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

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 often prioritize one objective at the expense of others, leading to either communication bottlenecks caused by fine-grained partitioning or load imbalance stemming from temporal variations. We present Real, an efficient distributed system for DGNN training that breaks this trilemma. Real leverages a novel incremental aggregation mechanism to decouple computation from structural updates, effectively pruning redundant FLOPS via snapshot- and group-level reuse. To orchestrate these optimizations, we formulate the workload distribution as a makespan minimization problem and propose a cross-group scheduling algorithm that co-optimizes data reuse and resource utilization within a triple-stream execution pipeline. Furthermore, we design two complementary scheduling policies: Real-L, which leverages ILP for globally optimal planning, and Real-G, a greedy fallback capable of scaling to massive graphs. Extensive experiments on six dynamic graph datasets and four DGNN models show that Real-L and Real-G achieve up to 3.26× and 3.16× speedup over state-of-the-art baselines, respectively, with excellent scalability.

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 [1, 4, 16, 26, 29, 33, 39, 45, 46, 49, 52, 53, 55, 56, 58, 59, 62, 63]. 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 [9, 10, 50], 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* [13, 14, 18, 25, 36, 43, 48], 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, 11, 41]. 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 [11] 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 [41] 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 [18, 19, 40, 48] have shown that temporally adjacent snapshots exhibit substantial topology similarity, revealing opportunities for reusing invariant data to reduce redundant computation and I/O. However, exploiting such reuse inherently introduces computational irregularity, as the degree of redundancy varies dynamically across snapshots. This

Author's Contact Information:

variation exacerbates workload imbalance in distributed settings, yet no existing system jointly optimizes data reuse and load balancing to mitigate this reuse-induced skew.

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, 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 state-of-the-art 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 state-of-the-art method, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Dynamic Graph Neural Networks

DGNNs consist of blocks that combine structural and temporal encoding. Each block contains a structure encoder for neighborhood information aggregation and a time encoder for capturing temporal evolution. Different DGNN models implement these encoders in various ways. For example, EvolveGCN [33] dynamically adjusts its graph convolutional network (GCN) [24] parameters over time to accommodate the evolving nature of the graph. WD-GCN [29] combines a GCN with a long

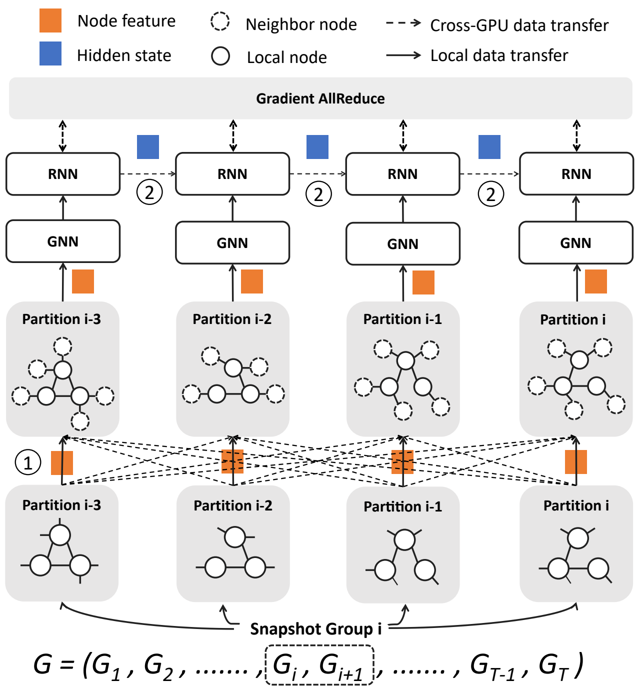


Fig. 1. **Workflow of distributed training for DGNNs.** *The dynamic graph is processed in snapshot groups, with partitions distributed across GPUs (gray boxes). Solid and dashed circles represent local and cross-partition neighbors, respectively. The workflow highlights two data transfer overheads: ① **Neighbor Feature Transfer** (dashed arrows) exchanges boundary-node features (orange squares) prior to GNN aggregation; ② **Hidden State Transfer** synchronizes boundary-node states (blue squares) to maintain temporal consistency across snapshots.*

short-term memory (LSTM) [22] network to capture both spatial and temporal features in dynamic graphs. TGCN [4] integrates a GCN with a gated recurrent unit (GRU) [7] to effectively capture spatial and temporal dynamics in dynamic graphs. GAT-LSTM [55] leverages a Graph Attention Network (GAT)[47] for capturing structural information while using an LSTM to model temporal dependencies. These models form the foundation for many subsequent variants, such as TTGCN [26], DRAIN [1], SGNN-GR [49], ROLAND [58], Dyngraph2vec [16], and DySAT [39].

2.2 Workflow of Distributed DGNN Training

Training a DGNN on multiple GPUs necessitates efficient coordination of graph data and model states to mitigate communication bottlenecks. As illustrated in Figure 1, the dynamic graph $\mathcal{G} = (\mathcal{G}_1, \ldots, \mathcal{G}_T)$ is partitioned into *snapshot groups*, where each group serves as a basic scheduling unit processed in one iteration. Within a distributed environment, the training workflow consists of three distinct phases characterized by their data dependencies:

Data Loading. Each iteration commences with transferring the assigned graph partition and input node features from host to device memory (HtoD). This step prepares data for GPU execution but incurs significant PCIe bandwidth overhead, particularly for large-scale graphs.

Spatial Aggregation. For a node v in snapshot $\mathcal{G}_t$, the graph neural network (GNN) aggregates features from its local neighborhood $\mathcal{N}(v)$:

$$\mathcal{H}_t(v) = \text{SpatioAggr}_v(\mathcal{W}_{\text{gnn}}, \{\mathcal{X}_t(u) \mid u \in \mathcal{N}(v)\}, \mathcal{X}_t(v)) \quad (1)$$

Category	Dimension	AliGraph [64]	DistDGL [60]	ESDG [3]	DGC [5]	BLAD [11]	DynaHB [41]	Real
Data Transfer	*No Neighbor Feature Transfer*	×	×	✓	×	✓	✓	✓
	No Hidden State Transfer	✓	✓	×	×	✓	✓	✓
	No Extra CPU-GPU Transfer	✓	✓	✓	✓	✓	×	✓
Resource Util.	*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).*

Due to graph partitioning, the neighborhood $\mathcal{N}(v)$ may span across device boundaries. Consequently, calculating $\mathcal{H}_t(v)$ for boundary nodes triggers *neighbor feature transfer* (①), where feature vectors (indicated as orange squares in Figure 1) must be fetched from remote GPUs. This irregular memory access pattern can dominate runtime in dense graphs.

Temporal Evolution. The spatially aggregated embeddings are subsequently fed into a temporal model (e.g., RNN) to capture historical dependencies:

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

This step enforces a sequential dependency on the previous hidden state h_{t-1}. If consecutive snapshots (or the relevant boundary nodes) reside on different devices, *hidden state transfer* (②) is required to synchronize the boundary hidden states (blue squares). This creates a synchronization barrier that can lead to GPU idling.

Parameter Synchronization. Following the forward and backward passes for the entire snapshot group, a global *Gradient AllReduce* is performed to aggregate gradients and synchronize the GNN/RNN parameters across all participating GPUs before the next iteration.

2.3 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 [60, 64]. 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 [11] 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.

2.4 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 [29] on three

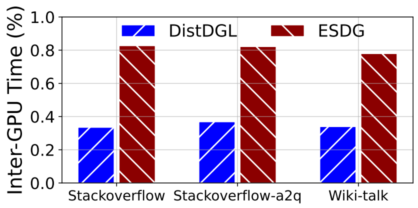


Fig. 2. *Inter-GPU data transfer time during forward pass for DistDGL and ESDG.*

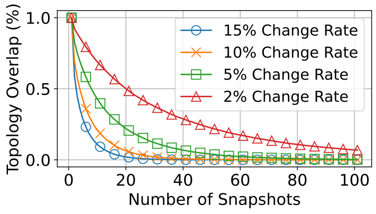


Fig. 3. *Effect of change rate and snapshot count on graph topology overlap.*

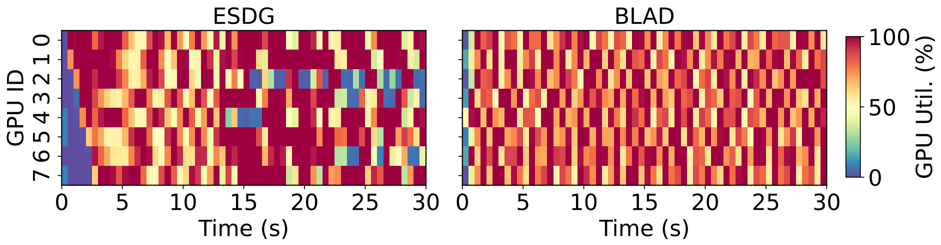


Fig. 4. *GPU utilization of ESDG and BLAD on the Stackoverflow dataset.*

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 idle while waiting for hidden states to be transferred. Using WD-GCN [29] 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 [42] 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.

Ineffective Utilization of Topological Similarity. Existing strategies fail to effectively translate topological similarity into distributed performance gains due to their limited applicability and neglect of system-level side effects. For instance, DGC [5] reuses graph embeddings across epochs, but ignores topological similarity between graphs. In the single-GPU setting, PiPAD [48] adopts a global structured reuse scheme by identifying the global intersection across snapshots. 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), diminishing its reuse potential. Similarly, reINC [19] employs incremental aggregation to prune

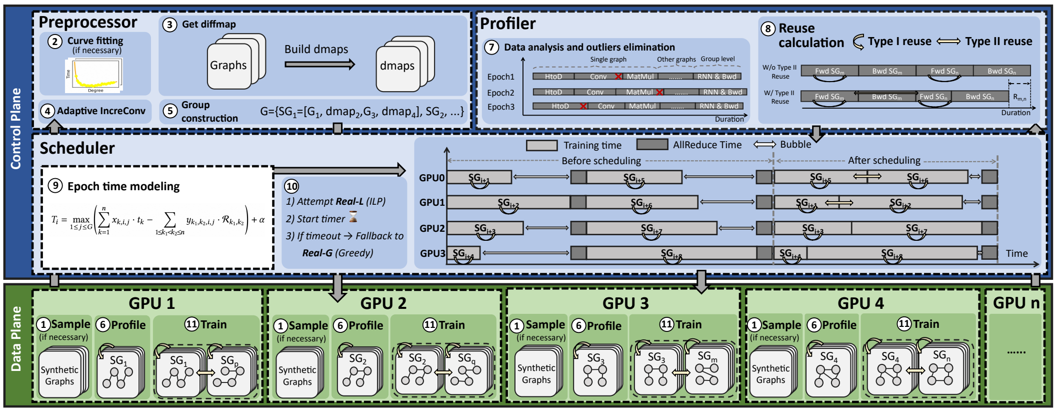


Fig. 5. **Overview of the Real architecture.** *The system decouples the Control Plane (CPU) for preprocessing and scheduling from the Data Plane (GPU) for parallel training. Guided by runtime profiles, the Scheduler dynamically selects between the ILP solver (Real-L) and the greedy heuristic (Real-G) to dispatch optimized, load-balanced workloads.*

redundant computations. Crucially, however, neither PiPAD nor reINC supports GATs. Efficiently updating the global non-linear softmax normalization using solely incremental information remains a significant challenge. Furthermore, in the distributed context, reINC also fails to co-optimize reuse with load balancing. It ignores the computational heterogeneity induced by varying reuse rates across snapshots, which exacerbates load imbalance and straggler effects among GPUs.

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 Real system.

Overview. Real adopts a two-plane architecture to decouple coordination from execution. The control plane (CPU) orchestrates graph analysis and workload planning via the *Preprocessor*, *Profiler*, and *Scheduler*. Conversely, the data plane (multi-GPU) is dedicated to executing the distributed DGNN training workflow.

Preprocessor. The preprocessor analyzes the DTDG topology to guide efficient 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. 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 $\mathcal{SG}_i = [\, \mathcal{G}_i, \texttt{dmap}_{i+1}, \mathcal{G}_{i+2}, \texttt{dmap}_{i+3} \,]$. Details are provided 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.

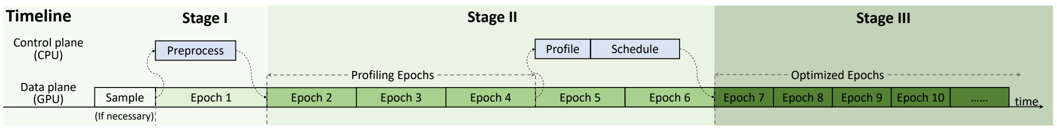


Fig. 6. **Pipelined execution timeline.** *REAL overlaps control plane overheads (preprocessing and scheduling) with data plane training to hide latency, enabling a non-blocking transition to the optimized schedule in Stage III.*

3.3 Overlapping Control Plane Overhead

Figure 6 shows the training 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.

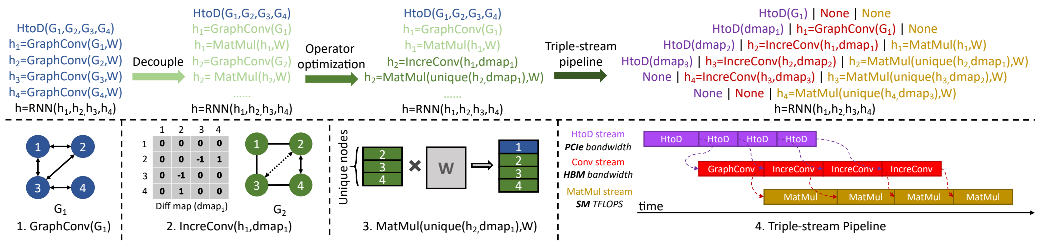


Fig. 7. **Snapshot-level data reuse.** *Using GCN with aggregation preceding transformation as an example.*

4 TOPOLOGY-AWARE DATA REUSE

In this section, we present our design to exploit the topology 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 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 7) 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. Note that `IncreConv` reuses cached aggregation results from the previous snapshot, which are already retained for subsequent RNN computation, thus incurring no extra memory cost.

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 a pair of integer attributes: its degree in the current full snapshot ($d^{(t)}$) and its degree in the preceding base snapshot

($d^{(t-1)}$). This design allows us to handle topological updates locally without maintaining a global edge weight table. Specifically, for a newly added edge (i, j), we compute the normalized weight using the current degrees $d^{(t)}$; conversely, for a deleted edge, we reconstruct its obsolete weight on-the-fly using the stored base degrees $d^{(t-1)}$ to strictly subtract its prior contribution from the aggregation. By encapsulating these dual degree states within the sparse `dmap`, `IncreConv` ensures mathematical equivalence to standard GCN while significantly reducing memory footprint by eliminating the need to cache weights for all existing edges.

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 $h_j^{(t-1)}$ from snapshot $t-1$:

$$\mathcal{D}_j^{(t-1)} = \sum_{i \in \mathcal{N}(j,t-1)} \exp(e_{ij}^{(t-1)}) \tag{3}$$

$$h_j^{(t-1)} = \sum_{i \in \mathcal{N}(j,t-1)} \frac{\exp(e_{ij}^{(t-1)})}{\mathcal{D}_j^{(t-1)}} \mathcal{W} h_i \tag{4}$$

where $e_{ij}^{(t-1)}$ is the unnormalized attention score on edge (i, j), and $\mathcal{W}$ 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 first compute the incremental change of the denominator:

$$\Delta\mathcal{D}_j = \sum_{i \in \Delta\mathcal{E}_j} \pm \exp(e_{ij}^{(t)}) \tag{5}$$

$$\mathcal{D}_j^{(t)} = \mathcal{D}_j^{(t-1)} + \Delta\mathcal{D}_j \tag{6}$$

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

$$\tilde{h}_j = h_j^{(t-1)} \cdot \frac{\mathcal{D}_j^{(t-1)}}{\mathcal{D}_j^{(t)}} \tag{7}$$

$$h_j^{(t)} = \tilde{h}_j + \sum_{i \in \Delta\mathcal{E}_j} \pm \frac{\exp(e_{ij}^{(t)})}{\mathcal{D}_j^{(t)}} \mathcal{W} h_i \tag{8}$$

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. Selecting the optimal aggregation target is non-trivial due to the non-monotonic performance characteristics of the GNN operator. Profiling indicates that `GraphConv` latency exhibits a U-shaped correlation with node density, meaning that processing a `dmap`—even with fewer edges—can paradoxically degrade performance compared to the original graph $\mathcal{G}$ depending on the sparsity regime. To reconcile this, we implement a self-adaptive selection mechanism based on a fitted degree–latency cost curve (see Appendix §A). At runtime, the system estimates the execution costs for both $\mathcal{G}$ and `dmap` based on their current average degrees and dispatches the computation to the path with the lowest predicted cost.

4.1.2 Transformation Phase Reuse. The applicability of transformation reuse depends on the dependency order between the linear projection (`MatMul`) and neighbor aggregation (`GraphConv`). *Case 1: Pre-aggregation Transformation (e.g., GAT).* In architectures where feature transformation precedes aggregation, the input node features are *time-invariant* within a snapshot group. Consequently, the projection is computed only once for the first snapshot and shared across all subsequent snapshots, eliminating redundant computations entirely. *Case 2: Post-aggregation Transformation (e.g., GCN).* When transformation follows aggregation, the input to `MatMul` depends on the temporally evolving topology. Here, we employ *selective execution*: only nodes with topologically updated neighborhoods (identified via `dmap`) trigger new `MatMul` operations, while unaltered nodes directly reuse cached results from the preceding snapshot.

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 7, `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` $\longrightarrow$ `MatMul` $\longrightarrow$ `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 leverages the property that model parameters remain frozen within a single distributed training iteration. Consequently, when consecutive snapshot groups share overlapping graph structures, their associated data can persist in resident GPU memory across group boundaries. This mechanism effectively eliminates redundant `HtoD` transfers and expensive `GraphConv` computations for the overlapping portions. To fully exploit this opportunity, the Real scheduler adopts a similarity-driven scheduling strategy. Instead of adhering to a naive order that results in disjoint pairs (*e.g.*, processing $[\mathcal{G}_1, \mathcal{G}_2]$ followed by $[\mathcal{G}_3, \mathcal{G}_4]$), the scheduler proactively reorders the execution sequence to maximize structural overlap (*e.g.*, chaining $[\mathcal{G}_1, \mathcal{G}_2]$ with $[\mathcal{G}_2, \mathcal{G}_3]$). This similarity-aware rescheduling constructs a reuse-centric execution chain that significantly reduces I/O and compute overhead while maintaining workload balance. The concrete *cross-group scheduling* algorithm is 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

We consider a set of snapshot groups $\mathcal{SG} = \{1, \ldots, n\}$, a set of available GPUs (devices) $\mathcal{D} = \{1, \ldots, D\}$, and a sequence of training iterations $\mathcal{I} = \{1, \ldots, m\}$. The primary objective of Real is to minimize the total per-epoch training time $T = \sum_{i \in \mathcal{I}} T_i$, where T_i denotes the duration (makespan) of iteration i. We define binary decision variables $x_{k,i,j} \in \{0, 1\}$ to indicate whether snapshot group $k \in \mathcal{SG}$ is assigned to GPU $j \in \mathcal{D}$ in iteration $i \in \mathcal{I}$. Let t_k be the standalone execution time of group k, and $\mathcal{R}_{k_1,k_2}$ be the reuse benefit (time saving) if groups k_1 and k_2 are co-located. The actual processing time (load) of GPU j in iteration i, denoted as $\mathcal{L}_{i,j}$, accounts for both computation cost and reuse reduction:

$$\mathcal{L}_{i,j} = \underbrace{\sum_{k \in \mathcal{SG}} t_k \cdot x_{k,i,j}}_{\text{Base Computation}} - \underbrace{\sum_{k_1 < k_2} \mathcal{R}_{k_1,k_2} \cdot \mathbb{I}(x_{k_1,i,j} = 1 \wedge x_{k_2,i,j} = 1),}_{\text{Reuse Benefit}} \tag{9}$$

where $\mathbb{I}(\cdot)$ is the indicator function. The duration of iteration i is determined by the bottleneck GPU plus synchronization overhead α:

$$T_i = \max_{j \in \mathcal{D}} \{\mathcal{L}_{i,j}\} + \alpha. \tag{10}$$

Minimizing T requires jointly optimizing the assignment x to balance loads and maximize reuse. This problem generalizes the classical makespan minimization on parallel machines [15] and is NP-hard. To ensure system stability, we impose a capacity constraint L (e.g., $L = 2$) on the number of groups per GPU per iteration.

5.2 REAL-L: ILP-based Scheduling

To solve the formulation efficiently, we propose REAL-L, which models the problem as an Integer Linear Programming (ILP) task. Standard formulations of Eq. (10) often involve the max operator and logical AND conditions, which are non-linear. We linearize these components as follows:

Assignment and Capacity Constraints. We strictly enforce that every snapshot group is scheduled exactly once, and no GPU exceeds its capacity limit L:

$$\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{D}} x_{k,i,j} = 1, \quad \forall k \in \mathcal{SG}, \tag{11}$$

$$\sum_{k \in \mathcal{SG}} x_{k,i,j} \le L, \quad \forall i \in \mathcal{I}, \forall j \in \mathcal{D}. \tag{12}$$

Reuse Linearization (McCormick Envelopes). To handle the conditional reuse term, we introduce auxiliary binary variables $y_{k_1,k_2,i,j}$ for all pairs $k_1 < k_2$. We enforce $y_{k_1,k_2,i,j} = 1 \iff x_{k_1,i,j} = 1 \wedge x_{k_2,i,j} = 1$ using standard McCormick envelope constraints:

$$\begin{aligned} &y_{k_1,k_2,i,j} \le x_{k_1,i,j}, \\ &y_{k_1,k_2,i,j} \le x_{k_2,i,j}, \qquad \forall i \in \mathcal{I}, \forall j \in \mathcal{D}, \forall k_1 < k_2. \\ &y_{k_1,k_2,i,j} \ge x_{k_1,i,j} + x_{k_2,i,j} - 1. \end{aligned} \tag{13}$$

Min-Max Formulation via Epigraph Transformation. A direct linearization of the max operator in Eq. (10) typically requires Big-M constraints, which loosen the linear relaxation and hinder solver performance. Instead, we treat T_i as a continuous decision variable and apply an epigraph-style formulation. We constrain T_i to be lower-bounded by the load of every GPU in iteration i:

$$T_i \ge \underbrace{\sum_{k \in \mathcal{SG}} t_k \cdot x_{k,i,j}}_{\text{Total Cost}} - \underbrace{\sum_{k_1 < k_2} \mathcal{R}_{k_1,k_2} \cdot y_{k_1,k_2,i,j}}_{\text{Reuse Benefit}} + \alpha, \quad \forall j \in \mathcal{D}. \tag{14}$$

Since the objective is to minimize $\sum T_i$, the solver naturally tightens T_i to the maximum load among GPUs ($\max_{j \in \mathcal{D}} L_{i,j} + \alpha$) without requiring auxiliary binary indicators. This formulation significantly tightens the relaxation bound, pruning the branch-and-bound search space efficiently.

We implement REAL-L using the Gurobi solver [21]. Given the NP-hardness, we set a termination criterion based on a preset optimality gap (e.g., 10%) to balance schedule quality and solving time.

Algorithm 1: Real-G

```
Input  : GPU count $D$, Reuse matrix $\mathcal{R}$, Snapshot group times $\mathcal{SG} = [t_1, t_2, \ldots, t_n]$
Output : Schedule strategy $\mathbb{X}$
1  Initialize $\mathcal{SG} \leftarrow$ Preprocess($\mathcal{SG}$), $n \leftarrow |\mathcal{SG}|$, $\mathbb{X} \leftarrow \emptyset$;
2  while $|\mathcal{SG}| > D$ do
3      $targets \leftarrow$ GetCandidateTargets($\mathcal{SG}, \mathcal{R}, n$);
4      $W_{min} \leftarrow \infty$;
5      foreach $\mathcal{T} \in targets$ do
6          $pairs \leftarrow$ GetPairs($\mathcal{T}, \mathcal{R}, \mathcal{SG}, D, n$);
7          $W \leftarrow D \cdot \max\Big(\max_j \ pairs[j].T, \ \mathcal{T}\Big)$;
8          $W \leftarrow W - \mathcal{T} - \sum_{j=1}^{D-1} pairs[j].T$;
9          if $W < W_{\min}$ then $W_{\min} \leftarrow W$;
10     Update $\mathcal{SG}, \mathbb{X}$;
11 Record rest groups in $\mathbb{X}$;
12 return $\mathbb{X}$;
--------------------------------------------------------------
13 Function Preprocess($\mathcal{SG}$):
14     Add $t_0 = 0$ to $\mathcal{SG}$ and sort ascending;
15     return $\mathcal{SG}$;
16 Function GetCandidateTargets($\mathcal{SG}, \mathcal{R}, n$):
17     Initialize $targets \leftarrow \emptyset$;
18     for $i = 0$ to $n - 1$ do
19         Add $\{\mathcal{SG}[i] + \mathcal{SG}[n] - \mathcal{R}[i][n]\}$ to $targets$;
20     return $targets$;
21 Function GetPairs($\mathcal{T}, \mathcal{R}, \mathcal{SG}, D, n$):
22     Initialize $pairs \leftarrow \emptyset$;
23     while $|pairs| < D - 1$ do
24         Initialize $head, tail, best\_pair$;
25         while $head < tail$ do
26             $T \leftarrow \mathcal{SG}[head] + \mathcal{SG}[tail] - \mathcal{R}[head][tail]$;
27             if $|T - \mathcal{T}| < |best\_pair.T - \mathcal{T}|$ then
28                 $best\_pair \leftarrow [T, head, tail]$;
29             $T < \mathcal{T}$ ? $head$ ++ : $tail$ −−
30         Add $best\_pair$ to $pairs$;
31     return $pairs$;
```

5.3 Real-G: Greedy-based Scheduling

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. The following describes these steps in detail.

Input. The algorithm takes as input the GPU count D, the reuse matrix $\mathcal{R}$ capturing type II reuse savings, and the group list $\mathcal{SG} = [t_1, t_2, \dots, t_n]$ 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.

Candidate target generation. Line 3 generates a set of potential makespan targets, denoted as *targets*, using `GetCandidateTargets` (lines 16-20). This procedure pairs the group with the longest remaining execution time (denoted as $\mathcal{SG}[n]$ after sorting) with other groups to project potential bottleneck durations for the current iteration.

Group selection and waste time calculation. Lines 5–9 evaluate each candidate target $\mathcal{T} \in$ *targets*. For each target $\mathcal{T}$, `GetPairs` (lines 21-31) performs a two-point search to efficiently identify the group combination that best matches $\mathcal{T}$. The algorithm then calculates the wasted time W_i (lines 8–9) to quantify the quality of the schedule:

$$W_i = \underbrace{D \cdot (T_i - \alpha)}_{\text{Allocated Capacity}} - \underbrace{\sum_{j \in \mathcal{D}} \mathcal{L}_{i,j}}_{\text{Total Effective Load}} . \quad (15)$$

The algorithm selects the schedule that yields the minimum waste W_i, ensuring efficient group scheduling for the current iteration.

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 IMPLEMENTATION

We implement REAL on PyTorch [34] (training backend) and DGL [50] (graph backend). REAL introduces system-level extensions in two places: the GNN operator and the data-loading runtime.

Operator extensions. We override DGL's `GraphConv` and `GATConv` to expose intermediate aggregation states across snapshots and to support incremental execution. The operators take diffmaps as auxiliary inputs and selectively update the affected nodes while keeping full compatibility with DGL's operator interfaces.

Scheduling runtime. We replace DGL's default dataloader with a custom `REALDataLoader`. It treats snapshot groups as training units, materializes snapshots or diffmaps on demand and dispatches groups according to the REAL-L or REAL-G schedule. `REALDataLoader` integrates with PyTorch's training loop through standard iterator semantics and performs asynchronous prefetching to overlap data preparation with GPU execution.

7 EVALUATION

In this section we first present our experimental setup, including the testbed, models, datasets, and baselines (§7.1). We evaluate REAL for training throughput, data transfer time, `GraphConv` FLOPs, and load imbalance (§B.1), conduct ablation studies (§B.2), and confirm its accuracy (§7.4). We also assess scalability via simulations (§7.5).

7.1 Methodology

Testbed. We evaluate Real on two testbeds summarized in Table 2: a flagship H200-based cluster and a lower-tier H20-based cluster. Cluster A contains four servers, each equipped with 192-core Intel

Component	Cluster A (4×8)	Cluster B (2×8)
CPU	192-core Intel Xeon Platinum 8558 @ 4.0 GHz	192-core Intel Xeon Platinum 8575C @ 4.0 GHz
GPU	8× NVIDIA H200	8× NVIDIA H20
Intra-node GPU BW	900 GB/s NVLink	900 GB/s NVLink
Inter-node Network	3.2 Tbps RoCE (8× ConnectX-7)	1.6 Tbps RoCE (4× ConnectX-7)

Table 2. **Hardware configuration of our testbeds.** *The main evaluation is conducted on Cluster A, while results for Cluster B are provided in the Appendix B.*

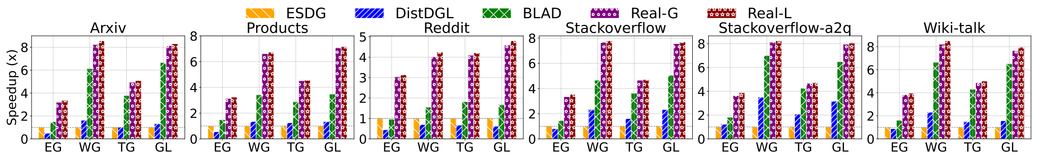
Dataset	$\overline{\lvert\mathcal{V}\rvert}$	$\overline{\lvert\mathcal{E}\rvert}$	$\overline{\lvert\mathcal{V}_d\rvert}$	$\overline{\lvert\mathcal{E}_d\rvert}$	d_v	β	γ
Arxiv [23]	169.3k	1.8M	88.7k	242.6k	128	10.6	100
Products [23]	2.4M	36.7M	1.5M	5.9M	100	15.0	100
Reddit [38]	232.9k	31.9M	211.6k	5.5M	602	137.3	100
Stackoverflow [42]	2.6M	23.1M	790.3k	3.1M	128	8.9	100
Stackoverflow-a2q [42]	2.5M	10.8M	554.3k	1.5M	128	4.4	100
Wiki-talk [32]	1.1M	6.3M	182.8k	439.4k	128	5.5	100
Arxiv†	169.3k	1.8M	89.3k	245.9k	128	10.6	300
Products†	2.2M	18.6M	1.6M	3.6M	100	7.6	300

Table 3. **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. The first six datasets are used for single-server (1×8 GPUs) experiments, while the $\gamma = 300$ variants (Arxiv† and Products†) are constructed for multi-server evaluation.*

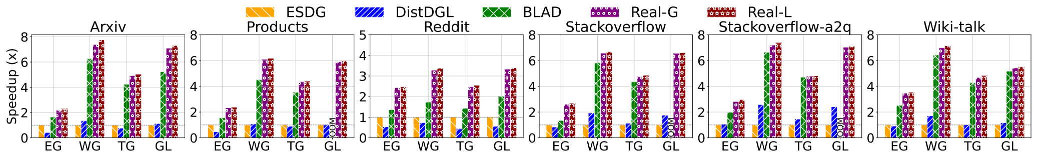
Xeon Platinum 8558 CPUs and eight NVIDIA H200 GPUs (141 GB HBM3e per GPU) interconnected via full-bandwidth NVLink (900 GB/s). Each server also integrates eight ConnectX-7 NICs, providing up to 3.2 Tbps RoCE bandwidth in multi-server experiments. Cluster B comprises two servers with 192-core Intel Xeon Platinum 8575C CPUs and eight NVIDIA H20 GPUs (141 GB HBM3e per GPU) per server, using the same NVLink (900 GB/s) topology and four ConnectX-7 NICs (1.6 Tbps RoCE) per server. We report results on Cluster A in the main text and include Cluster B results in the Appendix B. All experiments are conducted within the DGL NGC Container (version 24.07-py3) [31], which provides DGL v2.4 [50] for scalable graph processing, PyTorch v2.4.0 [34] as the deep learning backend, and CUDA 12.5 for GPU acceleration.

Datasets. We evaluate on six DTDG datasets (Table 3), including both real-world temporal networks and synthetic dynamic graphs derived from static datasets. The real-world datasets are Stackoverflow and Stackoverflow-a2q [42], which record user interactions on the Stack Exchange website, and Wiki-talk [32], which captures edits among Wikipedia users. We also build dynamic graphs from three static graph datasets: Arxiv [23], Products [23], and Reddit [38]. Following BLAD [11], 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 [33], WD-GCN [29], TGCN [4], and GAT-LSTM [55]. 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. For each DGNN, we evaluate both one-layer and two-layer variants, where a layer consists of a spatial aggregation module (GNN) followed by a temporal update module (RNN). The hidden dimension is set to 64 across all model configurations.

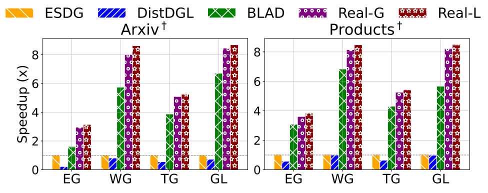


(a) One-layer DGNN models

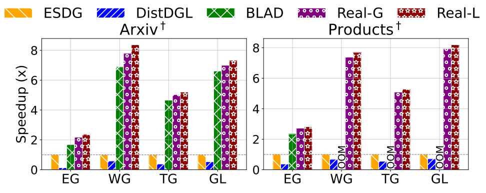


(b) Two-layer DGNN models

Fig. 8. **Training speedup of different systems across DGNN models and datasets on a single server (8 GPUs) of Cluster A.** *Speedup is normalized to ESDG [3].* ***EG**, **WG**, **TG**, and **GL** denote EvolveGCN, WD-GCN, TGCN, and GAT-LSTM, respectively.*



(a) One-layer DGNN models



(b) Two-layer DGNN models

Fig. 9. **Training speedup of different systems across DGNN models and datasets on four servers (32 GPUs) of cluster A.** *Speedup is also normalized to ESDG [3].*

Baselines. We compare Real-G and Real-L with existing state-of-the-art distributed DGNN training methods, including ESDG [3], DistDGL [60], and BLAD [11], as follows:

- ESDG [3]: Snapshots in a snapshot group are evenly distributed across GPUs based on their temporal intervals.
- DistDGL [60]: Each snapshot is partitioned into subgraphs and distributed across GPUs.
- BLAD [11]: Utilizes a two-stage pipeline to 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 [41] is an *asynchronous* training framework, thus not included in the comparison.

7.2 Experimental results

7.2.1 Overall Performance. We first compare the training speedups of all methods on a single server in Cluster A. We prioritize this setting to evaluate the intrinsic system performance, preventing the heavy inter-GPU communication overhead of baselines from dominating the execution time due to limited inter-node bandwidth. As shown in Figure 8, for one-layer DGNN models, Real-L delivers substantial gains over all baselines, achieving a 3.14×–8.56× speedup over ESDG (avg. 5.73×), 2.26×–7.76× over DistDGL (avg. 4.68×), and 1.11×–3.26× over BLAD (avg. 1.83×). Real-G

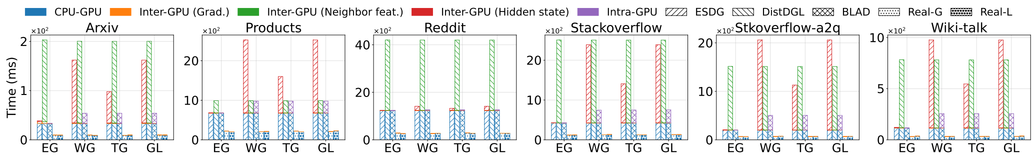


Fig. 10. **Breakdown of data transfer time on a single server in Cluster A.** *The time is decomposed into CPU-GPU transfer, inter-GPU communication (gradients, neighbors, hidden states), and intra-GPU transfer.*

also delivers consistent improvements, with a 3.05×–8.24× speedup over ESDG (avg. 5.57×), 2.23×–7.42× over DistDGL (avg. 4.52×), and 1.10×–3.16× over BLAD (avg. 1.77×). For two-layer models, Real also maintains its performance advantage. Specifically, Real-L achieves speedups of 2.28×–7.77× over ESDG (avg. 4.88×), 2.79×–6.57× over DistDGL (avg. 4.63×), and 1.02×–1.99× over BLAD (avg. 1.39×). Similarly, Real-G achieves a 2.18×–7.39× speedup over ESDG (avg. 4.78×), 2.65×–6.42× over DistDGL (avg. 4.53×), and 1.02×–1.94× over BLAD (avg. 1.36×). 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. Moreover, BLAD requires concurrently loading two groups of data into GPU memory to support its pipelining. This high peak memory footprint leads to Out-Of-Memory (OOM) errors in memory-intensive scenarios, preventing it from running on large datasets.

We further report results on 4 servers (32 GPUs) in Figure 9. While the speedups of Real over ESDG and BLAD remain comparable to the single-server setting, the performance gap with DistDGL widens significantly—with Real-L achieving average speedups of 9.93× (One-layer) and 12.43× (Two-layer). DistDGL is more adversely affected in the multi-machine setting, as its vertex-level partitioning necessitates cross-machine neighbor aggregation, incurring heavy communication overhead due to limited inter-node bandwidth. In contrast, ESDG avoids cross-machine hidden state transfers when the group size matches the per-server GPU count (i.e., four), effectively confining hidden states within each server and enabling efficient inter-group data parallelism.

7.2.2 *Data transfer time breakdown.*

Figure 10 illustrates the breakdown of per-epoch data transfer time on a single server in Cluster A. 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. For example, 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.

7.2.3 `GraphConv` *FLOPs.*

We profile the total `GraphConv` floating-point operations (FLOPs) within one training epoch on a single server in Cluster A. As shown in Figure 11, Real significantly

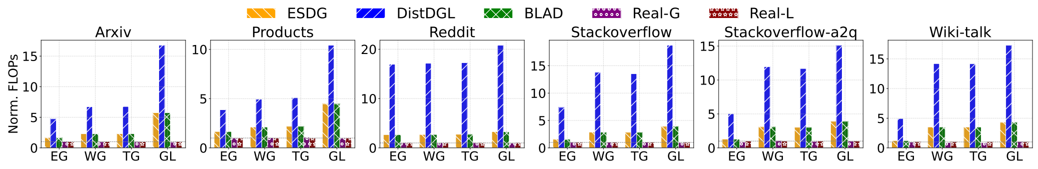


Fig. 11. **Normalized GraphConv FLOPs on a single server in Cluster A.** *FLOPs is normalized to Real-L.*

reduces computational redundancy. Compared to DistDGL, Real-G achieves a FLOPs reduction of 3.88×–21.70× (avg. 11.92×), while Real-L achieves 3.87×–20.80× (avg. 11.64×). Against ESDG and BLAD—which perform full-graph computation and thus incur identical FLOPs—Real-G reduces operations by 1.27×–6.45× (avg. 2.95×), and Real-L by 1.22×–5.74× (avg. 2.87×). 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 structural reuse to strictly limit computation to the incremental updates, effectively pruning these redundant operations.

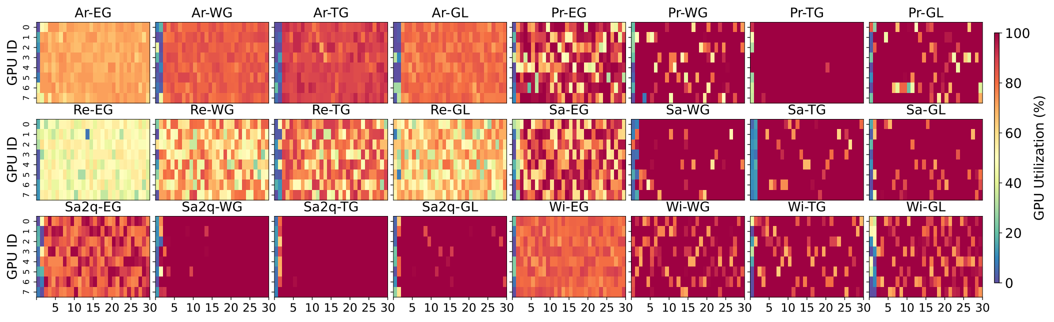


Fig. 12. **GPU utilization heatmaps of Real-L across different DGNN models and datasets on a single server in Cluster A.** ***Ar***, ***Pr***, ***Re***, ***Sa***, ***Sa2q***, *and* ***Wi*** *denote the datasets Arxiv, Products, Reddit, Stackoverflow, Stackoverflow-a2q, and Wiki-talk, respectively.*

7.2.4 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 12, while the results for other baselines are provided in the supplementary material §C. The heatmaps demonstrate that Real-L maintains a consistently high overall GPU utilization, mainly driven by the *Triple-Stream Pipeline* in Type I reuse. By effectively overlapping HtoD transfers with concurrent Conv and MatMul computation streams, the system successfully masks data movement latency and eliminates execution bubbles. Furthermore, Real-L also achieves a remarkably balanced workload distribution across GPUs. This is primarily attributed to its *cross-group* scheduling, which strategically optimizes group assignments to mitigate inter-GPU variance, effectively alleviating the bottleneck effect.

7.3 Ablation Study

To evaluate the individual contribution of each component in Real, we perform a stepwise ablation study normalized to the *Partition-by-Snapshot-Group* (PSG) baseline, where each GPU sequentially

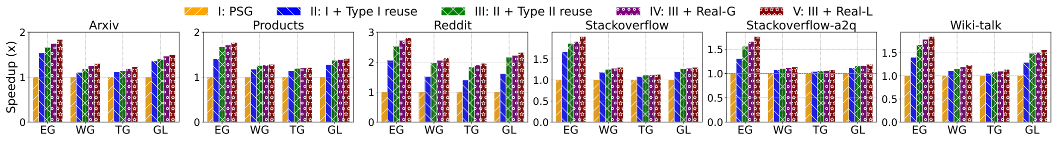


Fig. 13. **Speedup ablation across different models and datasets.** *Speedup is normalized to PSG.*

trains one complete snapshot group per iteration without reuse or load balancing. Figure 20 reports cumulative speedups across models and datasets.

The PSG baseline achieves 1.00× by definition, serving as the starting point.First, adding **type I reuse**—via `IncreConv`, selective `MatMul`, and the triple-stream pipeline—targets intra-group redundancy. This improves performance to 1.04×–2.05× by effectively eliminating redundant computations and overlapping data transfers between snapshots. Next, incorporating **type II reuse**, which by default pairs two consecutive snapshot groups to maximize inter-group locality, further raises speedups to 1.05×–2.52×. This gain stems from removing redundant `HtoD` transfers and `GraphConv` operations across group boundaries. To address the potential imbalance introduced by reuse, the greedy cross-group scheduler (**Real-G**) is introduced; it balances workloads while preserving reuse benefits, achieving 1.06×–2.73×. Finally, replacing the greedy scheduler with the ILP solver (**Real-L**) delivers the highest cumulative speedups of 1.08×–2.81×, benefiting from a global view that optimizes the entire epoch's schedule rather than making local decisions.

7.4 Convergence Analysis

Compared to ESDG, Real effectively increases the batch size per iteration by a factor of K. We analyze the convergence properties of Real under standard assumptions, including L-smoothness of the loss function and bounded gradient variance (detailed in Appendix D). Our analysis hinges on the effective variance reduction achieved by the synchronized distributed updates. Specifically, by aggregating gradients across N GPUs each processing K independent mini-batches, the variance of the gradient estimator $\tilde{\nabla} f(\boldsymbol{x}_t)$ is bounded as follows (Lemma D.7):

$$\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{x}_t) - \nabla L(\boldsymbol{x}_t)\|^2] \leq \frac{\sigma^2}{NK}. \tag{16}$$

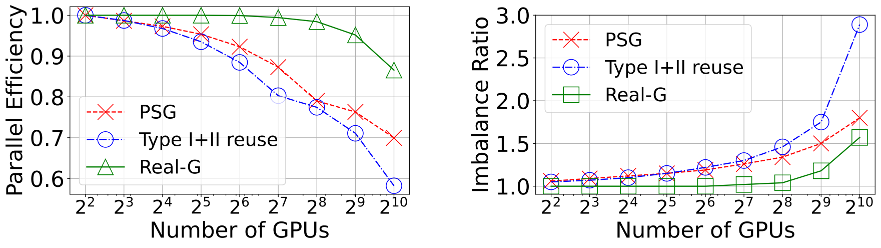
Leveraging this variance reduction, we establish the following convergence guarantee:

Theorem 7.1 (Convergence Rate of Real). *Suppose the loss function L is L-smooth and the stochastic gradients have variance bounded by σ^2. Let the learning rate be set as $\eta = \frac{c}{\sqrt{T}}$. Then, for T iterations, the average squared gradient norm of Real satisfies:*

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2] \leq \frac{C_1}{\sqrt{T}} + \frac{C_2 \cdot \sigma^2}{\sqrt{T} \cdot NK}, \tag{17}$$

where C_1 and C_2 are constants depending on L and the initial optimality gap.

Theorem 7.1 demonstrates that Real converges to a first-order stationary point at a rate of $O(1/\sqrt{T})$. Crucially, the second term reveals that the error floor induced by stochastic noise is suppressed by a factor of NK, theoretically justifying the scalability of our approach. We also provide empirical accuracy and loss comparisons in Appendix E, demonstrating that Real matches ESDG in both convergence behavior and final model quality.



(a) Parallel efficiency

(b) Imbalance Ratio

Fig. 14. *Simulated parallel efficiency and imbalance ratio on clusters with 4 to 1024 GPUs.*

7.5 Scalability

Due to hardware limitations, we extend the evaluation to larger clusters through simulation. This simulation approach is justified by two key factors intrinsic to our problem setting:

(1) **Workload Independence:** The training time of a specific snapshot group on a single GPU depends solely on the GPU's compute capability and the group's intrinsic characteristics (e.g., topology and feature size). It remains invariant regardless of the cluster scale, allowing us to use profiled real-world execution times as reliable ground truth.
(2) **Minimal Communication Overhead:** For the methods compared (PSG, Reuse-only, and REAL-G), the unique cross-GPU data transfer involved is the gradient AllReduce. Since DGNN models are typically compact, this overhead is minimal—accounting for less than 0.1% of the total training time in our measurements. Nevertheless, to ensure rigor, we explicitly model this overhead (α in Equation 10) following the communication modeling methodology from Libra [27].

To evaluate performance at larger scales, we conduct simulations using 10,000 snapshots generated by DyGraph [30]. In this scenario, solving with ILP-based methods is time-consuming, so we evaluate REAL-G, which can be solved quickly even at large scales, against baselines that only incorporate data reuse without load balancing and the naive PSG method. The scheduling strategies from different methods are applied to Equation 10 for time simulation; detailed simulation settings are provided in the Appendix F.

Figure 14 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 14 (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 14 (right) further demonstrates that this scalability improvement correlates with lower imbalance ratios, where REAL-G consistently outperforms baselines.

8 RELATED WORK

8.1 Distributed GNN Training System

Distributed training systems for static graphs have laid the foundation for scalability by optimizing communication and resource utilization. Systems like P3 [12] and NeutronStar [51] minimize transfer overheads through pipelined push-pull strategies and hybrid dependency management, respectively, while Sancus [35] employs decentralized staleness-aware communication to bypass synchronization barriers. Addressing sampling bottlenecks, GNNLab [57] utilizes pre-sampling caching to optimize GPU memory, and DSP [2] dynamically adapts sampling policies for load balancing. At the kernel level, MGG [54] enables fine-grained intra-kernel pipelining to overlap sparse-dense operations. Most recently, research has shifted towards memory hierarchy and feature management: XGNN [44] abstracts unified global memory spaces for billion-scale datasets, F^2CGT [28] leverages two-level feature compression to alleviate processing bottlenecks, and LeapGNN [6] proposes a feature-centric paradigm that migrates lightweight models to data locations to minimize remote retrieval overhead.

8.2 Single-GPU DGNN Training Acceleration

DGNN training acceleration within a single GPU focuses on maximizing memory bandwidth efficiency and eliminating redundant computations. PiPAD [48] introduces a holistic pipeline that reconstructs the training paradigm to overlap graph loading, transfer, and computation, effectively hiding memory access latency. Targeting memory bottlenecks, SEIGN [37] proposes a scalable framework that utilizes parameter-free message passing to decouple spatial aggregation from temporal evolution, enabling efficient mini-batch training without expensive neighborhood sampling. ETC [13] further optimizes data access by enforcing temporal locality through a coherent execution plan, grouping contiguous snapshots to minimize cache misses. Additionally, WinGNN [65] reduces model complexity by eliminating temporal encoders, instead employing a random gradient aggregation window to efficiently capture temporal dependencies.

8.3 CTDG Training Optimization

While REAL targets DTDGs, significant progress continues in optimizing CTDGs. TGL [61] establishes a baseline for billion-scale CTDG learning with specialized temporal data structures and parallel sampling. Addressing data movement bottlenecks, SIMPLE [14] optimizes CPU-GPU transfers through a dynamic data placement strategy that adapts to temporal access patterns. For low-latency streaming, D3-GNN [20] employs a distributed dataflow architecture to efficiently manage cascading updates. On the modeling front, CTAN [17] leverages ODEs to capture long-range dependencies, while SIG [8] integrates causal inference for interpretable link prediction.

9 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×–4.53× and 1.10×–3.99× speedup over SOTA baseline, respectively.

References

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

[2] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN Training with Multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 392–404. https://doi.org/10.1145/3572848.3577528

[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] Weijian Chen, Shuibing He, Haoyang Qu, and Xuechen Zhang. 2025. LeapGNN: Accelerating Distributed GNN Training Leveraging Feature-Centric Model Migration. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 255–270. https://www.usenix.org/conference/fast25/presentation/chen-weijian-leap

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[8] Lanting Fang, Yulian Yang, Kai Wang, Shanshan Feng, Kaiyu Feng, Jie Gui, Shuliang Wang, and Yew-Soon Ong. 2024. SIG: Efficient Self-Interpretable Graph Neural Network for Continuous-time Dynamic Graphs. arXiv:2405.19062 [cs.LG] https://arxiv.org/abs/2405.19062

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

[10] Matthias Fey, Jinu Sunil, Akihiro Nitta, Rishi Puri, Manan Shah, Blaž Stojanovič, Ramona Bendias, Alexandria Barghi, Vid Kocijan, Zecheng Zhang, Xinwei He, Jan Eric Lenssen, and Jure Leskovec. 2025. PyG 2.0: Scalable Learning on Real World Graphs. arXiv:2507.16991 [cs.LG] https://arxiv.org/abs/2507.16991

[11] 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

[12] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[13] 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.

[14] 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.

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

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

[17] Alessio Gravina, Giulio Lovisotto, Claudio Gallicchio, Davide Bacciu, and Claas Grohnfeldt. 2024. Long Range Propagation on Continuous-Time Dynamic Graphs. arXiv:2406.02740 [cs.LG] https://arxiv.org/abs/2406.02740

[18] 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.

[19] Mingyu Guan, Saumia Singhal, Taesoo Kim, and Anand Padmanabha Iyer. 2025. ReInc: Scaling Training of Dynamic Graph Neural Networks. arXiv:2501.15348 [cs.LG] https://arxiv.org/abs/2501.15348

[20] Rustam Guliyev, Aparajita Haldar, and Hakan Ferhatosmanoglu. 2024. D3-GNN: Dynamic Distributed Dataflow for Streaming Graph Neural Networks. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 2764–2777. https:

//doi.org/10.14778/3681954.3681961
[21] Gurobi Optimization, LLC. 2022. Gurobi - The Fastest Solver. https://www.gurobi.com Accessed: 2022-01-01.
[22] S Hochreiter. 1997. Long Short-term Memory. *Neural Computation MIT-Press* (1997).
[23] 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.
[24] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
[25] 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.
[26] 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.
[27] Yunzhuo Liu, Bo Jiang, Shizhen Zhao, Tao Lin, Xinbing Wang, and Chenghu Zhou. 2023. Libra: Contention-Aware GPU Thread Allocation for Data Parallel Training in High Speed Networks. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. 1–10. https://doi.org/10.1109/INFOCOM53939.2023.10228922
[28] Yuxin Ma, Ping Gong, Tianming Wu, Jiawei Yi, Chengru Yang, Cheng Li, Qirong Peng, Guiming Xie, Yongcheng Bao, Haifeng Liu, and Yinlong Xu. 2024. Eliminating Data Processing Bottlenecks in GNN Training over Large Graphs via Two-level Feature Compression. *Proc. VLDB Endow.* 17, 11 (July 2024), 2854–2866. https://doi.org/10.14778/3681954.3681968
[29] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (2020), 107000.
[30] 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
[31] NVIDIA. 2024. Deep Graph Library (DGL) Container. https://catalog.ngc.nvidia.com/orgs/nvidia/containers/dgl. Accessed: 2025-08-20.
[32] 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
[33] 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.
[34] 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.
[35] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.* 15, 9 (May 2022), 1937–1950. https://doi.org/10.14778/3538598.3538614
[36] 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.
[37] 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)*. 2850–2863. https://doi.org/10.1109/ICDE55515.2023.00218
[38] Reddit. n.d.. Reddit Dataset. https://www.reddit.com/.
[39] 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.
[40] 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.
[41] 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.

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

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

[44] Dahai Tang, Jiali Wang, Rong Chen, Lei Wang, Wenyuan Yu, Jingren Zhou, and Kenli Li. 2024. XGNN: Boosting Multi-GPU GNN Training via Global GNN Memory Store. *Proc. VLDB Endow.* 17, 5 (Jan. 2024), 1105–1118. https://doi.org/10.14778/3641204.3641219

[45] 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*.

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

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

[48] 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.

[49] 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.

[50] 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*.

[51] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1301–1315. https://doi.org/10.1145/3514221.3526134

[52] 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

[53] 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)*.

[54] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 779–795. https://www.usenix.org/conference/osdi23/presentation/wang-yuke

[55] 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.

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

[57] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 417–434. https://doi.org/10.1145/3492321.3519557

[58] 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.

[59] 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).

[60] 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

[61] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: a general framework for temporal GNN training on billion-scale graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1572–1580.

[62] 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.

[63] 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.

[64] 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

[65] Yifan Zhu, Fangpeng Cong, Dan Zhang, Wenwen Gong, Qika Lin, Wenzheng Feng, Yuxiao Dong, and Jie Tang. 2023. WinGNN: Dynamic Graph Neural Networks with Random Gradient Aggregation Window. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 3650–3662. https://doi.org/10.1145/3580305.3599551

Supplementary Materials

A Performance Modeling for IncreConv

Figure 15 profiles the runtime of the GraphConv operator across varying graph scales (100k to 5M nodes) and average degrees. The x-axis represents the average node degree on a logarithmic scale, while the y-axis reports execution time (ms). A key observation is that GraphConv exhibits a characteristic **U-shaped latency curve**: runtime initially decreases as the graph transitions from extremely sparse to moderately connected, but subsequently rises in high-degree regimes due to increased aggregation overheads. We capture this non-monotonic behavior using the following regression model:

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

where N and d denote the node count and average degree, respectively.

This fitted model serves as the oracle for our adaptive execution strategy. At runtime, we evaluate the predicted latency for two potential execution paths: (i) applying GraphConv on the full baseline graph $\mathcal{G}$, and (ii) applying GraphConv on the differential dmap. The system then dynamically dispatches the computation to the target that yields the minimized predicted cost, ensuring optimal efficiency across varying density regimes.

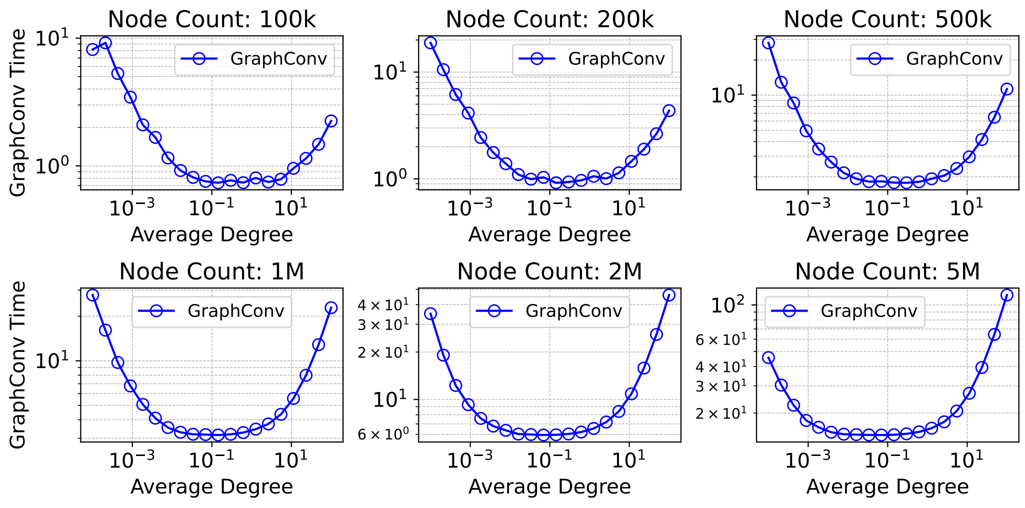
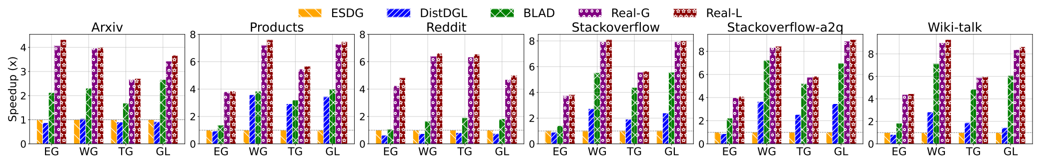


Fig. 15. GraphConv *runtime exhibits a U-shaped trend with respect to node degree.*

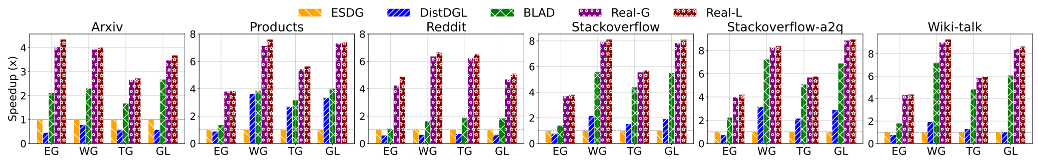
B Supplementary Evaluation

B.1 Experimental results

B.1.1 Overall Performance. We first compare the training time speedups across all methods. As shown in Figure 16a, 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



(a) Single-machine setup (testbed A)



(b) Multi-machine setup (testbed B)

Fig. 16. **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.*

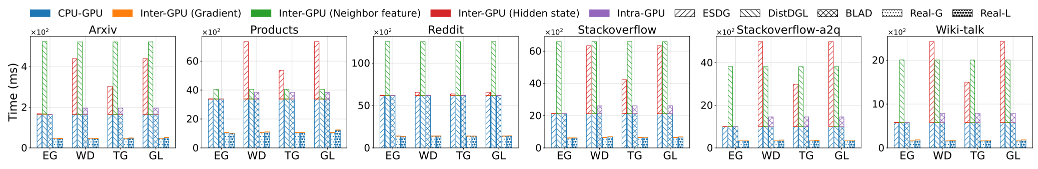


Fig. 17. **Breakdown of data transfer time per epoch on the testbed A.** *Each bar is decomposed into CPU-GPU transfer, inter-GPU transfer (gradients, neighbors, hidden states), and intra-GPU transfer.*

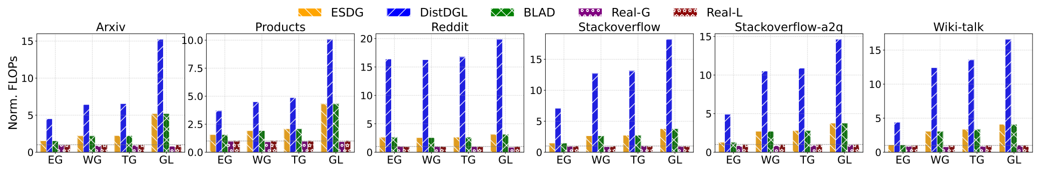
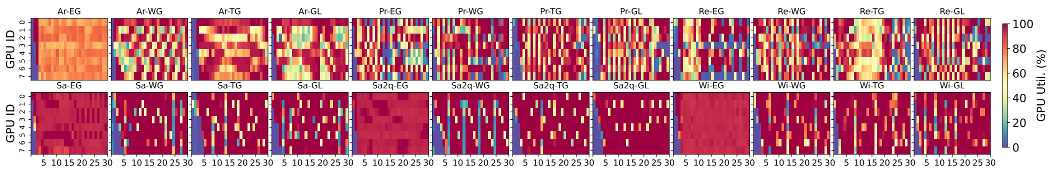


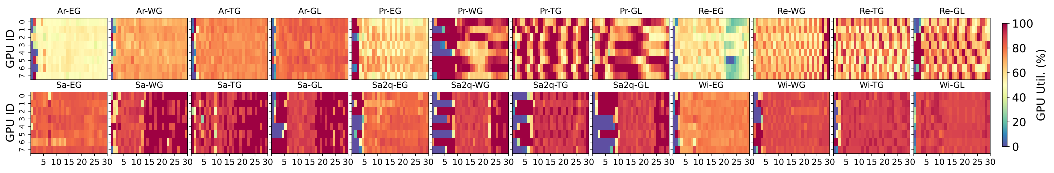
Fig. 18. **Normalized `GraphConv` FLOPs per epoch on the testbed A.** *FLOPs is normalized to* Real*-L.*

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.

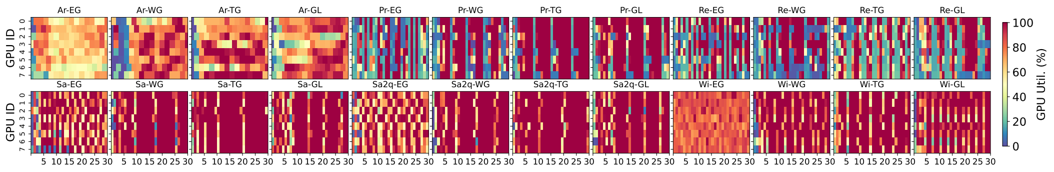
We further report results on testbed B in Figure 16b, where the speedups of Real remain comparable to testbed A for all baselines except DistDGL. DistDGL is more adversely affected in the multi-machine setting, as its vertex-level partitioning requires cross-machine neighbor aggregation, incurring heavy 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



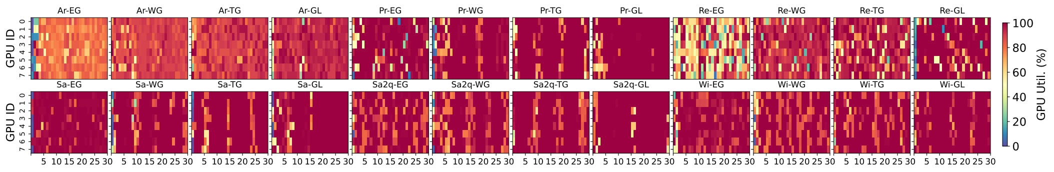
(a) ESDG



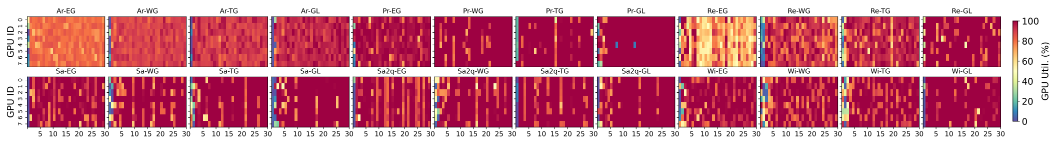
(b) DistDGL



(c) BLAD



(d) REAL-G



(e) REAL-L

Fig. 19. **GPU utilization heatmaps across different dynamic GNN models and datasets.** We compare (a) ESDG, (b) DistDGL, (c) BLAD, (d) REAL-G, and (e) REAL-L. Labels **Ar**, **Pr**, **Re**, **Sa**, **Sa2q**, and **Wi** denote the datasets *Arxiv*, *Products*, *Reddit*, *Stackoverflow*, *Stackoverflow-a2q*, and *Wiki-talk*, respectively.

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.

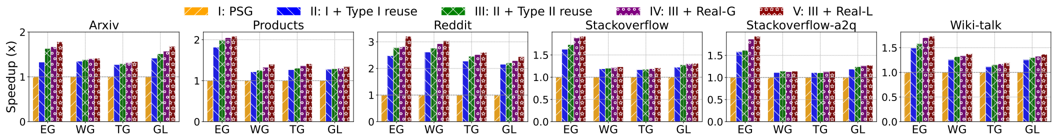


Fig. 20. **Speedup ablation across different models and datasets.** *Speedup is normalized to PSG.*

B.1.2 Transfer time breakdown. Figure 17 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.

B.1.3 `GraphConv` *FLOPs.* We profile the total `GraphConv` computation within one training epoch. As shown in Figure 18, 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.

B.1.4 GPU Utilization and Load Balance. We profile the utilization of all GPUs during the first 30 seconds of training. We present the GPU utilization heatmaps Real and all baselines in Figure **??**. 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.

B.2 Ablation Study

To evaluate the individual contribution of each component in Real, we perform a stepwise ablation study normalized to the *Partition-by-Snapshot-Group* (PSG) baseline, where each GPU sequentially trains one complete snapshot group per iteration without reuse or load balancing. Figure 20 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

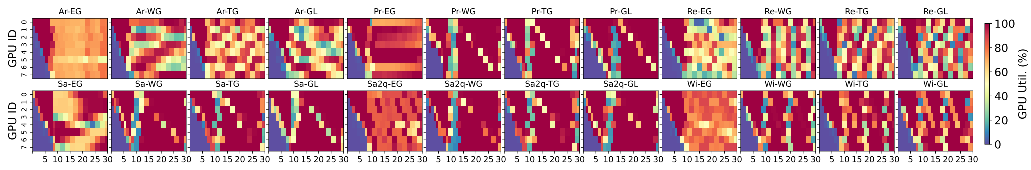


Fig. 21. *GPU utilization heatmaps of ESDG across different dynamic GNN models and datasets.*

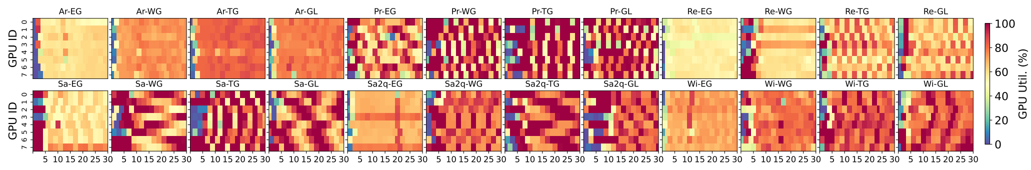


Fig. 22. *GPU utilization heatmaps of DistDGL across different dynamic GNN models and datasets.*

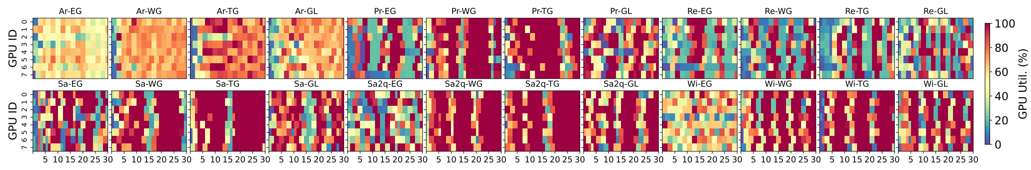


Fig. 23. *GPU utilization heatmaps of BLAD across different dynamic GNN models and datasets.*

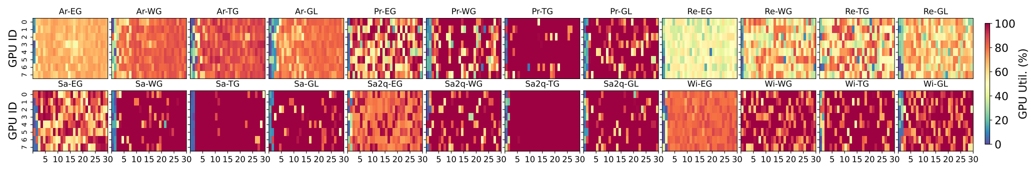


Fig. 24. *GPU utilization heatmaps of REAL-G across different dynamic GNN models and datasets.*

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.

C GPU Utilization of Other Methods

We also visualize the per-GPU utilization heatmaps for the remaining four baselines in Figures 21–24, measured during the first 30 seconds of training. For **ESDG** (Figure 21), 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 22) 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 23) 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 24) combines load balancing with triple-stream overlap on each rank, reducing inter-rank imbalance and improving overall GPU utilization.

D Convergence Analysis

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

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

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

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

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

D.2 Algorithm Description

Definition D.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:

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

D.3 Detailed Steps

Lemma D.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 D.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:

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

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 D.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 D.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 D.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:

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

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

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

Then:

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

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

$$\begin{aligned}\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla L(\boldsymbol{x}_t)\|^2] &\leq \frac{2\sqrt{T}(L(\boldsymbol{x}_1) - L(\boldsymbol{x}^*))}{Tc(2 - \frac{Lc}{\sqrt{T}})} + \frac{L\sigma^2 c}{2NK\sqrt{T}}\\ &= \frac{2(L(\boldsymbol{x}_1) - L(\boldsymbol{x}^*))}{c(2\sqrt{T} - Lc)} + \frac{L\sigma^2 c}{2NK\sqrt{T}} = O\left(\frac{1}{\sqrt{T}}\right).\end{aligned}$$

□

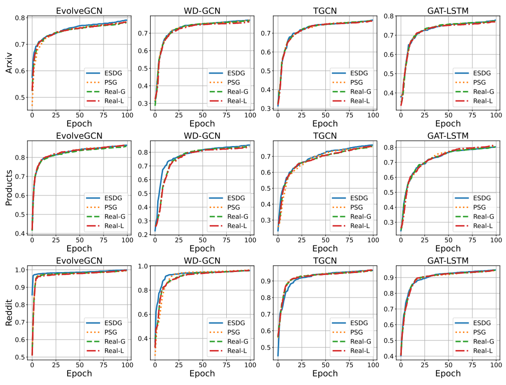


Fig. 25. *Model accuracy.*

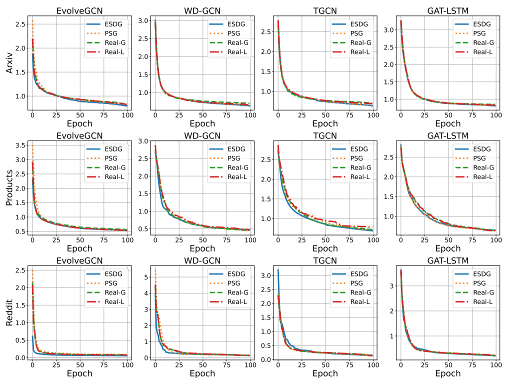


Fig. 26. *Training loss.*

E 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 25 and 26. The Stackoverflow, Stackoverflow-a2q and wiki dataset, lacking labels, is not included in the accuracy comparison.

From Figure 25, 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 26 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.

F 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, 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 real 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:

$$t_{snapshot_i} = \alpha_1 \cdot N_{node_i} + \alpha_2 \cdot N_{edge_i} + \alpha_3 \tag{18}$$

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:

$$\mathcal{R}_{i,j} = \sum_{k=i+1}^{max(i,i+tw-(j-i))} t_{snapshot_k} \cdot RATIO \tag{19}$$

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

$$t_{combined} = t_{group_i} + t_{group_j} - \mathcal{R}_{i,j} \tag{20}$$

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 27. This model was then used to simulate the execution time of each snapshot in our evaluation.

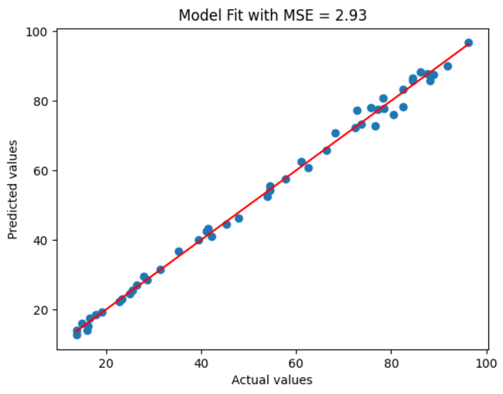


Fig. 27. *Comparison between predicted and measured snapshot training time.*