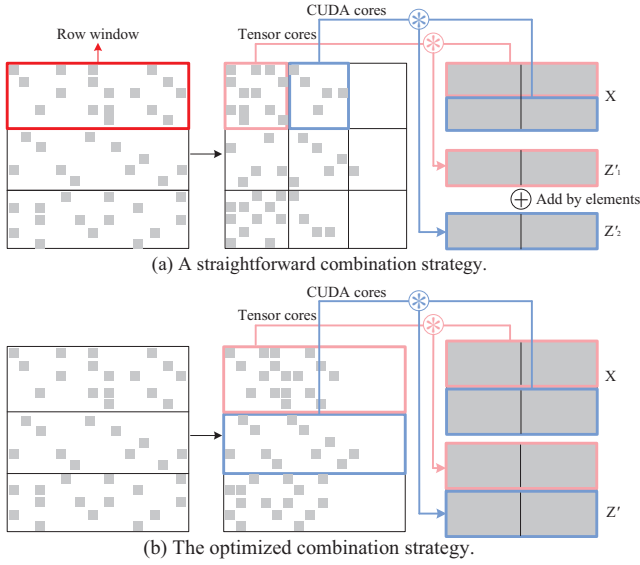(b) The optimized combination strategy.

Fig. 4. Comparison of combination strategies.

into the essential characteristics that influence the efficiency of CUDA and Tensor 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 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-B).

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

---

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

Algorithm 1: SpMM on CUDA cores

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

1 **for** $i = 0$ *to* $m - 1$ *in parallel* **do**
2   **for** $j = 0$ *to* $n - 1$ *in parallel* **do**
3       $res \leftarrow 0$;
4   **for** $k = rowPtr[i]$ *to* $rowPtr[i + 1]$ **do**
5          $res += val[k] * X[colInd[k], j]$;
6       $Z[i, j] = res$;

Algorithm 2: SpMM on Tensor cores

Input: rowPtr, colInd, val and embedding 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   Convert CSR into ASh in matrix format;
5   Load submatrices of X into XSh;
6   Each warp calculates $ASh \times XSh$ using Tensor cores;
7   Store the result into Z;

---

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 [33], 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 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-B).

B. Adaptive Core Selection

In this subsection, we quantify the difference between CUDA and Tensor cores and identify key characteristics of row windows that impact the performance of these cores. Following these observations, we devise an intelligent core allocation method.

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

---

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

TABLE I
COMPUTING AND MEMORY ACCESS COSTS OF GPU CORES IN SPMM[5].

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**

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 of a row window as 16×32 and the embedding 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.
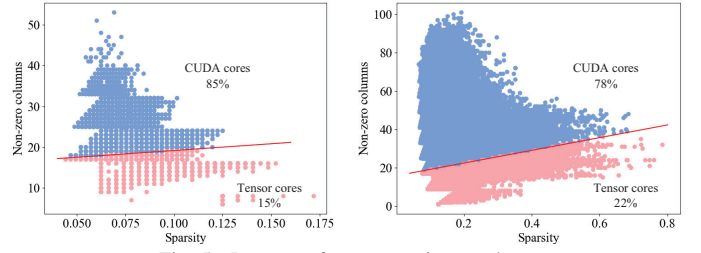

Fig. 5. Layouts of representative graphs.

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[6] compared to the aforementioned two characteristics, therefore we choose to disregard them. To further discover the difference between CUDA and Tensor cores, we evaluate the computing and memory access costs of both types of cores in SpMM, 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.

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 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. Note that this model is contingent upon hardware specifications and is agnostic to the specific data, necessitating only one-time training if the GPU remains unchanged. The logistic regression model is lightweight and efficient, allowing for rapid computation. The logistic regression model can complete the selection of GPU cores for a row window in a few nanoseconds, with an accuracy greater than 90%. More details of the logistic regression model are provided in Appendix A.

V. LAYOUT OPTIMIZATION

As discussed in § IV-B, CUDA cores are ideally suited for memory access-intensive row windows, while Tensor cores

---

[5]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.

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

Algorithm 3: 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     $RW \leftarrow$ the first vertex in $soList$ hasn't been visited;
5   **for** $i = 1$ *to* 15 **do**
6         $CanVtxs \leftarrow$ vertices with the highest *computing intensity* within VW;
7         $v \leftarrow$ vertex with the highest degree in $CanVtxs$;
8         $RW \leftarrow RW \cup \{v\}$;
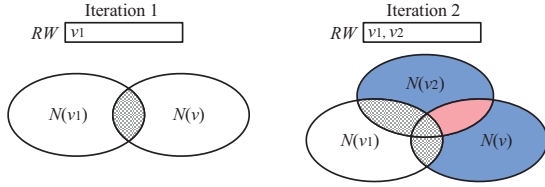9 $NRW \leftarrow NRW.append(RW)$;
10 Reorder G using NRW;

Fig. 6. An example of avoiding redundant computing.

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

excel in computing-intensive row ones. Real-world graphs typically exhibit row windows with numerous non-zero columns 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 5. 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.

$$computing\ intensity = \frac{\#nonzero\ elements}{\#nonzero\ columns} \quad (5)$$

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-B. 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 the row window 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 range window VW to confine the search scope. As outlined in Algorithm 3, we initially sort all vertices based on their neighbors' smallest index (line 1).

For each row window, we select the first unvisited vertex from the sorted list as the initial vertex and add it to the set RW (line 4). Subsequently, we compute the *computing intensity* when considering adding each vertex to the current row window according to Equation 5. 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 3. For instance, as depicted in Figure 6, 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, in subsequent iterations, we can optimize the set

union to avoid redundant computations. In general, in the i-th iteration, assuming the appended vertex is v_i from the previous iteration, the calculation can be formulated as Equation 6.

$$N(v) \cup N(v_i) - \cup_{u \in RW \setminus \{v_i\}} N(u) \qquad (6)$$

In addition, we calculate the intersection instead of the union result. This enables efficient calculation by traversing the vertices in $N(N(v_i))$. The optimized layout reformatting algorithm is illustrated in Algorithm 4. 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* using the formula $\frac{N(v) + \Sigma_{u \in RW} |N(u)|}{|N(v)| + |\cup_{u \in RW} N(u))| - |N(v) \cap (\cup_{u \in RW} N(u))|}$. Subsequently, the vertex with the highest *computation intensity* is appended to the row window (line 15). At the end of each iteration, the intermediate results are updated in lines 16-19.

VI. Overall Consideration

In § IV and § V, we present a novel SpMM kernel that combines CUDA cores and Tensor cores, along with a specialized algorithm for adjusting graph layouts. Now, in this section, we will integrate these components into the entire training pipeline and conduct a holistic optimization process, aiming to maximize the efficiency and performance of the overall GNN training process.

A. Kernel Fusion

Existing implementations, such as DGL [29] and TC-GNN [33], 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[7].

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 [14] and the forward propagation of GIN [38]. 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. Implementations 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 an embedding 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 embedding matrix stored in global memory.

Generalization. In scenarios where the embedding dimension is not a multiple of 32, using one 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 9(a).

Memory Management. Optimizing memory access in the SpMM kernel on CUDA cores commonly involves storing the embedding 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[8], 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.

---

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

[8]Loading all the data 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.
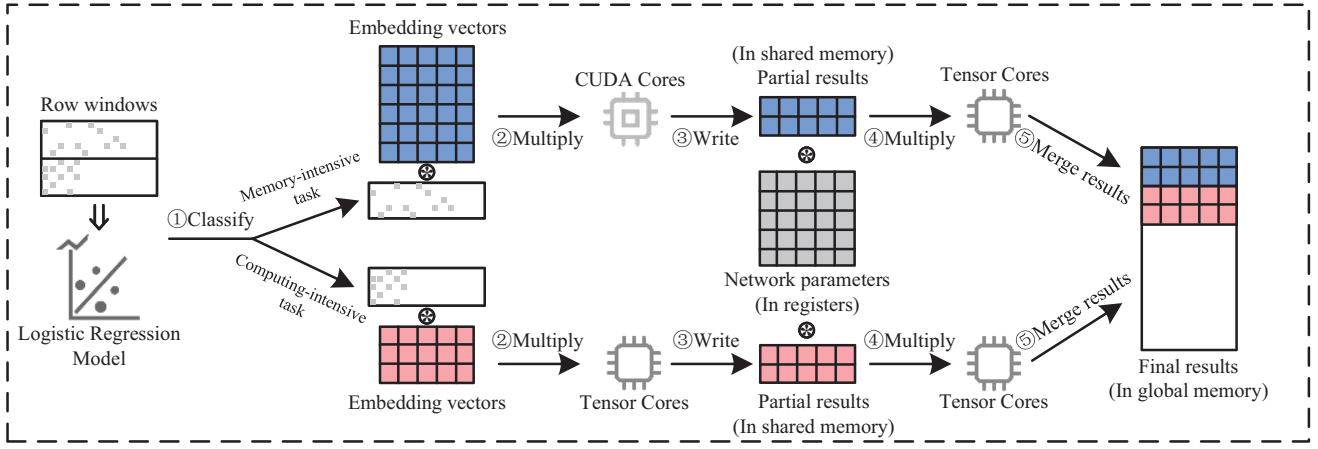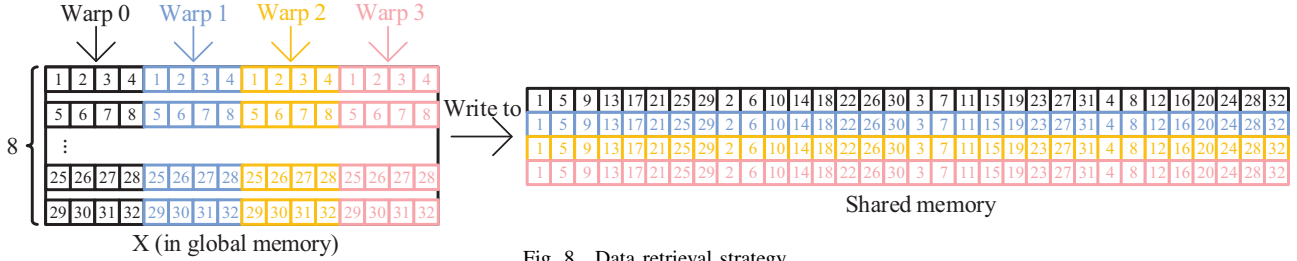
Fig. 7. Process in a GNN layer with kernel fusion.
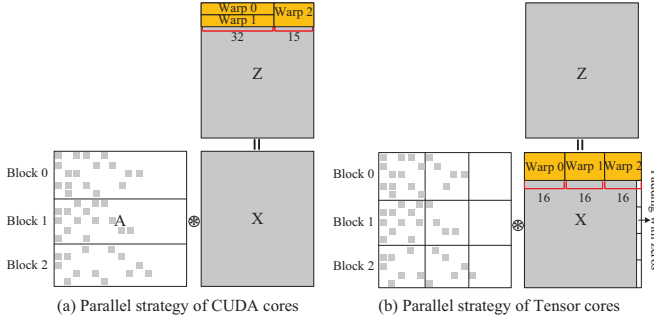

Fig. 8. Data retrieval strategy.

(a) Parallel strategy of CUDA cores
(b) Parallel strategy of Tensor cores

Fig. 9. Parallel strategy of different cores.

C. Implementations on Tensor Cores

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

Specifically, as illustrated in Figure 8, rather than retrieving 2 or 1 row, a warp is utilized to fetch 8 rows of an 8 matrix, with each row containing 4 elements. 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.

Additionally, We adopt the parallel strategy for SpMM on Tensor cores as presented in [33], which is illustrated in Figure 9(b). Following collaboratively loading the required data from X by warps within a thread block, 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 the GPU and enhances computational efficiency.

VII. EXPERIMENTS

In this section, we evaluate HyGNN 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 B.

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 [8], [9], [29], [33]. All the datasets are accessible at SNAP[9], TUDataset[10] and KONECT[11]. Table II summarizes the properties of the 13 datasets. Most datasets do not have information on the number of classes, so we uniformly use 22.

Baselines. We compare HyGNN with existing methods from two aspects: SpMM and GNN forward/backward propagation.

1) SpMM kernels: Sputnik [10] is a library developed by Google, which provides the state-of-the-art unstructured SpMM kernel for full precision on CUDA cores. GE-SpMM [13] is another efficient SpMM kernel on CUDA cores specifically designed for SpMM operation in GNN. cuSPARSE [20] 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 [33] is a method to use Tensor

---

[9]snap.stanford.edu/data/index.html

[10]chrsmrrs.github.io/datasets/docs/datasets

[11]konect.cc/networks

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

cores to accelerate the full-precision unstructured SpMM operation in GNNs. DTC-SpMM [8] is the state-of-the-art SpMM kernel using Tensor cores. Notably, solely employing a certain type of cores in HyGNN yields comparable or even worse performance to the aforementioned methods.

2) GNN training frameworks: We conduct end-to-end experiments and compare the frameworks specifically targeting the optimization of SpMM. Due to the calculation of the GNN method remaining unchanged, the training results of these frameworks are identical. Other frameworks, such as [32], [45], are not considered in experiments, because they are orthogonal to our work. GE-SpMM and TC-GNN integrate the optimized SpMM kernels into PyTorch. These two frameworks are state-of-the-art works for accelerating GNN by optimizing SpMM on CUDA and Tensor cores, respectively. As the efficiency of GE-SpMM and TC-GNN is much more significant[12] 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 HyGNN in C++ under Nvidia CUDA 12.2 and integrate it into PyTorch 1.8.

B. Comparison with SpMM kernels

To substantiate the efficacy of our methodology, we engage in a comprehensive comparative analysis of our proposed SpMM kernel integrated within HyGNN against SOTA 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, a lightweight profiling tool. When profiling the program, there exists a bit of overhead before calling and launching the kernel introduced by nvprof, while no overhead is incurred when executing the kernel. The absolute numbers of the execution time are detailed in Appendix B1. Our results show that HyGNN consistently outperforms all compared methods across all datasets. On average, HyGNN achieves speed improvements of $4.94\times$ over cuSPARSE, $1.33\times$ over Sputnik, $1.40\times$ over GE-SpMM, $3.01\times$ over TC-GNN and

---

[12]TC-GNN achieves $1.7\times$ speedup on average over DGL, while Ge-SpMM brings up to $3.7\times$ speedup over PyG.

$1.56\times$ over DTC-SpMM. Notably, in the best cases, HyGNN achieves a remarkable speedup of $19.56\times$ over cuSPARSE on *DP*, $1.57\times$ speedup over Sputnik on *TT*, $1.57\times$ speedup over GE-SpMM on *RD*, $6.76\times$ and $3.00\times$ speedup over TC-GNN and DTC-SpMM on *PM*. We also compare HyGNN with the SpMM kernel in PyTorch executed on CPU, which achieves an average speedup of $183.77\times$. These results underscore the effectiveness of leveraging hybrid cores to perform SpMM. HyGNN 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 HyGNN has no results merged costs because of the combination strategy proposed in § IV-A even if we employ two cores for calculation simultaneously.

The speedup ratios of HyGNN vary across different datasets when contrasted against state-of-the-art methods. Compared with Sputnik and GE-SpMM which employ CUDA cores for computation, the improvements of HyGNN 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, HyGNN, 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 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 B3.

C. 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 the forward and backward propagation stages in GNNs. To achieve this goal, we integrate our SpMM kernel into PyTorch and compare its performance with two efficient GNN training frameworks. Specifically, We implement GCN [14] and GIN [38] 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
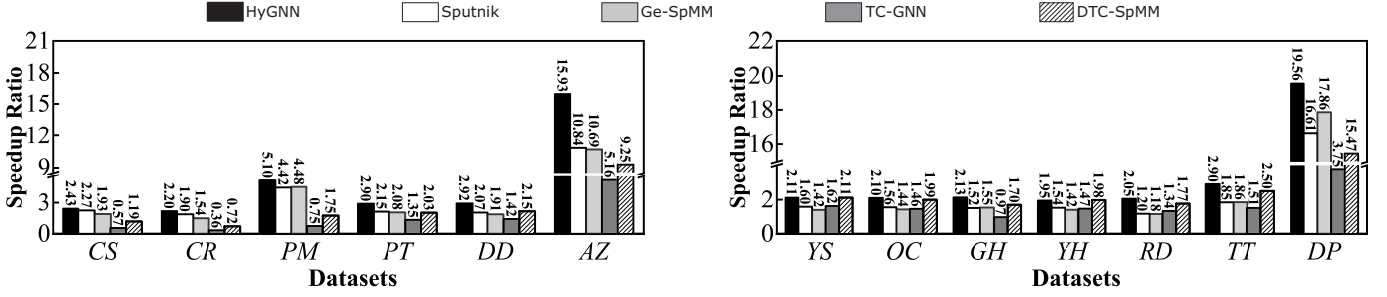
Fig. 10. Overall performance of SpMM kernels.

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*. All the reported time below is the average execution time of one epoch unless noted otherwise. Because of the space constraints and similar experimental outcomes, we report the performance of HyGNN on 5 representative datasets with a large number of edges. The absolute numbers of the execution duration are reported in Appendix B2.

Figure 11(a) illustrates the performance of GNN training frameworks in forward propagation of GCN. Overall, HyGNN outperforms GE-SpMM and TC-GNN across all datasets. Specifically, HyGNN 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 HyGNN in forward propagation is similar to that of SpMM in § VII-B. The acceleration of HyGNN becomes more pronounced for large graphs, such as *RD* and *TT*. As previously discussed, there are more dense parts in large graphs and Tensor cores are efficient on the dense parts.

Figure 11(b) illustrates the performance of backward propagation of GCN among the three methods. It is evident that HyGNN demonstrates the best performance in all instances. In general, HyGNN achieves a 1.48× speedup over TC-GNN and a 1.33× speedup over GE-SpMM on average. Notably, HyGNN 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.

Regarding GIN, Figure 12 depicts the results. On average, HyGNN 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× speedup in the forward and backward propagation on average. The performance is similar to GCN, hence warranting no further in-depth elucidation.

D. Layout optimization

In this section, we evaluate the effectiveness of our proposed layout optimization algorithm LOA.

The improvements brought by LOA are depicted in Figure 13. On average, LOA achieves a speedup of 8.40% over
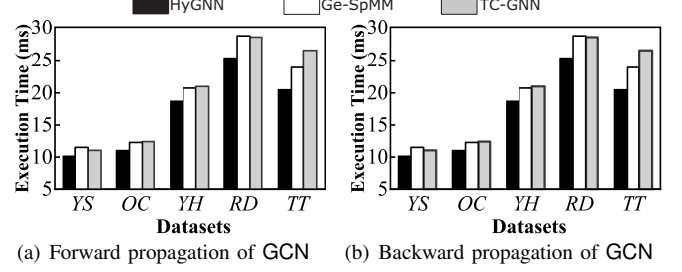

(a) Forward propagation of GCN   (b) Backward propagation of GCN

Fig. 11. Comparison of GCN propagation.

(a) Forward propagation of GIN   (b) Backward propagation of GIN
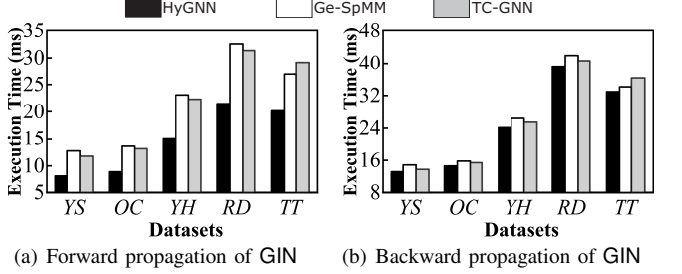
Fig. 12. Comparison of GIN propagation.

HyGNN without LOA 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* can be 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 intelligently adjusts the original layout based on the metric named *computing intensity*, 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

Fig. 13. Improvements of layout optimization.

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

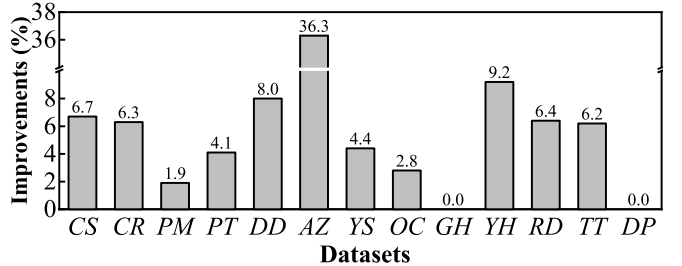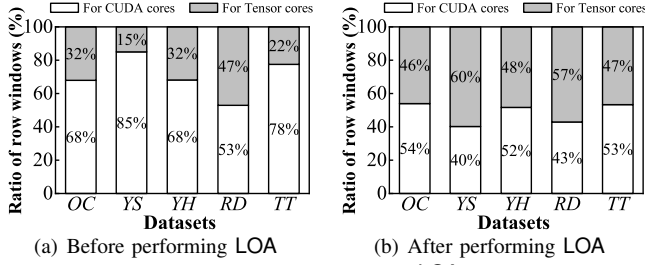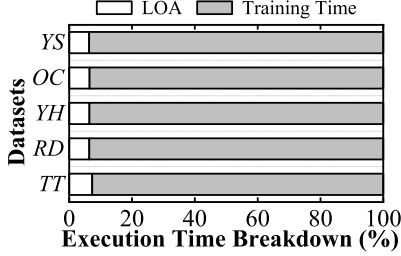Fig. 14. Effectiveness of LOA.

Fig. 15. Overhead of LOA in the entire training.

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%

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%

Kernel Fusion. We assess the execution time of a single GNN layer during backward propagation with and without kernel fusion. The results are illustrated in Table III. 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 § VI-C, 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.

Generalization of SpMM on CUDA Cores. 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 IV 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.

Memory Management of SpMM on CUDA Cores. In § VI-B, shared memory is used to store the column indices of CSR. We evaluate the effectiveness of this technique and report the results in Table V. 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.

Data Loading Strategy of SpMM on Tensor Cores. Finally, we conduct experiments to validate the effectiveness of the data loading strategy. In this evaluation, we only record the calculation time of Tensor cores. Table VI 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. It is worth noting that although our data loading strategy enhances performance, data loading remains the bottleneck of SpMM on Tensor cores

Tensor cores and CUDA cores before and after employing LOA. An increase in the count of row windows suitable 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 14. 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, which leads to more row windows suitable for CUDA cores. However, CUDA cores are less efficient than Tensor cores. We can improve the graph layout with little overhead by leveraging LOA. As shown in Figure 14(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 15 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. Note that the execution of LOA is offline, which has constant overhead regardless of the number of epochs and layers. Conversely, the benefit of LOA in accelerating SpMM will accumulate as the epochs and layers 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[13]. In such scenarios, we advocate for utilizing LOA.

E. Evaluation of the overall considerations

In this section, we evaluate the effectiveness of the techniques proposed in § VI.

---

[13]For example, the overhead of LOA accounts for only about 3% of the training time in the case of 400 epochs, while the speedup ratio of LOA remains unchanged.

TABLE V
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 VI
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%

and requires further exploration.

VIII. CONCLUSION

We present HyGNN, a novel algorithm for accelerating the SpMM operation and GNN training. Being the first GPU accelerator to utilize hybrid GPU cores, i.e., CUDA and Tensor cores, HyGNN offers pioneering ideas that lay the foundation for future research. It intelligently selects hardware structures suitable for submatrices, fully leveraging the computational characteristics of different hardware structures of GPUs. We further devise LOA to improve the graph layout, optimizing it for Tensor cores and releasing the computational potential of Tensor cores. In the integration of HyGNN into the training pipeline, we propose a kernel fusion technique, memory management method, and data loading strategy to further optimize the training process. We integrate HyGNN into PyTorch, and the experimental results consistently demonstrate its superiority over existing SpMM and GNN training frameworks.

REFERENCES

[1] A. Buluc and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP*, pages 503–510, 2008.
[2] 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.
[3] 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.
[4] 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.
[5] G. V. Demirci, A. Haldar, and H. Ferhatosmanoglu. Scalable graph convolutional network training on distributed-memory systems. *PVLDB*, 16(4):711–724, 2023.
[6] 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.
[7] R. Fan, W. Wang, and X. Chu. Fast sparse gpu kernels for accelerated training of graph neural networks. In *IPDPS*, pages 501–511, 2023.
[8] 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.
[9] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop*, 2019.
[10] T. Gale, M. Zaharia, C. Young, and E. Elsen. Sparse gpu kernels for deep learning. In *SC*, pages 1–14, 2020.
[11] 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.
[12] 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.
[13] 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.
[14] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.
[15] C.-T. Li, Y.-C. Tsai, and J. C. Liao. Graph neural networks for tabular data learning. In *ICDE*, pages 3589–3592, 2023.
[16] S. Li, K. Osawa, and T. Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC*, pages 1–15, 2022.
[17] 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.
[18] H. Lin, M. Yan, X. Ye, D. Fan, S. Pan, W. Chen, and Y. Xie. A comprehensive survey on distributed training of graph neural networks. *Proceedings of the IEEE*, 111(12):1572–1606, 2023.
[19] 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.
[20] Nvidia. Cuda sparse matrix library. 2023. https://developer.nvidia.com/cusparse.
[21] Nvidia. Dense linear algebra on gpus. 2023. https://developer.nvidia.com/cublas.
[22] T. Nvidia. V100 gpu architecture: The world's most advanced datacenter gpu. *NVIDIA Corporation*, 2017.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] Q. Wang, Y. Zhang, H. Wang, C. Chen, X. Zhang, and G. Yu. Neutron-star: distributed gnn training with hybrid dependency management. In *SIGMOD*, pages 1301–1315, 2022.
[31] Y. Wang, B. Feng, and Y. Ding. Qgtc: accelerating quantized graph neural networks via gpu tensor core. In *PPoPP*, pages 107–119, 2022.
[32] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. Gnnad-visor: An adaptive and efficient runtime system for gnn acceleration on gpus. In *OSDI*, pages 515–531, 2021.
[33] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding. Tc-gnn: Bridging sparse gnn computation and dense tensor cores on gpus. In *USENIX ATC*, pages 149–164, 2023.
[34] Z. Wen, Y. Fang, and Z. Liu. Meta-inductive node classification across graphs. In *SIGIR*, pages 1219–1228, 2021.
[35] 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.
[36] 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.

[37] 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.

[38] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

[39] 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.

[40] 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.

[41] C. Yang, A. Buluç, and J. D. Owens. Design principles for sparse matrix multiplication on the gpu. In *Euro-Par*, pages 672–687, 2018.

[42] 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.

[43] J. Zhang, C. Gao, D. Jin, and Y. Li. Group-buying recommendation for social e-commerce. In *ICDE*, pages 1536–1547, 2021.

[44] 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.

[45] 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.

[46] Y. Zhang and A. Kumar. Lotan: Bridging the gap between gnns and scalable graph analytics engines. *PVLDB*, 16(11):2728–2741, 2023.

[47] 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.

[48] Z. Zhang, P. Cui, and W. Zhu. Deep learning on graphs: A survey. *TKDE*, page 249–270, 2022.

[49] Z. Zhang, H. Wang, S. Han, and W. J. Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *HPCA*, pages 261–274, 2020.

[50] C. Zheng, H. Chen, Y. Cheng, Z. Song, Y. Wu, C. Li, J. Cheng, H. Yang, and S. Zhang. Bytegnn: efficient graph neural network training at large scale. *PVLDB*, 15(6):1228–1242, 2022.

[51] 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.

[52] 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. Details of the Logistic Regression Model

The training pipeline of the logistic regression model in § IV-B consists of 4 procedures: (1) sparse matrices generating, (2) execution results collecting, (3) model training, and (4) model encoding.

Sparse matrices generating. In this step, we generate a set of matrices for the evaluation of GPU cores. As mentioned in § IV, the logistics regression model is responsible for identifying the appropriate GPU cores for each row window, which has 16 rows. Therefore, the number of rows for the generated matrices is set to 16. The columns of the generated matrices range from 1 to 120. For each column, we ensure that there is at least one non-zero element. As a result, the sparsity of matrices ranges from $\frac{11}{16}$ to $\frac{15}{16}$, and the corresponding number of non-zero elements ranges from $\#columns$ to $\#columns \times 5$. To ensure each column is a non-zero column, we first generate an element for each column, the position of the row is generated by a uniform random number function. Subsequently, the row and column positions of the remaining elements are generated randomly by the uniform random number function.

TABLE VII
OVERHEAD OF SpMM ON DIFFERENT GPUs. ($\times 10^{-6}$ s)

		HyGNN	Sputnik	Ge-SpMM	TC-GNN	DTC-SpMM	cuSPAESE
CS	3090	**5.25**	5.63	6.62	22.40	10.73	12.77
	4090	**3.93**	4.06	4.19	18.08	8.45	9.89
	A100	**10.66**	14.30	15.43	56.07	27.07	23.07
CR	3090	**6.05**	6.98	8.64	6.98	18.33	13.28
	4090	**4.83**	5.44	5.98	29.92	12.80	9.76
	A100	**13.76**	17.44	20.83	76.39	42.11	25.02
PM	3090	**11.62**	13.41	13.22	78.53	33.79	59.23
	4090	7.87	7.46	**7.20**	72.48	26.72	47.52
	A100	**15.43**	25.02	21.54	211.27	84.26	90.91
PT	3090	**17.76**	23.90	24.70	38.21	25.40	51.49
	4090	11.23	**7.84**	8.90	25.22	12.00	27.71
	A100	**17.82**	23.46	27.87	59.11	32.48	61.25
DD	3090	**121.57**	171.42	185.98	249.98	164.76	354.85
	4090	73.54	**67.68**	106.50	138.27	112.90	249.54
	A100	178.79	**155.78**	267.30	433.26	202.24	509.10
AZ	3090	**240.67**	353.57	358.53	743.17	414.25	3833.33
	4090	**112.61**	118.37	584.84	293.83	171.62	856.68
	A100	**271.97**	292.26	384.36	1051.22	471.43	1081.33
YS	3090	581.41	769.09	866.75	756.80	**581.36**	1226.88
	4090	**461.51**	479.24	585.10	548.04	498.15	1048.91
	A100	743.82	790.67	1181.46	1211.83	**669.47**	1286.10
OC	3090	**624.58**	841.21	909.57	899.20	660.56	1313.76
	4090	**505.99**	533.29	634.89	627.75	577.55	1178.73
	A100	825.17	851.28	1000.53	1451.10	**786.56**	1471.58
GH	3090	**568.41**	850.59	836.10	1330.56	764.60	1296.15
	4090	**441.32**	475.97	549.74	806.22	534.95	1104.04
	A100	**761.20**	1046.29	1102.03	2221.16	811.76	1863.42
YH	3090	**1045.92**	1395.39	1510.75	1461.47	1085.29	2145.85
	4090	**869.32**	917.48	1038.70	1045.17	954.03	1901.68
	A100	**1032.05**	1431.06	2011.01	2073.99	1129.12	2502.35
RD	3090	**1574.69**	2441.08	2469.04	2182.26	1651.71	2917.27
	4090	**1436.98**	1484.53	1639.77	1597.62	1474.62	2945.31
	A100	2013.03	2617.16	2870.19	3167.70	**1846.44**	2856.08
TT	3090	**1382.53**	2164.82	2153.66	2426.94	1601.94	4003.09
	4090	**1126.61**	1276.56	1347.54	1541.43	1231.86	3109.70
	A100	**1880.32**	2245.99	2918.77	2073.99	1963.72	5752.35
DP	3090	**16718.30**	19696.12	18312.22	87216.13	21148.42	327079.96
	4090	11350.72	12061.82	**11206.81**	95456.71	14240.56	169181.07
	A100	**13778.36**	16926.84	15532.93	178816.99	15808.13	145479.16

Execution results collecting. Using the matrices generated in the last step, we execute the SpMM kernel on Tensor cores and CUDA cores, respectively. For each matrix, we run 100 times to evaluate the average execution time, which is measured by the Nvidia profiling tool named nvprof. Note that the kernels are the same as the deployed kernel in SpMM, with the same parameter settings.

Model training. We employ the logistics regression model API provided by Sklearn. Each matrix in the previous step is a sample of the training data. The sample consists of two features, namely, the sparsity and the number of non-zero columns. As mentioned in § IV-B, these two features dominate the computing and memory access respectively, which represent the primary difference between CUDA and Tensor cores. 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, we set the label as 1. Otherwise, the label is 0. Then we train the model offline until it converges.

Model encoding. Finally, we extract the coefficients of the logistics regression model and hard encode them into the cores selecting function. The selection will be efficient because it only involves a few addition and multiplication operations.

B. Supplementary Experimental Results

1) Absolute Numbers of SpMM kernels

In this section, we list the absolute numbers related to the execution times of SpMM kernels in Table VII. To evaluate the generality of the logistic regression model, we also compare the SpMM kernels on Nvidia A100 and RTX 4090 GPUs. Our proposed SpMM kernel is superior to other SpMM kernels

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

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

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

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

in most cases, and the performance of the logistic regression model is stable on different types of GPUs.

2) Absolute Numbers of End-to-end Training

In this section, we report the absolute numbers of the training overhead of GCN in Table VIII (corresponding to Figure 11) and GIN in Table IX (corresponding to Figure 12).

3) Additional Experiments

To verify the adaptability of HyGNN 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

TABLE X
EXECUTION TIMES OF SPMM KERNELS ON MATRICES WITH VARIOUS
SPARSITY. ($\times 10^{-6}$ s)

Methods \ Sparsity	80%	85%	90%	95%
HyGNN	**7.49**	**6.62**	**5.73**	**5.31**
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

in Table X. HyGNN achieves the best performance on all synthetic matrices. It is worth noting that 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)—otherwise, the opposite.