REAL: Efficient Distributed Training of Dynamic GNNs with Reuse-Aware Load Balancing Scheduling

Abstract

Dynamic Graph Neural Networks (DGNNs) have emerged as powerful models for learning from evolving graphs by jointly capturing structural and temporal dependencies. However, training DGNNs at scale is challenging due to three conflicting requirements: maximizing resource utilization, minimizing data transfers, and maximizing data reuse. Existing distributed systems either overlook these objectives or handle them in isolation, thus leading to severe inefficiencies. We present REAL, an efficient distributed system for DGNN training. REAL exploits snapshot- and group-level reuse, and incorporates a cross-group scheduling algorithm that co-optimizes data reuse and resource utilization while avoiding extra data transfer overhead. Furthermore, we design two complementary scheduling policies: REAL-L, which leverages ILP for globally optimal planning, and REAL-G, a greedy fallback suitable for large-scale scenarios. Extensive experiments on six dynamic graph datasets show that REAL-L and REAL-G achieve up to 4.53× and 3.99× speedup over state-of-the-art baseline, respectively.

1 INTRODUCTION

Dynamic graphs are graphs whose structures and attributes change continuously over time. Dynamic Graph Neural Networks (DGNNs) have emerged as state-of-the-art methods for processing dynamic graphs, exhibiting a strong ability to capture both structural and temporal dependencies [2, 4, 11, 17, 18, 22, 26, 31, 32, 34, 36–41, 43, 44]. Depending on the event model, dynamic graphs are categorized into discrete-time dynamic graphs (DTDGs) and continuous-time dynamic graphs (CTDGs). DGNNs are categorized similarly according to the dynamic graphs they process. In this work, we focus on DGNNs for DTDGs, the widely used form in existing benchmarks and frameworks [1, 6, 35], which model temporal dynamics with discrete snapshots.

Significant efforts have been made to improve the efficiency of DGNN training. Some work focuses on efficient training on *a single GPU* [8, 9, 12, 16, 24, 30, 33], addressing various factors such as memory footprint and data access overhead. However, as datasets and models grow, distributed training across multiple GPUs has become increasingly important [3, 5, 7, 28]. Representative distributed systems adopt different partitioning and scheduling strategies, each with unique trade-offs. ESDG [3] distributes temporally adjacent snapshots across different GPUs, which requires transferring expensive hidden states. This also causes GPU idling and hence low utilization, as RNN's sequential dependency forces each GPU to wait for the previous hidden state. BLAD [7] avoids such transfer overhead by processing each group of temporally adjacent snapshots on the same GPU while assigning different groups on different GPUs. However, variations across groups lead to load imbalance, which also results in inefficient resource utilization. While DGC [5] adopts chunk-based graph partitioning to simultaneously pursue load balance and reduce cross-GPU data transfer, these two objectives are often conflicting during partitioning and interfere with each other. DynaHB [28] also achieves load balancing through graph partitioning with CPU vertex caching without incurring additional cross-GPU data transfer, but this introduces extra CPU-GPU data transfer overhead. Meanwhile, several works [12, 27, 33] have shown that temporally adjacent snapshots exhibit substantial topology similarity, revealing opportunities for reusing invariant data to reduce redundant computation and I/O. However, no existing distributed DGNN training system exploits such reuse, especially in conjunction with load balancing.

In this paper, we optimize the efficiency of distributed DGNN training by simultaneously addressing three key objectives: (1) *maximizing resource utilization*, (2) *minimizing data transfer overhead* (both cross-GPU and CPU-GPU), and (3) *maximizing data reuse*. Note these objectives are usually at odds: improving utilization often relies on fine-grained partitioning for load balance, which inflates cross-GPU data transfer; while enforcing reuse limits scheduling flexibility, making it harder to evenly distribute workloads.

To solve this challenging problem, we design REAL, a distributed DGNN training system that jointly optimizes the three objectives above and balances the trade-offs among them. Our key insight to leverage the inherent training process of DGNNs that *different snapshot groups are treated as independent training samples in DGNNs*, allowing them to be flexibly combined and scheduled in arbitrary order to meet the above objectives. **First**, REAL adopts *snapshot groups* as the basic scheduling unit, which naturally avoids cross-GPU data transfers. **Second**, REAL enables topology-aware data reuse at both snapshot and group levels. REAL designs new operators that explicitly exploit structural similarities between snapshots and overlapping snapshots across groups, significantly reducing redundant computation and data loading overhead. In addition, REAL overlaps data transfer with computation across heterogeneous resources in a pipeline manner, further improving resource utilization. **Third**, REAL performs cross-group combination to balance workload across GPUs. The cross-group scheduling process jointly considers inter-group data reuse and load balance to improve overall resource utilization. Based on these designs,

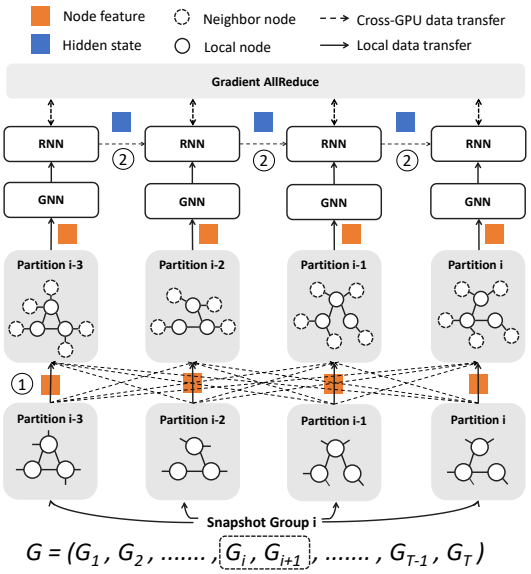


Figure 1. *Workflow of distributed training for DGNNs.*

we formulate an optimal scheduling problem to minimize per-epoch time. Depending on the scenario, REAL solves the problem using either Integer Linear Programming (ILP) or a greedy heuristic, resulting in two variants: REAL-L and REAL-G. Through real-world experiments, we demonstrate that REAL-L and REAL-G achieve 1.13×-4.53× and 1.11×-3.99× higher throughput, respectively, compared to SOTA method. In simulations, as GPU scale increases, REAL-G shows good scalability, scaling to 512 GPUs with 95% parallel efficiency.

We make the following main contributions.

- We identify key inefficiencies in existing distributed DGNN training methods, including insufficient resource utilization, redundant data transfers, and limited data reuse.
- We propose REAL, a system that integrates snapshot- and group-level reuse, and a cross-group scheduling algorithm to jointly optimize resource utilization and data reuse without incurring extra transfer overhead.
- We introduce two variants of the scheduling algorithm: REAL-L and REAL-G, based on ILP and a greedy heuristic, respectively. The system first attempts REAL-L and falls back to REAL-G when ILP solving exceeds the time limit.
- We evaluate REAL-L and REAL-G on six dynamic graph datasets and four DGNN models. REAL-L and REAL-G achieve up to 4.53× and 3.99× higher throughput compared to SOTA method, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Workflow of Distributed DGNN Training

Training a DGNN on multiple GPUs requires careful coordination of graph data across devices (Figure 1). The dynamic graph $\mathcal{G} = (\mathcal{G}_1, \ldots, \mathcal{G}_T)$ is split into snapshot groups, each processed in one iteration. For a node v in $\mathcal{G}_t$, the aggregation

$$\mathcal{H}_t(v) = SpatioAggr_v(\mathcal{W}_{gnn}, \{\mathcal{X}_t(u) | u \in \mathcal{N}(v)\}, \mathcal{X}_t(v)) \quad (1)$$

combines its features with those of its neighbors. If a neighbor lies on another GPU, *neighbor feature transfer* (①) is required, which can dominate runtime in dense graphs.

The graph embedding is then passed to a temporal model (*e.g.*, RNN) to capture time dependencies:

$$h_t = TemporalAggr(\mathcal{W}_{rnn}, \mathcal{H}_t, h_{t-1}), \quad h_0 = 0 \quad (2)$$

Non-initial steps require the previous hidden state; if on another GPU, *hidden state transfer* (②) is performed. Finally, loss is computed and gradients synchronized via allreduce.

2.2 Dataset Partition Strategy

In distributed training of DTDGs, the main dataset partitioning methods include *vertex-based partitioning*, *snapshot-based partitioning*, and *snapshot group-based partitioning*. The vertex-based partitioning method divides the dataset into subsets of nodes, each GPU responsible for processing these nodes and their associated edges, commonly used in frameworks like DistDGL and AliGraph [42, 45]. The snapshot-based partitioning method divides the dataset by snapshots, with each GPU processing a snapshot. This ensures that each GPU has all the node information for the snapshot, avoiding the overhead of transferring neighbor features, as proposed in ESDG [3]. BLAD [7] proposes a snapshot group-based partitioning aimed at reducing data transfer volume. In each iteration, each GPU processes a complete snapshot group, since each snapshot group includes all prior information of the target snapshot, eliminating hidden state transfer.

2.3 Dilemma in Distributed Training of DGNNs

Efficient distributed DGNN training requires jointly optimizing three objectives: high resource utilization, minimal data transfer overhead, and maximal data reuse. Existing approaches exhibit a range of strengths and weaknesses, as shown in Table 1; yet none of them address all key objectives.

Vertex-Based Partitioning. Approaches such as DistDGL adopt vertex-level partitioning to achieve fine-grained load balancing across GPUs by evenly distributing nodes and their associated computations. However, this strategy fragments the graph structure within each snapshot, resulting in frequent neighbor feature exchanges across GPUs. Consequently, it incurs substantial inter-GPU data transfer during message passing. We profile the forward pass of WD-GCN [18] on three real-world datasets and observe that DistDGL incurs notable data transfer overhead, accounting for an average of 34% of the total forward pass time (Figure 2).

Snapshot-Based Partitioning. To reduce neighbor feature data transfer, ESDG performs snapshot-level scheduling by assigning each snapshot to a single GPU, ensuring that neighbor aggregation is local. However, it introduces hidden state exchange across GPUs when modeling temporal dependencies, and variations in snapshot sizes causes inter-GPU load imbalance. Moreover, the sequential dependency of RNNs exacerbates resource underutilization, as GPUs often

Category	Dimension	AliGraph [45]	DistDGL [42]	ESDG [3]	DGC [5]	BLAD [7]	DynaHB [28]	REAL
Data Transfer	*No Neighbor Feature Transfer*	×	×	✓	×	✓	✓	✓
	No Hidden State Transfer	✓	✓	×	×	✓	✓	✓
	No Extra CPU-GPU Transfer	✓	✓	✓	✓	✓	×	✓
Resource Utilization	*Resource-Aware Optimization*	×	×	×	×	✓	×	✓
	Load Balance	✓	✓	×	✓	×	✓	✓
Data Reuse	*Topology-Aware Reuse*	×	×	×	×	×	×	✓

Table 1. *Comparison of existing solutions across six dimensions in three categories: data transfer (neighbor feature, hidden state, CPU–GPU transfer), resource utilization (resource-aware optimization, load balance), and data reuse (topology-aware reuse).*

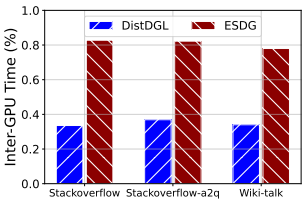


Figure 2. *Inter-GPU data transfer time during forward pass for DistDGL and ESDG.*

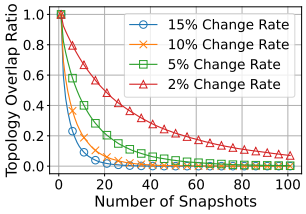


Figure 3. *Effect of change rate and snapshot count on graph topology overlap.*

idle while waiting for hidden states to be transferred. Using WD-GCN[18] on three real-world datasets, we observe that inter-GPU data transfer (including idle time) accounts for an average of 81% of forward pass time (Figure 2). In parallel, GPU utilization on Stackoverflow [29] dataset (Figure 4) also reveals severe underutilization of compute resources.

Snapshot Group-Based Partitioning. BLAD extends snapshot-level scheduling to snapshot groups to improve training efficiency. This approach reduces both neighbor feature transfer and hidden state data transfer, as snapshots within the same group are processed locally and sequentially. However, grouping multiple different snapshots together amplifies the imbalance problem: variations in graph size and structure accumulate within a group, making it harder to evenly distribute workload across GPUs. We also visualize GPU utilization under BLAD in Figure 4, which shows uneven utilization caused by imbalance.

Lack of Topology-Aware Reuse. No prior work on distributed DGNN training proactively exploits topological similarity for data reuse over time. DGC [5] reuses graph embeddings across epochs, but ignores topological similarity between graphs. In the single-GPU setting, PiPAD [33] adopts a global structured reuse scheme by decomposing each snapshot into the global intersection across time and its exclusive part, transmitting and computing the intersection only once. However, its scalability is limited: as snapshots grow, the globally shared intersection quickly shrinks. With fixed edge change rates (2%–15%), the intersection ratio drops below 20% after 30 snapshots (Figure 3), showing limited reuse potential at scale. Furthermore, this approach is incompatible with GATs, since attention normalization (*i.e., softmax*) requires full neighborhoods. Moreover, applying such single-GPU strategies in distributed settings may aggravate load imbalance as reuse gains vary across GPUs.

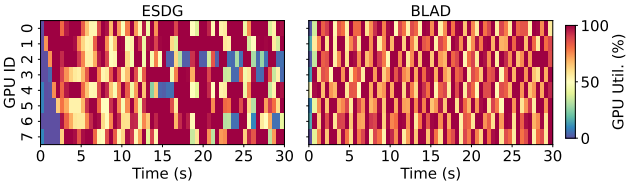


Figure 4. *GPU utilization of ESDG and BLAD on the Stackoverflow dataset.*

3 SYSTEM OVERVIEW

In this section, we first introduce our design principles (§3.1), then present the architecture of REAL in detail (§3.2), and detail REAL's pipelined execution design (§3.3).

3.1 Design Principles

Minimize Data Transfer. REAL employs a *snapshot group-based partitioning strategy* that limits cross-GPU data transfer to all-reduce gradient only. In addition, REAL proactively identifies *reusable graph structures* across snapshots to reduce redundant CPU-GPU data transfer.

Maximize Data Reuse. REAL proactively considers *both snapshot-level and group-level topology similarity*, optimizing data reuse and reducing redundant computations.

Maximize Resource Utilization. REAL improves resource utilization through two techniques: (1) a *triple-stream pipeline* that overlaps heterogeneous operations, and (2) *cross-group scheduling* that balances workloads across GPUs.

3.2 REAL Architecture

Figure 5 (top) provides a high-level illustration of REAL, showing how data preprocessing, profiling, and scheduling come together to achieve these goals. We next provide a module-by-module description of the system.

Overview. REAL follows a two-plane architecture: the control plane, running on the CPU, includes the Preprocessor, Profiler, and Scheduler modules for graph analysis, profiling, and schedule generation; the data plane, running on multiple GPUs, is responsible for executing the actual DGNN training.

Preprocessor. The preprocessor performs structural analysis of DTDGs to guide subsequent scheduling and execution. As an offline profiling step, the preprocessor fits degree–runtime cost curves on synthetic graphs only once during system initialization; the curves are reused across different datasets as the basis for operator selection (*e.g.,*

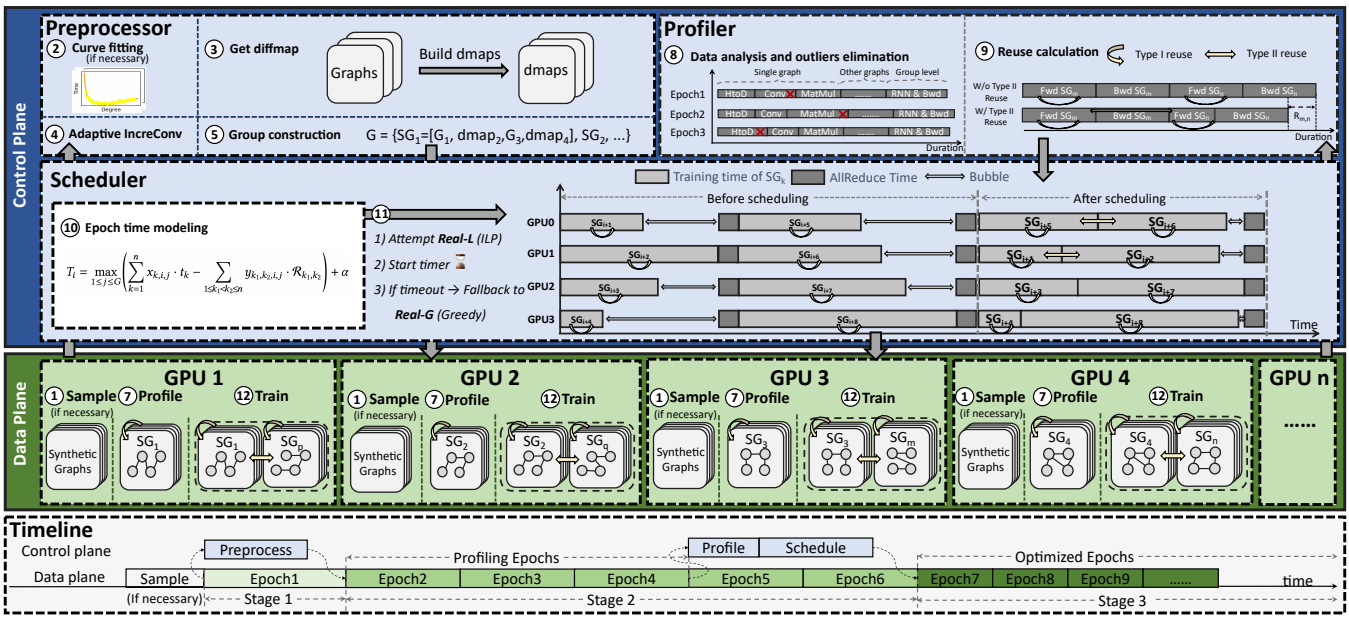


Figure 5. *Real architecture (top) and execution timeline (bottom).*

SpMM vs. GraphConv). Then, for each snapshot $\mathcal{G}_i$ ($i > 1$), it computes a sparse difference map (dmap) by diffing against its predecessor $\mathcal{G}_{i-1}$, which captures localized structural changes. Finally, depending on the estimated cost from dmap and the fitted curves, the preprocessor selects for each snapshot an appropriate form, either the full graph or its dmap. For example, a four-size group can be constructed as $\mathcal{SG}_i = [\,\mathcal{G}_i, \text{dmap}_{i+1}, \mathcal{G}_{i+2}, \text{dmap}_{i+3}\,]$. The details are described in §4.1.

Profiler. The profiler quantifies training costs and reuse opportunities to support scheduler simulation. It first collects fine-grained training time details, including graph-level metrics such as HtoD, Conv, and MatMul times, as well as group-level RNN and backward times. To ensure reliable statistics, the profiler analyzes multiple training epochs and eliminates outliers caused by runtime noise. Based on these profiled values, it further estimates the potential reuse time $\mathcal{R}_{i,j}$ when two groups $\mathcal{SG}_i$ and $\mathcal{SG}_j$ are scheduled to the same GPU in the same iteration. This reuse time captures the overlapping data transfer and computation in HtoD and GNN phases. The detailed $\mathcal{R}_{i,j}$ modeling is shown in ⑨ of Figure 5. Both the profiled timeline and $\mathcal{R}_{i,j}$ table are later used by the scheduler to simulate the total epoch time.

Scheduler. The scheduler estimates the training time of an epoch by leveraging profiling statistics, incorporating both the standalone execution cost of each group and the reuse $\mathcal{R}_{i,j}$ when groups with overlapping snapshots are co-located on the same GPU. Based on this performance model, Real supports two scheduling backends: an ILP-based solver (Real-L) and a greedy heuristic (Real-G), described in detail in §5. To balance scheduling quality and runtime efficiency, Real adopts a dynamic selection strategy. At the beginning of training, Real first attempts to solve the scheduling problem using Real-L, and a timer is started upon invocation. If the solver fails to return a feasible schedule within a predefined threshold (*e.g.,* 10 s), the scheduler falls back to Real-G.

3.3 Overlapping Control Plane Overhead

Figure 5 (bottom) shows the timeline, illustrating how Real overlaps control plane operations with data plane training. To achieve this, we decompose the training process into three stages, corresponding to the three colors of epochs. During the first stage, the data plane initiates training without any optimization, while the control plane concurrently performs preprocessing on the input graph. Once preprocessing completes, Real enters the second stage, where type I reuse (§4.1) is enabled based on the constructed difference maps, allowing the data plane to benefit from reduced computation without waiting for full scheduling decisions. Concurrently, the control plane collects runtime statistics across several profiling epochs. These statistics drive the profiling and scheduling modules to compute an optimized execution plan for the remaining training. In the third stage, the data plane adopts the optimized schedule, which incorporates type II reuse (§4.2) and cross-group scheduling, to further accelerate training. This pipelined design prevents CPU coordination from blocking GPU execution, sustaining high utilization.

4 TOPOLOGY-AWARE DATA REUSE

In this section, we present our design to exploit reuse opportunities in DGNN training, including both snapshot-level and group-level data reuse.

4.1 Type I: Snapshot-Level Data Reuse

In DTDGs, consecutive snapshots are usually highly similar: most nodes and edges remain unchanged, with only a small fraction updated. However, existing distributed DGNN

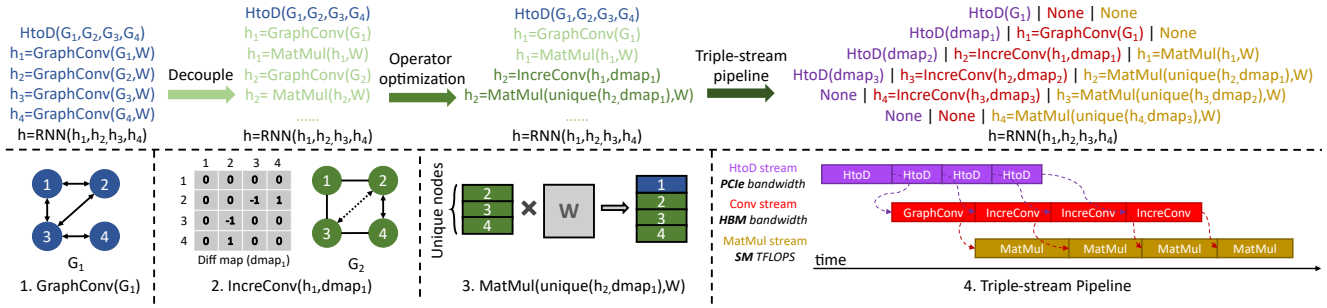


Figure 6. *Snapshot-level data reuse. Using GCN with aggregation preceding transformation as an example.*

systems treat each snapshot independently and rerun full `GraphConv`, causing substantial redundancy. This redundancy manifests in two main ways. First, in the aggregation phase of `GraphConv`, if a node's neighborhood remains unchanged across consecutive snapshots within the same group, its aggregated feature also stays identical, making repeated neighborhood aggregation redundant. Second, in the linear transformation phase, these identical aggregated features are repeatedly multiplied by the same weight matrix, consuming additional memory bandwidth and compute cycles. To address this, we propose a snapshot-level reuse strategy (Figure 6) that decouples aggregation and transformation in `GraphConv` for fine-grained reuse in both phases. We apply this mechanism at the first GNN layer, where identical snapshot features across time enable maximal reuse.

4.1.1 Aggregation Phase Reuse.

In the aggregation phase of `GraphConv`, each node collects its neighbors' features to form a structural summary of its local neighborhood. To avoid redundant aggregation, we use a difference map (`dmap`), where each entry `dmap[i][j]` marks structural changes in the local neighborhood. We define `dmap[i][j]` = 1 if the edge (i, j) is newly added, `dmap[i][j]` = -1 if it is removed, and `dmap[i][j]` = 0 otherwise. We then define an incremental aggregation operator, `IncreConv`, to capture the differential contribution of these local changes. During `IncreConv`, each nonzero entry in the `dmap` contributes by multiplying the neighbor feature with the corresponding edge weight. Finally, the `IncreConv` output is incorporated with the prior snapshot's aggregation to obtain the result.

Degree-Aware `IncreConv` for GCN. In GCN, the normalized edge weight is defined as $w_{ij} = \frac{1}{\sqrt{d_i d_j}}$, where d_i and d_j are the degrees of nodes i and j in the *full snapshot*. However, the `dmap` only records the changed edges of each node and does not preserve the complete neighborhood. As a result, the degree derived from the `dmap` is incomplete and cannot be directly used to compute normalized edge weights.

To address this, we augment each node in the `dmap` with an additional integer attribute that records its true degree in the full snapshot. This enables us to correctly compute the normalized weight for every newly added edge. Meanwhile, we maintain a lookup table that maps each existing edge to its previously used weight. When an edge is deleted in the next `dmap`, we directly retrieve its weight from the table to subtract its contribution from the aggregation. By maintaining per-node global degrees and per-edge cached weights, `IncreConv` ensures that incremental updates strictly follow the normalization rule of standard GCN.

Softmax-Aware `IncreConv` for GAT. In contrast to GCN, where normalized edge weights only depend on node degrees and can be incrementally updated from the `dmap` with degree tracking, GAT poses an additional challenge: due to the softmax operation, each attention coefficient α_{ij} depends on the *entire* neighborhood of node j. Thus, even a single edge update alters the denominator shared by all neighbors, making it impossible to derive updated attention weights solely from the `dmap`. To incrementally update GAT aggregation, we maintain for each node j the softmax denominator $\mathcal{D}_j^{(t-1)}$ and the aggregation result $h_j^{(t-1)}$ in snapshot $t-1$:

$$\mathcal{D}_j^{(t-1)} = \sum_{i \in \mathcal{N}(j,t-1)} \exp(e_{ij}^{(t-1)}), \quad h_j^{(t-1)} = \sum_{i \in \mathcal{N}(j,t-1)} \frac{\exp(e_{ij}^{(t-1)})}{\mathcal{D}_j^{(t-1)}} \mathcal{W} h_i.$$

where $e_{ij}^{(t-1)}$ is the unnormalized attention score on edge (i, j), $\mathcal{W}$ is the linear transformation weight. When snapshot t introduces an update set $\Delta\mathcal{E}_j$ of incident edges (with sign +1 for addition and −1 for deletion in `dmap`), we compute the incremental change of the denominator and update it as

$$\Delta\mathcal{D}_j = \sum_{i \in \Delta\mathcal{E}_j} \pm \exp(e_{ij}^{(t)}), \quad \mathcal{D}_j^{(t)} = \mathcal{D}_j^{(t-1)} + \Delta\mathcal{D}_j.$$

The previous aggregation is rescaled and then updated with the contributions of changed edges:

$$\tilde{h}_j = h_j^{(t-1)} \cdot \frac{\mathcal{D}_j^{(t-1)}}{\mathcal{D}_j^{(t)}}, \quad h_j^{(t)} = \tilde{h}_j + \sum_{i \in \Delta\mathcal{E}_j} \frac{\exp(e_{ij}^{(t)})}{\mathcal{D}_j^{(t)}} \mathcal{W} h_i.$$

This design allows unchanged neighbors to be updated in $O(1)$ by denominator rescaling, while only the affected edges incur $O(|\Delta\mathcal{E}_j|)$ cost. Overall, the per-snapshot complexity is reduced from $O(|\mathcal{E}|)$ to $O(|\Delta\mathcal{E}| + |\mathcal{V}_\Delta|)$, where $\Delta\mathcal{E}$ is the number of changed edges and $\mathcal{V}_\Delta$ the touched nodes.

Adaptive `IncreConv` Execution. Because `dmap` is stored in a sparse format, the most natural choice for its aggregation kernel is DGL's native `COO+SpMM` pipeline. However, profiling shows that when the original graph $\mathcal{G}$ becomes denser,

dmap is no longer sparse and the COO layout becomes memory–bandwidth–bound; in some cases, performance is worse than materializing dmap as a graph and running GraphConv, and even worse than running GraphConv directly on $\mathcal{G}$. To reconcile these contrasting behaviors, we design a degree-aware, self-adaptive strategy for IncreConv. The runtime of GraphConv follows a characteristic U-shape with respect to the average node degree, whereas COO+SpMM excels only in the very low-degree regime; this comparison is presented in the supplementary material §A. We fit the two curves and use them to predict the runtime of different kernels at execution time. At runtime, IncreConv first measures the average degrees of both the original graph $\mathcal{G}$ and its dmap. Using the fitted GraphConv and COO+SpMM performance curves, it predicts three runtimes: (i) GraphConv on $\mathcal{G}$, (ii) GraphConv on dmap, and (iii) COO+SpMM on dmap. The kernel with the lowest predicted cost is then selected for execution, ensuring optimal performance across varying sparsity patterns.

4.1.2 Transformation Phase Reuse. The position of the transformation phase varies across different DGNN designs: it may be applied before or after GraphConv. When the transformation precedes GraphConv, as in GAT, the input node features remain identical across all snapshots in a group. Consequently, the MatMul results are also identical, and all subsequent snapshots can directly reuse the result of the first snapshot. When the transformation follows GraphConv, as in GCN, only nodes with changed neighborhoods require new MatMul operations, while the others can reuse results from the previous snapshot. Both cases significantly reduce redundant MatMul computations.

4.1.3 Triple-Stream Pipeline. To further improve end-to-end efficiency, we extend our execution to three parallel CUDA streams: (i) an HtoD stream for host-to-device data transfer, (ii) a Conv stream for Conv operations, and (iii) a MatMul stream for feature transformation. These three stages stress different hardware resources: PCIe bandwidth, HBM bandwidth, and GPU compute, respectively, and thus can be effectively overlapped. The streams are coordinated via cudaEvent to preserve inter-operator dependencies while maximizing concurrency. Compared to conventional sequential execution, our triple-stream pipeline enables fine-grained inter-snapshot interleaving. As illustrated in Figure 6, HtoD, Conv, and MatMul operations from different snapshots can overlap, effectively hiding latency and improving pipeline throughput. When transformation precedes GraphConv (*e.g.,* GAT), the dependency becomes HtoD → MatMul → Conv. Here, MatMul is computed once and reused across snapshots, so the pipeline effectively overlaps HtoD and Conv.

4.2 Type II: Group-Level Data Reuse

Group-level data reuse arises when consecutive snapshot groups share overlapping snapshots and are scheduled within the same iteration, ensuring that model parameters remain unchanged. In this case, both the graph structure and the corresponding intermediate representations can be directly reused across group boundaries, avoiding redundant HtoD transfers and GraphConv computations. Building on this reuse opportunity, the Real scheduler employs a similarity-driven scheduling strategy to enhance cross-group sharing. Rather than forming disjoint pairs of snapshots (*e.g.,* $[\mathcal{G}_1, \mathcal{G}_2]$ and $[\mathcal{G}_3, \mathcal{G}_4]$), the scheduler reorganizes groups according to structural similarity (*e.g.,* prioritizing $[\mathcal{G}_1, \mathcal{G}_2]$ and $[\mathcal{G}_2, \mathcal{G}_3]$ to improve reuse opportunities). This similarity-aware rescheduling forms overlapping subgraph pairs that better exploit group-level reuse potential while maintaining workload balance. The concrete *cross-group scheduling* algorithm will be detailed in § 5.

5 CROSS-GROUP SCHEDULING

In this section, we present our cross-group scheduling algorithm, which jointly considers inter-group data reuse and load balance to improve performance.

5.1 Optimization Objectives

The primary objective of Real is to minimize the per-epoch training time T, which is critical in distributed environments where imbalanced group assignment and inefficient scheduling can significantly extend computation time. We formally define $T = \sum_{i=0}^{m} T_i$, where m is the number of iterations in one epoch and T_i denotes the duration of iteration i.

To incorporate benefits from *reuse*, we introduce binary decision variables $x_{k,i,j}$ and $y_{k_1,k_2,i,j}$. Specifically, $x_{k,i,j} = 1$ if snapshot group k is assigned to iteration i on GPU j, and 0 otherwise. The variable $y_{k_1,k_2,i,j} = 1$ if snapshot groups k_1 and k_2 are both assigned to iteration i on GPU j. Let t_k be the execution time of group k, $\mathcal{R}_{k_1,k_2}$ the time savings from Type II reuse, and α the overhead of synchronization (*e.g.,* gradient allreduce). The iteration time is then formulated as:

$$T_i = \max_{1 \le j \le G} \left(\sum_{k=1}^{n} x_{k,i,j} \cdot t_k - \sum_{1 \le k_1 < k_2 \le n} y_{k_1,k_2,i,j} \cdot \mathcal{R}_{k_1,k_2} \right) + \alpha, \quad (3)$$

where G is the number of GPUs.

Since different snapshot groups can be executed independently, determining the optimal scheduling strategy $Strategy = \{(a_1, g_1), (a_2, g_2), \ldots, (a_n, g_n)\}$ to minimize T is NP-hard, as it generalizes the classical makespan minimization problem on parallel machines [10]. In particular, for a single iteration ($m = 1$) without gradient all-reduce ($\alpha = 0$), this reduces to assigning tasks to GPUs to minimize the maximum completion time, which is already NP-hard for two or more machines [14]. To ensure system convergence while improving tractability, we constrain each GPU to handle at most two groups per iteration. Building on this formulation, we propose Real-L, an ILP-based approach, and Real-G, a greedy heuristic, to efficiently address the problem.

5.2 Real-L

To globally optimize the per-epoch training time T, we cast the formulation in Equation 3 as an ILP problem, with the following constraints and linearization steps.

Assignment Constraint. Each snapshot group must be assigned to exactly one iteration and one GPU:

$$\sum_{i=1}^{m}\sum_{j=1}^{G} x_{k,i,j} \;=\; 1, \quad \forall k \in \{1,2,\ldots,n\}. \tag{4}$$

GPU Capacity Constraint. No GPU can process more than L snapshot groups in any iteration (*e.g.*, $L = 2$):

$$\textstyle\sum_{k=1}^{n} x_{k,i,j} \;\le\; L, \quad \forall i \in \{1,2,\ldots,m\},\ \forall j \in \{1,2,\ldots,G\}. \tag{5}$$

Reuse Variables. For all $1 \le k_1 < k_2 \le n$, $1 \le i \le m$, and $1 \le j \le G$, we enforce $y_{k_1,k_2,i,j} = 1$ if and only if group $\{k_1, k_2\}$ are both assigned to GPU j in iteration i through:

$$x_{k_1,i,j} + x_{k_2,i,j} - 1 \;\le\; y_{k_1,k_2,i,j} \;\le\; \min\left(x_{k_1,i,j},\, x_{k_2,i,j}\right) \tag{6}$$

Linearization of the max **Function.** We first define the load of GPU j in iteration i as

$$z_{i,j} = \sum_{k=1}^{n} x_{k,i,j}\, t_k - \sum_{1\le k_1<k_2\le n} y_{k_1,k_2,i,j}\, \mathcal{R}_{k_1,k_2}. \tag{7}$$

The iteration time is then $T_i = \max_j z_{i,j} + \alpha$, where the max operator is nonlinear and must be linearized for ILP solving. We introduce binary variables $u_{i,j}$ and a sufficiently large constant $\mathcal{M}$, and impose:

$$z_{i,j} \;\le\; T_i \;\le\; z_{i,j} + \mathcal{M}(1 - u_{i,j}), \quad \forall j, \tag{8}$$

where $u_{i,j} \in \{0,1\}$. Constraint (8) bounds T_i between all $z_{i,j}$ values, while $\sum_{j=1}^{G} u_{i,j} = 1$ $(\forall i)$ enforces that exactly one GPU is selected. Together, they ensure T_i equals the maximum load among all GPUs, thus linearizing the max operator.

We solve the ILP (Equation 3) using standard linearization and the Gurobi solver [13]. As the problem is NP-hard, we terminate early once the solver finds a solution within a preset optimality gap (*e.g.*, 5%), balancing quality and runtime.

5.3 Real-G

For large-scale optimization problems, we use Real-G, a greedy heuristic algorithm, as the ILP fallback on timeout. Algorithm 1 provides the details of Real-G. The algorithm first derives candidate scheduling targets, then incrementally selects compatible groups guided by reuse and load balance considerations, and finally evaluates alternative schedules to minimize wasted time of each iteration.

Input. The algorithm takes as input the GPU count G, the matrix $\mathcal{R}$ capturing type II reuse savings, the remaining group count *Res* (initialized as n), and the group list $groups = [t_1, t_2, \ldots, t_n]$ with t_i being the execution time of group i.

Group preprocessing. Line 1 initializes preprocessing by calling `Preprocess` (lines 14–16), which inserts a *zero group* ($t_0 = 0$) as a dummy option. This ensures that any group can either pair with another group or with the zero group (*i.e.*, run alone). All groups are then sorted by execution time t_i in ascending order to facilitate efficient matching.

Algorithm 1: Real-G

```
   Input  : G, R, Res, groups = [t_1, t_2, ..., t_n]
   Output : Strategy
1  Initialize groups ← Preprocess(groups);
2  while Res > G do
3      targets ← GetCandidateTargets(groups, R, n);
4      W_min ← ∞;
5      for i = 0 to |targets| − 1 do
6          T_target ← targets[i];
7          pairs ← GetPairs(T_target, G, groups, n);
8          W ← max(max_j pairs[j].T, T_target) · G;
9          W ← W − Σ_{j=1}^{G−1} pairs[j].T − T_target;
10         W < W_min ? W_min ← W;
11     Update Res, groups, Strategy;
12 Record rest group in Strategy;
13 return Strategy;
14 Function Preprocess(groups):
15     Add t_0 = 0 to groups and sort ascending;
16     return groups;
17 Function GetCandidateTargets(t, R, n):
18     Initialize targets ← ∅;
19     for i = 0 to n − 1 do
20         Add {t[i] + t[n] − R[i][n]} to targets;
21     return targets;
22 Function GetPairs(T_target, R, G, t, n):
23     Initialize pairs ← ∅;
24     while |pairs| < G − 1 do
25         Initialize head, tail, best_pair;
26         while head < tail do
27             T ← t[head]+t[tail]−R[head][tail];
28             if |T−T_target| < |best_pair.T−T_target| then
29                 best_pair ← [T, head, tail];
30             T < T_target ? head++ : tail−−
31         Add best_pair to pairs;
32     return pairs;
```

Candidate target generation. Line 3 generates candidate target times T_{target} using `GetTargetList` (lines 17–21). This procedure pairs the group with the longest remaining time with arbitrary combinations of the other groups, producing potential scheduling targets.

Group selection and waste time calculation. Lines 5–10 evaluate each candidate T_{target}. For each case, `GetPairs` (lines 22–32) performs a two-point search to efficiently identify the group combination that best matches the target. The wasted time W_i, defined in Equation 9, quantifies the quality

Dataset	$\overline{\lvert\mathcal{V}\rvert}$	$\overline{\lvert\mathcal{E}\rvert}$	$\overline{\lvert\mathcal{V}_d\rvert}$	$\overline{\lvert\mathcal{E}_d\rvert}$	d_v	β	γ
Arxiv [15]	169.3k	1.8M	88.7k	242.6k	128	10.6	100
Products [15]	2.4M	36.7M	1.5M	5.9M	100	15.0	100
Reddit [25]	232.9k	31.9M	211.6k	5.5M	602	137.3	100
Stackoverflow [29]	2.6M	23.1M	790.3k	3.1M	128	8.9	100
Stackoverflow-a2q [29]	2.5M	10.8M	554.3k	1.5M	128	4.4	100
Wiki-talk [21]	1.1M	6.3M	182.8k	439.4k	128	5.5	100

Table 2. Attributes of the six datasets. *$\overline{\lvert\mathcal{V}\rvert}$ and $\overline{\lvert\mathcal{E}\rvert}$ are the average nodes and edges per snapshot; $\overline{\lvert\mathcal{V}_d\rvert}$ and $\overline{\lvert\mathcal{E}_d\rvert}$ are the average nodes and edges per* `dmap`*; d_v is the node feature dimension; β and γ are the average degree and snapshot count.*

of a schedule (lines 8-10):

$$W_i = G \cdot (T_i - \alpha) - \sum_{j=1}^{G} \left(\sum_{k=1}^{n} x_{k,i,j} t_k - \sum_{1 \le k_1 < k_2 \le n} y_{k_1,k_2,i,j} \mathcal{R}_{k_1,k_2} \right). \quad (9)$$

The algorithm then selects the T_{target} with the minimum W_i, ensuring efficient group scheduling.

Complexity analysis. The overall time complexity of Algorithm 1 is dominated by the main loop while running at $O(n^3)$, making it computationally feasible for large-scale scenarios. In practice, when n reaches the $O(10^3)$ scale, the solving time still remains within a few seconds.

6 EVALUATION

In this section we first present our experimental setup, including the testbed, models, datasets, and baselines (§6.1). We evaluate Real for training throughput, data transfer time, `GraphConv` FLOPs, and load imbalance (§6.2), conduct ablation studies (§6.3), and confirm its accuracy (§6.4). We also assess scalability via simulations (§6.5).

6.1 Methodology

Testbed. We conduct our experiments using both single-machine and multi-machine configurations.

A. Single-machine setup. A single machine has a 192 core Intel Xeon Platinum 8575C CPU @ 4.0 GHz, and eight NVIDIA H20 GPUs interconnected in a full NVLink topology, achieving an intra-node GPU-to-GPU bandwidth of 900 GB/s.

B. Multi-machine setup. Each machine in the cluster has the same hardware configuration as the single-machine setup. Additionally, each machine is equipped with four ConnectX-7 NICs, and two machines are interconnected via a 1.6 Tbps RoCE-enabled Ethernet fabric.

All experiments are conducted within the DGL NGC Container (version 24.07-py3) [20], which provides DGL v2.4 [35] for scalable graph processing, PyTorch v2.4.0 [23] as the deep learning backend, and CUDA 12.5 for GPU acceleration.

Datasets. We evaluate on six DTDG datasets (Table 2), including both real-world temporal networks and synthetic dynamic graphs derived from static datasets. The real-world datasets are Stackoverflow and Stackoverflow-a2q [29], which record user interactions on the Stack Exchange website, and Wiki-talk [21], which captures edits among Wikipedia users. We also build dynamic graphs from three static graph datasets: Arxiv [15], Products [15], and Reddit [25]. Following BLAD [7], we create snapshots by randomly deleting some of the edges of the static graph. The evolution pattern of the number of edges in these snapshots mirrors the trend observed in the Stackoverflow dataset. For all datasets, the snapshot group size is set to four.

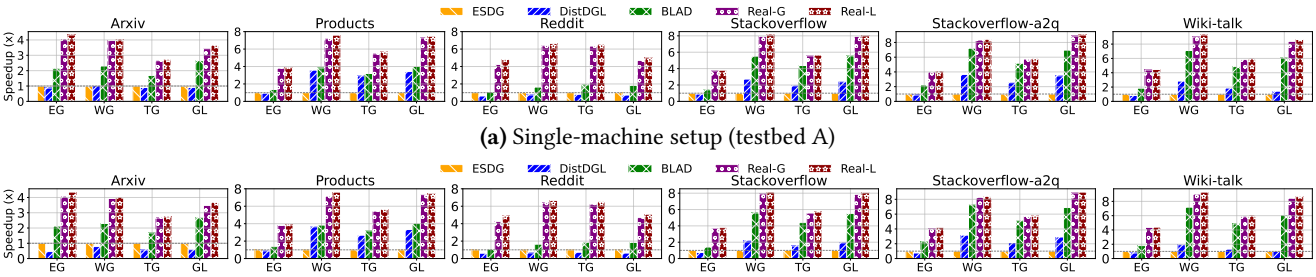
Benchmark DGNN models. We use four representative DGNNs: EvolveGCN [22], WD-GCN [18], TGCN [4], and GAT-LSTM [38]. The first three models are GCN-based DGNNs, while GAT-LSTM is a GAT-based model. These models are widely used due to their effectiveness in dynamic graph learning. Each DGNN model features a two-layer architecture, comprising a graph aggregation operation (GNN) and a temporal update operation (RNN), with the hidden dimension set to 64 for all models.

Baselines. We compare Real-G and Real-L with existing state-of-the-art distributed DGNN training methods, including ESDG [3], DistDGL [42] and BLAD [7]. In ESDG, snapshots in a snapshot group are evenly distributed across GPUs based on their temporal intervals. DistDGL partitions each snapshot into subgraphs and distributes them across GPUs. BLAD utilizes a two-stage pipeline to collaboratively train two consecutive snapshot groups. In contrast, Real-G and Real-L execute two scheduled groups sequentially. DGC [5] is excluded due to its *lack of open-source code*, and DynaHB [28] is an *asynchronous* training framework, thus not included in the comparison.

6.2 Experimental results

Overall Performance. We first compare the training time speedups across all methods. As shown in Figure 7a, on the testbed A, Real-L delivers substantial gains over all baselines, achieving a 2.71×–9.28× speedup over ESDG (avg. 5.98×), 1.13×–4.53× over BLAD (avg. 2.03×), and 1.94×–8.92× over DistDGL (avg. 4.23×). Real-G also delivers consistent improvements, with a 2.65×–8.99× speedup over ESDG (avg. 5.80×), 1.11×–3.99× over BLAD (avg. 1.95×), and 1.86×–8.59× over DistDGL (avg. 4.07×). The gains of Real are particularly pronounced in WD-GCN, TGCN, and GAT-LSTM, where all reuse mechanisms can be exploited. In contrast, for EvolveGCN, the weights in the `MatMul` stage change with each timestamp, preventing reuse in this phase. ESDG suffers from hidden state transfers between GPUs. The cost is minor for EvolveGCN with small hidden states, but becomes a major bottleneck for models with larger hidden states. DistDGL is further hindered by frequent neighbor aggregation, which limits its ability to scale throughput despite balanced workloads. BLAD, as the current SOTA, outperforms both ESDG and DistDGL mainly by reducing inter-GPU communication; however, its lack of data reuse and load balancing considerations still caps its maximum performance gains.

We further report results on testbed B in Figure 7b, where the speedups of Real remain comparable to testbed A for



(a) Single-machine setup (testbed A)

(b) Multi-machine setup (testbed B)

Figure 7. *Training speedup of different systems across DGNN models and datasets. Speedup is normalized to ESDG [3].* ***EG****,* ***WG****,* ***TG****, and* ***GL*** *denote EvolveGCN, WD-GCN, TGCN, and GAT-LSTM, respectively.*

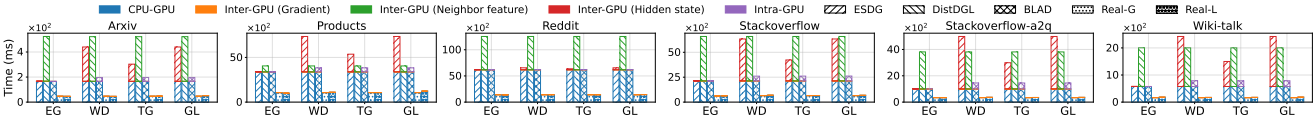


Figure 8. *Breakdown of data transfer time per epoch on the testbed A. Each bar is decomposed into CPU-GPU transfer, inter-GPU transfer (gradients, neighbors, hidden states), and intra-GPU transfer.*

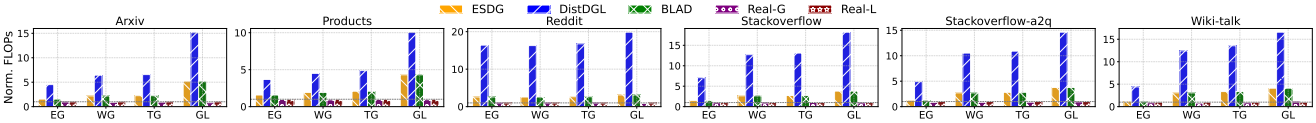


Figure 9. *Normalized* `GraphConv` *FLOPs per epoch on the testbed A. FLOPs is normalized to* Real-L.

all baselines except DistDGL. DistDGL is more adversely affected in the multi-machine setting, as its vertex-level partitioning requires cross-machine neighbor aggregation, incurring heavy communication overhead. Note that ESDG does not incur cross-machine hidden state transfers when the group size is four, since hidden states are confined within each 4-GPU group, resulting in inter-group data parallelism. Since BLAD's open-source version lacks multi-machine support, we use an optimistic estimate by extrapolating its performance on testbed A with a 2× scaling factor.

Transfer time breakdown. Figure 8 shows the breakdown of per-epoch data transfer time. In ESDG, inter-GPU hidden state transfer is the dominant cost for most models except EvolveGCN, and is particularly expensive when processing large single-graph datasets such as Stackoverflow. DistDGL suffers from neighbor aggregation overhead, especially on dense graphs or with high feature dimensions, where many neighbors are on different GPUs. In the Reddit dataset, with an average degree of 137 and feature dimension of 602, this results in particularly high transfer cost. BLAD avoids both types of cross-GPU data transfer but still loads the full graph, making `HtoD` transfer a bottleneck. Its pipeline further involves hidden state transfer across processes on the same GPU, which becomes significant on datasets such as Stackoverflow and Wiki. Real also avoids both types of cross-GPU data transfer and, within each GPU, leverages DiffMap-based reuse to greatly reduce `HtoD` volume.

`GraphConv` FLOPs. We profile the total `GraphConv` computation within one training epoch. As shown in Figure 9, compared to DistDGL, Real-G and Real-L reduce redundant FLOPs by 3.81×–23.23× and 3.68×–19.85×, respectively. Against both ESDG and BLAD (which always perform full `GraphConv` computation and thus incur identical FLOPs), the reductions are 1.26×–6.43× for Real-G, and 1.09×–5.22× for Real-L. These savings stem from skipping full `GraphConv` on structurally unchanged nodes. DistDGL's vertex-level partitioning still applies complete `GraphConv` to imported cross-GPU vertices whose outputs are never used in subsequent computation, incurring substantial redundant FLOPs. ESDG and BLAD overlook structural redundancy and perform full-graph convolution for every snapshot. In contrast, Real leverages type I and type II reuse to bypass both aggregation and transformation for unchanged nodes.

GPU Utilization and Load Balance. We profile the utilization of all GPUs during the first 30 seconds of training. Due to space constraints, we only present the GPU utilization heatmaps of Real-L in Figure 10, while the results for other baselines are provided in the supplementary material §B. The heatmaps show that Real-L maintains a high overall GPU utilization, mainly because the *Triple-Stream Pipeline* in Type I reuse overlaps `HtoD` transfers with `Conv` and `MatMul` operations, removing idle GPU time. Furthermore, Real-L also achieves balanced workload distribution across GPUs, primarily attributed to its *cross-group* scheduling.

6.3 Ablation Study

To evaluate the individual contribution of each component in Real, we perform a stepwise ablation study normalized to

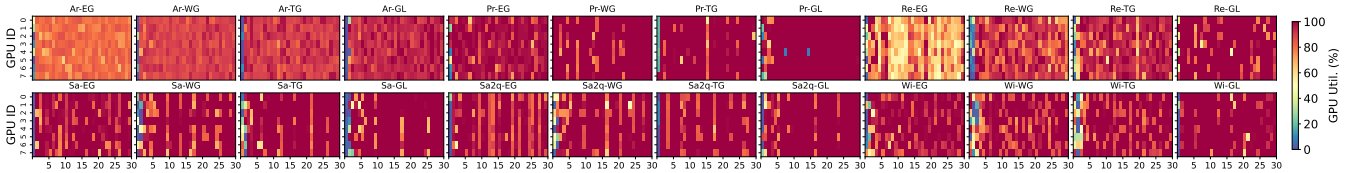


Figure 10. *GPU utilization heatmaps of* Real-L *across different DGNN models and datasets.* ***Ar***, ***Pr***, ***Re***, ***Sa***, ***Sa2q***, *and* ***Wi*** *denote the datasets Arxiv, Products, Reddit, Stackoverflow, Stackoverflow-a2q, and Wiki-talk, respectively.*

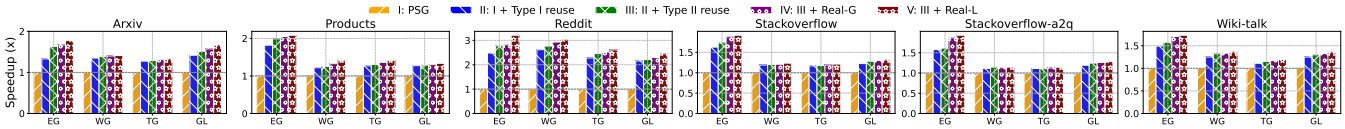


Figure 11. *Speedup ablation across different models and datasets. Speedup is normalized to PSG.*

the *Partition-by-Snapshot-Group* (PSG) baseline, where each GPU sequentially trains one complete snapshot group per iteration without reuse or load balancing. Figure 11 reports cumulative speedups across models and datasets.

The PSG baseline achieves 1.00× by definition, serving as the starting point. Adding **type I reuse**, via `IncreConv`, selective `MatMul`, and the triple-stream pipeline, improves performance to 1.11×–2.62× by eliminating redundant computations and data transfers between snapshots. Incorporating **type II reuse**, which by default pairs two consecutive snapshot groups to maximize reuse, further raises speedups to 1.11×–2.77× by removing redundant `HtoD` transfers and `GraphConv` across groups. Introducing the greedy cross-group scheduler (**Real-G**) balances workloads while preserving reuse benefits, achieving 1.12×–2.93×. Finally, replacing the greedy scheduler with the ILP solver (**Real-L**) delivers the highest cumulative speedups of 1.14×–3.21×, benefiting from a global view for optimal scheduling.

6.4 Model Accuracy

Compared to ESDG, Real effectively increases the batch size per iteration by a factor of K. Under standard assumptions for stochastic non-convex optimization (Lipschitz gradients, bounded variance, i.i.d. samples, synchronized updates), enlarging the batch size in this way does not change the asymptotic convergence rate. In particular, with a learning rate $\eta = \Theta(1/\sqrt{T})$, Real achieves the same $O(1/\sqrt{T})$ convergence guarantee as vanilla SGD.

We defer both the formal analysis and the empirical accuracy/loss comparisons to the supplementary materials (§C, §D), which show that Real matches ESDG in convergence behavior and final model quality.

6.5 Scalability

Due to hardware limitations, we extend the evaluation to larger clusters through simulation. To evaluate performance at larger scales, we conduct simulations using 10,000 snapshots generated by DynaGraph[12]. In this scenario, solving with ILP-based methods is time-consuming, so we evaluate Real-G, which can be solved quickly even at large

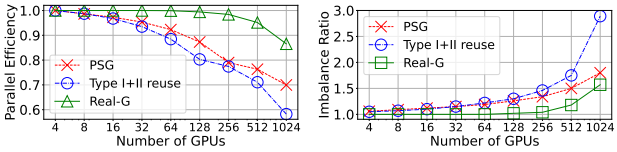


Figure 12. *Simulated parallel efficiency and imbalance ratio on clusters with 4 to 1024 GPUs.*

scales, against baselines that only incorporate data reuse without load balancing and PSG. The scheduling strategies from different methods are applied to Equations 3 for time simulation; detailed simulation settings are provided in the supplementary material §E. Figure 12 reports both *parallel efficiency*, defined as the ratio of per-GPU throughput at scale to the single-GPU throughput, and the *imbalance ratio*, defined as the training time of the most heavily loaded GPU divided by that of the least loaded one (excluding synchronization wait time). Figure 12 (left) shows that while baselines suffer from declining efficiency as the GPU count increases—particularly beyond 64 GPUs, Real-G sustains 95% efficiency at 512 GPUs and over 85% at 1024 GPUs. Figure 12 (right) further demonstrates that this scalability improvement correlates with lower imbalance ratios, where Real-G consistently outperforms baselines.

7 CONCLUSION

In this paper, we introduce Real, an efficient distributed training system for DGNNs. Real directly targets the three fundamental challenges of distributed DGNN training: maximize resource utilization, minimize data transfer, and maximize data reuse. Real integrates snapshot-level and group-level reuse with cross-group scheduling, and provides two variants: Real-L, which employs ILP to achieve globally optimized schedules, and Real-G, a greedy fallback used when ILP solving times out. Extensive evaluation on six DTDG datasets and four DGNN models demonstrates that Real-L and Real-G deliver 1.11×–6.13× and 1.10×–5.82× speedup over SOTA baseline, respectively.

References

[1] [n. d.]. PyG 2.0: Scalable Learning on Real World Graphs.

[2] Guangji Bai, Chen Ling, and Liang Zhao. 2022. Temporal domain generalization with drift-aware dynamic neural networks. *arXiv preprint arXiv:2205.10664* (2022).

[3] Venkatesan T. Chakaravarthy, Shivmaran S. Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient scaling of dynamic graph neural networks *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 77, 15 pages. https://doi.org/10.1145/3458817.3480858

[4] Bo Chen, Wei Guo, Ruiming Tang, Xin Xin, Yue Ding, Xiuqiang He, and Dong Wang. 2020. TGCN: Tag graph convolutional network for tag-aware recommendation. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 155–164.

[5] Fahao Chen, Peng Li, and Celimuge Wu. 2023. DGC: Training Dynamic Graphs with Spatio-Temporal Non-Uniformity using Graph Partitioning by Chunks. *Proc. ACM Manag. Data* 1, 4, Article 237 (dec 2023), 25 pages. https://doi.org/10.1145/3626724

[6] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[7] Kaihua Fu, Quan Chen, Yuzhuo Yang, Jiuchen Shi, Chao Li, and Minyi Guo. 2023. BLAD: Adaptive Load Balanced Scheduling and Operator Overlap Pipeline For Accelerating The Dynamic GNN Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 37, 13 pages. https://doi.org/10.1145/3581784.3607040

[8] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. ETC: Efficient Training of Temporal Graph Neural Networks over Large-scale Dynamic Graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1060–1072.

[9] Shihong Gao, Yiming Li, Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. SIMPLE: Efficient Temporal Graph Neural Network Training at Scale with Dynamic Data Placement. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–25.

[10] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[11] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. 2020. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge-Based Systems* 187 (2020), 104816.

[12] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–10.

[13] Gurobi Optimization, LLC. 2022. Gurobi - The Fastest Solver. https://www.gurobi.com Accessed: 2022-01-01.

[14] Ellis Horowitz and Sartaj Sahni. 1976. Exact and Approximate Algorithms for Scheduling Nonidentical Processors. *J. ACM* 23, 2 (April 1976), 317–327. https://doi.org/10.1145/321941.321951

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[16] Haoyang Li and Lei Chen. 2021. Cache-based gnn system for dynamic graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 937–946.

[17] Hongxi Li, Zuxuan Zhang, Dengzhe Liang, and Yuncheng Jiang. 2024. K-Truss Based Temporal Graph Convolutional Network for Dynamic Graphs. In *Asian Conference on Machine Learning*. PMLR, 739–754.

[18] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (2020), 107000.

[19] Andrew McCrabb, Hellina Nigatu, Absalat Getachew, and Valeria Bertacco. 2022. DyGraph: a dynamic graph generator and benchmark suite. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Philadelphia, Pennsylvania) *(GRADES-NDA '22)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3534540.3534692

[20] NVIDIA. 2024. Deep Graph Library (DGL) Container. https://catalog.ngc.nvidia.com/orgs/nvidia/containers/dgl. Accessed: 2025-08-20.

[21] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining* (Cambridge, United Kingdom) *(WSDM '17)*. Association for Computing Machinery, New York, NY, USA, 601–610. https://doi.org/10.1145/3018661.3018731

[22] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 5363–5370.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

[24] Xiao Qin, Nasrullah Sheikh, Chuan Lei, Berthold Reinwald, and Giacomo Domeniconi. 2023. Seign: A simple and efficient graph neural network for large dynamic graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2850–2863.

[25] Reddit. n.d.. Reddit Dataset. https://www.reddit.com/.

[26] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th international conference on web search and data mining*. 519–527.

[27] Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, Qi Guo, and Yunji Chen. 2021. Cambricon-G: A polyvalent energy-efficient accelerator for dynamic graph neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 1 (2021), 116–128.

[28] Zhen Song, Yu Gu, Qing Sun, Tianyi Li, Yanfeng Zhang, Yushuai Li, Christian S Jensen, and Ge Yu. 2024. DynaHB: A Communication-Avoiding Asynchronous Distributed Framework with Hybrid Batches for Dynamic GNN Training. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3388–3401.

[29] Stack-Overflow. 2023. Stack-Overflow Dataset. https://snap.stanford.edu/data/sx-stackoverflow.html. Accessed: [Insert the date you accessed the dataset].

[30] Junwei Su, Difan Zou, and Chuan Wu. 2024. PRES: Toward Scalable Memory-Based Dynamic Graph Neural Networks. *arXiv preprint arXiv:2402.04284* (2024).

[31] Yuxing Tian, Yiyan Qi, and Fan Guo. 2023. FreeDyG: Frequency Enhanced Continuous-Time Dynamic Graph Model for Link Prediction. In *The Twelfth International Conference on Learning Representations*.

[32] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning representations over dynamic graphs. In *International conference on learning representations*.

[33] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: pipelined and parallel dynamic GNN training on GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 405–418.

[34] Junshan Wang, Wenhao Zhu, Guojie Song, and Liang Wang. 2022. Streaming graph neural networks with generative replay. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and*

Data Mining. 1878–1888.

[35] 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*.

[36] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. 2021. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2628–2638. https://doi.org/10.1145/3448016.3457564

[37] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. In *International Conference on Learning Representations (ICLR)*.

[38] Tianlong Wu, Feng Chen, and Yun Wan. 2018. Graph attention LSTM network: A new model for traffic flow forecasting. In *2018 5th international conference on information science and control engineering (ICISCE)*. IEEE, 241–245.

[39] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962* (2020).

[40] Jiaxuan You, Tianyu Du, and Jure Leskovec. 2022. ROLAND: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*. 2358–2366.

[41] Zeyang Zhang, Xin Wang, Ziwei Zhang, Zhou Qin, Weigao Wen, Hui Xue, Haoyang Li, and Wenwu Zhu. 2024. Spectral invariant learning for dynamic graphs under distribution shifts. *Advances in Neural Information Processing Systems* 36 (2024).

[42] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2021. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. arXiv:2010.05337 [cs.LG] https://arxiv.org/abs/2010.05337

[43] Lekui Zhou, Yang Yang, Xiang Ren, Fei Wu, and Yueting Zhuang. 2018. Dynamic network embedding by modeling triadic closure process. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[44] Linhong Zhu, Dong Guo, Junming Yin, Greg Ver Steeg, and Aram Galstyan. 2016. Scalable temporal latent space inference for link prediction in dynamic social networks. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2765–2777.

[45] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. arXiv:1902.08730 [cs.DC] https://arxiv.org/abs/1902.08730

Supplementary Materials

A Kernel Selection for IncreConv

Figure 13 compares the runtime of IncreConv using GraphConv and COO+SpMM across graphs of different sizes (100k to 5M nodes) and varying average degrees. The x-axis shows the average node degree on a logarithmic scale, and the y-axis reports the execution time (ms) in log scale. GraphConv exhibits a characteristic U-shape: runtime decreases as degree increases from extremely sparse to moderately connected graphs, then rises again for dense graphs due to higher aggregation cost, which we model as

$$T_{graphconv}(N,d) \;=\; a_0 \cdot N \;+\; a_1 \cdot d \;+\; \frac{a_2}{d} \;+\; a_3.$$

COO+SpMM, in contrast, performs best in the very low-degree regime but quickly becomes memory–bandwidth–bound, showing a strictly increasing trend with degree in our measurements. Its runtime is captured by

$$T_{spmm}(N,d) \;=\; b_0 \cdot N \,\left(b_1 \;+\; b_2 \cdot d^{\,b_3}\right).$$

These fitted curves guide the runtime decision of IncreConv. For a given input, we evaluate: (i) GraphConv on the baseline graph, (ii) GraphConv on dmap, and (iii) COO+SpMM on dmap. Depending on the degrees of the baseline graph and dmap, any of the three can be the fastest option. SpMM on the baseline graph is excluded, since its runtime cost grows monotonically with both node count and degree, making it consistently slower than SpMM on the diffmap.

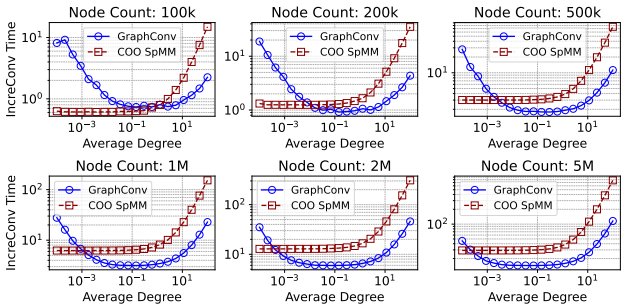


Figure 13. IncreConv *runtime on graphs of different sizes.*

B GPU Utilization of Other Methods

We also visualize the per-GPU utilization heatmaps for the remaining four baselines in Figures 14–17, measured during the first 30 seconds of training. For **ESDG** (Figure 14), each non-initial rank must wait for the RNN hidden state from other ranks, which frequently causes GPU idleness. This imbalance is mild for EvolveGCN due to its small hidden state size, but becomes more pronounced for other DGNN models. **DistDGL** (Figure 15) mitigates imbalance by partitioning graphs to equalize workload sizes across ranks, enabling fine-grained load control. However, its lack of overlap between computation and data transfer results in uniformly low utilization. **BLAD** (Figure 16) employs a two-stage pipeline per rank, but without inter-rank load balancing, it suffers from severe imbalance—especially on datasets with relatively small per-graph node counts (e.g., Arxiv, Reddit, Wiki). In contrast, **Real-G** (Figure 17) combines load balancing with triple-stream overlap on each rank, reducing inter-rank imbalance and improving overall GPU utilization.

C Convergence Analysis

C.1 Notation and Assumptions

We restate the relevant notation and assumptions for convenience:

- $\boldsymbol{x} \in \mathbb{R}^d$: model parameter vector.
- ζ: randomly sampled training data.
- Loss function $L(\boldsymbol{x}) = \mathbb{E}_\zeta[f(\boldsymbol{x};\zeta)]$, where $f(\boldsymbol{x};\zeta)$ is the loss on sample ζ.
- Model parameters at iteration t: $\boldsymbol{x}_t$.
- Learning rate: η.
- Number of GPUs: N.
- Number of batches processed per GPU per iteration: K (in our specific algorithm, $K = 2$).

Assumption C.1 (Independence)**.** For all i, the groups of samples $\{\zeta_t^{i,1}, \zeta_t^{i,2}, \ldots, \zeta_t^{i,K}\}$ processed by GPU i at iteration t are independent and also independent of other GPUs' samples.

Assumption C.2 (Bounded Gradients)**.** There exists a constant $G > 0$ such that for all $\boldsymbol{x}$ and ζ,

$$\|\nabla f(\boldsymbol{x};\zeta)\| \le G.$$

Assumption C.3 (Lipschitz Continuity of Gradients)**.** The gradient of $L(\boldsymbol{x})$ is L-Lipschitz continuous, i.e., for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$,

$$\|\nabla L(\boldsymbol{x}) - \nabla L(\boldsymbol{y})\| \le L\|\boldsymbol{x} - \boldsymbol{y}\|.$$

Assumption C.4 (Synchronization Consistency)**.** At each synchronization step, all GPUs have consistent model parameters, i.e., for all i, $\boldsymbol{x}_t^i = \boldsymbol{x}_t$.

C.2 Algorithm Description

Definition C.5 (Real)**.** At each iteration t, GPU i processes K mini-batches $\zeta_t^{i,1}, \zeta_t^{i,2}, \ldots, \zeta_t^{i,K}$ at the current model parameters $\boldsymbol{x}_t$ and computes:

$$\tilde{\nabla} f_i(\boldsymbol{x}_t) = \frac{1}{K}\sum_{k=1}^{K} \nabla f(\boldsymbol{x}_t;\zeta_t^{i,k}).$$

After each GPU i finishes, they synchronize to obtain:

$$\tilde{\nabla} f(\boldsymbol{x}_t) = \frac{1}{N}\sum_{i=1}^{N} \tilde{\nabla} f_i(\boldsymbol{x}_t).$$

Then the model is updated as:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta\tilde{\nabla} f(\boldsymbol{x}_t).$$

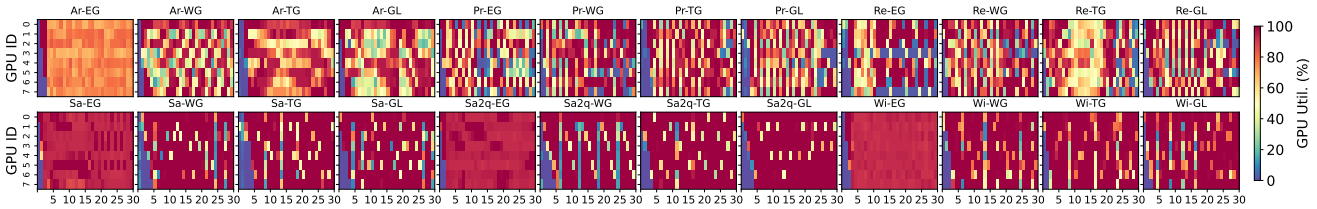


Figure 14. *GPU utilization heatmaps of ESDG across different dynamic GNN models and datasets.*

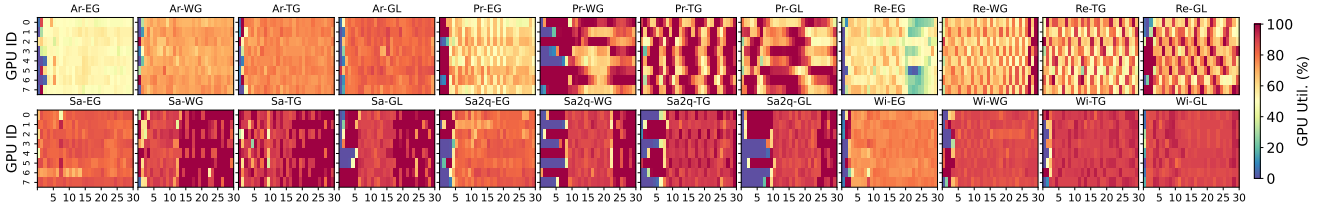


Figure 15. *GPU utilization heatmaps of DistDGL across different dynamic GNN models and datasets.*

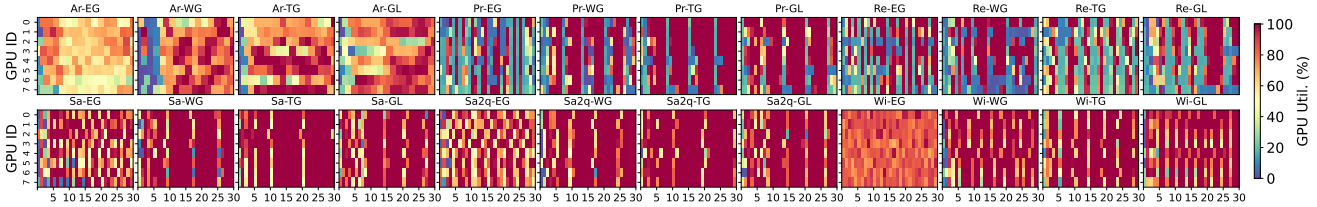


Figure 16. *GPU utilization heatmaps of BLAD across different dynamic GNN models and datasets.*

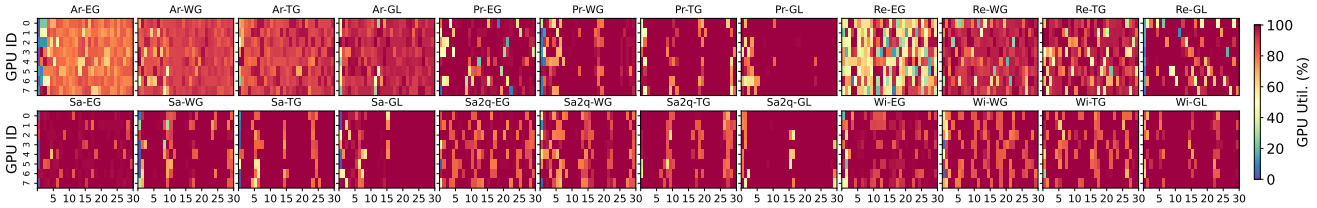


Figure 17. *GPU utilization heatmaps of Real-G across different dynamic GNN models and datasets.*

C.3 Detailed Steps

Lemma C.6 (Unbiasedness of the Gradient Estimator)**.** *Under the independence assumption, we have*

$$\mathbb{E}[\tilde{\nabla} f(\boldsymbol{x}_t)] = \nabla L(\boldsymbol{x}_t).$$

Proof. By definition:

$$\tilde{\nabla} f_i(\boldsymbol{x}_t) = \frac{1}{K}\sum_{k=1}^{K} \nabla f(\boldsymbol{x}_t; \zeta_t^{i,k}).$$

Since each $\zeta_t^{i,k}$ is drawn independently from the same distribution of ζ, we have:

$$\mathbb{E}[\nabla f(\boldsymbol{x}_t; \zeta_t^{i,k})] = \nabla L(\boldsymbol{x}_t).$$

Thus:

$$\mathbb{E}[\tilde{\nabla} f_i(\boldsymbol{x}_t)] = \frac{1}{K}\sum_{k=1}^{K} \mathbb{E}[\nabla f(\boldsymbol{x}_t; \zeta_t^{i,k})] = \nabla L(\boldsymbol{x}_t).$$

Averaging over N GPUs:

$$\mathbb{E}[\tilde{\nabla} f(\boldsymbol{x}_t)] = \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}[\tilde{\nabla} f_i(\boldsymbol{x}_t)] = \nabla L(\boldsymbol{x}_t),$$

as required. □

Lemma C.7 (Variance Bound)**.** *Let σ^2 be an upper bound on the variance of a single-sample stochastic gradient, i.e.,*

$$\mathbb{E}[\|\nabla f(\boldsymbol{x}_t; \zeta) - \nabla L(\boldsymbol{x}_t)\|^2] \le \sigma^2.$$

Then for the estimator $\tilde{\nabla} f(\boldsymbol{x}_t)$, we have

$$\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t) - \nabla L(\boldsymbol{x}_t)\|^2] \le \frac{\sigma^2}{NK}.$$

Proof. Each $\tilde{\nabla} f_i(\boldsymbol{x}_t)$ is the average of K independent stochastic gradients:

$$\tilde{\nabla} f_i(\boldsymbol{x}_t) = \frac{1}{K}\sum_{j=1}^{K} \nabla f(\boldsymbol{x}_t; \zeta_{ij}).$$

Since each $\nabla f(\boldsymbol{x}_t;\zeta_{ij})$ is independent and has variance bounded by σ^2, the variance of $\tilde{\nabla} f_i(\boldsymbol{x}_t)$ can be computed as:

$$\mathbb{E}[\|\tilde{\nabla} f_i(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2] = \mathbb{E}\left[\left\|\frac{1}{K}\sum_{j=1}^{K}(\nabla f(\boldsymbol{x}_t;\zeta_{ij})-\nabla L(\boldsymbol{x}_t))\right\|^2\right].$$

Expanding the squared norm and using the independence of the gradients, we have:

$$\mathbb{E}\left[\left\|\frac{1}{K}\sum_{j=1}^{K}\left(\nabla f(\boldsymbol{x}_t;\zeta_{ij})-\nabla L(\boldsymbol{x}_t)\right)\right\|^2\right] = \frac{1}{K^2}\sum_{j=1}^{K}\mathbb{E}\left[\|\nabla f(\boldsymbol{x}_t;\zeta_{ij})-\nabla L(\boldsymbol{x}_t)\|^2\right].$$

By the assumption that $\mathbb{E}[\|\nabla f(\boldsymbol{x}_t;\zeta_{ij})-\nabla L(\boldsymbol{x}_t)\|^2]\le\sigma^2$, it follows that:

$$\mathbb{E}[\|\tilde{\nabla} f_i(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2]\le\frac{\sigma^2}{K}.$$

The estimator $\tilde{\nabla} f(\boldsymbol{x}_t)$ is the average of N independent $\tilde{\nabla} f_i(\boldsymbol{x}_t)$:

$$\tilde{\nabla} f(\boldsymbol{x}_t)=\frac{1}{N}\sum_{i=1}^{N}\tilde{\nabla} f_i(\boldsymbol{x}_t).$$

Since each $\tilde{\nabla} f_i(\boldsymbol{x}_t)$ is independent and has variance bounded by $\frac{\sigma^2}{K}$, the variance of $\tilde{\nabla} f(\boldsymbol{x}_t)$ satisfies:

$$\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2] = \mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N}(\tilde{\nabla} f_i(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t))\right\|^2\right].$$

Expanding the squared norm and using the independence of the $\tilde{\nabla} f_i(\boldsymbol{x}_t)$, we have:

$$\mathbb{E}\left[\left\|\frac{1}{N}\sum_{i=1}^{N}\left(\tilde{\nabla} f_i(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\right)\right\|^2\right] = \frac{1}{N^2}\sum_{i=1}^{N}\mathbb{E}\left[\|\tilde{\nabla} f_i(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2\right].$$

Substituting the bound $\mathbb{E}[\|\tilde{\nabla} f_i(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2]\le\frac{\sigma^2}{K}$, we obtain:

$$\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2]\le\frac{\sigma^2}{NK}.$$

□

Theorem C.8 (Non-Convex Convergence Rate)**.**

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]\le O\left(\frac{1}{\sqrt{T}}\right).$$

Proof. From L-smoothness:

$$L(\boldsymbol{x}_{t+1})\le L(\boldsymbol{x}_t)+\langle\nabla L(\boldsymbol{x}_t),\boldsymbol{x}_{t+1}-\boldsymbol{x}_t\rangle+\frac{L}{2}\|\boldsymbol{x}_{t+1}-\boldsymbol{x}_t\|^2.$$

Since $\boldsymbol{x}_{t+1}=\boldsymbol{x}_t-\eta\tilde{\nabla} f(\boldsymbol{x}_t)$:

$$L(\boldsymbol{x}_{t+1})\le L(\boldsymbol{x}_t)-\eta\langle\nabla L(\boldsymbol{x}_t),\tilde{\nabla} f(\boldsymbol{x}_t)\rangle+\frac{L\eta^2}{2}\|\tilde{\nabla} f(\boldsymbol{x}_t)\|^2.$$

Taking expectation and using Lemma C.6:

$$\mathbb{E}[L(\boldsymbol{x}_{t+1})]\le\mathbb{E}[L(\boldsymbol{x}_t)]-\eta\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]+\frac{L\eta^2}{2}\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t)\|^2].$$

Decompose $\|\tilde{\nabla} f(\boldsymbol{x}_t)\|^2$:

$$\|\tilde{\nabla} f(\boldsymbol{x}_t)\|^2=\|\nabla L(\boldsymbol{x}_t)+(\tilde{\nabla} f(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t))\|^2.$$

Taking expectations:

$$\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t)\|^2]=\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]+\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2].$$

By Lemma C.7:

$$\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t)-\nabla L(\boldsymbol{x}_t)\|^2]\le\frac{\sigma^2}{NK}.$$

Thus:

$$\mathbb{E}[L(\boldsymbol{x}_{t+1})]\le\mathbb{E}[L(\boldsymbol{x}_t)]-\eta\,\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]+\frac{L\eta^2}{2}\left(\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]+\frac{\sigma^2}{NK}\right).$$

Rearranging:

$$\eta\,\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]\le\mathbb{E}[L(\boldsymbol{x}_t)-L(\boldsymbol{x}_{t+1})]+\frac{L\eta^2}{2}\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]+\frac{L\eta^2\sigma^2}{2NK}.$$

So:

$$(\eta-\tfrac{L\eta^2}{2})\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]\le\mathbb{E}[L(\boldsymbol{x}_t)-L(\boldsymbol{x}_{t+1})]+\frac{L\eta^2\sigma^2}{2NK}.$$

Summing over $t=1$ to T:

$$(\eta-\tfrac{L\eta^2}{2})\sum_{t=1}^{T}\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]\le L(\boldsymbol{x}_1)-L(\boldsymbol{x}_{T+1})+\frac{L\sigma^2\eta^2}{2NK}\cdot T.$$

Rearranging:

$$\frac{1}{T}(\eta-\tfrac{L\eta^2}{2})\sum_{t=1}^{T}\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]\le\frac{L(\boldsymbol{x}_1)-L(\boldsymbol{x}_{T+1})}{T}+\frac{L\sigma^2\eta^2}{2NK}.$$

Since $L(\boldsymbol{x})$ is bounded below, let $L(\boldsymbol{x}^*)$ be a lower bound:

$$L(\boldsymbol{x}_1)-L(\boldsymbol{x}_{T+1})\le L(\boldsymbol{x}_1)-L(\boldsymbol{x}^*).$$

Then:

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]\le\frac{2(L(\boldsymbol{x}_1)-L(\boldsymbol{x}^*))}{T\eta(2-L\eta)}+\frac{L\sigma^2\eta}{2NK}.$$

Let $\eta=\frac{c}{\sqrt{T}}$:

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2]\le\frac{2\sqrt{T}(L(\boldsymbol{x}_1)-L(\boldsymbol{x}^*))}{Tc(2-\frac{Lc}{\sqrt{T}})}+\frac{L\sigma^2c}{2NK\sqrt{T}}$$

$$=\frac{2(L(\boldsymbol{x}_1)-L(\boldsymbol{x}^*))}{c(2\sqrt{T}-Lc)}+\frac{L\sigma^2c}{2NK\sqrt{T}}=O\left(\frac{1}{\sqrt{T}}\right).$$

□

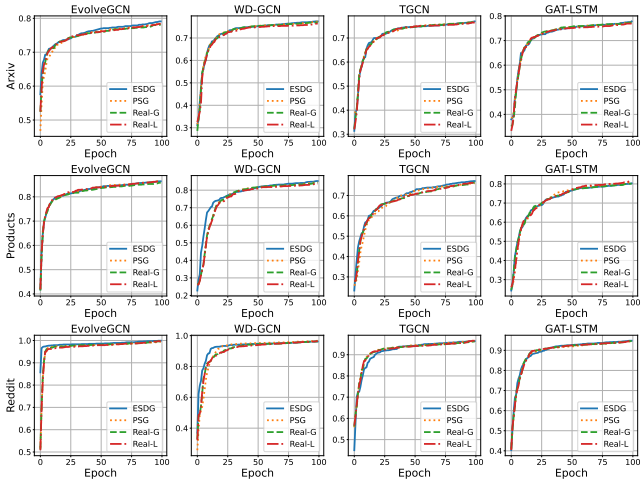


Figure 18. *Model accuracy.*

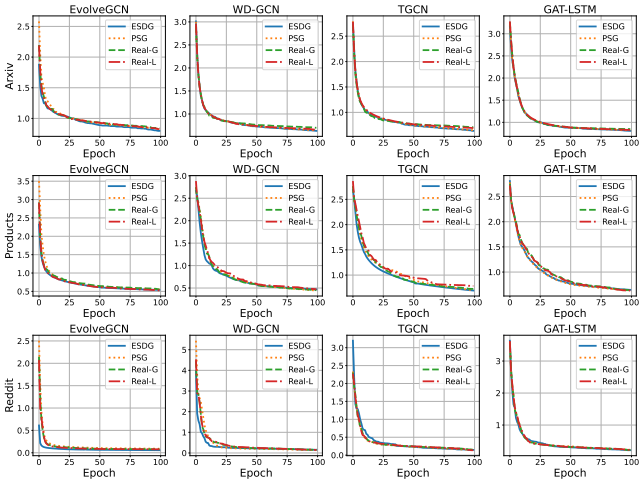


Figure 19. *Training loss.*

D Model Accuracy and Loss

The impact on model accuracy between Real-L and Real-G versus PSG and ESDG is directly analogous to increasing the training batch size. To ensure that this does not lead to any accuracy degradation, we compared the model accuracy and loss over 100 epochs between Real-L, Real-G, PSG and ESDG methods on the Arxiv, Products, and Reddit datasets, as shown in Figures 18 and 19. The Stackoverflow, Stackoverflow-a2q and wiki dataset, lacking labels, is not included in the accuracy comparison.

From Figure 18, it is evident that both Real-G and Real-L achieve accuracy levels comparable to ESDG throughout the entire training process. The accuracy curves for all three methods exhibit minimal divergence, suggesting that the scheduling techniques implemented in Real-G and Real-L do not degrade the model's ability to learn effective representations.

Similarly, Figure 19 illustrates the training loss trajectories over the course of 100 epochs. The loss curves for Real-G and Real-L closely align with that of ESDG, demonstrating nearly identical convergence behavior. This alignment indicates that the optimization process remains stable and efficient under the proposed scheduling strategies. This consistency in accuracy and loss highlights the robustness of the proposed scheduling approaches, as they maintain the model's performance without introducing significant deviations from the baseline method.

E Simulation of Scalability

In this section, we describe our comprehensive simulation setup and methodology for predicting snapshot execution times. Our experimental framework encompasses both the generation of dynamic graph snapshots and the development of an accurate time prediction model for computational resource optimization.

For the simulation of dynamic graph evolution, we followed the established methodology of Dygraph[19], generating 10,000 snapshots together with their corresponding `diffmap` representations to ensure a robust characterization of temporal network dynamics.

To develop an accurate prediction model for snapshot execution times, we conducted detailed profiling experiments on NVIDIA A100 GPUs. During profiling, we recorded both the total execution time and the potential reuse time for each snapshot, where reuse time represents computations that could be shared between snapshots. Our analysis revealed a strong linear correlation between snapshot training time and the graph's structural characteristics, specifically the number of nodes and edges. This observation led us to develop a linear regression model for predicting snapshot training times:

$$t_{snapshot_i} = \alpha_1 \cdot N_{node_i} + \alpha_2 \cdot N_{edge_i} + \alpha_3 \tag{10}$$

where $t_{snapshot_i}$ represents the predicted training time for snapshot i, N_{node_i} and N_{edge_i} denote the number of nodes and edges in snapshot i, respectively, and α_1, α_2, and α_3 are learned coefficients that capture the relationship between graph structure and computational requirements. For snapshots that are not the first in a group, we instead simulate their execution time using the corresponding `diffmap`, by replacing N_{node_i} and N_{edge_i} with the number of changed nodes and edges. For estimating the reusable computation time between snapshot groups, we developed a ratio-based model that accounts for temporal overlap:

$$\mathcal{R}_{i,j} = \sum_{k=i+1}^{max(i,i+tw-(j-i))} t_{snapshot_k} \cdot RATIO \tag{11}$$

where tw denotes the time window size. This allows us to estimate the effective execution time when combining unprofiled snapshot groups using:

$$t_{combined} = t_{group_i} + t_{group_j} - \mathcal{R}_{i,j} \tag{12}$$

where $t_{combined}$ represents the predicted execution time for two combined snapshot groups, t_{group_i} and t_{group_j} are their individual predicted times, and $\mathcal{R}_{i,j}$ quantifies the computational overlap between groups i and j based on our ratio model. This comprehensive approach enables us to effectively estimate the computational resources needed for processing both individual snapshots and combined groups, facilitating efficient resource allocation and workload planning.

Using another set of data for extrapolation, we found that the prediction error was less than 5%, as illustrated in Figure 20. This model was then used to simulate the execution time of each snapshot in our evaluation.

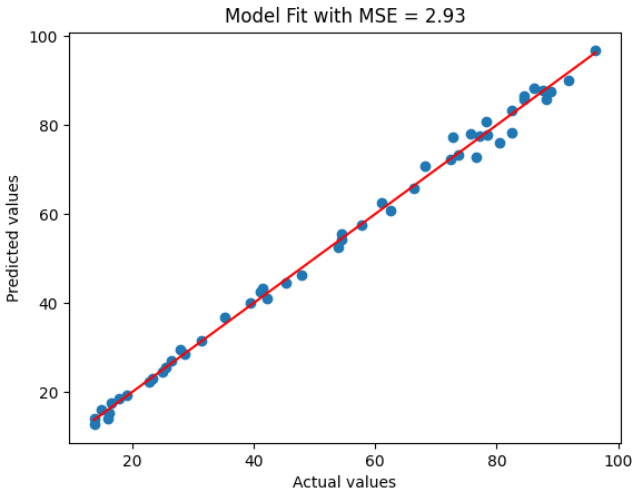


Figure 20. *Comparison between predicted and measured snapshot training time.*