A Survey of Conferences' Submission on GNN Domains

In our survey of NeurIPS/ICML/ICLR 2024 papers, a total of 76 papers are related to GNN domains. In 76 papers, 44.7% (34 papers) used full-graph training, and among them, 38.2% (13 papers) directly reported out-of-memory. In terms of experimental environments, a total of 62 papers reported their GPU environments, and 45 papers utilized a single GPU (72.6%). Also, some papers with full-graph training directly stated that larger-sized datasets can incur out-of-memory when running their experiments. This shows the importance of enabling full-graph training of large graphs under limited resources (e.g., a single GPU).

B Limitations of Existing Full-Graph Training Methods

Full-graph GNN training processes all vertices' activations and gradients in a single pass, requiring substantial memory.

Distributed Training: Distributed full-graph training (Tripathy et al., 2020; Peng et al., 2022; Jia et al., 2020a; Wan et al., 2022b;a) scales the number of GPUs to meet the memory requirement. However, it requires a costly multi-server cluster. Moreover, it significantly suffers from communication bottlenecks, especially inter-server communication. We break down the training time of 3-/5-layer GCN with the widely used distributed full-graph training methods (CAGNET (Tripathy et al., 2020) and Sancus (Peng et al., 2022)) in the four-server setup utilized in the evaluation (Section 8.1). They both suffer from severe communication bottlenecks from 80% to 98% of the execution time.

A few single-server approaches exist to address such issues. Micro-batch (Yang et al., 2023) and host memory offloaded (Wang et al., 2023a) training have tried to conduct full-graph training in GPU memory-limited environments. Figure 14 illustrates an example graph and discusses the drawbacks of the above methods based on the full-graph dependency.

Micro-Batch Training: Betty (Yang et al., 2023) (Figure 14c) accumulates gradients from message flow graphs (MFGs) with all neighbor information across all layers, followed by a single weight update. However, even a small number of GNN layers cause MFGs to expand rapidly (Figure 14b), often exceeding the GPU memory. Partitioning (Karypis & Kumar, 1998; Karypis et al., 1997; LaSalle & Karypis, 2013) can reduce MFG size but requires significant memory, presenting a practical bottleneck.

Host Memory Offloaded Training: HongTu (Wang et al., 2023a) (Figure 14d) reduces GPU memory usage by moving activations and gradients to host memory. A 1-hop partitioning approach extracts 1-hop graphs that fit in GPU memory.

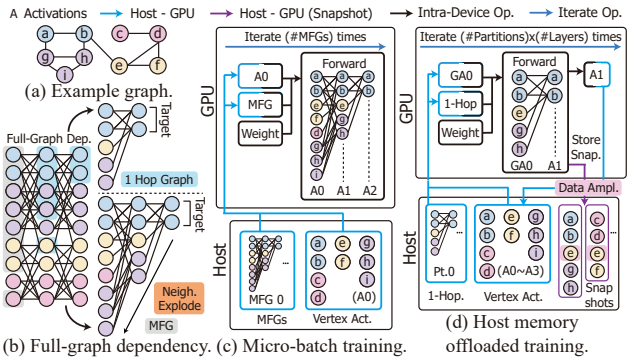


Figure 14. Full-graph training of single epoch for limited resources.

During an epoch, activations for these graphs are transferred to the GPU, processed, and offloaded back to host memory. While this saves GPU resources, it causes a *data amplification* problem: by saving 'snapshots' of 1-hop graphs for the backward pass, vertices appearing in multiple 1-hop graphs are stored repeatedly, increasing memory and I/O overhead.

Memory-Hungry Partitioning: All aforementioned methods rely on partitioning tools like METIS (Karypis & Kumar, 1998; Karypis et al., 1997; LaSalle & Karypis, 2013), which sequentially coarsen and refine graphs. This process consumes up to 4.8× the graph's size in memory (Kaur & Gupta, 2021), often exceeding the capacity of a typical single server. Hence, existing single-server full-graph methods face out-of-memory risks or resort to an external server.

C Limitation of Extending Storage-Based Mini-Batch Training to Full-Graph Training with Micro-Batch Training

While extending storage-based mini-batch training (e.g., Ginex (Park et al., 2022), MariusGNN (Waleffe et al., 2023), DiskGNN (Liu et al., 2025), GNNDrive (Jiang et al., 2024)) to full-graph training by setting the batch size to the entire node set and maximizing the neighbor size (i.e., micro-batch training from Betty (Yang et al., 2023)) may seem to enable the large-scale full-graph training on a limited environment, it faces several limitations[3].

First, as it still depends on the message flow graph structure (MFG), it faces the GPU memory limit like the original micro-batch training (Betty). Since micro-batch training needs to keep all neighbor information intact without sampling, it easily runs into out-of-memory due to neighbor explosion. For more details, please refer to Appendix B.

[3]Please note that since these systems are not designed for full-graph training workflows, adaptations may deviate from their original performance characteristics.

Table 5. **Performance of extending storage-based mini-batch training to full-graph training.** †: GNNDrive's GPU caching is statically preprocessed, so the fanout is restricted to 25 and not equivalent to full-graph training. *: Preprocessing failed because of excessive disk space usage.

Method	Products	IGBM	Papers
Ginex	9.00	OOM	17.72
DiskGNN	2.18	Preproc. Fail*	Preproc. Fail*
GNNDrive	6.33†	OOM	12.06†
GriNNder (Ours)	**0.12**	**0.93**	**9.07**

Second, they are mainly focused on handling initial features efficiently for mini-batch training, and are inefficient in supporting full-graph training without sampling. For instance, DiskGNN aggressively utilizes the preprocessing and prestores the mini-batch message flow graphs and related initial features to efficiently support large-scale mini-batch training with storage. However, since full-graph training (with micro-batch training) needs to handle all the features without dropping, the preprocessed data size easily exceeds the SSD capacity due to the redundantly saved data.

To show the above limitations directly, we evaluated DiskGNN (Liu et al., 2025) and GNNDrive (Jiang et al., 2024), which surpass the previous state-of-the-art storage-based mini-batch training (Ginex (Park et al., 2022), MariusGNN (Waleffe et al., 2023)) in Table 5. Ginex and GNNDrive encountered GPU out-of-memory on IGBM due to neighbor explosion without information dropping. On Papers, even with fanout 25, they were significantly slower than ours. DiskGNN uses offline preprocessing to pre-store all cacheable mini-batches with features. The preprocessing of IGBM/Papers fails by overflowing 4TB SSD, even with reduced fanout (25) from neighbor explosion. Our method is significantly faster for runnable cases (Products/Papers).

To sum up, while mini-batch storage-based systems can emulate full-graph training by micro-batch training, results show that this becomes infeasible on a large scale. This is due to either GPU memory exhaustion or prohibitive preprocessing disk usage. GriNNder avoids them by not relying on message flow graphs (MFGs) or redundant preprocessing.

D Overall Procedure of GriNNder

As GriNNder is the first work on storage offloaded full-graph GNN training, we carefully designed the framework to address the three challenges outlined in Section 1, whose overall procedure is listed in Algorithm 1. GriNNder first partitions the graph into smaller pieces, which should be done to incur minimal data transfer (line 2). Our contribution is on devising a lightweight partitioning algorithm that operates with significantly lower memory requirements while preserving the partitioning quality (Section 6). Then,

Algorithm 1 Overall procedure of GriNNder

Input: $\{W^i | 1 \leq i \leq L\}$: initial parameters, L: #layers
G: graph, F: initial features, P: #partitions to meet GPU mem. req.
Output: $\{W^i | 1 \leq i \leq L\}$: updated parameters
Notations:
T_p: 1-hop topologies (src→dst)
A^l_p: destination features/activations of layer l, partition T_p
GA^l_p: gathered source features/activations of layer l, partition T_p

1: **if** $METIS_{limit} \geq Host_{limit}$ **then**
2: $T_{(\cdot)} \leftarrow SA\_Partition(G, P)$ // Switching-aware partitioning (Sec. 6)
3: **else** $T_{(\cdot)} \leftarrow METIS(G, P)$ **end if**
4: // Do until finding proper P which makes all T_ps fit GPU memory limit.
5:
6: **for** $e = 1 \ldots \#epochs$ **do**
7: // Forward pass
8: **for** $l = 1 \ldots L$ **do**
9: $Storage\_to\_Host(A^{l-1}_{(\cdot)})$ // Partition-wise graph caching (Section 4)
10: **for** $p = 0 \ldots P-1$ **do**
11: $GA^{l-1}_p \leftarrow Gather(A^{l-1}_{(\cdot)})$
12: $Host\_to\_GPU(GA^{l-1}_p)$
13: $A^l_p \leftarrow FW(W^l, GA^{l-1}_p, T_p)$ // w/ Regathering (redundancy elimination) (Section 5)
14: $GPU\_to\_Host(A^l_p)$
15: **end for**
16: **end for**
17: // Backward pass
18: **for** $l = L \ldots 2$ **do**
19: // Partition-wise graph caching (Section 4)
20: $Host\_Upload\_or\_Intitialization(A^{l-1}_{(\cdot)}, \nabla A^{l-1}_{(\cdot)})$ // Host as write-back buffer
21: **for** $p = 0 \ldots P-1$ **do**
22: $Storage\_to\_Host(A^l_p, \nabla A^l_p)$
23: $GA^{l-1}_p \leftarrow Gather(A^{l-1}_{(\cdot)})$
24: $Host\_to\_GPU(GA^{l-1}_p)$ // Grad-engine activation regathering (Section 5)
25: $(\nabla GA^{l-1}_p, \nabla W^l) \xleftarrow{\cdot,+} BW(W^l, A^l_p, \nabla A^l_p, GA^{l-1}_p)$
26: $GPU\_to\_Host(GA^{l-1}_p)$
27: $\nabla A^{l-1}_{(\cdot)} \xleftarrow{+} Scatter(GA^{l-1}_p)$
28: **end for**
29: **end for**

for each partition (lines 10 and 21), forward and backward passes are performed on the GPUs (lines 11–14 and 22–27). To maximize the reuse of the data, GriNNder designs an efficient policy to cache intermediate data on the host memory (Section 4). During the forward/backward passes, much of the data transfer occurs between GPU-CPU due to checkpointing (lines 12, 14, 24, 26). This was originally designed toward reducing latency in previous work (Wang et al., 2023a), but it severely increases the amount of traffic and host memory usage for storage offloading scenarios. GriNNder redesigns the gradient engine with redundancy elimination, achieving significantly higher speedup and less memory requirement (Section 5).

E Profile of Dependency among Partitions

We additionally presented the profile of dependency among partitions on other datasets in Figure 15. When the size of a graph becomes larger, we need to partition the graph into a much larger number of partitions. This makes the

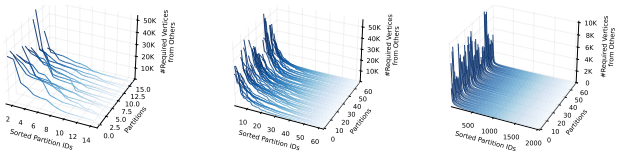
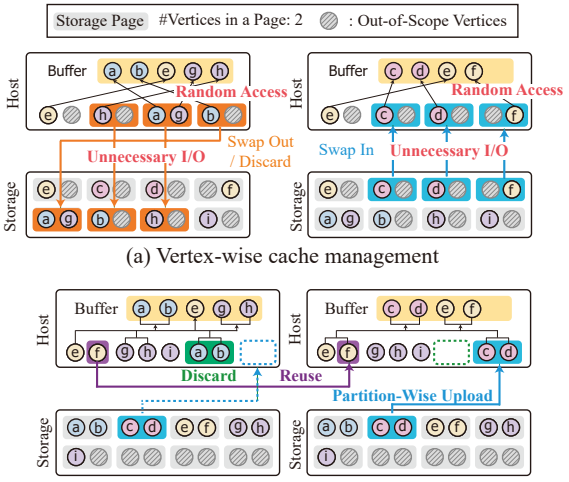


Figure 15. Partition dependency profile. (left) Products with 16 partitions, (mid) IGBM with 64 partitions, and (right) Papers with 2048 partitions. In the case of Papers, we only presented earlier 64 partitions for visibility.



(a) Vertex-wise cache management

(b) Partition-wise cache management

Figure 16. Advantage of partition-wise cache management compared to vertex-wise one.

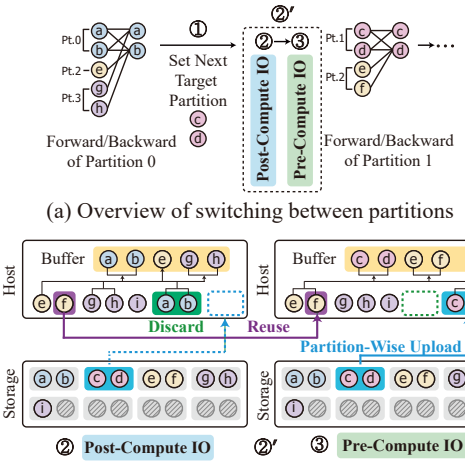
trend of power-law distribution clearer. For instance, in Figure 15(right), the Papers dataset with 2048 partitions shows a very vivid power-law distribution compared to the other two cases. This further enhances the scalability of GriNNder on large-scale graphs.

F Vertex-Wise Cache Management vs. Partition-Wise Cache Management

Figure 16 emphasizes the advantage of partition-wise cache management compared to the vertex-wise cache management. Since storage devices access data at a page granularity (e.g., 16KiB), vertex-wise cache management incurs a substantial amount of *unnecessary I/O*, denoted as 'Out-of-Scope' vertices. For instance, when processing the next partition, in Figure 16a, vertex-wise management needs to swap out (or discard) and swap in unnecessary data combined with the required data due to the page granularity of a storage device. In contrast, loading and evicting at a partition granularity alleviates such overhead.

G I/O Optimizations of GriNNder

G.1 Overlapping of Processing and Cache Management



(a) Overview of switching between partitions

(b) Overlapping cache management with computation

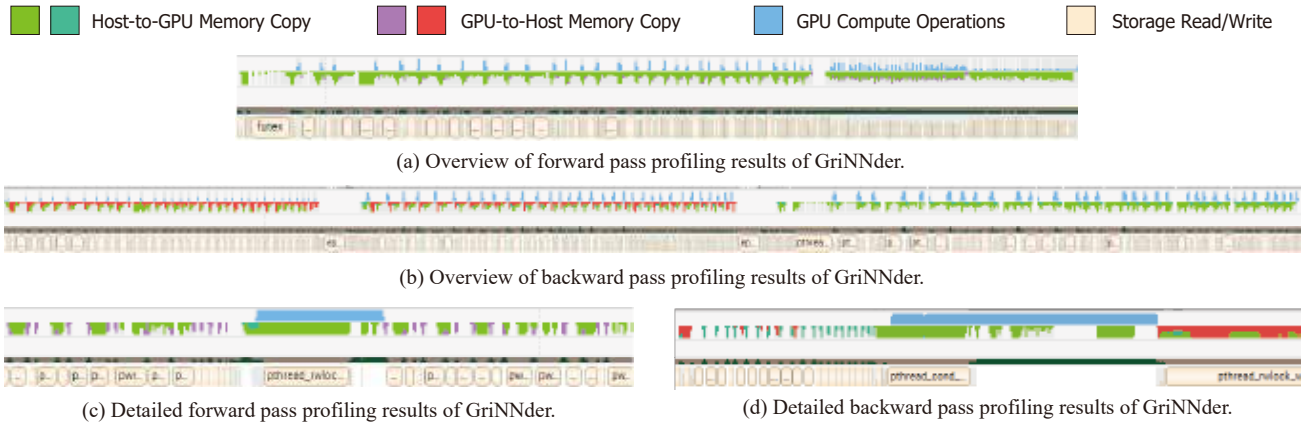
Figure 17. Overview of overlapping cache management with computation.

GriNNder schedules host memory cache evictions and prefetching to overlap with GPU computations, minimizing storage I/O latency as illustrated in Figure 17. ① We pick the next target partition to exploit already-cached neighbors, determined statically since 1-hop graphs are fixed. ② We discard partitions no longer needed. ③ We fetch only required partitions from storage while keeping reusable ones in the host memory. Because we keep a small extra buffer (dotted blue), uploading the dependency for partition 1 (pre-compute) does not have to wait for partition 0 computation and the succeeding evictions (post-compute), which enables overlapping these I/O operations (②') with ongoing computations. Also, we overlap the GPU compute and host-GPU I/O to further reduce latency.

We actually profile the training procedure of GriNNder, as illustrated in Figure 18. We profiled the 3-layer GCN on the IGBM dataset with #partitions=32. In both forward and backward passes, GriNNder overlaps the host memory and storage I/O with the GPU computation. Thus, in overall training, GriNNder enables aggressive latency overlapping of I/O and computation and provides superior training throughput.

G.2 In-Partition Vertex Ordering for Sequential Accesses

Another source of slowdown is in the gathering, which places vertex activations to be sent (GA) to the GPU in a dedicated host buffer. This involves multiple random memory accesses, as illustrated in Figure 16a, causing slowdown.



(a) Overview of forward pass profiling results of GriNNder.

(b) Overview of backward pass profiling results of GriNNder.

(c) Detailed forward pass profiling results of GriNNder.

(d) Detailed backward pass profiling results of GriNNder.

Figure 18. Profiling results of GriNNder's forward and backward pass.

To avoid this, after the graph is partitioned, we reorder the individual adjacency lists such that the neighbors are first sorted by their partition IDs and then by their vertex IDs. This replaces the random lookups with a single random lookup per partition, as in Figure 16b.

H In-depth I/O Volume and Memory Footprint Analyses

Table 6. **I/O analysis in forward pass**

Methods	GPU-Host	Host-Storage	GPU-Storage
HongTu w/ OS swap memory (i.e., `mmap`)	$(2\alpha+1)D$	$(2\alpha+1)D - Mem_{Host}$	
Ours	αD	$\alpha D - CacheHit$	D

* $|V||H| = D$. Topology data I/O is omitted for brevity.

Table 7. **Maximum memory usage analysis**

Methods	Host	Storage
HongTu	$(\alpha+1)D\|L\| + 2D$	
Ours	$D + D$	$D\|L\| + D$

* $|V||H| = D$. Considers activation and gradients.

Grad-engine activation regathering greatly reduces the I/O volume from snapshot store/load per layer and the memory footprint displayed in Table 6 and Table 7, where $D = |V||H|$.

I/O Volume: We assume host memory offloaded training (HongTu (Wang et al., 2023a)) to utilize OS swap memory (i.e., `mmap`), since it targets the host memory, not storage employment. In Table 6, compared to HongTu, the input activation-related GPU-host I/O volume ($2\alpha D$) is halved (αD) by skipping snapshots. GriNNder incurs $\alpha D - CacheHit$ amount of host-storage traffic for intra-layer partition-wise caching, but this is significantly less than utilizing `mmap` swap memory. Also, when host memory can handle the single-layer activations (D), this term becomes D from a full hit. When the host memory offloaded training faces the memory limit (Mem_{Host}), it needs to swap around $(2\alpha+1)D - Mem_{Host}$ data from/to storage. Given that α is around 3-10, the improvement is significant.

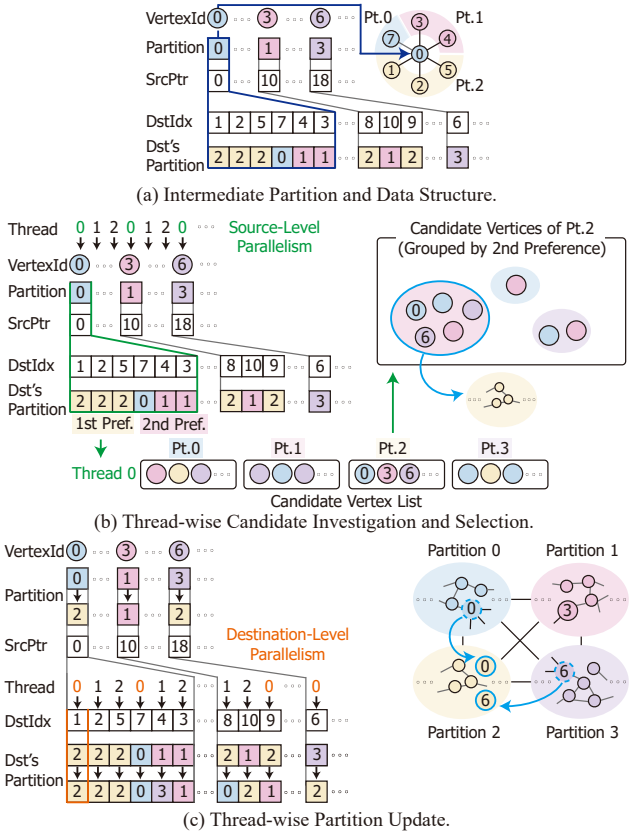
Memory Footprint: In Table 7, we report the peak memory usage of host offloaded training (Wang et al., 2023a) (HongTu) and GriNNder. For HongTu, the overhead mostly comes from storing snapshots for all layers. These redundant snapshots consume additional $\alpha D|L|$ on top of $D|L|$ activations. It needs to save $2D$ of gradients in backward pass to handle input and output gradients. In contrast, with grad-engine activation regathering and partition-wise graph caching, GriNNder consumes up to $D + D$ host memory for saving layer-wise activations and gradients. Regarding storage usage, GriNNder consumes $D|L|$ for saving activations and D for single-layer gradients.

I Insights and Details of Switching-Aware Partitioning

We draw inspiration from streaming partitioning (Spinner (Martella et al., 2017)), which applied traditional label propagation (Zhu & Ghahramani, 2002) to partitioning in distributed cloud graph systems (e.g., Pregel (Malewicz et al., 2010)). While lightweight label propagation suits our host memory constraints, Spinner's message-passing-based design is unsuitable for such limited environments.

Hence, we propose switching-aware partitioning, which adapts label propagation for limited resources with memory usage similar to CSR. We also introduce a group-wise propagation strategy suited for storage-offloaded full-graph training.

Switching-aware partitioning aims to find vertices with similar properties in different partitions and relocate them to



(a) Intermediate Partition and Data Structure.

(b) Thread-wise Candidate Investigation and Selection.

(c) Thread-wise Partition Update.

Figure 19. Switching-aware partitioning.

the same partition. Additionally, we need to balance the size of each partition to reduce the workload imbalance between partitions. To do so, we iteratively refine the partitions by selectively relocating vertices within a certain limit.

Figure 19 shows the detailed procedure of the proposed switching-aware partitioning. At first, we set the initial partitioning state ($S_0 = P_0, ..., P_{p-1}$) by randomly assigning each vertex to different partitions. We want to achieve high-quality partitioning while maintaining the number of vertices of all partitions close to $|V|/p$. $|\cdot|$ means the #vertices in a partition (or a graph), and p is the #partitions. We additionally define the maximum capacity term as β and set the maximum capacity limit of a single partition as $\beta \times |V|/p$. Here the capacity of a partition refers to the number of vertices allocated to said partition. In a state S_i, each partition j has the available relocation capacity ($RC_{(i,j)}$) as follows:

$$RC_{(i,j)} = \beta \times |V|/p - |P_j|, (0 \leq j < p) \qquad (1)$$

This is used to limit the number of vertices moved to the current partition. Figure 19a illustrates the intermediate state (S_i) where each partition has the available relocation capacity of six ($RC_{(i,j)} = 6$). Following the CSR format, our data structure comprises source pointers (`SrcPtr`) and destination indices (`DstIdx`). We manage another array (`Dst's Partition`) and fill this array with the partition of each destination index in `DstIdx`. For example in Figure 19b, the vertex 0 has neighbors of vertex $\{1, 2, 5, 7, 4, 3\}$. For each neighbor, we fill the `Dst's Partition` with its partition $\{2, 2, 2, 0, 1, 1\}$.

From a state (S_i) (Figure 19a), we calculate *kth preference* of a vertex: among the neighbors of the vertex, the partition ID of the kth largest frequency is the kth preference of the vertex. Then, using each vertex's first preference, each partition manages its own relocation candidate vertices from other partitions. In Figure 19b, vertex 0's neighboring vertices' partitions are $\{2, 2, 2, 0, 1, 1\}$. Among them, the partition that occurs most frequently is 2. Therefore, we put vertex 0 to the partition 2's relocation candidate (0 is now included in Pt.2 List in Figure 19b).

When selecting the final vertices to be relocated among candidates, we select them in a group-wise manner. In Figure 19b, we first use the 2nd preference partition of each vertex as a feature to help cluster vertices into different groups, unlike the baseline streaming partitioning algorithm. We then choose the largest group with the same 2nd preference to avoid vertices belonging to small, disparate clusters being relocated. In this example, vertex 0's 2nd preference partition is partition 1, and vertex 6's 2nd preference partition is also partition 1. Therefore, we put those two vertices into the same group. We choose to relocate the group including $\{0, 6\}$ because it is the largest group among the candidates. This provides a clustering effect and helps the convergence speed of partitioning. This can be generalized into comparing until kth preference, but we use $k = 2$ as default because it already empirically provides good performance.

To parallelize the procedure, we apply source-level parallelism, which distributes the source vertices to each thread. Each thread manages its own candidate lists for partitions as depicted in Figure 19b with the example of thread 0. We dedicate each thread to the equal available relocation capacity ($RC_{(i,j)}/\#threads$) to run threads in a fully parallel manner.

After selection, we update the relocation result to the `Dst's Partition` array. In Figure 19c, vertex 0 and 6 are selected to be relocated to partition 2. Therefore, we update the values of vertex 0 and 6 in `Dst's Partition` array to 2 (meaning partition 2). This procedure is conducted with destination-level parallelism, as illustrated in Figure 19c. After the update, using the updated data structure, iteration $i + 1$ proceeds. For each iteration i, we conduct the following procedure until reaching the termination condition, which will be discussed in the next subsection.

We discussed switching-aware partitioning as a procedural view. In the detailed algorithm, we need a penalty term for suppressing the propagation to reduce the imbalance among

the number of vertices in partitions. Therefore, in a state S_i, for a vertex v, the scoring term ($Score_{(v,I,j)}$) for each partition j and the final objective are as follows:

$$\begin{aligned} Penalty_{(i,j)} &= |P_j|/(\alpha_{balance} \times |V|/p), (0 \le j < p) \\ Score_{(v,i,j)} &= 1 + \#N(v,j)/\#N(v,\cdot) - Penalty_{(i,j)} \\ maximize &\sum_{v \in G} Score_{(v,i,j=partition_v)} \end{aligned} \tag{2}$$

where $\#N(v,j)$ denotes the frequency of partition j among the neighbors of the vertex v and $Penalty_{(i,j)}$ denotes the penalty term of state S_i of partition j. The penalty term reduces the preference when a partition already reaches the additional capacity $\alpha_{balance}$. The objective function calculates the total sum of the internal preferential scores of partitions. The partitioning halts when the objective does not improve over $\epsilon = 0.001$ for $w = 5$ times.

J Effect of Partitioning on Host↔GPU Traffic

Table 8. **Training time breakdown of storage-offloaded training.** Storage and Host↔GPU denote the storage I/O and host↔GPU I/O.

	Storage	Host↔GPU	Compute	Sync.	Etc.
w GriNNder	19.2%	48.4%	8.6%	12.0%	11.8%
w/o GriNNder	85.4%	—	2.1%	12.5%	

Partitioning is significantly helpful in reducing host ↔ GPU traffic. Once partition-wise scheduling reduces random storage I/O and host-memory caching minimizes storage traffic, host ↔ GPU traffic becomes the main bottleneck as illustrated in Table 8. We profiled a single epoch of storage-offloaded training of the IGBM dataset without/with GriNNder's optimizations. Since GriNNder is highly optimized to overlap compute and I/O, we employed a CPI-stacking-like method (Eyerman et al., 2006) to roughly break down execution time. When applying GriNNder optimizations, reduced random storage accesses from partition-wise I/O and minimized storage I/O from host memory caching make the host-to-GPU traffic the main bottleneck.

Without effective partitioning, the input activations required per partition—scaled by the expansion ratio (α)—would dramatically increase host ↔ GPU traffic, severely harming performance. In detail, for every vertex that belongs to a partition, we need to fetch all dependent extra vertices through the host ↔ GPU path redundantly. The factor that affects the *extra fetching* is our expansion ratio (α),

$$\alpha = \frac{\text{\# required vertices}}{\text{\# target vertices}}. \tag{3}$$

Put differently, α tells us how much larger the input tensor (to be sent between host ↔ GPU) becomes after we consider dependencies. The total input-activation traffic for one partition is then

$$\text{Traffic} = \alpha \times |H| \times |N_{target}|, \tag{4}$$

where $|H|$ is the hidden dimension and $|N_{target}|$ is the number of vertices the partition owns. Partitioning is designed to drive down α by co-locating vertices that are frequently accessed together as much as possible.

In practice, this helps GriNNder to provide stable training speed, even when long-tail effects may cause some partition dependencies to span multiple partitions. For instance, on the Papers dataset (2048 partitions), a target partition depends on 1602.07± 234.11 (Std.) other partitions. With the effect above, GriNNder provides significant speedup over other baselines by reducing storage I/O and the traffic between host and GPU.

K API Interface Details

```python
from torch_geometric.nn import GCNConv
from torch_sparse import SparseTensor
from models import GriNNderGNN
from utils.loader import GriNNderLoader

class GCN(GriNNderGNN):
    def __init__(..., loader: GriNNderLoader, ... ,
                 use_cache: bool, storage_offload: bool, ...):
        super().__init__(... , loader, ..., use_cache, storage_offload, ...)

        for i in range(num_layers):
            conv = GCNConv(in_dim, out_dim)
            self.convs.append(conv)

    def forward(self, x: Tensor, adj: SparseTensor, ...):
        for (conv, ...) in zip(self.convs[:-1], ...):
            h = conv(x, adj)

    def forward_layer(self, layer, x: Tensor, adj: SparseTensor, ...):
        h = self.convs[layer](x, adj)
```

Inheriting GriNNderGNN
GriNNderLoader for storage
Additional definition of forward_layer

Figure 20. User interface of GriNNder. Users inherit `GriNNderGNN` and implement `layer_forward` to enable layer-wise execution for partition-based full-graph training.

Figure 20 illustrates the user-facing API design. Existing PyG models require minimal changes: users inherit `GriNNderGNN` and implement the `layer_forward` method, which receives a partition ID and returns the layer's output for that partition. This enables the framework to orchestrate partition-wise execution transparently while maintaining compatibility with standard PyTorch training loops.

L Detailed Experimental Settings and Baselines

Models and datasets. We tested graph convolutional network (GCN) as the baseline GNN architecture and also used GAT (Veličković et al., 2018) and GraphSAGE (Hamilton et al., 2017). We set the hidden size as the widely-used 256, if not stated otherwise. We used three medium- (Products (Hu et al., 2020)) to large-scale (IGBM (Khatua et al., 2023) and Papers (Hu et al., 2020)) datasets (details in Table 9). Products is a co-purchasing network where vertices

Table 9. **Real-world graph datasets and hyper-parameters**

Name	Dataset Info.			Hyper-parameter		
	#Nodes	#Edges	Feat. size	lr	Dropout	#Epochs
Products (Hu et al., 2020)	2,449K	61.9M	100	0.003	0.3	500
IGBM (Khatua et al., 2023)	10,000K	120.1M	1024	0.01	0.5	500
Papers (Hu et al., 2020)	111,000K	1,600M	128	0.01	0.5	500

represent Amazon products and edges indicate products purchased together. IGBM and Papers are citation networks with vertices and edges representing research papers and citations, respectively. We also utilized Kronecker random graphs (Leskovec et al., 2010) (average degree=10) with random initial features of dimension 128 and #classes of 10 for scalability and versatility tests with ablation.

Hardware. We run main experiments on a single GPU workstation with an AMD Ryzen9 7950X 3D CPU (16C 32T), 128GB DDR5-5600 RAM, one RTX A5000 (24GB) GPU, a PCIe 5.0 NVMe SSD (4TB), and a total 4TB swap space for swap-based experiments. We utilized the NVMe SSD for the swap memory and GPUDirect Storage (GDS) (NVIDIA, 2021) and AIO (aio). We chose a single GPU setup to demonstrate how GriNNder breaks through the host/GPU memory limitations. For the multi-GPU extension, we utilized a multi-GPU server with four RTX4090 GPUs, 2×Intel Xeon Gold 6442Y, 512GB DDR5 DRAM, and 2TB PCIe5.0 NVMe SSD. We used a four-server cluster to test distributed full-graph training baselines, each server having four RTX A6000 GPUs, which aggregates to 16 GPUs. Intra-server GPUs are connected via NVLink Bridge (NVIDIA, 2023b), and servers are connected via Infiniband SDR (NVIDIA, 2023a). Each server has 512GB DDR4 RAM and an EPYC 7302 (16C 32T). For IGBM/Papers, we needed all 16 GPUs to fit the data in the GPU memory. For Products, using fewer GPUs could yield better performance, but we used all GPUs to maintain consistency among datasets.

Baselines. We compared four single-server baselines with GriNNder (denoted as 'GRD'). For MFG-based full-graph training, we used Betty (Yang et al., 2023) (called micro-batch training), the state-of-the-art full-graph training in limited environments, as our baseline. As Betty sometimes shows significant slowdowns due to slow MFG generation, we excluded the MFG generation time for comparison. To test extension of storage-based mini-batch training to full-graph training while utilizing SSD, we extend Ginex (Park et al., 2022) to micro-batch training (Yang et al., 2023). For host offloaded full-graph training, we faithfully implemented HongTu (Wang et al., 2023a) and used it as a baseline. When the training data overflows the host memory, we use storage swap memory to compare it with GriNNder regarding storage usage. We also tested the naïve extension of ROC (Jia et al., 2020a) to naïvely just use storage for offloading, but reported the results of it only in Appendix X because this extension was much slower than the others. In the appendix, we additionally tested two storage-based mini-batch training (DiskGNN (Liu et al., 2025) and GNNDrive (Jiang et al., 2024)) with micro-batch extension (Appendix C).

We also compared GriNNder with two distributed full-graph training baselines, CAGNET (Tripathy et al., 2020) and Sancus (Peng et al., 2022). CAGNET is one of the famous distributed full-graph training methods, and Sancus accelerated it by storing stale activations and gradients to reduce the communication bottleneck. Note that while Sancus is not the exact full-graph training from using stale activations and gradients, we still included it as it is one of the state-of-the-art distributed full-graph training frameworks. These two baselines ran on the cluster mentioned above. When GPU out-of-memory issues arise in distributed training baselines, we implement host memory activation checkpointing (indicated by '*') to attempt to make them executable.

For partitioning, we utilized the multi-threaded METIS (MT-METIS) (LaSalle & Karypis, 2013) as the baseline, which is one of the state-of-the-art METIS parallelizations (denoted as 'METIS'). Even when it does not run on the testbed due to insufficient memory, we assume it was preprocessed in another environment since all baseline methods rely on METIS. For comparisons with lightweight partitioners, we benchmarked Spinner (Martella et al., 2017) and an out-of-core partitioner (2PS-L (Mayer et al., 2022)).

M Comprehensive Analysis with Synthetic Graph on Scalability, Ablation, and Configuration

We conducted a comprehensive analysis using synthetic graphs, as summarized in Table 10. The tests utilized Kronecker synthetic graphs (Leskovec et al., 2010) with sizes ranging from 2^{22} to 2^{25} nodes (4.2–33.6M) and an average degree of 10.

Across all combinations of layers and datasets, all ablations of GriNNder consistently achieved significant speedups over HongTu. For smaller datasets, where host memory can store all intermediate activations and gradients, the configuration using only grad-engine activation regathering ('GRD-G') generally outperforms the storage-enabled version ('GRD-GC'), primarily due to cache management overhead. However, for larger datasets, employing storage alleviates host memory cache pressure, allowing the storage-based configuration ('GRD-GC') to deliver substantial speedups over both HongTu and GRD-G.

These results demonstrate that GriNNder is highly scalable for large datasets, with storage utilization being an effective strategy for handling large graphs on a single GPU. We also

Table 10. **Training time/epoch (min) for various-sized Kronecker synthetic graphs.** '-' denotes when the number of partitions is not enough for running. **Bold** is the fastest training time in each (#layers, dataset) pair.

#Layers	#Partitions	Method	4.2M	8.4M	16.8M	33.6M
3	16	HongTu	0.43	0.83	-	-
		GRD-G	**0.29**	**0.59**	-	-
		GRD-GC	0.31	0.63	-	-
	32	HongTu	0.57	1.11	7.25	-
		GRD-G	0.31	0.66	-	-
		GRD-GC	0.33	0.71	-	-
	64	HongTu	0.76	1.76	10.70	-
		GRD-G	0.41	0.77	-	-
		GRD-GC	0.43	0.81	-	-
	128	HongTu	1.05	5.32	18.96	36.31
		GRD-G	0.55	1.02	**1.93**	**3.73**
		GRD-GC	0.58	1.05	1.99	3.86
5	16	HongTu	0.83	1.99	-	-
		GRD-G	**0.57**	**1.14**	-	-
		GRD-GC	0.60	1.20	-	-
	32	HongTu	1.07	8.04	19.15	-
		GRD-G	0.60	1.30	-	-
		GRD-GC	0.63	1.37	-	-
	64	HongTu	1.48	11.43	24.08	-
		GRD-G	0.79	1.49	-	-
		GRD-GC	0.84	1.55	-	-
	128	HongTu	4.61	17.08	37.09	96.99
		GRD-G	1.08	1.96	**3.71**	10.87
		GRD-GC	1.13	2.02	3.82	**7.76**

observed that GriNNder occasionally requires a larger number of partitions (i.e., different configurations) than HongTu. This is due to the GPU memory overhead introduced by overlapping GDS operations and computation. Despite this, GriNNder continues to deliver significant performance improvements over HongTu. It is also important to note that the number of partitions is merely a configuration hyperparameter, and users are not burdened by the need to manually handle this difference.

N Cache Hit Rates

Table 11. **Cache hit rate.**

	Products	IGBM	Papers	kron-4.2M	kron-8.4M	kron-16.8M	kron-33.6M
Hit (%)	28.57	53.70	83.63	80.81	80.47	92.77	92.70

We report cache hit rates in Table 11. As larger datasets (> IGBM, 10M) incur more reuse from the higher number of partitions, the hit rate is more significant in them. A low hit rate is natural in small datasets (e.g., Products) because we employ only a few partitions, and most data are not reused. Thus, GriNNder's caching is promising in large-graph training.

O Convergence Trend and Practical Overhead of Switching-Aware Partitioning

Table 12. **Partitioning convergence trend.**

Dataset		Improvement (%) for Iterations										
Products (4 parts)	Iter.	1	5	10	15	20	25	28 (last)				
	Improve (%)	6.81	9.75	3.79	0.36	0.12	0.08	0.05				
IGBM (32 parts)	Iter.	1	5	10	15	20	25	30	35	40	45	50 (last)
	Improve (%)	11.13	7.78	3.66	1.96	0.66	0.77	0.39	0.21	0.16	0.10	0.08
Papers (2K parts)	Iter.	1	5	10	15	20	25	30	35	40	45	50 (last)
	Improve (%)	18.04	2.86	3.96	1.61	1.78	0.89	0.46	0.72	0.43	0.22	0.14

Switching-aware partitioning converges fast with low practical overhead. In Table 12, we report the trend of the partitioning quality (score of the objective function) improvement (convergence) from the adjacent previous iteration (e.g., iter 4 $\rightarrow$ 5). We observe that at most 50 iterations are enough for convergence, thus limiting partitioning to 50 iterations in our experiments.

Given that a single iteration takes 0.08sec/0.14sec/21.12sec on average and our lightweight partitioning only requires 2.49sec/6.96sec/17.60min, partitioning consumes 0.07/0.02/0.39% of the total training time (500 epochs) on Products/IGBM/Papers, respectively.

P Multi-GPU Extension

Our multi-GPU implementation employs two lightweight mechanisms:

(i) Partition parallelism: We divide the partitions into disjoint #(GPU) sets; each GPU performs forward/backward on its set independently. (ii) Weight/Gradient synchronization: During the backward pass, partial gradients of dependent vertices from different GPUs are atomically accumulated on the host, which accumulates the gradients of the vertices. Before the weight update, a weight gradient all-reduce operation is conducted between GPUs to synchronize the weights among them.

These straightforward enhancements enable effective multi-GPU execution with minimal overhead.

Q Configuration Sensitivity Results

We additionally conducted configuration sensitivity experiments in Table 13. From the efficient caching management and elimination of redundancy, GriNNder is much less sensitive to the number of partitions (configurations). This enhances the practicality of GriNNder for end-users as they are not required to carefully configure the number of partitions.

Table 13. **Configuration sensitivity on training time (sec).** The default number of partitions for PRODUCTS and IGBM are 2 and 32, respectively.

		Method	×1	×2	×4	×8
3-layer	PRODUCTS	HongTu	9.98	11.11	12.22	13.65
		GRD	**6.93**	**7.72**	**8.55**	**8.99**
	IGBM	HongTu	387.68	694.02	675.98	876.60
		GRD	**55.62**	**59.41**	**61.06**	**66.39**
5-layer	PRODUCTS	HongTu	19.14	21.46	23.42	26.22
		GRD	**13.65**	**15.22**	**16.38**	**17.60**
	IGBM	HongTu	894.09	958.20	1183.88	1425.36
		GRD	**91.46**	**92.60**	**98.99**	**114.76**

R HETEROGENEOUS GNN EXTENSION

We extended GriNNder to support heterogeneous graph training. Our implementation involved creating a heterogeneous graph dataloader (HeteroDataloader). We validated this extension using the IGBM-hetero dataset with a two-layer heterogeneous convolutional model, consisting of a GCN layer (paper-cite-paper relation) and GraphSAGE layers (other relations). With hidden dimensions set to 128 (except for the output layer with 19 classes), GriNNder achieved 71.58% accuracy after 100 epochs, while significantly reducing runtime compared to HongTu (26.92 sec/epoch vs. 55.41 sec/epoch).

S BENCHMARKING W/O GDS

GriNNder can be generally used when GDS is unavailable. In this case, Kvikio (used in GriNNder) automatically switches to POSIX. Thus, users can still utilize GriNNder without any modification. Also, please note that GDS is supported on GPUs with NVIDIA compute capability ¿6.x (e.g., V100 and after).

Table 14. **Sensitivity to GDS.**

Layers	GDS	Products	IGBM	Papers
3 layer	GDS	0.12	0.93	9.07
	w/o GDS	0.12	0.93	10.25
5 layer	GDS	0.23	1.52	12.03
	w/o GDS	0.23	1.52	13.73

We also benchmarked the performance (min) of GriNNder without GDS support in Table 14 as 'w/o GDS'. As Products and IGBM can be handled with host memory, the 'w/o GDS' performs similarly to the GDS cases. Even with Papers, where storage is highly utilized, there is only a 13-14% slowdown, demonstrating GriNNder's versatility.

T EFFECT OF GRAPH DISTRIBUTIONS

While power-law degree distributions are widely observed in real-world large-scale graphs (Leskovec et al., 2005), we also evaluate the robustness of GriNNder on non-power-law graphs. GriNNder's partitioning and caching strategies operate independently of specific graph structural properties, enabling performance gains across diverse graph patterns. To evaluate behavior on non-power-law graphs, we benchmark HongTu and GriNNder on a 10M Watts-Strogatz graph (Watts & Strogatz, 1998) (average degree=16) with randomized features using GCN in Table 15. GriNNder achieves 3.21×/7.01× speedups for 3-/5-layer models, demonstrating robustness beyond power-law graphs.

Table 15. **Performance on non-power-law graph.**

Time/Epoch (min)	HongTu	**GriNNder**	Speedup
3-Layer	3.50	**1.09**	3.21×
5-Layer	14.72	**2.10**	7.01×

U APPLICATION TO DYNAMIC GRAPHS

GriNNder can be extended to dynamic graphs by modifying the data-loading and partitioning components for graph updates. Switching-aware partitioning is especially well-suited for this: its streaming nature enables effective adaptation to incremental changes by starting from existing partitions and converging to high-quality solutions in a few lightweight iterations. Thus, we expect GriNNder to handle dynamic graphs without significant overhead.

V APPROACHES TO RESEMBLE FULL-GRAPH TRAINING WITH ALGORITHM CHANGE

Many works have been proposed to resemble the accuracy (effect) of full-graph training by addressing the information loss of mini-batch training. GNNAutoScale (Fey et al., 2021) utilizes stale activation to compensate for the information loss of mini-batch training. LMC (Shi et al., 2023) further addresses the information loss by compensating for it with gradients. In distributed full-graph training, many researchers have tried to address the communication bottleneck while resembling the full-graph training accuracy with staleness (Wan et al., 2022b; Peng et al., 2022) and error compensation (Wan et al., 2022a) through proportional dropping of communication. While the above compensation methods could be orthogonally applied to further enhance the performance of GriNNder, we did not apply them to implement the exact full-graph training without algorithm change.

W Functionality (Accuracy) Check of GriNNder

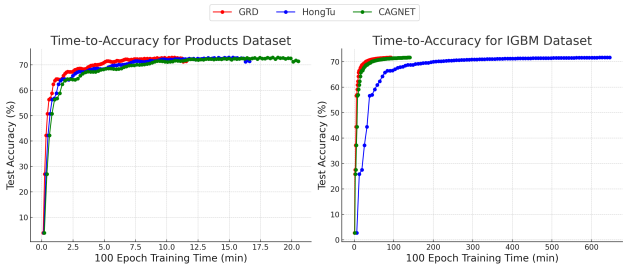


Figure 21. Functionality check of GriNNder.

While GriNNder does not change the algorithm of full-graph training, we tested the accuracy of GriNNder compared to full-graph training and HongTu for the functionality check, as illustrated in Figure 21. Full-graph training was conducted with a CAGNET distributed baseline because Sancus is not exact full-graph training. As depicted in Figure 21, while HongTu is much slower than the distributed setup, GriNNder provides significant speedup over the distributed CAGNET. Both baselines and GriNNder show the same accuracy, which demonstrates the correct functionality of GriNNder. We also additionally checked Sancus's result, which is not exact full-graph training as it utilizes stale activations and gradients. It shows similar accuracy to others but not exactly the same as them. On the Papers dataset, GriNNder achieves identical accuracy (63.04%) as CAGNET. We excluded the Papers in the figure because HongTu consistently encountered out-of-memory issues.

X Comparison with Naïve Baseline (Naïve Storage Extension of ROC (Jia et al., 2020a))

We also tested the naïve storage extension of ROC (Jia et al., 2020a) instead of HongTu, which is the state-of-the-art framework. While we tested with HongTu with OS-based swap (i.e., `mmap`), we made it directly utilize storage instead of OS-based management for the ROC extension. On this naïve extension, GriNNder provides 1.28/29.00× speedup on 3-layer GCN on Products and IGBM, respectively. The speedup is significant on IGBM because Products only use #partitions=2 while IGBM uses #partitions=32. Thus, GriNNder provides further speedup on IGBM, which has much redundancy issue with ROC.

Y Limitation

We evaluate an extensive set of datasets and demonstrate the effectiveness of our partition-wise cache management. We also found that partition-wise cache management remains effective even when long-tail effects may cause some partition dependencies to span multiple partitions (Appendix J). However, there can be a worst-case scenario: when dependencies are uniformly distributed across many partitions. In this case, partition-wise management may lead to overhead rather than performance improvement. We leave the handling of such a case to future work.

Z Additional Related Work

This section provides comprehensive discussion of related work beyond the core contributions in Section 10.

Full-Graph GNN Training Systems. Numerous methods have been proposed for learning representations from graphs (Hu et al., 2020; Ying et al., 2018; Song et al., 2022; Frasca et al., 2020). Full-graph training preserves complete input information, preferred for algorithmic validation (Wan et al., 2022b; Jia et al., 2020a; Wan et al., 2022a; Ma et al., 2019). Distributed approaches (Peng et al., 2022; Tripathy et al., 2020; Demirci et al., 2022; Wan et al., 2023; Liu et al., 2021b; Wang et al., 2022a) partition graphs across machines but incur substantial inter-device communication overhead. Single-server methods enable full-graph training without distribution costs. Betty (Yang et al., 2023) employs batch-level graph partitioning but suffers from neighbor explosion across layers. HongTu (Wang et al., 2023a) stores activations/gradients to host memory but remains limited by host capacity. Hardware acceleration through near-memory processing (Zhou et al., 2022), computational storage (Kwon et al., 2022; An et al., 2023), or smart storage devices (Lee et al., 2021) achieves high performance but requires specialized hardware unavailable to most research groups. GriNNder targets commodity systems with NVMe SSDs through software optimization.

Storage-Based Training Systems. Training large models with storage has become prevalent. Large language model training systems (Rajbhandari et al., 2021) partition optimizer states, gradients, and parameters across memory hierarchy including NVMe storage. FlexGen (Sheng et al., 2023) enables large model inference through offloading and scheduling. However, these LLM techniques fundamentally differ from GNN requirements. LLM activations exhibit sequential layer dependencies enabling straightforward layer-by-layer management, while GNN layers exhibit graph-structured dependencies requiring gathering from multiple partitions based on topology.

Mini-batch GNN systems often employ storage for initial feature management. Ginex (Park et al., 2022) introduces unified storage-memory-GPU hierarchy with feature caching. DiskGNN (Liu et al., 2025) employs pre-sampling and IO-aware scheduling. GNNDrive (Jiang et al., 2024)

optimizes feature storage layouts. MariusGNN (Waleffe et al., 2023) loads valid features with two-level partitioning. Helios (Sun et al., 2023b) enables direct GPU access to storage. These systems manage only initial features, as sampled subgraphs fit in GPU memory. The computational pattern involves: (1) sampling subgraphs, (2) fetching initial features, (3) performing forward-backward in GPU memory with all intermediate activations resident. Each mini-batch operates independently with no inter-batch dependencies for intermediate activations. On the other hand, full-graph training requires simultaneous management of all intermediate activations/gradients for all vertices across all layers with inter-partition dependencies spanning the entire graph, creating fundamentally different storage access patterns. Additional subgraph training frameworks (Hamilton et al., 2017; Zeng et al., 2020; Zheng et al., 2020; 2022; Kaler et al., 2022; 2023; Yang et al., 2022; Thorpe et al., 2021; Gandhi & Iyer, 2021; Lin et al., 2020; Sun et al., 2023a) address memory constraints through sampling (Zhu et al., 2019; Huang et al., 2018; Dong et al., 2021) but introduce input information loss.

Activation Management. Checkpointing trades computation for memory via recomputation (Chen et al., 2016; Bulo et al., 2018; Mohan et al., 2021; Gupta et al., 2024; Wang et al., 2023b; Nicolae et al., 2020; Eisenman et al., 2022), storing subset of activations and reconstructing others during backpropagation. Prior GNN checkpointing (Zhang et al., 2022; Wang et al., 2023a; Chakaravarthy et al., 2021; Wang et al., 2022b; Jiang et al., 2020) applies checkpointing to GNN training. HongTu (Wang et al., 2023a) stores snapshots of offloaded partitions but suffers from massive redundancy. Redundancy-free computation graphs (Jia et al., 2020b) address computational redundancy in message passing operations, whereas GriNNder targets storage redundancy elimination. GriNNder introduces regathering for gradient computation, regenerating gathered activations on-demand from compact partition outputs, reducing storage I/O.

Alternative memory optimization strategies include pruning (Han et al., 2016; Fang et al., 2023; He et al., 2019; 2017; Sanh et al., 2020; Liu et al., 2021a), quantization (Choi et al., 2022; 2021; Courbariaux et al., 2015; Lin et al., 2016; Zhou et al., 2016; Zhuang et al., 2018; Shen et al., 2020; Liu et al., 2022), and memory-efficient backpropagation (Gruslys et al., 2016; Gomez et al., 2017; Chakrabarti & Moseley, 2019). These techniques complement GriNNder by reducing model requirements but do not address memory scalability challenges from intermediate activations proportional to graph size and hidden dimensions.

Graph Partitioning. Partitioning is widely adopted for distributed graph processing (Ma et al., 2019; Wang et al., 2023a; Yang et al., 2023; Karypis & Kumar, 1998; Zhang et al., 2023; Jia et al., 2017; Chen et al., 2018; Zhang et al., 2020; Tripathy et al., 2020; Liu et al., 2021b; Wang et al., 2022a; Wan et al., 2023). METIS (Karypis & Kumar, 1998) employs multi-level framework with coarsening, partitioning, and refinement phases (Davis et al., 2020; Heuer & Schlag, 2017; Shaydulin & Safro, 2018; Akhremtsev et al., 2017), producing high-quality partitions minimizing edge cuts. Many distributed GNN frameworks (Wan et al., 2022b; Peng et al., 2022; Zheng et al., 2020; 2022; Liu et al., 2021b; Wan et al., 2023; Wang et al., 2022a) employ METIS for minimizing communication cost. G3 (Wan et al., 2023) proposes iterative METIS-based partitioning enhancing three-dimensional parallelism. However, METIS requires substantial memory—measurements indicate 4.8×–13.8× input graph size (Kaur & Gupta, 2021; LaSalle & Karypis, 2013). Alternative frameworks (Zhang et al., 2023; Jia et al., 2017; Chen et al., 2018) attempt load balancing but demand large memory. Streaming algorithms (Echbarthi & Kheddouci, 2016; Stanton & Kliot, 2012) and online partitioning (Tsourakakis et al., 2014) reduce memory requirements through single-pass processing. Label propagation methods (Karypis & Kumar, 1999; Raghavan et al., 2007) and scalable frameworks (Martella et al., 2017) offer alternative approaches. These methods target distributed systems or assume sufficient memory for loading entire graph structure. GriNNder introduces lightweight partitioning requiring only small working set memory, enabling partitioning in memory-limited environments where METIS cannot execute.