ThunderGNN: Unlocking Tensor Cores for Graph Neural Networks

YuAng Chen
The Chinese University of Hong Kong
yuangchen@cuhk.edu.cn

Siyi Teng
The Chinese University of Hong Kong
syteng@se.cuhk.edu.hk

Wenqi Zeng
Hong Kong University of Science and Technology
wzengad@connect.ust.hk

Jeffrey Xu Yu
Hong Kong University of Science and Technology (Guangzhou)
jeffreyxuyu@hkust-gz.edu.cn

ABSTRACT

Graph Neural Networks (GNNs) have emerged as the state-of-the-art methodology for learning on graph-structured data, yet their performance is severely constrained by a fundamental mismatch between irregular graph sparsity and the rigid parallelism of modern hardware. While modern GPUs rely on Tensor Cores (TCs) to deliver massive computational throughput, these units demand strictly tiled, dense inputs—a requirement that conflicts with the extreme sparsity of real-world graphs. Existing frameworks fail to resolve this design conflict: they either fallback to legacy SIMT cores, leaving TCs underutilized, or they incur prohibitive memory bloat by forcing sparse data into dense tiles via excessive padding. To bridge this gap, we propose ThunderGNN, a hardware-aware acceleration system designed to reconcile graph irregularity with Tensor Core rigidity. ThunderGNN employs a unified co-design strategy comprising three key optimizations: (1) a sparsity-aware reordering algorithm that logically groups graph rows to maximize local density; (2) a Condensed Binarized Abstraction (CBA) storage layout that physically organizes the adjacency matrix into TC-aligned blocks without explicit zero-padding; and (3) a hardware-aware execution engine that efficiently streams compressed blocks directly into TCs. Extensive experiments on NVIDIA A100 GPUs demonstrate that ThunderGNN significantly outperforms state-of-the-art systems, achieving geometric mean speedups of 1.89× over DGL and 2.59× over PyG.

PVLDB Reference Format:
YuAng Chen, Siyi Teng, Wenqi Zeng, and Jeffrey Xu Yu. ThunderGNN: Unlocking Tensor Cores for Graph Neural Networks. PVLDB, 19(10): 2699 - 2712, 2026.
doi:10.14778/3828612.3828625

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yuang-chen/ThunderGNN-VLDB2026.

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828625

1 INTRODUCTION

Graph Neural Networks (GNNs) have emerged as the state-of-the-art approach for learning on graph-structured data [21, 31, 35]. Their ability to model complex dependencies has driven adoption across diverse domains, including social network analysis [16], bioinformatics [13, 14], and recommendation systems [37, 41]. However, as real-world datasets grow increasingly massive and irregular, GNN processing has evolved from a computational challenge into a fundamental data management problem [42].

To address the performance demands of these workloads, prior research has primarily focused on scale-out strategies, such as optimizing inter-GPU data movement [24, 46] or employing execution planning to hide CPU-GPU latency [23]. Furthermore, techniques such as graph reordering [27, 39] and software-managed caches [29, 45] have been developed to improve locality. Nevertheless, most existing solutions continue to build upon mainstream frameworks like DGL [36] and PyG [12]. These frameworks effectively treat the GPU as a "black box," relying on legacy execution models that offload computations to general-purpose SIMT cores (i.e., CUDA cores). By failing to adapt to the underlying architecture, these systems miss significant opportunities for hardware-aware optimization.

This reliance on legacy models overlooks a critical paradigm shift in hardware architecture. Modern accelerators now rely on Tensor Cores (TCs) [1, 22], which are specialized units that deliver orders-of-magnitude higher throughput than SIMT cores. Yet, these specialized units impose a rigid constraint: they require strictly tiled, dense input layouts to function. This technological divergence creates a fundamental mismatch: while the hardware demands dense, structured regularity, graph data is inherently sparse and irregular. Consequently, because conventional GNN methods cannot naturally satisfy these rigid hardware constraints, they leave the vast majority of the GPU's computational potential largely unused.

This mismatch creates a critical conflict for GNN acceleration. Existing methods (e.g., DGL, PyG) must either ignore TCs, which wastes the GPU's immense computational potential. Or, they force data into dense tiles via excessive zero-padding, which creates prohibitive memory bandwidth bottlenecks. While recent advances in Large Language Models (LLMs), such as Flash-LLM [40], have successfully leveraged TCs for sparse computation, they operate in a regime of moderate sparsity (60–90%) using naive tiling strategies. These methods are ill-suited for graph data, which exhibits extreme, irregular sparsity (often exceeding 99%). Our analyses in Section 3 confirms that conventional approaches fail to reconcile these

Table 1: Comparison of GNN acceleration approaches. ThunderGNN leverages Tensor Cores for high-sparsity graphs with low storage overhead

	Work	Domain	Compute Unit	Supported Primitives	Graph Scale	Target Sparsity	Memory Overhead
SIMT-based Baselines (Legacy)	PyG [36]	GNN	SIMT Cores	Scatter, Gather	Large	~99%	Low
	DGL [12]	GNN	SIMT Cores	SpMM, SDDMM	Large	~99%	Low
Tensor Core-based Approaches	TC-GNN [38]	GNN	Tensor Cores	SpMM, SDDMM	Medium	~99%	Medium
	SMaT [28]	HPC	Tensor Cores	SpMM	Medium	~99%	High
	Flash-LLM [40]	LLM	Tensor Cores	SpMM	Small	60–90%	Low
	ThunderGNN (Ours)	**GNN**	**Tensor Cores**	**SpMM, SDDMM**	**Large**	**~99%**	**Low**

conflicting objectives: they either cannot meet the density requirements to saturate TCs or they incur memory expansion costs that negate any computational gains. Table 1 summarizes the design trade-offs of existing GNN and TC-based acceleration approaches and highlights the gap addressed by ThunderGNN.

To bridge this gap, we propose ThunderGNN, a hardware-aware acceleration system designed to reconcile the conflict between the rigid tiling requirements of TCs and the irregular sparsity of graphs. Unlike prior works that prioritize one metric at the expense of another, ThunderGNN employs a unified co-design strategy through a three-stage pipeline that transforms the graph data and operations to fit the hardware. Firstly, a similarity-driven reordering algorithm logically groups graph rows to maximize local density, creating the structural regularity necessary for efficient tiling. Then, this logical structure is mapped to Condensed Binarized Abstraction (CBA), a novel physical storage that organizes the matrix into TC-aligned blocks without explicit zero padding. Finally, a cache-aware execution engine streams these compressed micro-tiles directly into TCs, with improved cache efficiency and compute throughput.

We focus specifically on intra-GPU optimization to maximize the throughput of the execution engine itself. By solving the fundamental data representation and execution problems on a single device, ThunderGNN provides a highly optimized compute backend that is orthogonal to, and can serve as the foundation for, distributed scale-out systems. However, the integration of ThunderGNN into distributed frameworks is beyond the scope of this paper and is left for future work.

To summarize, we make the following key contributions:

- **Problem Exploration:** We develop in-depth experimental analyses to characterize the mapping of sparse operations to TCs. This identifies the mismatch between tile density and memory footprint as the primary bottleneck for existing methods.
- **Novel GNN System:** We propose ThunderGNN, a unified framework that encapsulates a similarity-driven reordering algorithm, a compact, TC-aligned storage scheme (CBA), and a cache-aware execution engine.
- **Performance Gain:** ThunderGNN outperforms state-of-the-art systems, achieving geometric mean speedups of 1.89× over DGL and 2.59× over PyG across representation GNN models on real-world graphs.

2 BACKGROUND

2.1 Graph Neural Networks

Let $G = (V, E)$ be a graph with $n = |V|$ nodes and $m = |E|$ edges. Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ denote the adjacency matrix and $\mathbf{X} \in \mathbb{R}^{n \times d}$ denote the node feature matrix, where d is the feature dimension. A GNN layer updates node features through two core operations: *message computation* and *node aggregation*.

First, the *message computation* function ψ calculates a scalar coefficient (i.e., message) $s_{vu}^{(k)}$ for the edge connecting node v and neighbor u. This coefficient is derived from the node features and edge attributes a_{vu}:

$$s_{vu}^{(k)} = \psi(\mathbf{x}_v^{(k)}, \mathbf{x}_u^{(k)}, a_{vu}) \tag{1}$$

These coefficients $s_{vu}^{(k)}$ collectively form the non-zero elements of a sparse score matrix $\mathbf{S}$, where non-zeros correspond to edges in E.

Second, the *node aggregation* function ϕ updates each node's representation by performing a weighted summation of neighbor features using the computed coefficients. This produces the aggregated feature vector $\mathbf{z}_v^{(k)}$:

$$\mathbf{z}_v^{(k)} = \phi\left(\sum_{u \in \mathcal{N}(v)} s_{vu}^{(k)} \cdot \mathbf{x}_u^{(k)}\right) \tag{2}$$

This operation corresponds to the matrix multiplication $\mathbf{Z} = \mathbf{S} \times \mathbf{X}$. The aggregated features are then transformed by MLPs before passing to the next layer. As these neural operations incur negligible cost compared to the graph operations [17, 38], we omit them from our analysis.

2.2 Matrix Multiplication in GNNs

To facilitate GPU acceleration, the two core GNN operations are often mapped to sparse matrix primitives.

Message Computation. First, the message function ψ (used in models like GAT [33, 35]) corresponds to *Sampled Dense-Dense Matrix Multiplication* (SDDMM). This operation computes the sparse edge coefficients (scores) based on the interaction of node features. Formally, it generates a sparse score matrix $\mathbf{S}$ via:

$$\mathbf{S} = (\mathbf{X}\mathbf{X}^T) \odot \mathbf{A} \tag{3}$$

Here, $\odot$ denotes the Hadamard product, which samples the dense dot-product results ($\mathbf{X}\mathbf{X}^T$) only at the indices of non-zero elements in the adjacency matrix $\mathbf{A}$.

Node Aggregation. Second, the node aggregation function ϕ corresponds to *Sparse Matrix-Matrix Multiplication* (SpMM). This operation updates node features by aggregating neighbor information. It computes the product of a sparse matrix (either the raw adjacency $\mathbf{A}$ or the dynamic scores $\mathbf{S}$ computed above) and the dense feature matrix $\mathbf{X}$ to produce the output $\mathbf{Z}$:

$$\mathbf{Z} = \mathbf{S} \times \mathbf{X} \tag{4}$$

2.3 GNN Execution on GPUs

GPUs are the de facto hardware for GNN workloads due to their massive parallelism. Existing frameworks typically deploy SpMM and SDDMM on general-purpose *SIMT cores* (i.e., CUDA Cores) [9, 10, 12, 36]. These cores excel at scalar, element-level computations, making them flexible enough to handle the sparse, irregular operations inherent in graphs.

In contrast, modern NVIDIA GPUs feature *Tensor Cores (TCs)*, specialized units designed to accelerate matrix multiplication with orders-of-magnitude higher throughput than SIMT cores (e.g., 16× speedup on A100 [22]). However, these units impose rigid hardware constraints: they require dense, structured input tiles (e.g., 16×16) to be loaded into *fragments* (specialized register files) for operation. This rigid tile-level granularity is fundamentally incompatible with the sparse, unstructured nature of graph data. Consequently, conventional GNN approaches fail to utilize these powerful accelerators, leaving significant computational throughput untapped.

3 PROBLEM ANALYSIS AND MOTIVATION

The transition of GNNs from academic exploration to large-scale industrial deployment has coincided with a fundamental paradigm shift in hardware architecture. While GNN models have grown in complexity and depth, the hardware accelerators designed to run them have become increasingly specialized. This divergence has created a critical impedance mismatch: the irregular, hypersparse structure of graph data is inherently at odds with the rigid, dense parallelism of modern hardware accelerators.

In this section, we analyze this conflict, demonstrating that existing acceleration paradigms fail to resolve the tension between hardware rigidity and data sparsity. We identify a "utilization gap" where current systems must choose between underutilizing compute units or overwhelming memory bandwidth, motivating the need for a holistic hardware-software co-design.

3.1 Mismatch: Hardware vs. Data

Modern GPU architectures have evolved to prioritize dense matrix multiplication, the computational workhorse of Deep Learning. Specialized units, such as NVIDIA's TCs on the A100 GPU, deliver orders-of-magnitude higher throughput than general-purpose CUDA cores. However, this performance comes with strict micro-architectural constraints: TCs typically operate on fixed-size micro-tiles (e.g., 16×16 fragments) and require data to be loaded in dense, contiguous blocks. To achieve peak efficiency, the input data must exhibit high structural regularity, allowing for coalesced memory access and warp-synchronous execution.

Real-world graphs, conversely, are defined by irregular sparsity. Datasets such as social networks, web graphs, or protein interactions often exhibit sparsity levels exceeding 99%. Furthermore, the non-zero elements (edges) are scattered unpredictably across the adjacency matrix, following power-law distributions rather than uniform patterns. This creates a structural conflict:

- **The Hardware** demands dense, regular 16×16 tiles to saturate its computing units.
- **The Data** provides scattered, individual non-zeros that rarely fill a tile.

Bridging this gap requires transforming the sparse data into a format the hardware can accept. However, the transformation leads to a fundamental trade-off between Tile Density (computational efficiency) and Memory Footprint (memory efficiency). If we preserve the natural sparsity, we cannot fill the tiles (low density). If we force the data into tiles, we introduce zeros (high memory overhead).

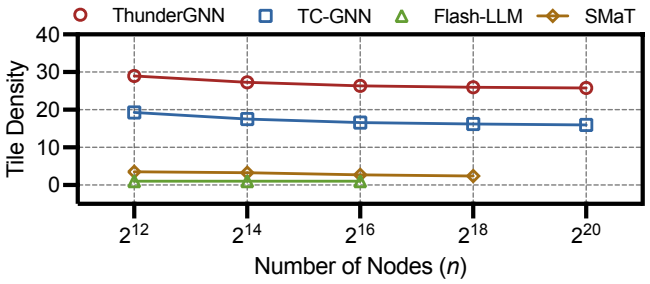


Figure 1: Tile density of different TC-based methods (graph degree $m/n = 20$).

3.2 The Failure of Existing Paradigms

Current GNN systems attempt to navigate this conflict through two primary strategies, both of which result in suboptimal performance because they fail to address the root cause of the mismatch.

The Legacy SIMT Approach (DGL, PyG): Mainstream frameworks like DGL and PyG entirely sidestep the TC architecture. They rely on "legacy" execution models that treat the GPU as a collection of scalar SIMT cores. By using primitives like scatter-gather or standard SpMM on CUDA cores, they maintain high memory efficiency—only storing and processing valid non-zeros.

These approaches leave the GPU's most powerful engine TCs completely idle. As shown in our analysis (Table 1), these frameworks fail to scale on modern hardware because they are bound by the lower throughput of general-purpose cores. They effectively treat a modern AI accelerator as a legacy graphics card, wasting the device's potential.

The Forced Density Approach (Naive Tiling & Padding): To utilize Tensor Cores, recent works such as Flash-LLM [40] adapt sparse data to dense units by partitioning the matrix into tiles and padding missing elements with zeros. While effective for the moderate sparsity (60-80%) seen in LLM pruning, it fails catastrophically for graph data characterized by extreme sparsity (e.g., 99%). For such sparse graphs, a naive tiling strategy results in extreme low tile density and/or prohibitive memory bloat.

We define *tile density* as the average number of nonzero elements per tile sized 16×16, and evaluate representative TC-based methods on power-law Kronecker [2] graphs with varying sizes. As shown in Fig. 1, Flash-LLM's naive tiling yields tile densities close to 1, corresponding to the worst-case scenario for TC utilization,

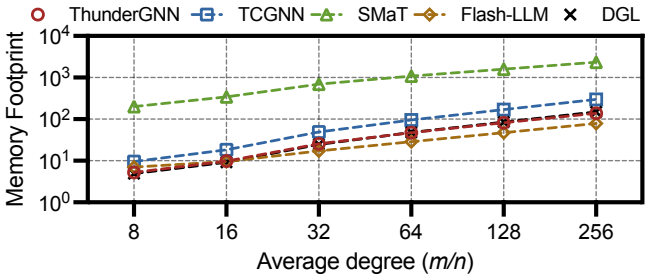


Figure 2: Memory footprint of different methods ($n = 2^{16}$).

rendering TCs severely underutilized. SMaT [28] improves locality through hash-based graph reordering, but achieves only marginal density gains, reaching at most ~3 nonzeros per tile.

Fig. 2 further illustrates the memory footprint incurred by different approaches. DGL serves as the baseline, as it stores only nonzero values in FP16 together with their corresponding indices in INT32, incurring minimal memory overhead. ThunderGNN remains close to DGL, incurring only 1.12× memory on average.

For TC–based tiling approaches, a tile containing a single edge must still materialize a full 16×16 fragment to satisfy TC input requirements, resulting in 255 padded zeros for just one nonzero element. Consequently, as shown in Fig. 2, SMaT incurs substantial memory costs ranging from 15× to 41× relative to DGL, because it explicitly pads sparse tiles with zeros in its storage format.

In contrast, Flash-LLM avoids explicit zero padding by storing only nonzero values and their indices in FP16, leading to low memory overhead. However, this design produces extremely low tile density for highly sparse graphs, severely limiting effective TC utilization. Overall, naive tiling approaches shift the performance bottleneck: TCs either stall while loading large volumes of padded data or operate at very low utilization, effectively nullifying their theoretical compute advantages.

The Intermediate Approach (TC-GNN): TC-GNN [38] attempts to find a middle ground by condensing sparse rows into dense blocks. Compared to naive padding, TC-GNN increases tile density, achieving approximately 15–19 nonzeros per tile. However, this improvement comes at the cost of complex metadata required to maintain the mapping between logical sparsity patterns and physical dense layouts. As a result, TC-GNN incurs substantial memory overhead, more than doubling costs relative to DGL.

Moreover, TC-GNN adopts a largely hardware-oblivious data management strategy. It buffers entire dense feature matrices in shared memory (L1 cache) despite the fact that these features are consumed once without reuse. As feature dimensionality grows, this design substantially increases pressure on the on-chip memory hierarchy, leading to frequent cache thrashing and reduced effective bandwidth. Thus, the lack of explicit control over data movement becomes an additional performance bottleneck, limiting TC-GNN's ability to fully exploit TCs even when tile density is improved.

3.3 The Necessity of Co-Design

This analysis reveals that the problem cannot be solved by hardware adaptation or software optimization in isolation. We face a design conflict:

- **Maximizing Compute Utilization** requires dense tiling (risking memory bloat).
- **Minimizing Memory Footprint** requires sparse formats (risking TC underutilization).

Existing methods occupy the extremes of this spectrum. To resolve this impedance mismatch, we need a system that can synthesize regularity from irregular data. This motivates ThunderGNN, which introduces a unified co-design pipeline.

4 THUNDERGNN DESIGN

To resolve the impedance mismatch between graph irregularity and hardware rigidity, ThunderGNN introduces a hardware-inspired software co-design that systematically reduces graph irregularity and exploits hardware capabilities. Our architecture is built upon three foundational design rationales that move beyond traditional hardware-oblivious optimizations, guiding the development of a unified three-stage pipeline (Fig. 3).

4.1 Design Rationale and System Overview

ThunderGNN's design is guided by three complementary rationales that address hardware–data conflicts in TC-based GNN execution:

Rationale 1: Local Regularity Induction. The first barrier to effective TC utilization is the absence of local structural regularity in sparse graph representations. TCs operate on fixed-size dense fragments (e.g., 16×16), but in real-world graphs, non-zero entries are scattered according to power-law degree distributions. When such matrices are tiled directly, most fragments contain very few non-zeros, resulting in extremely low arithmetic intensity and poor TC utilization.

A natural approach is to reorder the graph to improve locality. However, existing techniques remain insufficient for TC execution, because they prioritize global metrics over local density. Conventional reorderings such as BFS or Gorder [39] optimize global metrics (e.g., bandwidth or diameter) that do not guarantee high non-zero density within fixed-size blocks. SMaT [28] clusters rows with similar sparsity patterns but fails to achieves satisfactory density due to the coarse granularity of hash-based grouping.

Consequently, graphs that are well-ordered under traditional metrics may still produce TC fragments that are sparsely populated and computationally inefficient. This exposes a key insight: *TC performance fundamentally depends on local block density at the fragment granularity, rather global metrics.* ThunderGNN therefore adopts Local Regularity Induction as a core design guideline. It treats the adjacency matrix as a permutable logical layer and explicitly reorders rows to maximize non-zero overlap within TC-aligned windows. By grouping nodes with similar connectivity patterns, ThunderGNN synthesizes dense local regions from globally irregular graphs, creating the structural precondition required for efficient TC execution.

Rationale 2: Representation–Execution Decoupling. Although regularity induction increases local density, it cannot eliminate sparsity entirely. Even after reordering, many TC-aligned fragments remain only partially filled due to the intrinsic irregularity of graph data. Existing TC-based designs tightly couple the logical representation granularity of sparse data with the physical

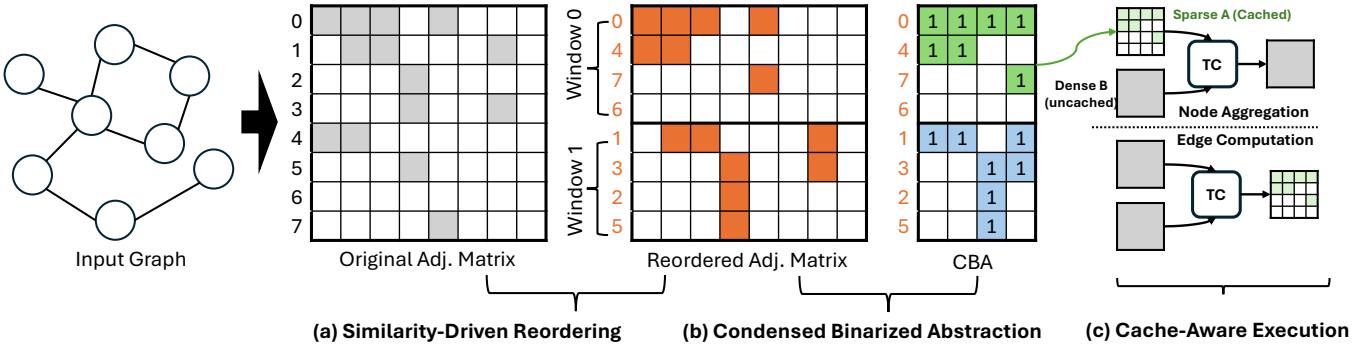


Figure 3: Overview of the ThunderGNN pipeline.

execution granularity of TC fragments: sparse matrices are organized into fixed tiles that match TC fragment size (e.g., Flash-LLM, TC-GNN), while some systems further pad the tiles with zeros (e.g., SMaT). The tight coupling forces sparse regions into rigid, hardware-aligned tiles. As a result, memory footprint and bandwidth consumption scale with fragment size rather than with the true number of non-zeros.

This inefficiency arises because sparse data representation and TC execution impose fundamentally different requirements. *Memory efficiency favors density-adaptive layouts that track actual sparsity, whereas TCs require fixed-size dense fragments for efficient execution.* Binding these two granularities together forces sparse regions to pay for empty space.

ThunderGNN copes with the granularity gap through Representation–Execution Decoupling. Sparse graph data is stored in a compact, density-adaptive logical format, while TC-compatible dense fragments are materialized on demand at execution time. Following this rationale, ThunderGNN introduces the Condensed Binarized Abstraction (CBA), a novel representation that encodes sparse tiles in a compact format and enables effective TC utilization even under extreme sparsity.

Rationale 3: Data Access Control. Even with TC-friendly structure and representation, performance can degrade if data movement is left to hardware-default caching behavior. Graph workloads exhibit asymmetric reuse patterns: structural metadata (e.g., adjacency patterns) is reused across feature dimensions and operations, whereas node features are typically streamed once with little or no temporal locality. Existing GNN frameworks (e.g., TC-GNN) are often hardware-unaware and cache these data types uniformly. This oversight causes *a fundamental mismatch between data reuse patterns and cache behavior*, resulting in cache pollution and reduced effective bandwidth.

ThunderGNN therefore incorporates Data Access Control as a core design requirement. By separating reusable structural metadata from non-reusable feature values, ThunderGNN prioritizes high-reuse structural data in on-chip caches, while allowing feature data to bypass caches and be consumed directly. Synchronizing data movement with reuse pattern is essential for translating induced regularity and compact representations into actual hardware throughput. This rationale leads to a cache-aware execution engine that provides specialized graph operators and fine-grained control over memory placement, optimizing the utilization of on-chip caches and registers.

System Overview. As illustrated in Fig. 3, these three rationales establish a unified pipeline composed by three-stage optimizations. (a) Similarity-driven reordering induces local structural regularity; (b) CBA decouples sparse representation from dense execution; and (c) the cache-aware execution engine orchestrates data movement to preserve both compute efficiency and memory locality. Together, they enable ThunderGNN to reconcile extreme graph sparsity with the rigid execution model of TCs. The remainder of this section details the design of each stage.

4.2 Similarity-Driven Graph Reordering

Guided by the rationale of Regularity Induction, ThunderGNN logically reshapes the graph to expose dense computation opportunities. ThunderGNN instead formulates reordering as a *similarity-driven partitioning problem*, explicitly targeting dense submatrices at the granularity of TC fragments. As outlined in Algo. 1, the process comprises two steps.

Algorithm 1: Similarity-Driven Graph Reordering

Input: Sparse matrix $A \in \mathbb{R}^{n \times n}$, neighbor count k, window size W
Output: Permuted matrix A'
// Step 1: Scalable Similarity Construction
$G_{\text{knn}} \leftarrow \emptyset$;
for $i \leftarrow 0$ **to** $n-1$ **do**
  $N_i \leftarrow \text{ApproxNearestNeighbors}(A[i], k, \text{Jaccard})$;
  $G_{\text{knn}}[i] \leftarrow \{(j, \text{score}) \mid j \in N_i\}$;
// Step 2: Fixed-Window Partitioning
$n_{\text{parts}} \leftarrow n/W$;
$\mathcal{P} \leftarrow \text{BalancedPartitioning}(G_{knn}, W)$;
// Permutation Generation
$\pi \leftarrow Concatenate(\mathcal{P}[0], \mathcal{P}[1], \ldots, \mathcal{P}[n_{\text{parts}} - 1])$;
return $A[\pi, \pi]$;

Step 1: Scalable Similarity Construction. We define the "affinity" between two graph rows using the Jaccard similarity of their non-zero column indices. High similarity implies that if two rows are grouped, their non-zeros will align in the same columns, maximizing the density of the resulting block.

$$J(u,v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \tag{5}$$

A naive implementation would require computing pairwise similarities between all row pairs, incurring $O(n^2)$ time and rendering the approach impractical for large graphs. ThunderGNN avoids this bottleneck by leveraging Nearest Neighbor Descent (NND) to construct an approximate k-nearest-neighbor graph G_{knn} over rows. NND iteratively refines neighbor candidates using local neighbor propagation, converging rapidly in practice while avoiding exhaustive comparisons. This reduces the complexity of similarity graph construction to $O(n^{1.14})$, enabling scalability to graphs with millions of nodes.

Importantly, the approximate nature of G_{knn} is sufficient for our purpose: the reordering only requires locally coherent groupings rather than exact similarity rankings. Empirically, we find that NND preserves enough high-affinity edges to guide effective row clustering, while dramatically reducing preprocessing overhead. Consequently, ThunderGNN achieves a practical balance between similarity fidelity and scalability, making similarity-driven reordering feasible for large-scale GNN workloads.

Step 2: Fixed-Window Partitioning. The similarity graph G_{knn} captures fine-grained structural affinity between rows, but is insufficient to ensure efficient execution of TCs. To translate similarity into performance gains, it must conform to the constraints imposed by TCs, which operate on fixed-size matrix fragments of dimension $F \times F$ (typically 16×16). Consequently, both *which* rows are grouped and *how many* rows are grouped together are critical for TC utilization.

A natural approach would be to apply similarity-based clustering or community detection algorithms, such as Louvain [3], to group highly related rows. While effective at identifying coherent communities, these methods are designed to optimize graph-theoretic objectives and therefore produce clusters with highly variable sizes. Such variability is ill-suited to TC execution: mapping variable-sized clusters onto fixed hardware fragments would require padding, splitting, or merging clusters, all of which erode local density and undermine the benefits of similarity-based grouping.

To avoid this mismatch, ThunderGNN reformulates row reordering as a fixed-window partitioning problem, in which similarity preservation and execution constraints are jointly enforced. Rather than discovering arbitrary communities, we partition the similarity graph G_{knn} into a set of disjoint partitions $\mathcal{P}$, each with a strictly fixed window size W. We set $W = F = 16$, matching the height of a TC fragment and defining the exact number of rows processed together by a single TC operation.

Under this formulation, the objective is to minimize the edge cut of G_{knn} subject to the fixed-size constraint. Minimizing the edge cut encourages rows with high pairwise similarity to be placed within the same partition, while the size constraint guarantees direct compatibility with the hardware. The final permutation π is constructed by concatenating the row indices from each partition $\mathcal{P}[p]$ into a single global vector. By applying π to the adjacency matrix, we guarantee that every contiguous window of W rows forms a structurally regularized submatrix.

In practice, we implement this phase using a size-constrained graph partitioner based on METIS [19], configured to enforce an exact partition size of $W = 16$. This ensures that rows within each 16-row window exhibit high column overlap, thereby minimizing the union of non-zero columns within each 16×16 block. Unlike prior reordering techniques that are agnostic to TC execution, our method explicitly aligns similarity structure with hardware-aligned computation. As a result, it directly increases the effective density seen by the TC and establishes the structural foundation for the subsequent condensation stage.

Complexity. Let n be the number of nodes and k be the number of nearest neighbors used for similarity construction. ThunderGNN first builds an approximate k-NN graph with kn edges using NND, which takes $O(n^{1.14})$ time in practice. It then partitions this graph with METIS in $O(kn)$ time. Thus, the overall preprocessing time is $O(n^{1.14} + kn)$, or $O(n^{1.14})$ when k is treated as a small constant, which is substantially more scalable than exact pairwise similarity computation with $O(n^2)$ time.

The auxiliary preprocessing memory is $O(n + kn)$ for node-level metadata and the temporary k-NN graph. This storage is released after the reordered layout is generated, and the one-time preprocessing cost can be amortized over multiple training epochs and inference runs.

4.3 Condensed Binarized Abstraction

The Representation-Execution Decoupling rationale is realized through CBA, which separates logical representation from physical execution to balance memory efficiency and compute utilization.

At the core of CBA is the observation that optimal storage granularity depends on graph sparsity, whereas TC execution enforces a fixed fragment size of $F \times F$ (with $F = 16$). Instead of binding storage to this rigid execution unit, CBA allows the logical tile size B to vary independently of F. Dense graphs can therefore map directly to full fragments to maximize compute utilization, while sparse graphs intentionally use smaller logical tiles to reduce metadata and memory traffic. Within this density-adaptive tiling framework, condensation and binarization operate as complementary mechanisms that reshape sparsity into a TC-friendly form.

Condensation. As plotted in Fig. 3b, CBA partitions the sparse matrix row-wise into fixed-height *row windows*, where the window height matches the TC fragment dimension (e.g., 16 rows). Within each row window, empty columns are removed, and the remaining non-zero columns are compacted and reassigned contiguous column indices. Formally, for the w-th row window $R_w \in \mathbb{R}^{F \times n}$, let

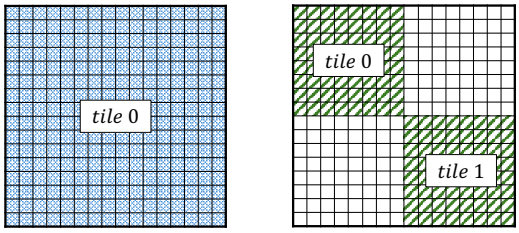
$$C_w = \{\, j \mid \exists\, r \in [0, F-1],\ R_w(r,j) \neq 0 \,\} \tag{6}$$

denote the set of active columns in the window. Condensation then constructs a compact submatrix

$$\widehat{R}_w = R_w[:, C_w] \in \mathbb{R}^{F \times |C_w|} \tag{7}$$

where the columns in C_w are reindexed contiguously.

This condensation step, following TC-GNN [38], transforms a sparse submatrix into a narrower, denser block suitable for dense computation. The effectiveness of condensation critically depends on row similarity within each window. When rows share overlapping non-zero patterns, the union of active columns is small, yielding a tightly condensed layout. Conversely, grouping dissimilar rows leads to wide unions and poor compaction. This directly



(a) Full (b) Diagonal

Figure 4: Tile placement on the 16×16 fragment.

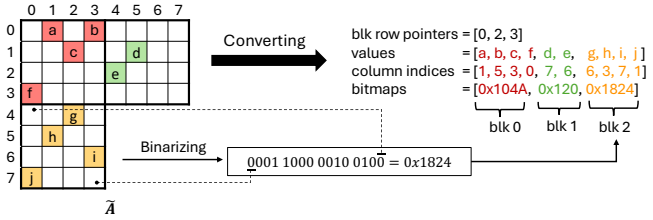


Figure 5: Condensed Binarized Abstraction (CBA) layout, illustrated using 4×4 tiles for clarity.

benefits from the similarity-driven reordering described in Section 4.2, which co-places structurally similar rows to maximize post-condensation density.

Binarization. To represent fine-grained sparsity within condensed blocks, CBA separates structure from values using bitmap-based binarization. Each logical $B \times B$ tile is encoded as a linearized bitmask, where each bit indicates the presence or absence of a non-zero. For a logical tile $T \in \mathbb{R}^{B \times B}$, its bitmap is defined as

$$M_T(u,v) = \begin{cases} 1, & T(u,v) \neq 0, \\ 0, & \text{otherwise}, \end{cases} \qquad (u,v) \in [0, B-1] \tag{8}$$

The non-zero values of T are stored in a dense array following the same linearized order as the set bits in M_T.

Numerical values are stored in dense arrays and streamed contiguously at runtime, satisfying GPU memory coalescing requirements. Element coordinates are reconstructed on the fly using efficient bitwise operations. This design deliberately trades lightweight computation for substantially reduced memory traffic, which is the dominant bottleneck in sparse GNN workloads.

Limitations of Fixed-Size Logical Tiles. Condensation and binarization alone are insufficient if the logical tile size is rigidly fixed to $B = F = 16$. A 16×16 bitmap incurs a fixed 256-bit metadata cost, regardless of how few non-zeros are present. As a result, memory savings are realized only when a tile contains more than eight non-zero entries. In practice, many regions remain well below this threshold even after reordering. For example, non-zeros confined to an 8×8 subregion still pay the full metadata cost when forced into a 16×16 representation, wasting up to 75% of the bitmap storage. This exposes a remaining mismatch between variable graph sparsity and rigid, hardware-aligned storage.

Adaptive Tiling. CBA resolves this mismatch by allowing the logical tile size B to shrink in sparse regions while preserving a fixed physical execution size F. Multiple $B \times B$ logical tiles are packed along the diagonal of a single $F \times F$ TC fragment. Let $\rho = F/B$ denote the packing factor. This design introduces a trade-off between compute utilization and memory efficiency:

$$\text{Tensor Utilization (TU)} = \frac{\rho \cdot B^2}{F^2} = \frac{1}{\rho}, \tag{9}$$

$$\text{Memory Savings (MS)} = \frac{F^2}{\rho \cdot B^2} = \rho. \tag{10}$$

Reducing B improves memory efficiency linearly, but proportionally decreases TC utilization. Since no single tile size is optimal across all sparsity regimes, ThunderGNN adopts a density-aware tiling policy. Specifically, ThunderGNN fixes $F = 16$ to match TC intrinsics and selects between two logical tile configurations based on the average graph degree (m/n). This decision is made once per graph to avoid runtime divergence and kernel specialization overhead:

- *Full Placement* ($B = 16$, $\rho = 1$): for denser graphs ($m/n > 8$), achieving full TC utilization with a 256-bit bitmap.
- *Diagonal Placement* ($B = 8$, $\rho = 2$): for sparser graphs ($m/n \leq 8$), packing two 8×8 logical tiles into the diagonal quadrants of a 16×16 fragment, reducing TC utilization to 50% while halving bitmap metadata to 128 bits.

As illustrated in Figure 4, diagonal placement effectively disables off-diagonal TC quadrants to minimize memory traffic. This explicitly recognizes that in sparse regimes, memory bandwidth, instead of compute throughput, is the primary performance limiter.

Final Layout. CBA materializes the combined effects of condensation, binarization, and adaptive tiling into a compact, execution-ready physical layout, illustrated in Fig. 5. The layout consists of five components:

(1) `rowPtr`: Indexes tile rows after condensation, enabling efficient traversal over row windows.
(2) `colIdx`: Stores the original column indices of each element, preserving the global matrix semantics.
(3) `bitmaps`: Encodes intra-tile sparsity using 64-bit bitmasks. FULL placement uses four bitmaps per tile, while DIA placement uses one bitmap per logical tile, reducing index storage by up to 32× compared to coordinate-based formats.
(4) `values`: A dense, contiguous array containing *only* non-zero FP16 values, ordered consistently with the bitmap encoding to enable coalesced memory access.
(5) `p`: A lightweight metadata scalar indicating the packing factor (`p`=1 for FULL, `p`=2 for DIA), used by the execution engine to interpret tile placement; omitted from the figure for clarity.

Together, these components form a unified storage abstraction that preserves structural topology, minimizes metadata overhead, and directly maps to TC execution.

4.4 Cache-Aware Execution Engine

The final stage of ThunderGNN operationalizes the previous transformations through a cache-aware execution engine that precisely controls data movement and coordinates computation with the GPU's fast on-chip cache hierarchy.

The execution engine follows three core principles. First, tile topology is decoded just-in-time and fused with data loading. Second, decoded structural information is buffered in on-chip shared memory and shared across thread warps to amortize decoding cost. Third, dense feature values are accessed through direct, non-cached loads into registers for immediate consumption, avoiding cache pollution. The engine explicitly separates reusable structural information from non-reusable feature values.

Just-in-Time Fused Decoding. ThunderGNN employs a "lazy" decoding mechanism that reconstructs logical graph tiles only at the moment of computation. Tile decoding is fused with value loading and never materializes intermediate sparse coordinates.

Algorithm 2: Just-in-Time Fused Decoding

Input: *bitmap*(s), *values*, *placement* mode
Output: decoded tile $\widehat{T}$

1 **if** *placement* = *FULL* **then**
2 **for** $i \leftarrow 0$ **to** 255 **do**
3 $n \leftarrow i/64$; $m \leftarrow i \bmod 64$;
4 $\widehat{T}[i] \leftarrow bitmaps[n][m]$? load(*values*) : 0;
5 **else**
6 **foreach** *diagonal region* $r \in \{0, 1\}$ **do**
7 **for** $i \leftarrow 0$ **to** 63 **do**
8 $\widehat{T}[\text{map}(r, i)] \leftarrow bitmap[i]$? load(*values*) : 0;

As outlined in Algo. 2, the engine supports two decoding modes corresponding to the storage layouts defined in CBA. In FULL placement, the bitmap (four 64-bit integers) spans all positions of the execution fragment, and decoding iterates over a fixed, fragment-aligned order. In DIA placement, decoding is restricted to diagonal subregions, and the mapping function `map(·)` projects local 8×8 coordinates into the corresponding fragment positions.

Crucially, decoding is fused with data loading. The bitmap value serves as a predicate: if a bit is set (1), the engine fetches the corresponding value from global memory; otherwise, it injects a zero. This predicate-based mechanism materializes a logically sparse tile in a dense, fragment-aligned layout, eliminating the need for intermediate sparse indices or coordinate lists.

The remaining principles are realized through operator designs that differentiate access paths for topology and features. Algo. 3 illustrates this mechanism for node aggregation, which computes the sparse-dense product $C = \tilde{A}B$ with $\tilde{A}$ stored in CBA format.

Algorithm 3: Node Aggregation in ThunderGNN

Input: graph $\tilde{A}$ in CBA format, features B
Output: aggregated features C

1 **for** $i \leftarrow \tilde{A}.rowPtr[row]$ **to** $\tilde{A}.rowPtr[row + 1] - 1$ **do**
2 $A_{\text{sm}} \leftarrow \text{DecodeTile}(\tilde{A}.bitmaps, \tilde{A}.values, i)$;
3 $fragA \leftarrow \text{Broadcast}(A_{\text{sm}})$;
4 $fragB \leftarrow \text{DirectGather}(B, \tilde{A}.colIdx)$;
5 $fragC \leftarrow \text{Compute}(fragA, fragB, fragC)$;
6 $C \leftarrow \text{Store}(\text{fragC})$;

Spatial Structure Reuse. Decoding compressed topology introduces extra bitwise operations. To amortize this cost, ThunderGNN leverages the spatial locality of the GPU's thread hierarchy. Instead of each thread decoding its own structure, a thread block cooperatively performs DecodeTile through Just-in-Time Fused Decoding in Line 2 and stages the decoded tile in *shared memory*. Once buffered, Broadcast in Line 3 broadcasts the decoded structure to all warps within the thread block. This allows the thread block to reuse one decoded tile while streaming different dense feature dimensions.

Streamed Dense Features. While the decoded tile is buffered and reused, dense feature data follows a non-reusable access path. Feature values are accessed via the original column indices stored in $\tilde{A}.colIdx$ and consumed exactly once, providing no temporal locality. Accordingly, DirectGather in Line 4 gathers these feature rows directly into registers, without staging them in shared memory. This preserves on-chip capacity for the decoded tile and avoids cache thrashing caused by high-volume feature traffic. Finally, Compute in Line 5 performs the fragment-level matrix multiplication on TCs.

Algorithm 4: Message Computation in ThunderGNN

Input: graph $\tilde{A}$ in CBA format, features B, features C
Output: edge messages D

1 **for** $i \leftarrow \tilde{A}.rowPtr[row]$ **to** $\tilde{A}.rowPtr[row + 1] - 1$ **do**
2 $A_{\text{sm}} \leftarrow \text{DecodeTile}(\tilde{A}.bitmaps, i)$;
3 **for** $k \in [0, K - 1]$ **do**
4 $mask \leftarrow \text{BroadcastMask}(A_{\text{sm}})$;
5 $fragB \leftarrow \text{DirectLoad}(B, row, k)$;
6 $fragC \leftarrow \text{DirectGather}(C, \tilde{A}.colIdx, k)$;
7 $fragD \leftarrow \text{Compute}(fragB, fragC, fragD)$;
8 $D \leftarrow \text{MaskedStore}(fragD, mask)$;

Algo. 4 follows the same principles but uses a different dataflow for message computation, which evaluates edge-wise feature interactions, e.g., $D_{ij} = B_i^\top C_j$ for each nonzero edge $(i, j) \in \tilde{A}$. Here, DecodeTile in Line 2 decodes the bitmap into a compact on-chip tile descriptor A_{sm}, rather than reconstructing weighted tile values. BroadcastMask in Line 4 derives and broadcasts the edge-validity mask from A_{sm} to the warps, and MaskedStore in Line 8 uses this mask to write only valid edge messages. Meanwhile, DirectLoad in Line 5 loads source features from the current row window into registers, and DirectGather in Line 6 gathers destination features indexed by $\tilde{A}$.colIdx into registers. Finally, Compute in Line 7 performs fragment-level matrix multiplication on TCs.

Overall, the execution engine makes bitmap-decoded topology the explicit driver of data movement and computation. By separating reusable structure from streamed features, ThunderGNN enables predictable execution and efficient utilization of GPU on-chip resources for irregular graph workloads.

5 EVALUATION

5.1 Experimental Setting

Platforms and Metrics. We implement ThunderGNN with Python for reordering algorithm and CUDA for compute engine. It incorporates the PyTorch [30] library for building GNNs. We evaluate

Table 2: GNN performance (ms) on NVIDIA A100. Bold and underline highlight the best and second-best performance, respectively. The rightmost column reports the geometric mean speedup of ThunderGNN over each competing method.

Model	Methods	A5	AR	CS	CA	CO	PP	PR	PF	PB	RD	SB	*Speedup*
GCN	DGL	25.39	4.26	3.73	19.73	3.67	5.40	**211.91**	3.69	3.42	68.18	6.76	1.70×
	PyG	28.70	5.53	3.36	20.30	3.46	8.99	532.04	4.49	3.79	302.47	9.59	2.40×
	GraphScope	26.49	6.37	3.80	19.75	3.84	9.29	297.49	4.68	3.50	184.06	10.92	2.25×
	TC-GNN	29.05	6.03	0.94	21.00	1.00	4.96	424.14	1.45	1.28	11257.88	9.89	2.03×
	ThunderGNN	**23.91**	**4.04**	**0.92**	**17.99**	**0.94**	**3.32**	236.61	**1.35**	**1.05**	**63.92**	**5.42**	1.00×
GAT	DGL	64.39	15.58	11.55	38.48	11.41	12.54	989.81	11.46	11.68	647.57	47.21	2.05×
	PyG	135.70	40.12	8.15	69.07	8.21	25.20	—	9.96	8.54	—	59.02	2.79×
	GraphScope	89.30	35.07	8.93	54.86	8.54	19.18	—	7.98	8.51	—	62.31	2.50×
	TC-GNN	109.13	33.35	2.56	39.16	2.62	17.69	2779.47	3.43	4.12	104600.84	69.51	2.68×
	ThunderGNN	**56.12**	**16.28**	**2.44**	**34.56**	**2.57**	**10.72**	**852.75**	**3.04**	**2.85**	**364.48**	**16.79**	1.00×

ThunderGNN on NVIDIA A100 GPU with 40GB memory using CUDA 12.8. The underlying MMA in TCs are shaped in $\langle 16, 16, 16\rangle$ with FP16, which is a standard precision for GNNs. Although our implementation targets NVIDIA A100 Tensor Cores, the reordering and CBA abstractions are parameterized by the fragment size F and can be adapted to other accelerators with fixed-shape matrix units.

The reported time represents the average of 100 runs. For GNN evaluation, we use two models: GCN [21] with 2 layers of 16 hidden dimensions, and GAT with 4 layers of 32 hidden dimensions [33]. GAT differs from GCN in its message computation where the edge features are computed before node aggregation.

Baselines. ThunderGNN is compared against several representative GNN systems. We include popular GNN frameworks such as DGL [36] and PyG [12], both of which rely on third-party libraries to execute graph operations on SIMT cores. We also evaluate GraphScope [11], an industrial-scale graph processing framework that supports a wide range of graph analytics and learning workloads. In addition, we compare against TC-GNN [38], the state-of-the-art GNN system specifically optimized for TCs.

At the operator level, we further evaluate ThunderGNN against several systems designed for sparse Transformer workloads, including Sputnik [15], VectorSparse [6], and Flash-LLM [40]. Sputnik targets sparse computation on SIMT cores, whereas VectorSparse and Flash-LLM are optimized for TC execution. For completeness, we also evaluate the corresponding operators from TC-GNN and DGL. We omit SMaT [28] from the comparison due to its substantially lower performance relative to the other baselines.

Datasets. We test a diverse set of graph datasets, sourced from previous GNN research, all of which can be readily accessed via DGL or PyG's dataloader. They represent real-world graphs from various domains, including citation networks, biological networks, social networks, and co-purchase networks. Detailed information on the datasets is provided in Table 5. These datasets are firstly converted to CBA format offline, and then loaded into GPU memory for ThunderGNN to reuse across various tasks.

5.2 GNN Benchmarks

Table 2 reports the per-epoch training time of ThunderGNN and 4 representative GNN systems (DGL, PyG, GraphScope and TC-GNN) on an NVIDIA A100 GPU across a diverse set of graph datasets.

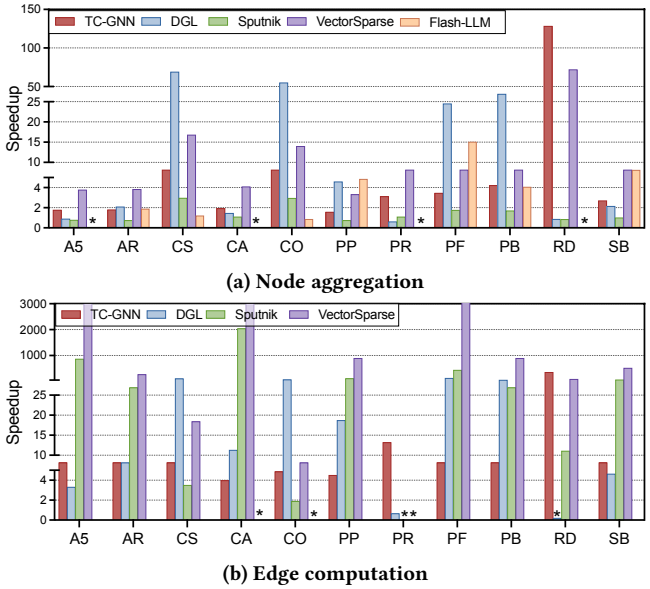
Table 5: Graph datasets for evaluation

Graph	**Abbr.**	n	m	$\overline{d}$	Dim.
amazon0505	A5	410,236	4,878,874	11.89	96
artist	AR	50,515	1,638,396	32.43	100
citeseer	CS	3,327	9,228	2.77	3,703
com-amazon	CA	334,863	1,851,744	5.53	96
cora	CO	2,708	10,556	3.90	1,433
ppi	PP	56,944	818,716	14.38	50
products	PR	2,449,029	123,718,152	50.52	100
PROTEINS_full	PF	43,466	162,088	3.73	29
pubmed	PB	19,717	88,651	4.50	500
reddit	RD	232,965	114,615,892	491.99	602
soc-BlogCatalog	SB	88,784	2,093,195	23.58	128

ThunderGNN consistently delivers the best or second-best performance for both GCN and GAT, and achieves the fastest runtime on nearly all dataset–model combinations. Overall, competing systems are slower than ThunderGNN by geometric mean factors ranging from 1.70× to 2.40× for GCN, and from 2.05× to 2.79× for GAT.

ThunderGNN exhibits larger performance gains for GAT than for GCN due to differences in their computational pattern. GCN primarily consists of node aggregations, whereas GAT additionally performs message computation to generate edge-wise attention scores. Message computation incurs higher memory traffic than node aggregation because it requires accessing and updating both node and edge features, and is more sensitive to graph sparsity. Moreover, GAT contains more computational stages than GCN, increasing the overall workload per layer. These characteristics amplify the benefits of ThunderGNN, leading to larger speedups for GAT.

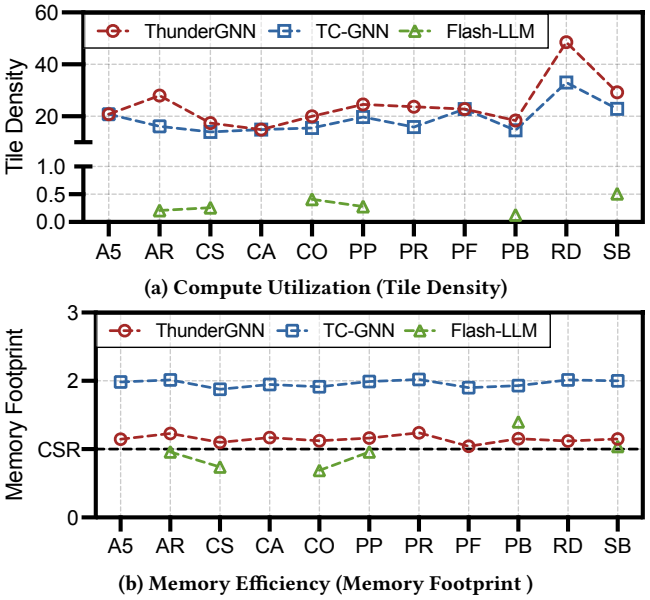
Among the baselines, DGL generally achieves the second-best performance. This is largely attributed to its reliance on highly optimized SIMT-based sparse operators from the cuSPARSE [9] library. GraphScope and PyG consistently underperforms due to their reliance on pytorch's scatter–gather primitives for graph operations, which introduce substantial overhead. Furthermore, both PyG and GraphScope do not scale to large graphs such as *products* (PR) and *reddit* (RD), where it encounters out-of-memory errors when running GAT.



(a) Node aggregation

(b) Edge computation

Figure 6: ThunderGNN's speedup over competing methods.



(a) Compute Utilization (Tile Density)

(b) Memory Efficiency (Memory Footprint)

Figure 7: Compute utilization and memory efficiency of TC-based systems.

TC-GNN performs competitively on small graphs with low average degree, such as *cora* (CO) and *citeseer* (CS). However, its performance degrades substantially on larger graphs and on graphs with high-degree vertices. In particular, for RD with average degree $\overline{d} = 491.99$, TC-GNN is two to three orders of magnitude slower than the other systems. The poor performance of TC-GNN on high-degree graphs stems from its inefficient memory layout, which requires loading numerous local and global column indices per tile. This results in excessive memory traffic and poor memory coalescing, significantly limiting achievable performance.

5.3 Graph Operation Benchmarks

Fig. 6 compares ThunderGNN against TC-GNN, DGL, and three sparse computation libraries originally designed for Transformer-style models: Sputnik [15], VectorSparse [6], and Flash-LLM [40]. Both DGL and Sputnik only utilzes CUDA cores; both VectorSparse and Flash-LLM leverage TCs. The evaluation focuses on the two graph operations: node aggregation, which is common to both GNNs and sparse Transformer models, and edge computation, which is critical for attention-based GNNs such as GAT.

Node Aggregation. ThunderGNN consistently outperforms all baselines across all graph datasets, achieving geometric mean speedups of 4.24×, 4.83×, 1.25×, 8.99×, and 3.17× over TC-GNN, DGL, Sputnik, VectorSparse, and Flash-LLM, respectively. VectorSparse delivers the weakest performance, as it is specifically designed for structured sparsity with moderate sparsity levels (50%–90%). When applied to highly sparse and irregular graphs (around 99% sparsity), its vectorized execution becomes inefficient. Sputnik performs better under high sparsity by regularizing memory access for SIMT, but remains slower than ThunderGNN. The performance of DGL is primarily limited by its partitioning-related pre-processing overhead. TC-GNN performs competitively on small graphs but experiences significant slowdowns as graph size increases. Flash-LLM fails to process medium and large graphs due to prohibitively long pre-processing time (exceeding 30 minutes for *ama*), and thus does not scale to realistic GNN workloads.

Edge Computation. ThunderGNN further widens its performance advantage for edge computation, achieving geometric mean speedups of 8.89×, 10.42×, 59.49×, and 473.80× over TC-GNN, DGL, Sputnik, and VectorSparse, respectively. This substantial improvement directly enables ThunderGNN to achieve the fastest end-to-end training performance for GAT models, as shown in Section 5.2. Both Sputnik and VectorSparse exhibit severe performance degradation on this operation. This is properly attributed to their compatibility issue in CUDA versions (from 10.1 to 12.8) and architecture shift (from Volta to Ampere). Flash-LLM does not provide operator for edge computation and is therefore excluded from comparison.

5.4 Resolving the Data-Hardware Conflict

In this section, we demonstrate the effectiveness of ThunderGNN in resolving the fundamental conflict between hardware-aligned execution and extreme sparsity. Unlike prior works that sacrifice one for the other, ThunderGNN bridges this gap through two critical dimensions: Compute Utilization and Memory Efficiency. Fig. 7 summarizes the compute utilization and memory efficiency of ThunderGNN in comparison with other TC-based systems.

Compute Utilization. We measure the compute utilization of ThunderGNN using *tile density*, defined as the average number of non-zero elements per tile. Higher tile density indicates more effective aggregation of nonzeros within each tile, leading to better utilization of TC compute resources and higher computational intensity. Since different TC-based systems adopt different tiling granularities, we normalize tile density to a standard 16 × 16 tile for fair comparison.

ThunderGNN produces substantially denser tiles than prior approaches, achieving an average of 24.40 nonzeros per tile, compared to 17.91 for TC-GNN and only 0.31 for Flash-LLM. The improvement over TC-GNN is primarily attributed to ThunderGNN's similarity-driven reordering, which co-place rows with overlapping nonzero patterns and concentrates sparsity into TC-aligned tiles.

In contrast, Flash-LLM exhibits extremely low tile density due to its coarse-grained tiling strategy. Flash-LLM operates on 256×64 tiles, each spanning 32 underlying 16×16 TC tiles. Because nonzeros are distributed across the large tile without fine-grained compaction, most individual TC tiles remain empty or nearly empty, resulting in severe underutilization of TC compute resources.

Memory Efficiency. We evaluate the memory efficiency of ThunderGNN using *memory footprint*, defined as the ratio of the memory consumption of a given representation to that of DGL. DGL serves as a highly memory-efficient baseline by executing GNN operations on SIMT cores without introducing additional metadata. A lower memory footprint indicates a more compact representation with reduced metadata and index storage, which lowers memory traffic and improves overall performance.

ThunderGNN maintains a near-DGL memory footprint (1.11×), indicating that it preserves much of DGL's memory efficiency despite introducing TC-oriented optimizations. In contrast, TC-GNN nearly doubles memory usage (1.94×). This gap highlights the inefficiency of TC-GNN', which introduces substantial metadata and index overhead to enforce fixed-size TC execution. Flash-LLM achieves a slightly lower memory footprint than DGL (0.97×); however, this reduction comes at the cost of extremely sparse tiles, which severely limit compute utilization and reduce its effectiveness for graph workloads.

Overall, ThunderGNN resolves the fundamental hardware–data conflict in TC-based GNN execution by jointly optimizing tile density and memory efficiency. It substantially increases tile density to better utilize TCs while preserving a memory footprint close to DGL, avoiding the memory expansion that typically accompanies TC-oriented designs. This balance enables high performance on sparse and irregular graph workloads and distinguishes ThunderGNN from existing TC-based approaches that optimize compute or memory in isolation.

5.5 Cache Utilization

A key contributor to ThunderGNN's performance is its high cache utilization, faciliated by its cache-aware execution engine. As shown in Fig. 8, ThunderGNN consistently achieves higher cache hit rates than both DGL and TC-GNN at the L1 and L2 levels for node aggregation and edge computation. Averaged across datasets, ThunderGNN attains a cache hit rate of approximately 90%, compared to 52% for DGL and 64% for TC-GNN.

TC-GNN suffers from frequent cache misses, particularly on high-degree graphs. For example, on *reddit*, ThunderGNN achieves an L1 cache hit rate of 94.02%, while TC-GNN attains only 25.61%. This gap is largely due to TC-GNN's unnecessary caching of dense matrices in shared memory, which increases cache pressure. In addition, the high-degree structure of *reddit* leads to irregular memory access patterns, further exacerbating cache inefficiency.

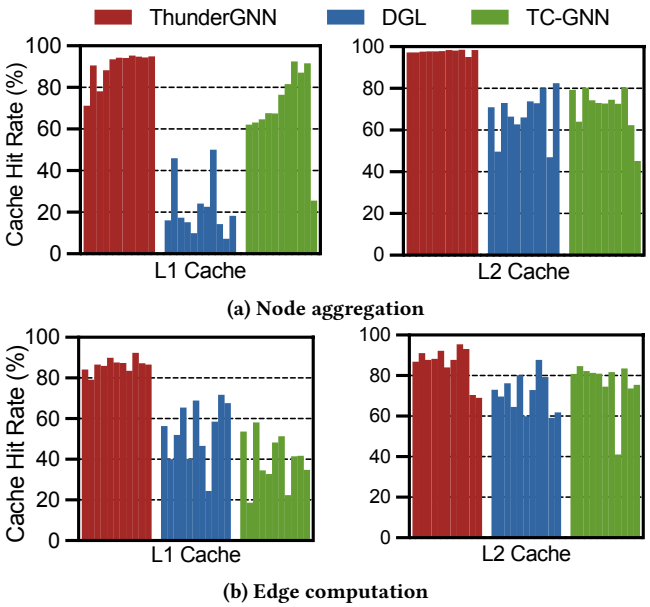


Figure 8: L1 and L2 cache hit rates of ThunderGNN and competing systems.

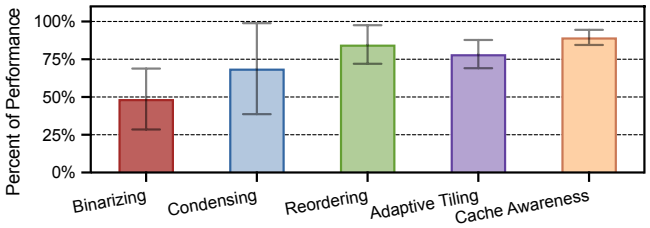


Figure 9: Normalized performance of ThunderGNN without individual optimization components.

ThunderGNN mitigates these issues by avoiding redundant caching of dense matrices in shared memory and instead loading them directly into registers, improving L1 cache efficiency. At the L2 level, graph data within the same tile are stored contiguously in memory, increasing spatial locality. Together, these design choices substantially improve utilization of the GPU memory hierarchy, enabling higher throughput for graph workloads.

5.6 Ablation Study

We conduct an ablation study to quantify the contribution of key optimization components in ThunderGNN, including graph reordering, graph condensing, binarization, adaptive tiling, and direct register loading. Each variant disables a single component and is compared against the fully optimized implementation.

Fig. 9 shows the performance impact of removing individual components. Removing *binarization* results in the largest degradation, with a 51% performance drop, highlighting its dominant role in reducing data footprint and simplifying GPU processing. This benefit is consistent across datasets, indicating that binarization is a critical optimization for irregular graph workloads.

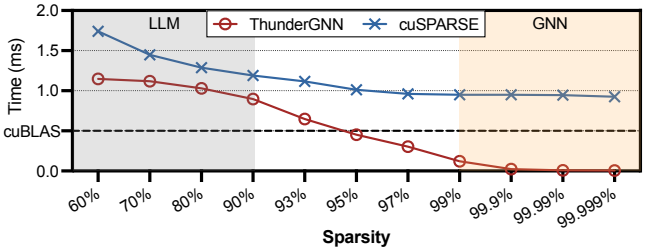


Figure 10: SpMM kernel on a pruned weight matrix from OPT-30B. Grey and orange areas indicate typical sparsity of LLMs and GNNs, respectively. cuBLAS is the dense kernel.

Disabling *graph condensing* leads to a 31% performance reduction, as more tiles must be processed without eliminating empty or low-utilization regions. *Adaptive tiling* contributes a 22% performance improvement by adapting tile shapes to graph structure. Removing it increases execution overhead due to suboptimal tile granularity.

Graph reordering yields a 15% performance gain by improving data locality and access regularity. It yields substantially higher gains on large and highly irregular graphs (up to 31% on *reddit*), but provides limited benefit on smaller graphs (about 2% on *citeseer*). Finally, removing *direct register loading* and falling back to shared memory incurs a 10% performance loss, reflecting additional memory traffic and synchronization overhead.

Overall, the ablation results show that ThunderGNN's performance gains arise from the combined effect of multiple complementary optimizations, with binarization and graph condensing providing the largest benefits. In addition, reordering, adaptive tiling, and cache-awareness offer substantial incremental improvements.

5.7 Application to Sparse LLMs

The pruning of LLM is a common practice to reduce the model size and improve the inference speed by converting the dense weight matrices into unstructured sparse matrices [40]. We extract a weight matrix size 7168×7168 from OPT-30B [44] and prune it with varying sparsity levels following Flash-LLM [40]. As plotted in Fig. 10, ThunderGNN' SpMM is compared with NVIDIA cuSPARSE's SpMM and cuBLAS's dense kernel [8].

Graphs often contains millions of nodes, whereas weight matrices in LLMs are much smaller in dimension (e.g., $7K \times 7K$). LLM weight matrices also exhibit only moderate sparsity (e.g., 60–90%) to balance accuracy and performance [40]. In this sparsity regime and dimension, cuBLAS's dense kernel outperforms sparse alternatives. ThunderGNN begins to outperform cuBLAS when sparsity exceeds 95%, and its advantage increases with sparsity, reaching an 88.37× speedup at 99.999% sparsity. cuSPARSE performs the worst in this setting due to substantial preprocessing overhead on small matrices. Although accelerating LLM workloads is beyond the scope of this paper, this analysis clarifies ThunderGNN's applicability: it is most effective for graph workloads with high sparsity and less suitable for the moderate sparsity levels (<90%) typical of LLMs.

6 RELATED WORK

Graph Partitioning. Graph partitioning has been widely studied as a fundamental technique for improving the scalability and locality of graph analytics and GNN training [20, 32, 34]. Classical partitioners such as METIS [20] aim to minimize edge cuts while balancing the number of vertices across partitions. Streaming partitioners, such as Fennel [34], reduce preprocessing overhead by assigning vertices online according to locality-aware objectives. In distributed GNN systems, partitioning is commonly used to reduce cross-machine communication [26, 47]. Cluster-GCN [7] constructs mini-batches by partitioning the graph into clusters, where each batch corresponds to one or several subgraphs, thereby reducing neighborhood expansion during training. Unlike these partitioning methods, ThunderGNN is explicitly designed for TCs, bridging the gap between irregular GNN sparsity and TC-friendly execution.

Graph Condensation. Graph condensation aims to compress a large training graph into a much smaller representative graph while preserving model accuracy [18, 25, 48]. Early methods such as GCond [18] optimize synthetic node features and labels so that GNNs trained on the condensed graph can achieve comparable performance to those trained on the original graph. Subsequent work further considers structure learning, trajectory matching, and distribution matching to improve condensation quality and scalability [25, 48]. By reducing the number of nodes and edges, graph condensation can substantially lower training cost and memory consumption. In contrast, ThunderGNN does not modify the graph or approximate the learning problem. It accelerates the sparse graph operators on GPUs and can also be applied to the execution of condensed graphs.

Graph Sampling. Sampling is an important direction for scaling GNN training [4, 5, 16, 49]. GraphSAGE [16] samples a fixed number of neighbors for each target node, reducing the exponential growth of computation across GNN layers. FastGCN [4] interprets graph convolution as an integral transformation and performs layer-wise sampling to reduce training complexity. LADIES [49] improves sampling quality by selecting layer-dependent nodes, while GraphSAINT [43] samples subgraphs to preserve local graph structures during mini-batch training. ThunderGNN is complementary to sampling-based GNN, since the aggregation over sampled neighbor subgraphs is still executed as sparse operators such as SpMM, which ThunderGNN is designed to accelerate.

7 CONCLUSION

In this paper, we present ThunderGNN, a framework optimized to leverage TCs for GNN. We introduce three levels of optimizations: (1) a reordering algorithm inducing local density for sparse graphs (2) a condensed, binarized graph representation to reduce memory footprint, and (3) a cache-aware execution engine that efficiently streams tiles directly into TCs. Evaluations show ThunderGNN consistently outperforms state-of-the-art works in GNN execution and graph operations. Future work will expand ThunderGNN to support more GNN models and integrate with distributed GNNs.

ACKNOWLEDGMENT

This work is supported by The Research Grants Council of Hong Kong, China, No.14205520.

REFERENCES

[1] NVIDIA Corporation & affiliates. 2022. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2024-01-15.

[2] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[4] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *Proceedings of the 6th International Conference on Learning Representations*.

[5] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 941–949.

[6] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. 2021. Efficient Tensor Core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[7] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 257–266.

[8] NVIDIA Corporation. 2012. cuBLAS, Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas. Accessed: 2024-06-05.

[9] NVIDIA Corporation. 2014. cuSPARSE, the CUDA sparse matrix library. https://docs.nvidia.com/cuda/cusparse/. Accessed: 2024-06-05.

[10] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algo.s for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/. Accessed: 2024-01-05.

[11] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[12] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[13] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. *Advances in neural information processing systems* 30 (2017).

[14] Pablo Gainza, Freyr Sverrisson, Frederico Monti, Emanuele Rodola, Davide Boscaini, Michael M Bronstein, and Bruno E Correia. 2020. Deciphering interaction fingerprints from protein molecular surfaces using geometric deep learning. *Nature Methods* 17, 2 (2020), 184–192.

[15] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[17] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 119–132.

[18] Wei Jin, Lingxiao Zhao, Shichang Zhang, Yozen Liu, Jiliang Tang, and Neil Shah. 2022. Graph Condensation for Graph Neural Networks. In *Proceedings of the 10th International Conference on Learning Representations*.

[19] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[20] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. In *Proceedings of the International Conference on Parallel Processing*. 113–122.

[21] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[22] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. 2020. NVIDIA Ampere Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth. Accessed: 2024-01-15.

[23] Zhiyuan Li, Xun Jian, Yue Wang, Yingxia Shao, and Lei Chen. 2024. DAHA: Accelerating GNN Training with Data and Hardware Aware Execution Planning. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1364–1376.

[24] Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. 2022. SCARA: scalable graph neural networks with feature-oriented optimization. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3240–3248.

[25] Mengyang Liu, Shanchuan Li, Xinshi Chen, and Le Song. 2022. Graph Condensation via Receptive Field Distribution Matching. *arXiv preprint arXiv:2206.13697* (2022).

[26] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1–14.

[27] Nikolai Merkel, Pierre Toussing, Ruben Mayer, and Hans-Arno Jacobsen. 2024. Can Graph Reordering Speed Up Graph Neural Network Training? An Experimental Study. *arXiv preprint arXiv:2409.11129* (2024).

[28] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2024. High performance unstructured spmm computation using tensor cores. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[29] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2024. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1227–1240.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[31] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[32] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1222–1230. https://doi.org/10.1145/2339530.2339722

[33] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735* (2018).

[34] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. ACM, 333–342.

[35] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[36] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

[37] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. 2019. Kgat: Knowledge graph attention network for recommendation. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 950–958.

[38] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 149–164.

[39] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.

[40] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity. *Proceedings of the VLDB Endowment* 17, 2 (2023), 211–224.

[41] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[42] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1241–1254.

[43] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *Proceedings of the 8th International Conference on Learning Representations*.

[44] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[45] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A dual-cache training system for graph neural networks on giant graphs with the GPU. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[46] Yuhao Zhang and Arun Kumar. 2023. Lotan: Bridging the gap between gnns and scalable graph analytics engines. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2728–2741.

[47] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph

Neural Network Training for Billion-Scale Graphs. In *Proceedings of the 2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE, 36–44.

[48] Xin Zheng, Miao Zhang, Chunyang Chen, Quoc Viet Hung Nguyen, Xingquan Zhu, and Shirui Pan. 2023. Structure-free Graph Condensation: From Large-scale Graphs to Condensed Graph-free Data. In *Advances in Neural Information Processing Systems*, Vol. 36.

[49] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. In *Advances in Neural Information Processing Systems*, Vol. 32. 11249–11259.