# FROST: Accelerating Temporal Graph Neural Network Training via a Unified Multi-GPU Framework

Technical Report

LEI ZHAO, Texas State University, USA
JOSEPH NKE, Texas State University, USA
JONGRYOOL KIM, SK hynix America, USA
CHUL-HO LEE, Texas State University, USA

Temporal graph neural networks (TGNNs) are the dominant approach for dynamic link prediction. Memory-based TGNNs (e.g., TGN and JODIE) compress interaction history into per-node states, while memoryless long-context models (e.g., CAWN, DyGFormer, and GraphMixer) aggregate temporal context on-the-fly for higher accuracy. As real-world temporal graphs can contain hundreds of millions of interactions, multi-GPU training has become essential. However, scaling these models to multiple GPUs remains an open challenge. While memory-based models suffer from severe synchronization stalls, multi-GPU training frameworks for recent popular memoryless models are currently unavailable. We present FROST, a unified multi-GPU training framework for TGNNs. For memory-based models, FROST employs a forward-looking planner and three-tier routing to direct updated dynamic states—node memories and messages—via the fastest available path across the CPU and GPUs, including direct inter-GPU transfers. This preserves exact per-macro-batch freshness while substantially reducing GPU idle time. For memoryless models, FROST provides the first multi-GPU training solution by integrating them into the same staged host-worker pipeline. Extensive experimental results using eight TGNN models and six benchmark datasets demonstrate FROST's superior performance in both accuracy and throughput.

CCS Concepts: • **Computer systems organization → Parallel architectures**; • **Computing methodologies → Neural networks**; *Parallel algorithms.*

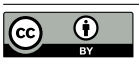
Additional Key Words and Phrases: Temporal Graph Neural Networks, Multi-GPU Training, Dynamic State Management, Pipeline Scheduling, Inter-GPU Communication

**ACM Reference Format:**
Lei Zhao, Joseph Nke, Jongryool Kim, and Chul-Ho Lee. 2026. FROST: Accelerating Temporal Graph Neural Network Training via a Unified Multi-GPU Framework: Technical Report. *Proc. ACM Manag. Data* 4, 6 (SIGMOD), Article 350 (December 2026), 40 pages. https://doi.org/10.1145/3856345

## 1 Introduction

Temporal (dynamic) graphs, which evolve over time in their topology and associated attributes, appear in various domains ranging from social networks and e-commerce to knowledge graphs [3, 9] and financial transactions [15, 24, 32]. The central learning task in temporal graphs is *temporal link prediction* [1]. Given the history of interactions between nodes up to time $t$, the goal is to predict which pairs of nodes will interact next. Temporal graph neural networks (TGNNs) have been proposed and proven highly successful for this task [22, 29].

Authors' Contact Information: Lei Zhao, Texas State University, San Marcos, Texas, USA, sqp17@txstate.edu; Joseph Nke, Texas State University, San Marcos, Texas, USA, pmh72@txstate.edu; Jongryool Kim, SK hynix America, San Jose, California, USA, jongryool.kim@sk.com; Chul-Ho Lee, Texas State University, San Marcos, Texas, USA, chulho.lee@txstate.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/12-ART350
https://doi.org/10.1145/3856345

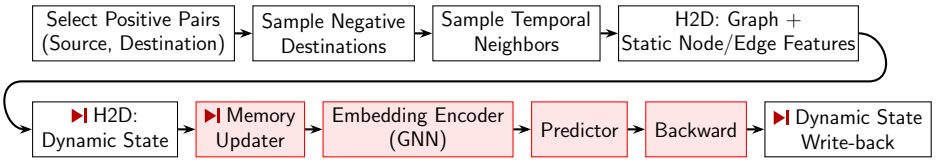



Fig. 1. Standard TGNN training workflow.

Existing TGNNs fall into two classes. The first class comprises *memory-based* models, such as TGN [22], JODIE [16], and APAN [27]. They maintain per-node vectors, each of which accumulates interaction history via a recurrent update rule. This design compresses temporal history into a compact per-node state and often requires only shallow, or even zero-hop, neighbor expansion, making memory-based models attractive for high-throughput training and inference. The second class consists of *memoryless long-context* models, such as TGAT [29], CAWN [28], GraphMixer [5], and DyGFormer [30]. They forgo per-node states and instead aggregate temporal context directly from sampled walks or sequences of interacting neighbors. These models often achieve higher accuracy, but this comes at a higher computational cost due to heavier temporal-context expansion, walk sampling, or sequence processing.

Despite advances in TGNN architectures, training these models on large graphs remains a computationally demanding task, as real-world temporal graphs can contain hundreds of millions of interactions [2, 10, 17, 31]. The sheer graph size makes single-GPU training prohibitively slow and motivates the use of multi-GPU parallelism. However, scaling the TGNNs to multiple GPUs is non-trivial for several reasons.

*First, general-purpose frameworks for scalable or distributed GNN training, such as Graph-Learn, DistDGL and PyG, do not directly resolve this problem [6, 7, 12, 26, 33, 36].* Their data loaders are designed under the assumption that mini-batches can be sampled and dispatched largely independently. In TGNN training, however, batches must respect chronological order and temporal cutoff constraints, and the sampler itself must retrieve *historical* neighbors or interactions that occur before each query timestamp rather than unrestricted neighborhoods as in ordinary static-graph GNN sampling. Moreover, memory-based TGNNs maintain persistent dynamic states that must be updated, stored, and made visible across batches. As a result, these static-graph data-loading pipelines are not directly applicable without a substantial redesign.

*Second, there exists a memory synchronization bottleneck for memory-based models.* The key to leveraging multiple GPUs for higher throughput is *pipelining*, which involves overlapping sampling, data transfer, and computation so that every GPU stays busy. However, pipelining collides with a *strict temporal ordering constraint* inherent to these models. Each batch's node memory must reflect all updates produced by the preceding batch. In a multi-GPU setting, enforcing this freshness means that updated states must be written back and redistributed across workers before the next batch can proceed, creating synchronization stalls that leave GPUs idle.

Fig. 1 illustrates the standard TGNN training workflow.[1] For memory-based models, the host samples temporal subgraphs and transfers them to the GPU, along with associated features and dynamic states (each consisting of node memory and messages). The forward and backward passes then follow to update the states and model weights. The iteration concludes by writing the updated states back to CPU DRAM. Under this workflow, strict multi-GPU training requires writing updated node states back to CPU DRAM and reloading them before processing the next batch, which creates significant GPU idle bubbles, as shown in Fig. 2a. We empirically observe that this serialized round-trip path (read and write) accounts for up to 83.5% of wall-clock training time.

[1] Shaded stages represent active GPU computation. Stages marked with ▶| do not exist in memoryless models.

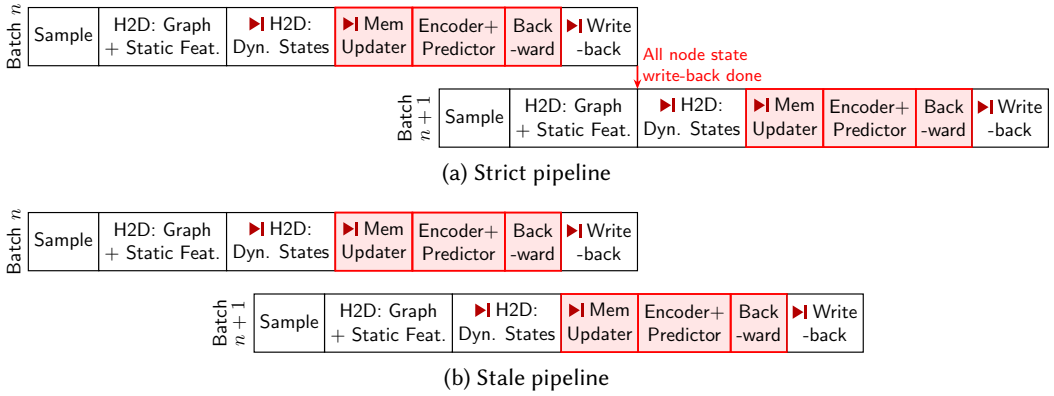



Fig. 2. Conventional TGNN pipelines.

Most existing multi-GPU systems sacrifice the exact freshness of dynamic states to reduce GPU idle time (see Fig. 2b). This trade-off occurs because maintaining strict per-macro-batch freshness imposes the steep CPU-GPU synchronization penalties described above. For example, TGL [34] mitigates GPU idle time by reading dynamic states with a staleness of roughly one macro-batch. MSPipe [23] fully pipelines execution to maximize GPU utilization, but at the cost of increased staleness that can span several macro-batches under long write-back latencies. However, these stale reads degrade prediction accuracy, as demonstrated in [23] and discussed in Sections 3.1 and 5.

*Third, there is currently no multi-GPU support for recent memoryless models.* Popular architectures such as CAWN, DyGFormer, and GraphMixer face a different but equally pressing problem. Existing implementations (e.g., DyGLib [30]) are restricted to a single GPU with no pipelining—sampling, data loading, and computation run sequentially, leaving the GPU idle during preprocessing. As temporal graphs grow, this lack of both parallelism and pipeline overlap makes training prohibitively slow and restricts the practical adoption of these otherwise strong models.

To address these limitations, we present **FROST**[2] (**F**orward-looking **R**outing and **O**rchestrated **S**ynchronization for **T**GNNs), a multi-GPU TGNN training system that substantially reduces GPU idle time across both model classes, while preserving exact per-macro-batch freshness for memory-based architectures. For these models, FROST tackles the accuracy-throughput trade-off by (i) determining *where* each updated dynamic state will be consumed next, and (ii) routing it there via the fastest available path before the consumer's next macro-batch begins. Specifically, FROST introduces a centralized forward-looking planner, running on a dedicated host process, and employs a three-tier routing mechanism. For every updated node, the planner determines any local reuse or direct inter-GPU transfer needed by the next macro-batch and, independently, whether to write the state back asynchronously to CPU shared memory. In addition, we design a staged host-worker pipeline that accommodates both memory-based and recent memoryless models, alongside optimizations for both inter-GPU and host-worker communication. Finally, FROST adopts a highly modular, component-based design. The current implementation natively supports eight TGNN models spanning both model classes, together with multiple temporal and negative-sampling mechanisms, via a configuration-driven interface without sacrificing distributed performance.

Our contributions can be summarized as follows:

[2]Source code is available at https://github.com/dttutty/frost.

Table 1. Notation summary.

| Symbol | Description |
|---|---|
| $\mathcal{E}, \mathcal{V}$ | Temporal edge sequence, node set |
| $d_e$, $d_n$, $d_{\text{mem}}$ | Dimensions for edge, node, and memory |
| $d_{\text{msg}}$, $L_{\text{box}}$ | Message dimension and APAN mailbox length |
| $n_{\text{neg}}$ | Negative destinations per positive edge |
| $B$ | Per-GPU mini-batch size |
| $\hat{B}$ $(= n_{\text{GPU}}B)$ | Macro-batch size |
| $n_{\text{GPU}}$ | Number of worker GPUs |
| $b$ | Macro-batch index |
| $\mathbf{s}_v(t)$ | Dynamic per-node memory vector |
| $\mathbf{m}_v(t)$ | Per-interaction message vector |
| $\sigma(v, b)$ | Read-time staleness |

- We formally characterize the importance of memory freshness and the memory synchronization bottleneck in multi-GPU memory-based TGNN training, which is the dominant contributor to pipeline stalls.
- We design FROST with a staged pipeline that substantially reduces GPU idle time, alongside a forward-looking planner and three-tier dynamic-state routing that preserve exact memory freshness for memory-based models.
- We realize FROST with two graph representations, communication optimizations, and a component-based design that supports eight TGNN models—including, to our knowledge, the first multi-GPU training support for recent popular memoryless models such as CAWN, TCL, GraphMixer, and DyGFormer.
- We present extensive experimental results demonstrating that FROST achieves superior throughput and accuracy compared to TGL for memory-based models (up to 12.61× and 4.31% in average precision, respectively), and significantly higher throughput compared to DyGLib for memoryless models (up to 26.39×). It also outperforms PipeTGL and scales efficiently to large graphs.

## 2 Background and Related Work

### 2.1 Temporal Graph Learning

A temporal graph is represented as a sequence of timestamped events $\mathcal{E} = \{(u_i, v_i, t_i, \mathbf{e}_i)\}_{i=1}^{M}$, where $u_i, v_i \in \mathcal{V}$ are source and destination nodes, $t_i \in \mathbb{R}^+$ is the interaction timestamp, and $\mathbf{e}_i \in \mathbb{R}^{d_e}$ is an edge feature. Nodes may also be associated with static features $\mathbf{x}_v \in \mathbb{R}^{d_n}$. Given a temporal graph with interactions up to $t^-$, the *temporal link prediction* task is to predict whether an edge $(u, v, t)$ will occur, which serves as the primary benchmark focus in this work.

TGNN training employs *contrastive batching*. Each positive edge $(u_i, v_i, t_i)$ is paired with $n_{\text{neg}}$ randomly sampled negative destination nodes. A mini-batch of size $B$ therefore contains $B(2 + n_{\text{neg}})$ root-node occurrences before deduplication. In multi-GPU training, $n_{\text{GPU}}$ consecutive mini-batches—one per worker GPU—are grouped into a *macro-batch* of size $\hat{B} = n_{\text{GPU}}B$. Table 1 summarizes the main notation used throughout the paper.

### 2.2 Memory-based TGNNs

Memory-based TGNNs maintain a dynamic per-node memory $\mathbf{s}_v(t) \in \mathbb{R}^{d_{\text{mem}}}$, which is subsequently utilized by a graph encoder to generate node embeddings. For each interaction $(u, v, t)$, they update

the memories of $u$ and $v$. Specifically, for node $u$, they construct a message $\mathbf{m}_u(t^-) \in \mathbb{R}^{d_{\text{msg}}}$ by concatenating its previous memory $\mathbf{s}_u(t^-)$, $v$'s previous memory $\mathbf{s}_v(t^-)$, and the edge feature of dimension $d_e$, resulting in a message dimension of $d_{\text{msg}} = 2d_{\text{mem}} + d_e$. The memory of node $u$ is then updated as

$$\mathbf{s}_u(t) = \text{updater}(\mathbf{s}_u(t^-), \mathbf{m}_u(t^-)) . \tag{1}$$

The memory for node $v$, $\mathbf{s}_v(t)$, is updated symmetrically.

Models in this class employ various update mechanisms, such as standard RNNs (JODIE [16]), GRUs (TGN [22]), or Transformer attention (APAN [27]). However, they all share a core recurrent structure that creates a strict sequential dependency across batches. This design inherently enforces strict temporal causality. Eq. (1) requires that the memory visible to the current batch incorporates all updates committed by prior batches. This dependency cannot be relaxed without compromising memory freshness and, consequently, altering the model's semantics.

### 2.3 Memoryless TGNNs

A second family of TGNNs forgoes persistent per-node states entirely. These *memoryless* encoder models aggregate temporal context dynamically from sampled neighborhoods or walks at each prediction step. For example, TGAT [29] applies a temporal graph attention mechanism over sampled neighbors, encoding time gaps with a learnable time kernel. CAWN [28] samples causal anonymous walks from the temporal graph and aggregates walk-level position encodings to capture structural and temporal motifs. TCL [25] encodes sampled temporal neighbors with a Transformer and learns representations through a temporal contrastive objective. GraphMixer [5] replaces attention with an MLP-Mixer architecture, in which a link encoder summarizes recent neighbor timestamps and a node encoder aggregates neighbor features. DyGFormer [30] extracts co-occurrence neighbor sequences for source and destination nodes and processes them with a patched Transformer, capturing long-range temporal patterns.

Because these models do not maintain dynamic states across batches, they are free from the temporal causality constraint of Eq. (1). However, they trade per-node memory for heavier per-prediction computation, which includes walk sampling, multi-hop neighbor expansion, or long-sequence processing, making single-GPU training slow on large graphs. Despite this scalability need, existing implementations (e.g., DyGLib [30]) are restricted to a single GPU without pipelining. Sampling, data loading, and computation run sequentially, leaving both multi-GPU parallelism and pipeline overlap entirely unsupported for these widely used models.

### 2.4 Multi-GPU TGNN Training

Prior work on static-graph GNN training has studied multi-GPU and distributed parallelism [4, 8, 13, 14, 18, 33]. Memoryless temporal GNNs are amenable to similar techniques, yet no multi-GPU solutions currently exist for them. Because they carry no dynamic states across mini-batches, data preprocessing and transfer can be readily overlapped with computation. Memory-based TGNNs, however, introduce strict inter-batch dependencies. Specifically, processing batch $b$ requires the committed memory state from batch $b-1$. This dependency prevents standard overlap techniques from fully hiding synchronization overheads.

To reason precisely about the freshness guarantees of different TGNN systems, particularly in multi-GPU settings, we introduce *read-time staleness*. Let $\ell^\star(v, b)$ denote the macro-batch index of the latest update to node $v$ that *should* be visible when macro-batch $b$ reads $v$ in a sequential exact-memory execution, and let $\ell(v, b)$ denote the macro-batch index of the latest update that is *actually* visible in a given system. We define the read-time staleness as

$$\sigma(v, b) = \ell^\star(v, b) - \ell(v, b) \geq 0. \tag{2}$$

Table 2. Key properties of existing TGNN systems.

| Paper | System | Pipeline? | Multi-GPU? | Staleness $\sigma$ |
|---|---|---|---|---|
| [34] | TGL | ✓ | ✓ | $\approx 1$ |
| [35] | DistTGL | ✓ | multi-server | $\approx 1$ |
| [23] | MSPipe | ✓ | ✓ | dynamic |
| [19] | PipeTGL | ✓ | ✓ | $0^{\dagger}$ |
| [30] | DyGLib | ✗ | ✗ | 0 |
| | **FROST** | ✓ | ✓ | **0** |

$^{\dagger}$Permits model-parameter staleness.

Table 3. Read-time staleness example at macro-batch 5.

| | Notation | FROST | TGL |
|---|---|---|---|
| Should be visible | $\ell^{\star}(v,5)$ | $b-1=4$ | $b-1=4$ |
| Visible in the system | $\ell(v,5)$ | $b-1=4$ | $b-2=3$ |
| Staleness | $\sigma=\ell^{\star}-\ell$ | 0 | 1 |

A value of $\sigma(v,b)=0$ indicates that the freshest correct state is read, while $\sigma(v,b)>0$ indicates that the read lags behind by that many macro-batch updates.

For example, suppose node $v$ is updated in every macro-batch and is read in macro-batch $b=5$, as shown in Table 3. Sequential exact-memory execution requires the update from macro-batch 4, so $\ell^{\star}(v,5)=b-1=4$ for both systems. FROST exposes the same update as sequential execution, so $\ell(v,5)=4$ for FROST. However, TGL may see only the update from macro-batch 3 while macro-batch 4 remains in flight, so $\ell(v,5)=3$ for TGL. Therefore, read-time staleness under FROST is $\sigma=0$, whereas it is $\sigma=1$ for TGL.

This notion naturally extends to message states by comparing the latest message write that *should* be visible against what is *actually* available. Note that for APAN, the relevant version is the latest mailbox insertion.

Table 2 compares existing TGNN systems across pipeline parallelism, multi-GPU execution, and read-time staleness ($\sigma$). TGL [34] adopts a host-worker framework in which the host process performs sampling and GPU worker processes perform training, relying on shared memory to maintain dynamic states. In its default multi-GPU setting, TGL reads stale memory and message states that lag one macro-batch behind ($\sigma\approx 1$), trading strict correctness for throughput. DistTGL [35] targets *multi-server* distributed TGNN training. It decomposes the per-node state into a *dynamic memory* component and a *static embedding* component built from structural graph features. The static part compensates for the information loss introduced by batched, asynchronous updates. Like TGL, DistTGL embraces staleness as a design choice and thus does not provide exact-memory guarantees.

More recently, MSPipe [23] introduces a stall-free pipelining framework that allows each training iteration to fetch node memories that are $k\geq 1$ iterations stale, where $k$ is determined at runtime to maximize throughput. However, *exact* memory freshness is fundamentally compromised. Every batch reads memories that lag behind the most recent updates, and in practice, this staleness ($\sigma$) can extend substantially beyond a single macro-batch. PipeTGL [19], which supports only TGN, explores a different trade-off. It preserves chronological memory continuity by enforcing inter-minibatch memory dependencies in a runtime DAG. However, to overlap multiple minibatches under its round-robin GPU schedule, PipeTGL permits *model-parameter staleness*, meaning a minibatch may

compute its loss and gradients using an earlier version of the model parameters. Thus, PipeTGL trades stale model parameters for fresh node memories, thereby altering its convergence behavior, as detailed in Section 5.10.

In contrast, for recent memoryless TGNNs, no analogous multi-GPU or pipelining efforts exist. The widely adopted reference implementation, DyGLib [30], unifies recent memoryless TGNNs (e.g., CAWN, DyGFormer, GraphMixer, and TCL) under a single training loop, but runs strictly on a single GPU with sampling, data loading, and computation executed sequentially. Thus, how to best leverage pipeline parallelism and enhance scalability for this increasingly popular class of TGNNs remains an open problem.

Prior multi-GPU TGNN systems accept some form of staleness to avoid synchronization stalls for memory-based models. TGL, DistTGL, and MSPipe permit stale dynamic states, whereas PipeTGL preserves memory continuity but permits stale model parameters. Moreover, none of these systems support recent memoryless models in a multi-GPU setting. FROST avoids both forms of staleness. It preserves exact memory read-time freshness ($\sigma = 0$) without introducing model-parameter staleness, by routing updated dynamic states to the next consumer via local reuse and direct inter-GPU transfers before the consumer macro-batch begins. It also serves as the first multi-GPU training solution for memoryless models such as CAWN, TCL, GraphMixer, and DyGFormer.

## 3 Technical Challenges

In this section, we discuss the technical challenges in supporting multi-GPU training for memory-based and memoryless TGNNs.

### 3.1 Why Fresh Memory Matters

Stale memory is not merely a semantic impurity; it introduces a quantifiable prediction distortion. To isolate the effect of memory staleness, we fix all other inputs to the predictor, including the peer-node states and any edge or node features, and consider only the dependence on node $v$'s memory. Let $\{t_i\}_{i\geq 1}$ denote the ordered interaction timestamps involving node $v$, where $t_1$ is the time of the first interaction, $t_2$ the second, and so forth. Let $\mathbf{s}_v(t_i)$ denote the fresh memory immediately after the update at time $t_i$, and let $\tilde{\mathbf{s}}_v(t_i) = \mathbf{s}_v(t_{i-\Delta})$ be a stale copy with lag $\Delta \geq 1$. If the resulting predictor $g$ is $L_g$-Lipschitz and the memory evolves according to Eq. (1), then we have

$$|g(\mathbf{s}_v(t_i)) - g(\tilde{\mathbf{s}}_v(t_i))| \leq L_g \sum_{j=i-\Delta+1}^{i} \|\mathbf{s}_v(t_j) - \mathbf{s}_v(t_{j-1})\|, \tag{3}$$

which follows from the triangle inequality.

Thus, the prediction distortion is bounded by the cumulative memory evolution omitted over the stale interval. This implies that data samples with small prediction margins are the most likely to have their predictions changed under staleness. Therefore, it is desirable to keep the node memories as fresh as possible. As demonstrated in Section 5, preserving freshness consistently outperforms or at least matches stale-memory baselines in predictive quality.

### 3.2 Memory Synchronization Bottleneck

It is expensive to preserve fresh memory in a multi-GPU setting. Each mini-batch needs to update the dynamic states of $2B$ root nodes (sources and destinations), assuming no node deduplication for simplicity. All updates must be globally visible before any GPU can begin the next mini-batch. In the strict pipeline, as shown in Fig. 2a, this synchronization is fully serialized. Each GPU first writes back its updated states to CPU shared memory and then fetches the states required for the next mini-batch.

Take TGN as an example. The write-back covers $2B$ root nodes, while the fetch loads the memories and messages of $(2 + n_{\text{neg}})B(1 + f_0)$ nodes, where $n_{\text{neg}}$ is the number of negative samples, and $f_0$ denotes the first-hop fanout. Let $BW_{\text{PCIe}}$ denote the effective host-to-device bandwidth, which is bounded by the slower of DRAM and PCIe. If we assume float32 storage, the per-step GPU idle time can be computed as $T_{\text{idle}} = T_{\text{wb}} + T_{\text{fetch}}$, where

$$T_{\text{wb}} = \frac{2B \cdot (d_{\text{mem}} + d_{\text{msg}}) \cdot 4}{BW_{\text{PCIe}}}, \text{ and}$$
$$T_{\text{fetch}} = \frac{(2 + n_{\text{neg}})B(1 + f_0) \cdot (d_{\text{mem}} + d_{\text{msg}}) \cdot 4}{BW_{\text{PCIe}}}.$$

We empirically observe that memory update and fetch together account for up to 74.5% of the end-to-end epoch time for TGN. For APAN, the synchronization cost is even higher. Its multi-slot mailbox ($L_{\text{box}}$ slots per node) replaces the single message vector, and each root's message is delivered to its sampled 1-hop neighbors, further increasing the transfer volume.

FROST addresses this synchronization bottleneck by judiciously combining local reuse, direct inter-GPU transfers, and asynchronous CPU write-backs. This maintains *exact memory* freshness (i.e., $\sigma(v, b) = 0$ for every node $v$ in every macro-batch $b$, as defined in Eq. (2)) while substantially reducing GPU idle time. Section 4 details the framework design of FROST.

### 3.3 Lack of Acceleration for Memoryless Models

Memoryless models sidestep the aforementioned synchronization bottleneck entirely, yet their training throughput remains limited by the framework in which they execute. DyGLib [30], the de facto reference implementation for recent memoryless models such as CAWN, TCL, GraphMixer, and DyGFormer, runs on a single GPU with no pipeline overlap. That is, neighbor sampling, feature loading, and model execution proceed sequentially, leaving the GPU *idle* during CPU-bound preprocessing. Furthermore, key operations such as temporal neighbor sampling and model-specific feature construction are implemented with unvectorized Python loops rather than batched native kernels. Beyond these computational bottlenecks, the entire training loop is bound to a single GPU using synchronous host-device data transfers, offering no way to scale with additional hardware.

FROST overcomes these limitations by replacing sequential execution with three key optimizations: (1) a C++/OpenMP neighbor sampler that processes the full batch in parallel with the Python global interpreter lock (GIL) released, (2) dense GPU-vectorized tensor operations for all encoder-specific preprocessing, and (3) a pipelined host-worker architecture that overlaps sampling with multi-GPU training, as detailed in Section 4.

## 4 Framework Design

In this section, we present the framework design of FROST. We begin with an overview of the host-worker architecture (§4.1) and the key observations that lay the foundation for FROST's pipeline design (§4.2). We then describe the forward-looking planner for memory-based models, which determines how updated dynamic states should be routed across the CPU and GPUs (§4.3). Next, we detail the three-tier routing mechanism that delivers these states via local reuse, direct inter-GPU transfers, or asynchronous CPU write-backs (§4.4). Following this, we explain the pipeline staging strategy (§4.5) and the optimizations for host-worker and inter-GPU communication (§4.6). Finally, we discuss what FROST supports, ranging from encoder architectures to negative sampling mechanisms (§4.7). Additional implementation details are provided in Appendix B.

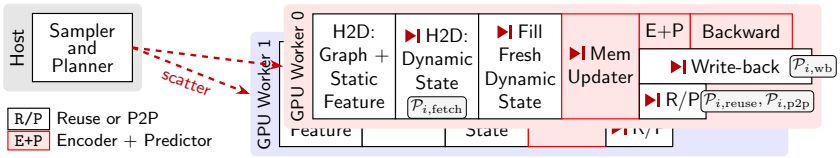



Fig. 3. FROST host-worker architecture.




Fig. 4. Fresh memory readiness and inter-GPU fabric idle window.

### 4.1 Architecture Overview

FROST follows a *host-worker* model, as illustrated in Fig. 3. A dedicated host process runs the temporal-graph sampler and assembles per-GPU mini-batches into macro-batches for both classes of TGNNs. It also executes the forward-looking planner to determine the routes of dynamic states across the CPU and GPUs for memory-based models. Each worker process manages one GPU and executes the training loop. The host and workers use a hybrid communication path where bulk sampled graph tensors and dynamic states reside in CPU shared memory, while per-batch metadata and control signals are exchanged among workers via collective communication across the GPUs.

In FROST, we use two different graph representations—one for data transfer and the other for computation. Specifically, each mini-batch is represented as a contiguous 1D graph object, called *FlatGraph*, that stores the sources, destinations, negative samples, and their neighbors in a serial manner. This simple representation is highly efficient for data transfers from the CPU to the GPUs and across GPUs. We apply data deduplication to node feature transfers and their states, meaning redundant data is not transmitted while the structural presence of duplicate node IDs is preserved. Prior to model execution, each worker converts this 1D object into a hierarchical graph object, called *HierGraph*, which represents the computational graph and facilitates dense-tensor operations.

### 4.2 Key Observations

While existing systems lack multi-GPU support for recent memoryless models, it is relatively straightforward to pipeline sampling, feature loading, and computation in their training loop. In contrast, optimizing pipeline parallelization for memory-based models remains technically challenging. We observe two important properties of the memory-based training loop, which underpin FROST's pipeline design.

First, each node's updated memory becomes available early in the training iteration. Specifically, the memory updater is executed as the *first* stage of the forward pass. Once it completes, all fresh node memories for the current mini-batch are materialized on the GPU, even though the encoder and predictor stages have not yet begun. This means that updated dynamic states can be dispatched for the next macro-batch *immediately* via local reuse or direct inter-GPU transfer, with CPU write-back planned separately, overlapping this distribution with the remainder of the forward computation.

Second, between the end of the memory updater and the first gradient synchronization, the inter-GPU fabric (e.g., NVLink) is completely idle, as no collective communication occurs in this interval under standard data-parallel training (Fig. 4). FROST exploits this idle window to enable direct inter-GPU transfers of fresh dynamic states at minimal cost, requiring only a lightweight

Table 4. Notation for Algorithms 1 and 2.

| Symbol | Description |
|---|---|
| $A$ | Access set (the set of all nodes in the batch) |
| $U$ | Update set (the set of sources and destinations in the batch) |
| $\tilde{U}$ | Deduplicated update set after ownership resolution |
| $M$ | Residual demand set ($A_{i,b+1} \setminus U_b$) |
| $C$ | Carry-over set ($A_{i,b} \setminus U_b$) |
| $F$ | CPU-fetch set (the set of nodes fetched from CPU) |
| $D$ | GPU topology distance matrix |
| $R_{ij}$ | Routed nodes from worker $i$ to worker $j$ |
| $\mathcal{P}$ | Dynamic-state execution plan |
| $\mathcal{S}$ | Static-feature execution plan |

**Algorithm 1:** Forward-Looking Dynamic-State Planner

**Input:** $\{A_{i,b+1}, U_{i,b+1}\}_i$, $\{A_{i,b}, U_{i,b}\}_i$, $\{F_{i,b}\}_i$, $D$
**Output:** $\{F_{i,b+1}\}_i$, $\{\mathcal{P}_i\}_i$

$U_b \leftarrow \bigcup_i U_{i,b}$; $U_{b+1} \leftarrow \bigcup_i U_{i,b+1}$;
$\{\tilde{U}_{i,b}\}_i \leftarrow$ ResolveUpdateOwnership($\{U_{i,b}\}_i$);
// Find states for local reuse and P2P transfer from $i$ to $j$
$\{R_{ij}\}_{i,j} \leftarrow$ UpdatedStateRoute($\{A_{i,b+1}\}_i, \{\tilde{U}_{i,b}\}_i$);
**if** `route_carry_over` **then**
  **for** $i = 1, \ldots, n$ **do** $M_{i,b+1} \leftarrow A_{i,b+1} \setminus U_b$; $C_{i,b} \leftarrow A_{i,b} \setminus U_b$;
  // Receiver-centric carry-over routing from the nearest GPU that already has required nodes
  $\{F_{i,b+1}\}_i, \{R^c_{ij}\}_{i,j} \leftarrow$ CarryOverRoute($\{M_{i,b+1}\}_i, \{C_{i,b}\}_i, D$);
  **for** $i, j = 1, \ldots, n$ **do** $R_{ij} \leftarrow R_{ij} \cup R^c_{ij}$;
**else**
  **for** $i = 1, \ldots, n$ **do** $F_{i,b+1} \leftarrow A_{i,b+1} \setminus U_b$;
**for** $i = 1, \ldots, n$ **do**
  $\mathcal{P}_{i,\text{reuse}} \leftarrow R_{ii}$; $\mathcal{P}_{i,\text{p2p}} \leftarrow \{R_{ij}\}_{j \neq i}$;
  $\mathcal{P}_{i,\text{wb}} \leftarrow \tilde{U}_{i,b} \setminus U_{b+1}$; $\mathcal{P}_{i,\text{fetch}} \leftarrow F_{i,b}$;
**return** $\{F_{i,b+1}\}_i$, $\{\mathcal{P}_i\}_i$;

synchronization hook to guarantee that data transfers complete before the gradient aggregation phase begins.

### 4.3 Forward-Looking Planners

For memory-based models, FROST employs two planners: one for dynamic states and one for static features. Both planners utilize a *one-step lookahead* in the sense that for a given macro-batch $b$, the sampling process for the next batch $b + 1$ completes before any worker begins model training on batch $b$.

We begin with the dynamic-state planner. For worker $i \in \{1, \ldots, n\}$, the dynamic-state plan comprises four components: (1) the CPU-fetch set $\mathcal{P}_{i,\text{fetch}}$, containing nodes whose states must be fetched from CPU shared memory; (2) the local-reuse set $\mathcal{P}_{i,\text{reuse}}$, containing nodes whose states already reside on the worker; (3) the P2P transfer set $\mathcal{P}_{i,\text{p2p}}$, which is a collection of node sets whose states need to be transferred from worker $i$ to other workers $j$; and (4) the CPU write-back

**Algorithm 2:** Forward-Looking Static-Feature Planner

**Input:** $\{A_{i,b+1}\}_i$, $\{A_{i,b}\}_i$, $\{F_{i,b}\}_i$
**Output:** $\{F_{i,b+1}\}_i$, $\{\mathcal{S}_i\}_i$
**for** $i = 1, \ldots, n$ **do**
  $\mathcal{S}_{i,\text{reuse}} \leftarrow A_{i,b} \cap A_{i,b+1}$; $\mathcal{S}_{i,\text{fetch}} \leftarrow F_{i,b}$;
  $F_{i,b+1} \leftarrow A_{i,b+1} \setminus A_{i,b}$;
**return** $\{F_{i,b+1}\}_i$, $\{\mathcal{S}_i\}_i$;

set $\mathcal{P}_{i,\text{wb}}$, containing nodes whose updated states must be written back to CPU shared memory. Fig. 3 illustrates where each component plan is consumed in the worker's pipeline.

Table 4 summarizes the key notation used in our algorithms. Let $A_{i,b}$ and $U_{i,b}$ denote the access and update sets that worker $i$ handles for macro-batch $b$, respectively. The access set $A_{i,b}$ comprises all nodes involved in the batch (including sources, destinations, negative samples, and their sampled neighbors), while the update set $U_{i,b}$ contains only the sources and destinations whose dynamic states are updated during the batch. When multiple workers are assigned the same node for a state update, a deterministic "higher-rank-wins" rule resolves ownership, producing the deduplicated set $\tilde{U}_{i,b}$. Because the chronological batch splitting ensures that a higher-rank worker processes interactions at a later timestamp, this rule guarantees that a node's state is always updated based on its most recent interaction. Finally, let $F_{i,b}$ denote the CPU-fetch set identified based on the previous batch $b-1$ for the current batch $b$ at worker $i$, and let $R_{ij}$ denote the set of nodes whose states must be transferred from worker $i$ to worker $j$.

As detailed in Algorithm 1, the dynamic-state planner takes as input the access and update sets for the current and next batches ($\{A_{i,b}, U_{i,b}\}_i$ and $\{A_{i,b+1}, U_{i,b+1}\}_i$), the current CPU-fetch set $\{F_{i,b}\}_i$, and the GPU topology distance matrix $D$, which ranks the fabric speeds between GPUs. It outputs the dynamic-state plan ($\mathcal{P}_{i,\text{fetch}}$, $\mathcal{P}_{i,\text{reuse}}$, $\mathcal{P}_{i,\text{p2p}}$, and $\mathcal{P}_{i,\text{wb}}$) together with the next-batch CPU-fetch set $\{F_{i,b+1}\}_i$.

The key steps of Algorithm 1 are as follows. First, the UpdatedStateRoute procedure, a *sender-centric* operation, identifies $R_{ij}$ for all pairs of workers $i$ and $j$. Specifically, for worker $i$, it intersects $\tilde{U}_{i,b}$ with the next-batch access set of every worker $j$, $A_{j,b+1}$. This yields the local-reuse set $R_{ii}$ and the P2P transfer sets $R_{ij}$ for $j \neq i$. Second, nodes in $\tilde{U}_{i,b}$ whose states are *not* updated again in the next batch (computed as $\tilde{U}_{i,b} \setminus U_{b+1}$) are assigned to the CPU write-back set, $\mathcal{P}_{i,\text{wb}}$. This ensures their exact states can be safely recovered in later batches.

Finally, an optional CarryOverRoute procedure allows the states of nodes that were accessed but not updated in batch $b$ to be reused or transferred to other workers for batch $b+1$. In contrast to the previous step, this is a *receiver-centric* operation. For each worker $i$, it examines the residual demand set $M_{i,b+1} = A_{i,b+1} \setminus U_b$. It then greedily routes these nodes from the nearest GPU that already holds them (based on $D$), falling back to a CPU fetch ($F_{i,b+1}$) only for the remainder. This trades additional P2P traffic for reduced PCIe and DRAM pressure, which is beneficial when the required CPU fetch volume exceeds available PCIe bandwidth. If CarryOverRoute is not used, the next-batch CPU-fetch set $F_{i,b+1}$ is simply computed as $A_{i,b+1} \setminus U_b$.

By construction, the combined updated-state and carry-over plan ensures that for every node $v$ accessed in batch $b+1$, the worker receives the state produced by the *most recent* committed update to $v$ in the system. Consequently, in terms of the read-time staleness defined in Eq. (2), FROST guarantees that $\sigma(v, b+1) = 0$ for every accessed node $v$ across all macro-batches $b$.

The static-feature planner operates on similar principles but is considerably simpler. Because static node features are immutable and have no cross-batch freshness dependencies, FROST only

needs to handle same-worker cache retention and CPU fetches. As detailed in Algorithm 2, the planner outputs the static-feature plan, consisting solely of the local-reuse set ($\mathcal{S}_{i,\text{reuse}}$) and the CPU-fetch set ($\mathcal{S}_{i,\text{fetch}}$), together with the next-batch CPU-fetch set $\{F_{i,b+1}\}_i$.

### 4.4 Three-Tier Dynamic-State Routing

Based on the dynamic-state plan generated by the forward-looking planner, FROST plans next-batch routing and CPU write-back independently. First, if the next macro-batch consumes an updated entry on the same worker, FROST reuses that freshly updated local entry directly, avoiding both a CPU refetch and an inter-GPU transfer. Second, entries needed by a peer GPU are sent directly via collective communication. The transfer utilizes direct inter-GPU fabrics; when the peer GPUs are connected via NVLink, this serves as the preferred fast path. Such an inter-GPU transfer is issued during the forward pass and must complete before the first gradient synchronization during the backward pass. This ordering is enforced by a lightweight communication hook, detailed in Section 4.6. Separately, updated entries that are not updated again in the next macro-batch are written back to the CPU shared memory asynchronously in the background while GPU computation proceeds, even when they are also supplied to that batch through local reuse or P2P.

It is worth noting that APAN [27] differs from other memory-based models such as TGN and JODIE, in that each root node's post-interaction message must be delivered to its sampled 1-hop neighbors. This makes the set of message updates typically larger than the set of memory updates. In FROST, we develop a dedicated planner for routing and writing back these neighbor-level message updates. These updates are inserted into the multi-slot mailboxes of the sampled neighbors, where each slot contains a single message. To avoid transmitting a complete mailbox across the GPUs for every update, FROST sends only the newly updated message slot per node and merges it into the receiver's mailbox.

### 4.5 Pipeline Staging

FROST's pipeline runs sampling, CPU-to-GPU static data loading (sampled subgraphs and their associated features), CPU-to-GPU dynamic state fetching, inter-GPU dynamic state routing, and model training (forward and backward passes) concurrently as overlapping stages. Fig. 5 illustrates the schematic pipeline with two worker GPUs, where each round advances two mini-batches (a macro-batch). Specifically, each worker first moves the 1D graph object FlatGraph and its associated features to the GPU. It then fetches only the necessary dynamic states from the shared memory that are neither locally reused nor received from peer workers via inter-GPU transfers. The complete set of required dynamic states is materialized once the locally reused, peer-transferred, and CPU-fetched states are combined. While FlatGraph is used to streamline all data transfers, it is converted into a HierGraph immediately after the memory update phase. This newly formed HierGraph is then used for the remainder of the forward and backward passes.

Each worker process runs a dedicated background thread to dispatch its NCCL collective communication for inter-GPU dynamic states, ensuring completion before the first gradient synchronization. Note that the host-to-device (H2D) gate in Fig. 5 ensures that the states of nodes that may not appear in consecutive batches are written back before being read again. For example, if a node appears in Batches 0–1, is absent from Batches 2–3, and reappears in Batches 4–5, the H2D gate ensures that Batches 4–5 read the most recently updated version of the memory rather than a stale one.

### 4.6 Communication Optimizations

FROST optimizes two distinct communication paths: *inter-GPU* communication during training, which applies to memory-based models whose dynamic states must be exchanged across GPUs,

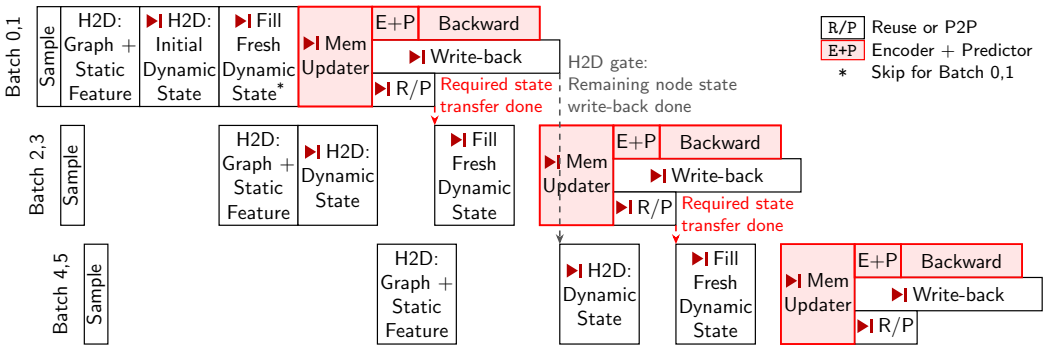



Fig. 5. FROST pipeline.

and *host-worker* data transfer during batch dispatch, which benefits both memory-based and memoryless models.

*Inter-GPU communication.* As explained in Section 4.2, the inter-GPU fabric is idle between the end of the memory updater and the first gradient synchronization, creating a natural window for direct dynamic-state transfers across GPUs. However, because both the inter-GPU transfers and gradient synchronization rely on NCCL, a deterministic ordering must be enforced. All inter-GPU transfers must be posted before any gradient synchronization begins. Violating this ordering would lead to a deadlock on the shared communication channel.

FROST resolves this by registering a custom communication hook. The hook is armed before the backward pass. When the *first gradient synchronization* is triggered during the backward pass, the hook waits to acquire a semaphore. This semaphore is released by the inter-GPU dispatch stage only after it has posted the NCCL communication for the dynamic-state tensors. Once the semaphore is acquired, the hook issues the first gradient synchronization, allowing the backward pass to proceed with subsequent synchronizations. This achieves maximum overlap. The inter-GPU communication happens concurrently with the forward pass and the early portion of the backward pass (all gradient computation up to the first gradient synchronization), effectively hiding the direct inter-GPU transfer behind GPU computation.

*Host-worker communication.* In existing frameworks such as TGL, the host dispatches each mini-batch by serializing a complete graph object via standard Python serialization and scattering it to every worker. This serialization and deserialization overhead scales proportionally with the sampled graph size. As described in Section 4.1, FROST decouples bulk graph data from control metadata. FlatGraph tensors reside in shared memory and are accessed by workers directly without data duplication, while the host scatters only a lightweight metadata object per worker (containing batch progress, shared-memory slot references, and routing plans), the size of which is independent of the sampled graph. Because the bulk data bypasses serialization entirely, the remaining per-batch serialization cost is negligible. We empirically observe that this design reduces host-side scatter time by 61–81% compared to TGL, with the performance gap widening at larger batch sizes.

### 4.7 Framework Modularity and Model Coverage

Unlike existing frameworks such as TGL, which tightly couple model logic with system execution, FROST adopts a component-based design. The temporal graph learning pipeline is decoupled into orthogonal components, ensuring that the host-worker execution engine, planner, and tensor data path remain unchanged even as the TGNN architectures vary. While users can easily integrate custom model architectures, FROST natively covers a wide range of TGNNs, including three memory-based models (TGN, JODIE, and APAN) and five memoryless models (TGAT, CAWN,

TCL, GraphMixer, and DyGFormer). This coverage is enabled by the following five configurable components:

- *Negative samplers:* FROST supports standard random sampling, historical and inductive strategies [21], and precomputed negative samples—including the evaluation sets provided by TGB [11].
- *Temporal neighbor samplers:* The C++ neighbor sampling module supports standard 1-hop neighbor sampling as well as the multi-hop and time-interval-aware sampling required by CAWN.
- *Memory updaters:* FROST supports standard RNN- and GRU-based memory updaters as well as more complex variants, such as APAN's mailbox mechanism.
- *Encoders:* A unified encoder interface natively supports both memoryless models and memory-based models that require recurrent state updates.
- *Predictors:* FROST provides a variety of predictors, ranging from standard link predictors to model-specific heads, such as pair-aware predictors (CAWN and DyGFormer) and merge-layer predictors (TCL).

Thanks to FROST's modular design, swapping any component—such as the neighbor sampler, memory updater, or negative sampling strategy—requires only configuration changes. The pipeline, routing, and tensor layouts then automatically adapt.

## 5 Experiments

In this section, we present an extensive experimental evaluation of FROST. We first describe the experimental setup (§5.1). Next, we evaluate the link prediction accuracy of both memory-based and memoryless models using FROST and two primary baseline systems, specifically verifying how preserving exact memory freshness improves accuracy (§5.2). We then demonstrate the substantial training throughput improvements achieved by FROST compared to the baselines (§5.3). We also analyze GPU utilization (§5.4) and memory footprint (§5.5), quantify FROST's traffic reduction (§5.6), present an ablation study (§5.7), and evaluate multi-GPU scalability (§5.8) and interconnect sensitivity (§5.9). Finally, we compare FROST against PipeTGL (§5.10) and evaluate large-graph training efficiency (§5.11).

### 5.1 Setup

*Hardware.* Unless otherwise stated, experiments are conducted on a single server equipped with four NVIDIA H200 GPUs. The GPUs are interconnected in an all-to-all NVLink topology, with 18 NVLink lanes per GPU, and PCIe 5.0 ×16 host links.

*Datasets.* We use Wiki, MOOC, Reddit, and LastFM [16, 22] for the main link prediction experiments and GDELT and MAG [34] for the large-graph experiments, as summarized in Table 5. Wiki and Reddit are social interaction datasets [22], whereas MOOC and LastFM are user-item interaction datasets [16]. GDELT is a global event knowledge graph, and MAG is a paper citation graph [34]. We use the standard 70/15/15 chronological train/validation/test split for the four small datasets and the year-based splits defined by TGL for GDELT and MAG.

*Baselines and covered models.* We compare FROST against two primary baselines, TGL [34] and DyGLib [30], for memory-based and memoryless models, respectively. TGL is a single-node multi-GPU baseline that provides a common pipelined acceleration framework for memory-based models (TGN, JODIE, and APAN). Note that TGL adopts a one-batch-behind stale-read policy ($\sigma \approx 1$). DyGLib is a single-GPU baseline for memoryless models (TGAT, CAWN, TCL, GraphMixer, and DyGFormer). FROST reuses these architectures unchanged, so the memoryless experiments focus on throughput. To keep the comparison fair, we modify DyGLib to store raw features in CPU DRAM and transfer them to the GPU on demand, matching FROST's host-memory setup.

Table 5. Dataset statistics.

| Dataset | Domain | $\lvert\mathcal{V}\rvert$ | $\lvert\mathcal{E}\rvert$ | $\lvert\mathcal{E}\rvert/\lvert\mathcal{V}\rvert$ | $d_v$ | $d_e$ | Duration |
|---|---|---|---|---|---|---|---|
| Wiki | Social | 9,227 | 157,474 | 17.1 | 0 | 172 | 1 month |
| MOOC | Interaction | 7,144 | 411,749 | 57.6 | 0 | 4 | 17 months |
| Reddit | Social | 10,984 | 672,447 | 61.2 | 0 | 172 | 1 month |
| LastFM | Interaction | 1,980 | 1,293,103 | 653.1 | 0 | 0 | 1 month |
| GDELT | Knowledge | 16,682 | 191,290,882 | 11,466.9 | 413 | 182 | 5 years |
| MAG | Citation | 121,751,665 | 1,297,748,926 | 10.7 | 768 | 0 | 120 years |

*Batch sizes.* Unless otherwise specified, we use the following default *macro-batch sizes* in terms of the number of interactions (events). For memory-based models, these are 600 events for Wiki, 1,200 for MOOC, 2,000 for Reddit, and 4,000 for LastFM. We chose these dataset-specific values to ensure the number of macro-batches per epoch is reasonably balanced across the four datasets. Specifically, these choices yield roughly 300 macro-batches per epoch on each dataset, ensuring stable model convergence.[3] For memoryless models, we use 600 events as the default macro-batch size, as their performance remains stable across all architectures. However, for the throughput benchmark, we sweep macro-batch sizes of 600, 1,200, 2,000, and 4,000 across all four datasets for both model classes.

*Metrics.* We evaluate FROST using two primary metrics: temporal link prediction accuracy and training throughput. For accuracy, we report *average precision* (AP), a standard metric in the literature [30, 34]. Specifically, we score positive temporal edges together with sampled negative edges and rank them by their predicted probabilities. AP is then computed as the area under the precision-recall curve based on these rankings. Test AP results are averaged over five independent runs. For throughput, FROST and TGL use two GPUs, while DyGLib uses one GPU because it is a single-GPU framework. We report throughput in events per second per GPU (ev/s/GPU), computed by dividing the aggregate throughput (total number of events processed per second) by the number of participating GPUs. We discard the first five macro-batches as warm-up and report the average throughput over the subsequent 100 macro-batches.

*Training setup.* We train all models for 100 epochs using the Adam optimizer with a learning rate of $10^{-4}$. For memory-based models, we configure the memory dimension as $d_{\text{mem}} = 128$. For all experiments, FROST and the baselines adopt the same random negative sampling strategy, following the setup in DyGLib [30].

As representative cases, we focus on TGN and DyGFormer trained on LastFM. We select LastFM because it contains the most events among the four main datasets, providing a suitable workload for performance evaluation while avoiding the long training times associated with the large-scale datasets. We use TGN as the representative memory-based model because it is the most canonical memory-based architecture and achieves the highest accuracy in its class. Similarly, we select DyGFormer as the representative memoryless model because it is the most recently proposed architecture and yields the highest accuracy among the evaluated memoryless models.

## 5.2 Link Prediction Accuracy

Table 6 reports the link prediction accuracy for both memory-based and memoryless models. Bold-faced entries indicate the best performance. Differences below 0.05 percentage points are treated as ties, and all tied entries are bolded. For memory-based models, FROST outperforms or ties TGL in

[3]JODIE lacks both a neighbor-aggregation GNN encoder and a message-diversity mechanism (e.g., APAN's attention mailbox). Because it relies heavily on per-interaction message generation to propagate information across nodes, large batches can destabilize its convergence. We thus use one-fifth of the default macro-batch size for JODIE.

Table 6. Test AP (%) for memory-based and memoryless models.

(a) Memory-based models

| Dataset | System | TGN | JODIE | APAN |
|---|---|---|---|---|
| Wiki | TGL | 98.46 ± 0.07 | 97.51 ± 0.07 | 95.75 ± 0.28 |
| | FROST | **98.72 ± 0.02** | **97.71 ± 0.15** | **98.27 ± 0.12** |
| MOOC | TGL | 86.71 ± 0.62 | 78.77 ± 1.75 | 69.55 ± 1.55 |
| | FROST | **91.02 ± 0.61** | **82.96 ± 1.08** | **70.16 ± 1.40** |
| Reddit | TGL | **98.66 ± 0.02** | 98.20 ± 0.04 | **97.87 ± 0.14** |
| | FROST | **98.65 ± 0.05** | **98.27 ± 0.02** | 97.29 ± 0.25 |
| LastFM | TGL | 82.55 ± 1.51 | 64.80 ± 6.34 | 69.69 ± 1.46 |
| | FROST | **85.17 ± 0.46** | **67.82 ± 5.23** | **72.46 ± 2.77** |

(b) Memoryless models

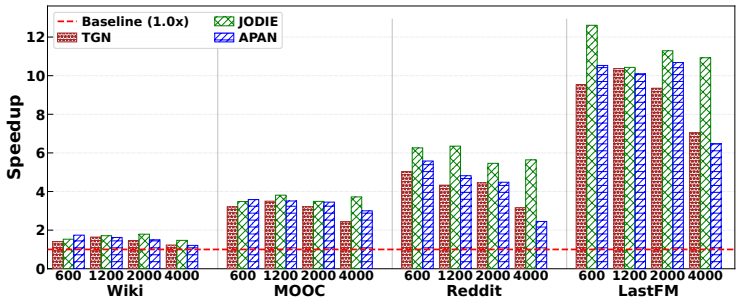
| Dataset | System | TGAT | CAWN | TCL | GraphMixer | DyGFormer |
|---|---|---|---|---|---|---|
| Wiki | DyGLib | **98.45 ± 0.06** | **98.76 ± 0.03** | 97.77 ± 0.00 | 97.25 ± 0.03 | **99.07 ± 0.01** |
| | FROST | 97.57 ± 0.08 | **98.74 ± 0.02** | **97.95 ± 0.07** | **97.33 ± 0.02** | **99.10 ± 0.01** |
| MOOC | DyGLib | **85.84 ± 0.15** | 80.15 ± 0.25 | 82.38 ± 0.24 | **82.78 ± 0.15** | 87.20 ± 0.27 |
| | FROST | 85.74 ± 0.31 | **81.56 ± 0.25** | **83.50 ± 0.52** | **82.75 ± 0.07** | **88.33 ± 0.43** |
| Reddit | DyGLib | 97.88 ± 0.00 | **99.11 ± 0.01** | **97.53 ± 0.02** | **97.31 ± 0.01** | 98.73 ± 0.38 |
| | FROST | **98.50 ± 0.01** | 98.98 ± 0.00 | 97.33 ± 0.02 | **97.29 ± 0.32** | **99.24 ± 0.01** |
| LastFM | DyGLib | 73.42 ± 0.21 | 86.99 ± 0.06 | **75.52 ± 0.37** | 75.61 ± 0.24 | 93.25 ± 0.00 |
| | FROST | **76.52 ± 0.18** | **88.20 ± 0.06** | 75.42 ± 0.49 | **77.53 ± 0.13** | **93.92 ± 0.04** |

11 of the 12 model-dataset settings, with particularly large gains on MOOC and LastFM. Even in the single setting where FROST underperforms, the deficit is a marginal 0.6 AP. These results show that stale reads under TGL's one-batch-behind policy ($\sigma \approx 1$) can substantially degrade accuracy, and that preserving exact memory freshness ($\sigma = 0$) yields measurable gains without sacrificing throughput, as we demonstrate in the next subsection.

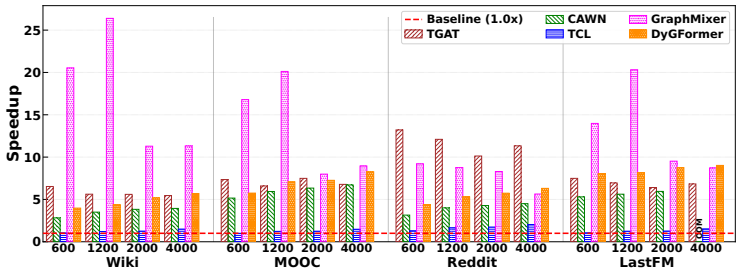
For memoryless models, FROST and DyGLib produce comparable test AP across all settings, as expected. Memoryless architectures carry no dynamic state between batches, so there is no freshness gap between the two systems. The minor fluctuations in both directions reflect only run-to-run variance and differences in multi-GPU versus single-GPU batching order.

### 5.3 Training Throughput

Fig. 6a summarizes the memory-based throughput benchmarks, demonstrating that FROST is faster than TGL across all 48 settings. Note that this comparison is conservative for FROST because the measured speedups are obtained against TGL under its default one-batch-behind stale-read policy, whose read-time staleness is approximately one macro-batch ($\sigma \approx 1$), whereas FROST simultaneously preserves exact memory freshness ($\sigma = 0$). FROST delivers the largest gains (up to 12.61×) on LastFM, while the advantage narrows on Wiki. A primary reason for the smaller relative gain on Wiki is that it has a much lower interaction density than the other datasets (Table 5), meaning adjacent macro-batches are less likely to access the same nodes. This leaves fewer opportunities for same-worker local reuse and direct inter-GPU transfers, so more dynamic states must later be written




(a) Speedup of FROST over TGL




(b) Speedup of FROST over DyGLib

Fig. 6. Speedup of FROST.

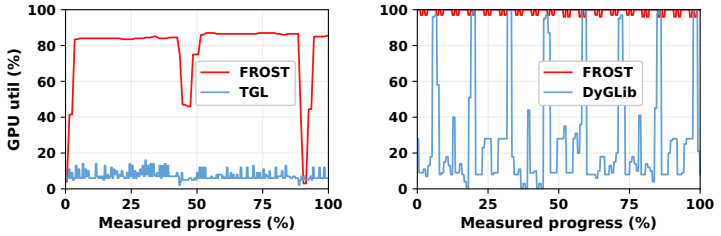
back to and fetched from CPU memory. In contrast, on higher-degree datasets such as LastFM, adjacent macro-batches overlap heavily, allowing FROST to exploit local reuse and direct inter-GPU transfers much more often.

Fig. 6b summarizes the memoryless throughput benchmarks. FROST consistently outperforms DyGLib on all models, with speedups of up to 26.39×. These gains come from a different source than in memory-based models. Memoryless architectures carry no mutable node state, so they do not benefit from same-worker state reuse or direct inter-GPU transfers of updated states. Instead, FROST accelerates them mainly through pipelining that hides sampling and data transfer behind GPU computation, a C++/OpenMP neighbor sampler, and vectorized encoder-side preprocessing. However, FROST and DyGLib perform similarly on TCL because TCL samples only short histories, its preprocessing in DyGLib is already highly vectorized, and its runtime is heavily dominated by pairwise attention computation rather than by sampling, feature movement, or Python-side preprocessing.

### 5.4 GPU Utilization

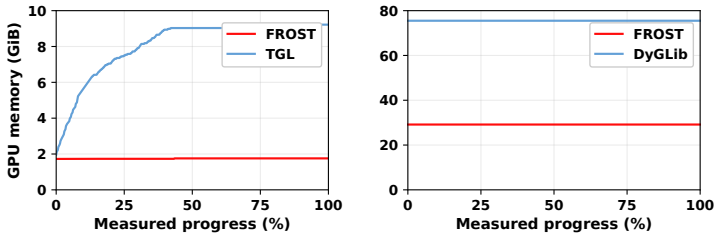
In Fig. 7, we show representative GPU utilization traces of FROST and the baselines using TGN-LastFM and DyGFormer-LastFM when the macro-batch size is 4,000. These experiments are conducted on two GPUs when comparing FROST with TGL, and on a single GPU when comparing FROST with DyGLib. Here, we sample GPU utilization every 50 ms, discard the first 10 macro-batches as warm-up, and focus our attention on the next 500 measured macro-batch intervals, which allow us to observe cross-epoch utilization.

As shown in Fig. 7a, FROST achieves substantially higher GPU utilization than TGL. FROST reaches an average GPU utilization of 77.9%, whereas TGL reaches only 7.5%. Here, the x-axis indicates normalized progress over the 500 measured macro-batches. It is worth noting that there are a few dips in FROST's trace. These are concentrated near epoch boundaries, where the system




(a) TGN-LastFM (b) DyGFormer-LastFM

Fig. 7. GPU utilization on LastFM.




(a) TGN-LastFM (b) DyGFormer-LastFM

Fig. 8. Memory footprint on LastFM.

must rebuild dynamic states and refill static features rather than benefiting from cross-batch local reuse or direct inter-GPU transfers.

In Fig. 7b, we zoom in on the GPU utilization of FROST and DyGLib for 500 consecutive samples. Here, the x-axis indicates normalized progress, but this time over the 500 sample measurements. This zoomed-in view is necessary because DyGLib's GPU utilization oscillates rapidly, a pattern that would be obscured if plotted over the full 500 macro-batch window. As shown in Fig. 7b, FROST's average utilization reaches 98.9%, while DyGLib reaches only 24.6%. DyGLib's DyGFormer exhibits larger short-term GPU utilization fluctuations because much of its pre-Transformer input preparation is done on the CPU. In particular, neighbor co-occurrence counting is heavily dominated by per-sample operations on the CPU. As a result, the GPU is fed in short bursts and experiences more frequent idle gaps. In summary, FROST keeps the GPUs substantially busier in both memory-based and memoryless settings.

### 5.5 GPU Memory Footprint

Fig. 8 shows the GPU memory footprint of FROST and the baselines with a macro-batch size of 4,000, measured using PyTorch reserved memory. As shown in Fig. 8a, FROST's memory footprint stays around 1.74 GiB per GPU, while TGL's footprint reaches a plateau of around 9.22 GiB. The primary reasons for this significant difference are their data management for two consecutive macro-batches and the amount of write-back data. Due to the pipelined design of FROST and TGL, two consecutive macro-batches naturally coexist in memory. However, FROST exploits the overlap across the batches and reuses the necessary data for both. In contrast, TGL treats them independently, resulting in redundant data duplication. Furthermore, while FROST minimizes the number of dynamic states to be written back to CPU shared memory, TGL requires all updated states to be written back unconditionally. This further prolongs the temporal overlap between the previous batch and the next batch. Finally, the underlying data structures also matter. FROST packs each batch into a more regular FlatGraph, whereas TGL operates on heavier graph objects. As

Table 7. Traffic reduction of FROST.

| | | Wiki | MOOC | Reddit | LastFM |
|---|---|---|---|---|---|
| Macro-batch size ($\hat{B}$) | | 600 | 1,200 | 2,000 | 4,000 |
| Sampled nodes | | 17,282 | 32,830 | 61,838 | 127,328 |
| Unique nodes | | 2,328 | 583 | 7,705 | 2,779 |
| H2D fetch | | 1,938 | 282 | 5,592 | 619 |
| Local reuse | | 189 | 139 | 1,027 | 1,079 |
| P2P transfer | | 200 | 162 | 1,087 | 1,080 |
| Unique positive roots | | 393 | 324 | 1,811 | 1,750 |
| D2H write-back | | 326 | 225 | 1,504 | 1,090 |
| CPU fetch ↓ | Deduplication | 86.53% | 98.22% | 87.54% | 97.82% |
| | Plan and Routing | 16.74% | 51.63% | 27.42% | 77.73% |
| Write-back ↓ | Deduplication | 67.07% | 86.43% | 54.50% | 78.01% |
| | Plan and Routing | 16.90% | 30.56% | 16.95% | 37.71% |

a result, FROST's footprint is substantially smaller than TGL's (just one-fifth of TGL's memory footprint).

As shown in Fig. 8b, FROST's memory footprint remains at 29.16 GiB per GPU, while DyGLib's reaches 75.5 GiB per GPU. Thus, FROST reduces GPU memory usage by 61.4%. The underlying cause of this disparity, however, is fundamentally different. DyGLib repeatedly allocates intermediate tensors, whereas FROST follows a highly regular, vectorized path that builds root and neighbor sequences only once and employs more efficient tensor management throughout the process. Repeatedly allocating short-lived tensors of varying shapes is highly inefficient for the CUDA caching allocator because it exacerbates memory fragmentation and inflates the amount of reserved memory. Therefore, DyGLib reaches a much higher sustained reserved-memory level, while FROST maintains a substantially smaller temporary working set, resulting in a significantly reduced overall footprint.

### 5.6 Traffic Reduction

To quantify how three-tier routing contributes to traffic reduction, we instrument FROST's host-side dynamic planner and routing module when training TGN on two GPUs, using one negative sample per positive edge and 10 sampled neighbors per node. We record the realized per-worker routing decisions, compute the node counts separately for each worker, and sum them per macro-batch before averaging over one epoch. Table 7 reports the epoch-averaged node counts at each stage of the training pipeline. First, it is important to note the impact of the data deduplication employed in FROST. Deduplication alone eliminates 86.53–98.22% of the sampled instances that would otherwise each trigger an independent host-to-device transfer in a naive design. It also reduces the device-to-host write-back traffic by 54.50–86.43%.

Building upon this, FROST's three-tier routing (local reuse, inter-GPU transfers, and residual CPU fetches) further reduces the residual traffic within this deduplicated population. Specifically, on the CPU fetch side, FROST needs to read data for only a small subset of unique nodes involved in a batch—on average 1,938 nodes on Wiki, 282 on MOOC, 5,592 on Reddit, and 619 on LastFM—which corresponds to a 16.74–77.73% reduction compared to the number of unique nodes. On the write-back side, FROST commits only a reduced subset of positive roots (sources and destinations)—averaging 326 nodes on Wiki, 225 on MOOC, 1,504 on Reddit, and 1,090 on LastFM. This corresponds

Table 8. Throughput of FROST and its variants (ev/s/GPU).

| Case | Macro-batch size | | | |
|---|---|---|---|---|
| | 600 | 1,200 | 2,000 | 4,000 |
| C1: No plan, no P2P/reuse, post-step write-back | 28,547 | 50,199 | 73,931 | 96,078 |
| C2: No plan, no P2P/reuse, immediate write-back | 32,239 | 56,690 | 83,798 | 123,517 |
| C3: Plan for deduplication, no P2P/reuse, immediate write-back | 32,370 | 63,121 | 100,549 | 175,864 |
| C4: FROST | 33,743 | 66,487 | 108,043 | 195,908 |

to a 16.90–37.71% reduction compared to the total number of unique positive roots, which is the number of nodes whose updated states would need to be written back to the CPU without three-tier routing.

### 5.7 Ablation Study

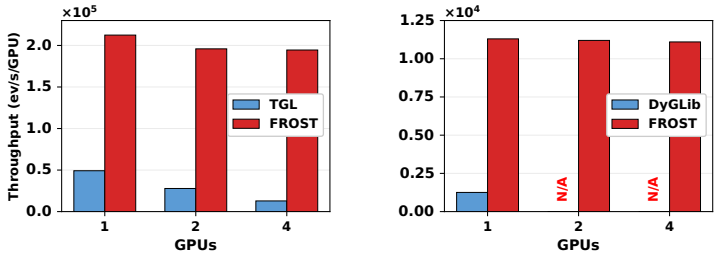
Following the discussion on the traffic reduction of FROST, we further evaluate the impact of FROST's data deduplication, forward-looking planner, and three-tier routing on training throughput. Table 8 reports the throughput of FROST (C4) and its stripped-down variants (C1–C3) when using TGN-LastFM for macro-batches of 600, 1,200, 2,000, and 4,000.

Cases C1–C3 lack the forward-looking planner and three-tier routing while still preserving exact memory freshness. Among these variants, data deduplication is enabled only in C3. For C1–C3, since workers can only read dynamic states from the CPU, macro-batch $b$+1 cannot start its process until macro-batch $b$ has finished its CPU write-back of updated states. C1 is the most stripped-down variant. Every batch reads all required states directly from the CPU shared memory, and the resulting updated states are written back only after the model weights are updated. This is similar to the strict pipeline illustrated in Fig. 2a. C2 is identical to C1 but performs memory persistence earlier, meaning that updated states are synchronously written back to the CPU immediately after the memory updater finishes updating node memories. C3 is the same as C2, but with a current-batch plan and data deduplication for both host-to-device and device-to-host data transfers.

Comparing C2 to C1 demonstrates that when dynamic states can only be read from the CPU, performing memory persistence as soon as the updated memory is ready yields a 12.9–28.6% throughput improvement. Moving from C2 to C3, deduplicating the current batch's write-back set and fetch set provides 0.4–42.4% higher throughput. C4 provides a further 4.2–11.4% throughput improvement over C3. The forward-looking planner and three-tier routing in FROST not only reduce the residual CPU fetch and write-back traffic within the deduplicated population, but, more importantly, remove the memory synchronization bottleneck in memory-based models. Instead of forcing macro-batch $b$+1 to wait for macro-batch $b$'s CPU write-back to refetch fresh states from the CPU shared memory, FROST preemptively identifies the dynamic states that still need to be fetched from the CPU. It transfers these states as soon as macro-batch $b-1$'s CPU write-back completes, while serving the shared states through local reuse or direct inter-GPU transfers.

### 5.8 Scalability with Number of GPUs

We evaluate the scalability of FROST using TGN-LastFM and DyG-Former-LastFM across 1, 2, and 4 worker GPUs while keeping the per-GPU batch size fixed at 2,000 events. This yields macro-batch sizes of 2,000, 4,000, and 8,000 for 1, 2, and 4 GPUs, respectively. Fig. 9 summarizes the resulting per-GPU throughput. As shown in Fig. 9a, FROST maintains substantially higher throughput than TGL at every GPU count and exhibits modest degradation as the GPU count increases. A key reason for this performance gap is the difference in host-side pipeline depth. In FROST, the host can buffer many future macro-batches through deeper host-side queues, allowing sampling and




(a) TGN-LastFM (b) DyGFormer-LastFM

Fig. 9. Scalability on LastFM.

access planning to run ahead of the workers so that batches wait in a "ready-to-dispatch" state before the workers request them. In contrast, TGL does not maintain an analogous host-side queue for prepared batches. Its host process follows a strictly sequential execution path. For each training step, it samples the next batch, constructs the corresponding subgraphs (Python objects), and then scatters these objects to the workers. As the GPU count increases, these sequential operations become a severe bottleneck.

In addition, as shown in Fig. 9b, FROST exhibits better scalability on DyGFormer compared to memory-based models such as TGN. As the GPU count increases, the per-GPU throughput of FROST for DyGFormer remains highly stable. DyGFormer is a long-context encoder whose runtime is heavily dominated by the encoder computation itself, meaning the majority of the time is spent on pure GPU compute. For memory-based models, however, a larger fraction of the step is spent on host-side sampling, subgraph construction, access planning, memory persistence, and inter-GPU synchronization. These overheads become more pronounced as the GPU count increases because the host must serve more workers, increasing its sampling and planning load accordingly. As a result, memory-based models have less pure computation to amortize these system overheads. Nonetheless, FROST maintains high scalability for both memory-based and memoryless architectures.

### 5.9 Interconnect Sensitivity

To evaluate the impact of different hardware fabrics, we compare FROST against TGL on TGN-LastFM using four two-GPU platforms. Table 9 shows the results while varying the macro-batch size from 600 to 4,000, where P1 and P2 use NV4 (four bonded NVLink connections), P3 uses NV6 (six bonded connections) over NVSwitch, and P4 uses PCIe 5.0 GPUDirect without NVLink.

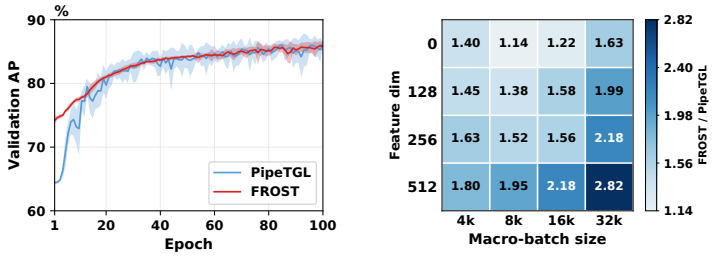
FROST remains substantially faster than TGL on all four platforms. P3 delivers the highest absolute throughput and the largest relative speedups. It reaches a throughput of 195.9k ev/s/GPU at a macro-batch size of 4,000 and achieves overall speedups of 7.04×–10.37× over TGL across the tested batch sizes. The NV4-connected P1 and P2 still provide consistent gains of 5.92×–6.51× and 5.65×–8.18×, respectively. Even on the PCIe-only P4, FROST reaches a throughput of 78.5k ev/s/GPU at a macro-batch size of 4,000 and remains 4.48×–5.43× faster than TGL. Overall, FROST exhibits larger relative speedups on higher-bandwidth peer fabrics, while retaining a substantial performance advantage on PCIe-only connectivity.

### 5.10 Comparison with PipeTGL

We next compare FROST against the latest system PipeTGL [19], which supports only TGN, in terms of convergence stability and throughput. Again, we use two H200 GPUs for the experiments. For convergence stability, we train TGN-LastFM using both systems for 100 epochs with ten seeds

Table 9. FROST's speedup over TGL across four platforms.

| | GPU | Peer Path | Macro-batch size | | | |
|---|---|---|---|---|---|---|
| | | | 600 | 1,200 | 2,000 | 4,000 |
| P1 | RTX A6000×2 | NVLink NV4 | 5.94× | 6.09× | 6.51× | 5.92× |
| P2 | RTX A5000×2 | NVLink NV4 | 8.18× | 7.38× | 6.61× | 5.65× |
| P3 | H200×2 | NVLink NV6 | 9.53× | 10.37× | 9.35× | 7.04× |
| P4 | RTX 5090×2 | PCIe 5.0 ×16 | 4.48× | 4.58× | 5.43× | 5.07× |




(a) Convergence (b) Speedup

Fig. 10. Comparison with PipeTGL.

and report the mean validation AP, with shaded bands indicating standard deviation. While both reach a similar final validation AP, Fig. 10a shows that FROST exhibits lower variability across seeds. We also observe that FROST achieves higher validation AP than PipeTGL during the early stages of training. This difference might stem from PipeTGL's stale model weights, which destabilize model convergence. In contrast, FROST keeps both model parameters and node memories fresh throughout training.

To evaluate throughput under diverse settings, we vary the global macro-batch size from 4,000 to 32,000 and use synthetically generated FP32 node features with dimensions ranging from 0 to 512. We report the throughput of the slower rank, taken as the median of three train-only runs after a full warm-up epoch. Fig. 10b shows that FROST outperforms PipeTGL by 1.14×–2.82×. From 0 to 512 dimensions, PipeTGL's throughput drops by 36.8%, 49.9%, 49.6%, and 48.3% at macro-batch sizes of 4,000, 8,000, 16,000, and 32,000, respectively. In contrast, the corresponding reductions for FROST are 18.6%, 13.8%, 10.3%, and 10.5%. At 512 dimensions, FROST's speedup over PipeTGL increases monotonically from 1.80× at a macro-batch size of 4,000 to 2.82× at 32,000. These results show that PipeTGL is increasingly bottlenecked by feature-handling overhead as dimensionality and macro-batch size grow, whereas FROST limits this overhead through its pipelined feature staging.

### 5.11 Large-Graph Training Efficiency

We finally evaluate the performance of FROST for large-scale training. To this end, we train the TGN model with FROST, TGL, and PipeTGL for five epochs on two large temporal graphs, MAG and GDELT, using two GPUs. We use a macro-batch size of 128,000 events and real on-disk features for both datasets. We choose this large batch size because smaller batches incur substantial overhead, leading to prohibitively long training times on graphs of this scale. Table 10 reports the resulting accuracy and throughput.

*Accuracy.* FROST and TGL both achieve a test AP of 99.29% at the fifth epoch on MAG, while PipeTGL fails to complete due to an NCCL communication timeout after 600 seconds. On GDELT, all three systems reach nearly identical test AP—97.91% for FROST and 97.90% for both TGL and

Table 10. Large-Graph Training Efficiency.

| Dataset | $d_v$ | $d_e$ | System | Per-GPU Throughput | Test AP (%) |
|---|---|---|---|---|---|
| GDELT | 413 | 182 | FROST | 493.93k ev/s | 97.91 |
| | | | TGL | 273.06k ev/s | 97.90 |
| | | | PipeTGL | 93.63k ev/s | 97.90 |
| MAG | 768 | 0 | FROST | 575.39k ev/s | 99.29 |
| | | | TGL | 400.80k ev/s | 99.29 |
| | | | PipeTGL | NCCL timeout | |

PipeTGL. Because these large datasets contain many training events, five full epochs provide sufficient optimization steps for the models to converge, which explains the small AP differences.

*Throughput.* FROST delivers 1.44× and 1.81× higher per-GPU training throughput than TGL on MAG and GDELT, respectively. The speedups at this scale are more modest because large macro-batches spend most of their execution time on GPU computation, making pipeline scheduling and synchronization overheads a smaller fraction of the total runtime and leaving less overhead for FROST to eliminate. Nevertheless, FROST remains faster than TGL on both large datasets. The lower speedup on MAG is also consistent with its lower node degrees, since adjacent macro-batches share a smaller fraction of nodes. Meanwhile, FROST is 5.28× faster than PipeTGL on GDELT, while PipeTGL fails to complete on MAG. This result is consistent with the scalability trend in Section 5.10. PipeTGL is substantially more sensitive to high-dimensional features at large macro-batch sizes. MAG combines a macro-batch size of 128,000 with 768-dimensional node features, pushing both factors beyond the ranges evaluated in Fig. 10b and presenting a substantially more demanding workload. Our profiling suggests that, under this demanding workload, PipeTGL encounters an implementation-level dependency-ordering issue in its inter-batch communication path. Under its round-robin schedule, a rank processing a successor batch may wait for incoming data before the corresponding communication becomes ready, preventing the NCCL operation from making progress until it times out. However, FROST avoids this serialization through planned state routing.

Additional experimental results and analyses are provided in the appendices, including training time breakdown (Appendix C), graph representation ablations (Appendix D), batch-size and multi-GPU scaling (Appendix E), and compatibility with TGB [9] and TPNet [20] (Appendices F and G).

## 6 Discussion

In this section, we discuss the broader implications of FROST's design and outline its current limitations.

*Why focus on link prediction?* Temporal link prediction is the *primary self-supervised training task* for learning embeddings in TGNNs. The learned embeddings can then support downstream tasks such as node classification and node-feature prediction using task-specific heads. Temporal link prediction itself creates a strong freshness constraint. Each batch couples source nodes, destination nodes, and negative samples under *temporal ordering*, meaning any stale dynamic state directly distorts the training signal. This makes link prediction more demanding for exact multi-GPU acceleration, whereas other downstream tasks are a natural next step once the learned embeddings are in place.

*Why memory-based models?* While memoryless long-context models such as CAWN and DyGFormer often achieve higher accuracy, memory-based models remain compelling because they summarize history in compact per-node states and thus can run with much shallower temporal

sampling. In practice, models such as JODIE require no neighbor expansion, while TGN and APAN use only shallow local context. This lower sampling footprint places memory-based TGNNs at a substantially different efficiency point from long-context memoryless models. FROST removes a critical yet previously overlooked systems penalty for this model family, allowing their inherent efficiency to translate into practical high-throughput training.

*Transferability of the routing abstraction.* It is important to note that the forward-looking planner and three-tier routing are not tied to a specific memory updater or encoder family. They apply whenever training maintains mutable per-entity states where the state consumers in batch $b$+1 can be determined during the processing of batch $b$. Thus, we can view FROST's routing mechanism as a reusable systems pattern that may extend to other stateful training pipelines beyond the TGNN instances evaluated in this work.

*Deeper lookahead windows.* For memory-based models, the current planner uses a one-step lookahead. It determines local reuse and direct inter-GPU transfers based on the sampled demand in the next macro-batch, and separately schedules CPU write-back for states that will not be updated in that macro-batch. This approach is already highly effective on higher-degree datasets such as LastFM, where adjacent macro-batches overlap heavily and many updated states can be kept resident on the GPUs through local reuse or peer-to-peer transfers. On lower-degree graphs such as Wiki, however, the overlap between consecutive macro-batches is much weaker. As a result, states that are not needed in batch $b$+1 may still be required again in batch $b$+2 or later. Nonetheless, a one-step planner must conservatively write these states back to CPU memory and fetch them again when they reappear. A natural extension is therefore to plan over a deeper lookahead window by evaluating the sampled demand of multiple future macro-batches, allowing selected states to be retained on the GPUs across a longer horizon. This could substantially improve throughput on sparse or low-degree temporal graphs, although it would require new policies for GPU memory budgeting and state eviction, as well as careful management of the increased planner overhead. We leave this direction as future work.

*Limitations.* The current implementation of FROST is limited to single-node, multi-GPU setups. Extending to a multi-node environment requires adding inter-node RDMA support, and the planner must also account for higher inter-node latency. We leave this as future work.

## 7 Conclusion

We presented FROST, a unified training system that, to our knowledge, is the first to provide efficient multi-GPU support for both memory-based and recent popular memoryless TGNNs. For memory-based models, FROST introduces a one-step forward-looking planner that orchestrates three-tier dynamic-state routing—same-worker local reuse, direct inter-GPU transfers, and asynchronous CPU write-backs—hiding the memory synchronization overhead that causes large pipeline stalls in existing systems. The planner ensures *optimal read-time freshness*, meaning that every accessed dynamic state is the most recently committed version in the system. For memoryless models, which previously lacked any multi-GPU or pipelined training runtime, FROST integrates them into the same staged host-worker pipeline, enabling scalable distributed training for the first time. Our evaluation across various TGNN architectures, datasets, and experimental settings demonstrates the efficiency and effectiveness of FROST for both memory-based and memoryless models. From multiple perspectives, including link prediction accuracy, training throughput, GPU utilization, and memory footprint, the results highlight FROST's capacity for large-scale training.

## Acknowledgments

This work was supported by the National Science Foundation under Grant Nos. 2209921 and 2209922, and an equipment donation from NVIDIA Corporation.

## References


[1] Claudio DT Barros, Matheus RF Mendonça, Alex B Vieira, and Artur Ziviani. 2021. A Survey on Embedding Dynamic Graphs. *ACM Computing Surveys (CSUR)* 55, 1 (2021), 1–37.

[2] Elizabeth Boschee, Jennifer Lautenschlager, Sean O'Brien, Steve Shellman, James Starz, and Michael Ward. 2015. ICEWS Coded Event Data.

[3] Borui Cai, Yong Xiang, Longxiang Gao, He Zhang, Yunfeng Li, and Jianxin Li. 2023. Temporal Knowledge Graph Completion: A Survey. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 6545–6553. Survey Track.

[4] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 130–144.

[5] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. 2023. Do We Really Need Complicated Model Architectures For Temporal Networks?. In *International Conference on Learning Representations*.

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

[7] Matthias Fey, Jinu Sunil, Akihiro Nitta, Rishi Puri, Manan Shah, Blaz Stojanovic, Ramona Bendias, Alexandria Barghi, Vid Kocijan, Zecheng Zhang, Xinwei He, Jan E. Lenssen, and Jure Leskovec. 2025. PyG 2.0: Scalable Learning on Real World Graphs. In *Temporal Graph Learning Workshop at KDD*.

[8] 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)*. 551–568.

[9] Julia Gastinger, Shenyang Huang, Mikhail Galkin, Erfan Loghmani, Ali Parviz, Farimah Poursafaei, Jacob Danovitch, Emanuele Rossi, Ioannis Koutis, Heiner Stuckenschmidt, et al. 2024. TGB 2.0: A Benchmark for Learning on Temporal Knowledge Graphs and Heterogeneous Graphs. *Advances in Neural Information Processing Systems* 37 (2024), 140199–140229.

[10] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1.

[11] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2023. Temporal Graph Benchmark for Machine Learning on Temporal Graphs. *Advances in Neural Information Processing Systems* 36 (2023), 2056–2073.

[12] Xin Huang, Jongryool Kim, Bradley Rees, and Chul-Ho Lee. 2022. Characterizing the Efficiency of Graph Neural Network Frameworks with a Magnifying Glass. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 160–170.

[13] Xin Huang and Chul-Ho Lee. 2025. Demystifying Distributed Training of Graph Neural Networks for Link Prediction. In *2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 199–209.

[14] Xin Huang, Weipeng Zhuo, Minh Phu Vuong, Shiju Li, Jongryool Kim, Bradley Rees, and Chul-Ho Lee. 2025. SDT-GNN: Streaming-based Distributed Training Framework for Graph Neural Networks. In *2025 IEEE International Conference on Big Data (BigData)*. IEEE, 162–172.

[15] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation Learning for Dynamic Graphs: A Survey. *Journal of Machine Learning Research* 21, 70 (2020), 1–73.

[16] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1269–1278.

[17] Kalev Leetaru and Philip A. Schrodt. 2013. GDELT: Global Data on Events, Location and Tone, 1979–2012. In *ISA Annual Convention*. San Francisco, CA, USA, 1–49.

[18] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 401–415.

[19] Jun Liu, Bingqian Du, Ziyue Luo, Sitian Lu, Qiankun Zhang, and Hai Jin. 2025. PipeTGL: (Near) Zero Bubble Memory-based Temporal Graph Neural Network Training via Pipeline Optimization. *Proceedings of the VLDB Endowment* 18, 8 (2025), 2722–2734.

[20] Xiaodong Lu, Leilei Sun, Tongyu Zhu, and Weifeng Lv. 2024. Improving Temporal Link Prediction via Temporal Walk Matrix Projection. *Advances in Neural Information Processing Systems* 37 (2024), 141153–141182.

[21] Farimah Poursafaei, Shenyang Huang, Kellin Pelrine, and Reihaneh Rabbany. 2022. Towards Better Evaluation for Dynamic Link Prediction. *Advances in Neural Information Processing Systems* 35 (2022), 32928–32941.

[22] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In *ICML Workshop on Graph Representation Learning*

*and Beyond (GRL+)*.

[23] Guangming Sheng, Junwei Su, Chao Huang, and Chuan Wu. 2024. MSPipe: Efficient Temporal GNN Training via Staleness-Aware Pipeline. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2651–2662.

[24] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2021. Foundations and Modeling of Dynamic Networks Using Dynamic Graph Neural Networks: A Survey. *IEEE Access* 9 (2021), 79143–79168.

[25] Lu Wang, Xiaofu Chang, Shuang Li, Yunfei Chu, Hui Li, Wei Zhang, Xiaofeng He, Le Song, Jingren Zhou, and Hongxia Yang. 2021. TCL: Transformer-based Dynamic Graph Modelling via Contrastive Learning. *arXiv preprint arXiv:2105.07944* (2021).

[26] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[27] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 2021. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *Proceedings of the 2021 International Conference on Management of Data*. 2628–2638.

[28] 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*.

[29] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive Representation Learning on Temporal Graphs. In *International Conference on Learning Representations*.

[30] Le Yu, Leilei Sun, Bowen Du, and Weifeng Lv. 2023. Towards Better Dynamic Graph Learning: New Architecture and Unified Library. *Advances in Neural Information Processing Systems* 36 (2023), 67686–67700.

[31] Zhongyi Yu, Jianqiu Wu, Zhenghao Wu, Shuhan Zhong, Weifeng Su, Chul-Ho Lee, and Weipeng Zhuo. 2025. TAMI: Taming Heterogeneity in Temporal Interactions for Temporal Graph Link Prediction. *Advances in Neural Information Processing Systems* 38 (2025), 117623–117657.

[32] Jiasheng Zhang, Jialin Chen, Menglin Yang, Aosong Feng, Shuang Liang, Jie Shao, and Rex Ying. 2024. DTGB: A Comprehensive Benchmark for Dynamic Text-Attributed Graphs. *Advances in Neural Information Processing Systems* 37 (2024), 91405–91429.

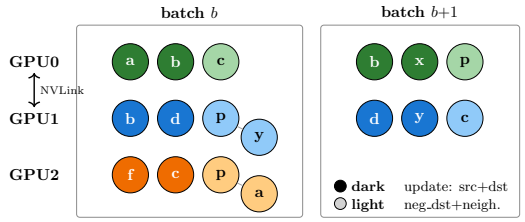
[33] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.

[34] 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.

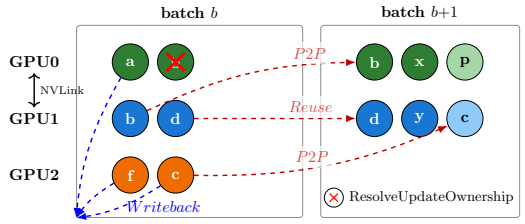
[35] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor Prasanna. 2023. DistTGL: Distributed Memory-Based Temporal Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[36] 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. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.

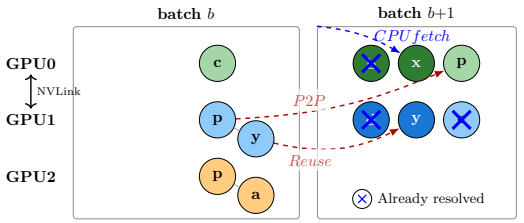
Received 17 April 2026; revised 19 August 2026; accepted 12 September 2026




(a) Setup.




(b) Updated-state route.




(c) Carry-over route.

Fig. 11. Toy example overview for the forward-looking dynamic state planner.

## A Toy Example: Forward-Looking Dynamic State Planner

We walk through Algorithm 1 on a concrete three-GPU example illustrated in Fig. 11. Table 4 summarizes the notation used throughout.

*Setup.* Fig. 11a shows the full node placement for batch $b$ and batch $b$+1. Dark nodes denote the source and destination nodes whose memory is updated by an interaction between them in this batch; light nodes denote negative destinations and sampled temporal neighbors, whose memory is *not* updated because no interaction occurs on them in this batch. For simplicity, some nodes have no historical neighbors, and GPU2 has no task at batch $b$+1.

Assume three GPUs where GPU0 and GPU1 are connected by NVLink, and GPU2 communicates with the others through PCIe. The resulting topology distance matrix is

$$D = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \\ 2 & 2 & 0 \end{pmatrix}.$$

The per-GPU updated-node sets ($U$) and all-node sets ($A$) for batch $b$ and batch $b$+1 are listed in Table 11.

*Step 1: Resolve update ownership and plan updated-state routes.* Fig. 11b visualizes the updated-state routing stage. Since updated-state routing only depends on currently updated nodes, the figure keeps only the dark nodes in batch $b$.

Table 11. Per-GPU node sets for batch $b$ and batch $b$+1.

| | batch $b$ | | batch $b$+1 | |
|---|---|---|---|---|
| | $U^g$ | $A^g$ | $U^g$ | $A^g$ |
| GPU0 | $\{a, b\}$ | $\{a, b, c\}$ | $\{b, x\}$ | $\{b, x, p\}$ |
| GPU1 | $\{b, d\}$ | $\{b, d, p, y\}$ | $\{d, y\}$ | $\{d, y, c\}$ |
| GPU2 | $\{f, c\}$ | $\{f, c, p, a\}$ | $\emptyset$ | $\emptyset$ |

We use the ownership rule that, if the same updated node appears on multiple GPUs, the larger GPU ID owns it. Since the three GPUs process events in chronological order—GPU0 handles the earliest events and GPU2 the latest—this rule ensures that the memory produced by the *most recent* interaction wins.

We first form

$$U_b^{\cup} = U_b^0 \cup U_b^1 \cup U_b^2 = \{a, b, c, d, f\}.$$

Node $b$ appears in both $U_b^0$ and $U_b^1$, so ownership is resolved in favor of GPU1. This is marked by the red cross on node $b$ in GPU0 in Fig. 11b. After ownership resolution, we have

$$\tilde{U}_b^0 = \{a\}, \qquad \tilde{U}_b^1 = \{b, d\}, \qquad \tilde{U}_b^2 = \{f, c\}.$$

Updated-state routing then forwards each next-batch demand that belongs to $U_b^{\cup}$ from its owner. For GPU0,

$$A_{b+1}^0 \cap U_b^{\cup} = \{b, x, p\} \cap \{a, b, c, d, f\} = \{b\},$$

so node $b$ is forwarded from GPU1 to GPU0:

$$1 \to 0 : \{b\}.$$

For GPU1,

$$A_{b+1}^1 \cap U_b^{\cup} = \{d, y, c\} \cap \{a, b, c, d, f\} = \{d, c\}.$$

Node $d$ is owned by GPU1 itself, yielding local updated-state reuse:

$$1 \to 1 : \{d\}.$$

Node $c$ is owned by GPU2, yielding updated-state P2P:

$$2 \to 1 : \{c\}.$$

Thus,

$$\mathcal{R}_{\text{dyn}}^{\text{upd}} = \{\, 1 \to 0 : \{b\},\ 1 \to 1 : \{d\},\ 2 \to 1 : \{c\} \,\}.$$

Fig. 11b uses blue arrows to indicate nodes that require write-back. A node whose state is updated in batch $b$ but will *not* be updated again in batch $b$+1 must be written back to CPU DRAM; otherwise its updated state will be lost if the node does not appear as an updated node in a future batch. Conversely, if a node will be updated again in batch $b$+1, its batch-$b$ state will be superseded by the new update, so no write-back is needed. The next-batch updated-node union is

$$U_{b+1}^{\cup} = \{b, x, d, y\},$$

so

$$\mathcal{P}_{\text{wb}}^0 = \tilde{U}_b^0 \setminus U_{b+1}^{\cup} = \{a\},$$
$$\mathcal{P}_{\text{wb}}^1 = \tilde{U}_b^1 \setminus U_{b+1}^{\cup} = \emptyset,$$
$$\mathcal{P}_{\text{wb}}^2 = \tilde{U}_b^2 \setminus U_{b+1}^{\cup} = \{f, c\}.$$

*Step 2: Plan carry-over routes for the unresolved demands.* Fig. 11c visualizes the carry-over routing stage. Next-batch demands already resolved by updated-state routing are masked out, which is why $b$, $d$, and $c$ in batch $b$+1 are crossed out in blue. The remaining demands can still benefit from GPU residency: if a non-updated node is already held by some GPU from an earlier batch, FROST reuses that GPU-resident copy via P2P instead of fetching it from CPU DRAM, reducing PCIe traffic.

We first compute the carry-over sets

$$C_b^g = A_b^g \setminus U_b^{\cup}.$$

Hence,

$$C_b^0 = \varnothing, \qquad C_b^1 = \{p, y\}, \qquad C_b^2 = \{p\}.$$

Next, we compute the unresolved next-batch demands after excluding all current updated nodes:

$$M_{b+1}^g = A_{b+1}^g \setminus U_b^{\cup}.$$

For GPU0,

$$M_{b+1}^0 = \{b, x, p\} \setminus \{a, b, c, d, f\} = \{x, p\}.$$

For GPU1,

$$M_{b+1}^1 = \{d, y, c\} \setminus \{a, b, c, d, f\} = \{y\}.$$

For GPU2,

$$M_{b+1}^2 = \varnothing.$$

For GPU0, node $p$ is present in the carry-over sets of both GPU1 and GPU2. The planner chooses the nearest carry-over holder under $D$, so $p$ is routed from GPU1 to GPU0:

$$1 \to 0 : \{p\}.$$

Node $x$ is unavailable from all carry-over sets, so it is deferred to CPU fetch:

$$F_{b+1}^0 = \{x\}.$$

For GPU1, node $y$ is already present in GPU1's own carry-over set, so it is satisfied by local carry-over reuse:

$$1 \to 1 : \{y\}, \qquad F_{b+1}^1 = \varnothing.$$

For GPU2,

$$F_{b+1}^2 = \varnothing.$$

Therefore,

$$\mathcal{R}_{\text{dyn}}^{\text{co}} = \{\, 1 \to 0 : \{p\},\ 1 \to 1 : \{y\} \,\}.$$

*Step 3: Final plan.* Combining the updated-state and carry-over routes, the final routing plan is

$$\mathcal{R}_{\text{dyn}} = \mathcal{R}_{\text{dyn}}^{\text{upd}} \cup \mathcal{R}_{\text{dyn}}^{\text{co}} = \{\, 1 \to 0 : \{b, p\},\ 1 \to 1 : \{d, y\},\ 2 \to 1 : \{c\} \,\}.$$

The sender-centric per-GPU plans are summarized in Table 12. The fetch component uses $F_b^g$ as determined by the previous planner invocation.

The next planner state persists

$$F_{b+1}^0 = \{x\}, \qquad F_{b+1}^1 = \varnothing, \qquad F_{b+1}^2 = \varnothing.$$

The planner achieves *optimal freshness*: for every GPU and every accessed node, the state read at batch $b$+1 is the most recently committed version in the system. No alternative routing plan can provide a fresher read.

Table 12. Sender-centric per-GPU execution plan.

| | Fetch | Reuse | P2P | Write-back |
|---|---|---|---|---|
| GPU0 | $F_b^0$ | $\varnothing$ | $\varnothing$ | $\{a\}$ |
| GPU1 | $F_b^1$ | $\{d, y\}$ | $1 \rightarrow 0$:$\{b, p\}$ | $\varnothing$ |
| GPU2 | $F_b^2$ | $\varnothing$ | $2 \rightarrow 1$:$\{c\}$ | $\{f, c\}$ |

*Message routing.* The walk-through above uses node memory as the running example, but each updated node also produces a new *message* that must reach the same set of consumers in batch $b$+1. For TGN/JODIE-style models, these messages share the same update and access pattern as the memory, so FROST routes them with the exact same planner and the same updated-state / carry-over decomposition—the toy example applies verbatim. APAN is the only exception: its post-interaction messages are fanned out to each root's sampled 1-hop neighbors, so the message update set is typically larger than the memory update set and requires a dedicated planner with mailbox-aware delivery. We defer the APAN message-routing details to Appendix B.4.

## B Implementation Details

In this section, we provide the low-level implementation details omitted from the main text. We first describe FROST's temporal-graph index, FlatGraph and HierGraph representations, and shared static-feature and dynamic-state storage (§B.1). We then explain how FROST realizes the staged pipeline with worker-side threads (§B.2), performs temporal graph sampling and accounts for the GPU interconnect topology (§B.3), and implements APAN's distinct message-delivery and mailbox semantics (§B.4).

### B.1 Data Structures

Section 4.1 introduces FROST's shared-memory host-worker path and the FlatGraph-HierGraph dual representation. Here, we provide additional details.

*Temporal graph index.* The temporal graph is stored as a CSR adjacency structure with arrays `indptr`, `indices`, `eid`, and `ts`. For each node $v$, `indptr`$[v]$ and `indptr`$[v+1]$ specify the start and end offsets of its adjacency list. At each offset, `indices`, `eid`, and `ts` store the neighbor ID, edge ID, and timestamp of the same interaction, respectively, with interactions ordered by timestamp. Neighbor queries are served by a compiled C++/nanobind extension that returns the $k$ neighbors before a given timestamp via binary search, yielding $O(\log |N(v)|)$ lookup time, where $N(v)$ is the set of neighbors of $v$.

*FlatGraph.* This is a flat, tensor-based representation of a sampled temporal subgraph. Root nodes occupy positions $[0, n_{\text{src}}(2 + n_{\text{neg}}))$ in the order [src | pos_dst | neg_dst], and hop-$k$ neighbors immediately follow hop-$(k-1)$ neighbors. A FlatGraph stores this layout in three aligned 1D tensors `nid`, `eid`, and `ts`. The batch metadata $(n_{\text{src}}, n_{\text{neg}})$, per-hop fanouts, and root-expansion rule determine the root and hop ranges. Each FlatGraph is stored in a fixed-capacity shared-memory circular buffer mapped directly by the workers.

*HierGraph.* This is the hierarchical view consumed by the model, exposing the roots and each sampled hop as separate tiers. Conversion from FlatGraph to HierGraph is zero-copy because `torch.split` partitions each row-aligned tensor into tier-specific views that share the original storage, requiring only lightweight metadata.

*Shared static-feature storage.* FROST loads the complete static node- and edge-feature tables once on the host and exposes them to all workers through shared memory; large node-feature tables may instead use a shared file-backed mapping. Each worker's CPU prefetch thread stages the required

node-feature rows and valid edge-feature rows. The H2D thread then transfers the staged rows to its GPU and scatters them into the corresponding row-aligned FlatGraph positions.

*Shared dynamic-state storage.* FROST initializes all tensors in the dynamic-state store to zero and resets them at the beginning of each epoch. Node memories and their last-update timestamps are stored in shared-memory tensors of shape $[n_{\text{nodes}}, d_{\text{mem}}]$ and $[n_{\text{nodes}}]$, respectively, and are accessible to all workers. The shared message store consists of three tensors: a message tensor of shape $[n_{\text{nodes}}, L_{\text{box}}, d_{\text{msg}}]$, a timestamp tensor of shape $[n_{\text{nodes}}, L_{\text{box}}]$, and a write-position vector of shape $[n_{\text{nodes}}]$, where $L_{\text{box}}$ is the mailbox capacity. For TGN and JODIE, $L_{\text{box}} = 1$, so the storage holds only the latest message per node. In APAN, $L_{\text{box}} > 1$, and the same storage becomes a mailbox ring buffer that retains historical messages received by each node.

### B.2 Pipeline Thread Details

Section 4.5 describes the logical pipeline stages. Here, we present the worker-level implementation details behind that staging.

Each GPU worker realizes the pipeline as four bounded producer-consumer stages: receiver, host-to-device, dynamic assembly (memory models only), and training. The dynamic-assembly stage fills the required dynamic-state fields in FlatGraph without converting it to HierGraph. The model first applies the memory updater on FlatGraph and only then unflattens it into a HierGraph for the rest of the forward and backward passes.

For memory-based models, a P2P background thread runs concurrently on dedicated CUDA streams. It packages updated memory and message tensors, posts a fixed sequence of all-to-all operations, and uses separate write-back paths for memory and messages. For models with an APAN-style multi-slot mailbox, the message path writes into the mailbox ring buffer.

Worker-side queue capacities are set to 1 for memory-based models and 2–4 for memoryless models. The larger capacity for memoryless models is possible because they have no cross-batch memory dependencies, allowing FROST to materialize several batches ahead of time and keep them ready for training. The deeper host-side scatter queue buffers sampled batches awaiting dispatch, allowing CPU sampling to overlap GPU computation. Routing decisions are produced on the host by the planner.

For models with APAN-style message delivery, dynamic states and node features are assembled only for root positions. Sampled neighbors intentionally remain zero-filled where the forward path does not consume them as root-side dynamic inputs.

### B.3 Host-Side Sampling and P2P Topology

In FROST, we implement host-side temporal sampling in C++/OpenMP to avoid two costly GPU-based alternatives: either each GPU must hold the full temporal graph in VRAM, increasing memory pressure, or the graph must be sharded across GPUs, requiring cross-GPU synchronization during sampling. By keeping the graph in CPU memory, the host-side sampler overlaps with GPU computation.

In addition, FROST queries the system's GPU topology during initialization to build an inter-GPU distance matrix, assigning smaller distances to NVLink-connected peers than to PCIe-connected ones. This matrix is currently used for optional carry-over routing. This topology awareness also provides a basis for future multi-server extensions, where GPU communication paths are highly heterogeneous and routing states from the closest available source can significantly reduce communication costs.

Table 13. Per-epoch stage-time breakdown (s).

| Model | $d_{\text{mem}}$ | Sample | H2D | | Training | | Write-back | Total |
|---|---|---|---|---|---|---|---|---|
| | | | Graph | DynState | Forward | Backward | | |
| TGN | 128 | 0.55 | 0.83 | 3.97 | 1.17 | 1.23 | 1.43 | 9.18 |
| | 256 | 0.53 | 0.85 | 7.23 | 1.65 | 1.95 | 3.02 | 15.26 |
| | 512 | 0.53 | 0.92 | 14.00 | 3.00 | 3.99 | 10.60 | 33.01 |
| JODIE | 128 | 0.00 | 0.14 | 1.49 | 0.31 | 0.42 | 1.10 | 3.46 |
| | 256 | 0.00 | 0.17 | 1.79 | 0.37 | 0.52 | 1.60 | 4.45 |
| | 512 | 0.00 | 0.18 | 5.58 | 0.68 | 0.98 | 3.76 | 11.19 |
| APAN | 128 | 0.37 | 0.14 | 21.54 | 1.57 | 2.56 | 1.91 | 28.12 |
| | 256 | 0.36 | 0.16 | 27.49 | 3.09 | 5.07 | 3.41 | 39.61 |
| | 512 | 0.38 | 0.15 | 48.19 | 7.22 | 13.38 | 6.76 | 76.01 |

### B.4 APAN Message and Mailbox

Section 4.4 outlines FROST's dedicated routing path. Here, we describe the implementation details of neighbor message fan-out, message planning, and mailbox merging for APAN.

*Message delivery.* The message delivery step expands only the positive-root portion (src + pos_dst) to neighbors. Root nodes are not self-delivered, and their own dynamic states are handled by the standard memory-update path.

*Dedicated message planner.* Unlike TGN and JODIE, APAN propagates information via message delivery, resulting in a different mailbox update rule. We therefore use a dedicated message planner for APAN. The key differences are: (i) APAN analyzer uses all roots as the message fetch set, while message updates are 1-hop neighbors of positive roots only; (ii) APAN message planner, like the generic dynamic planner, plans sender-centric routing for message updates, but it also writes back message updates to CPU memory.

*Single-slot mailbox merge.* APAN's mailbox is a ring buffer with $L_{\text{box}}$ slots. Sending the full ring buffer through direct inter-GPU transfer would multiply traffic by $L_{\text{box}}$. Instead, the P2P path transmits only a single new message slot per node. The dynamic state assembler preloads the old ring-buffer contents and inserts each received message into the corresponding slot. This reduces per-node mailbox-related P2P traffic from $L_{\text{box}} \cdot d_{\text{msg}}$ to $d_{\text{msg}}$.

## C Training Time Breakdown and P2P Communication Analysis

In this section, we examine whether FROST can hide dynamic-state peer-to-peer (P2P) communication (i.e., inter-GPU data transfer) behind computation. We first report the per-epoch stage-time breakdown (§C.1). We then quantify the GPU-fabric idle window for P2P communication (§C.2) and measure the gradient-synchronization delay introduced by the P2P guard-wait overhead (§C.3).

### C.1 Per-Epoch Stage-Time Breakdown

Section 3.2 identifies serialized CPU write-back followed by host-to-device (H2D) dynamic-state fetching as a major synchronization bottleneck. In Table 13, we provide the full per-epoch stage-time breakdown to show this is indeed the case. The measurements here are collected without pipelining, with an explicit GPU synchronization barrier at each stage boundary, so they include host-side control, kernel launches, data movement, and completion of asynchronous GPU work. As shown in Table 13, H2D dynamic-state fetching and CPU write-back are time-consuming for all configurations, and both costs increase with $d_{\text{mem}}$. The high cost of these stages motivates us to hide their latency through pipeline overlap.

Table 14. Device-side training time breakdown and GPU-fabric idle window (ms).

| Model | $d_{\text{mem}}$ | Forward | | | Backward | | Total | Idle |
|---|---|---|---|---|---|---|---|---|
| | | Mem. Upd. | ① Encoder | ② Predictor | ③ Pre-sync | Post-sync | | ≈①+②+③ |
| TGN | 128 | 3.55 | 6.75 | 0.76 | 7.22 | 1.73 | 20.56 | 15.27 |
| | 256 | 6.22 | 8.72 | 0.75 | 4.78 | 6.31 | 27.34 | 14.80 |
| | 512 | 12.04 | 8.44 | 0.59 | 4.91 | 15.18 | 41.68 | 14.47 |
| JODIE | 128 | 1.59 | 0.69 | 0.81 | 2.04 | 1.16 | 6.74 | 3.99 |
| | 256 | 2.33 | 0.95 | 1.42 | 2.56 | 1.01 | 8.87 | 5.54 |
| | 512 | 2.95 | 0.78 | 1.53 | 2.11 | 4.72 | 12.72 | 5.06 |
| APAN | 128 | 5.39 | 0.24 | 1.15 | 10.01 | 1.99 | 19.31 | 11.93 |
| | 256 | 9.43 | 0.06 | 0.39 | 8.38 | 8.40 | 26.94 | 9.11 |
| | 512 | 20.46 | 0.08 | 0.59 | 0.89 | 36.70 | 58.95 | 1.79 |

### C.2 GPU-Fabric Idle Window

Section 4.2 identifies the interval from the end of the memory updater to the first gradient synchronization as a GPU-fabric idle window that FROST leverages for direct P2P dynamic-state transfers. Here, we quantify this window with CUDA events on the execution stream. Since the memory updater finishes early in the forward pass, this GPU-fabric idle window spans the subsequent encoder and predictor computations, as well as the early portion of the backward pass before the first gradient collective.

Table 14 reports the details of the device-side training time measured on rank 0. The *Total* column denotes the device-side training time (i.e., the sum of the forward and backward passes). *Mem. upd.*, *Encoder*, and *Predictor* are major forward-pass components, while *Pre-sync* covers the portion of the backward pass before the first gradient synchronization. Since the directly measured GPU-fabric idle window starts when the memory update completes and ends when the first gradient synchronization begins, it is approximately *Encoder* plus *Predictor* plus *Pre-sync*. As shown in Table 14, every configuration has a nonzero GPU-fabric idle window during which direct P2P dynamic-state transfers can overlap with computation.

### C.3 P2P Guard-Wait Overhead

To quantify the delay introduced by FROST's semaphore guard before the first gradient synchronization (see §4.2), we instrument TGN on LastFM using two H200 GPUs and a macro-batch size of 4,000 events. Each routing round posts six asynchronous all-to-all operations. The first three exchange the node IDs, updated values, and timestamps for memory state, and the remaining three exchange the node IDs, new message vectors, and timestamps for mailbox state. FROST's DDP communication hook waits on a semaphore before posting the first gradient collective, and the P2P thread releases it after all six calls are posted. We measure the semaphore wait time from the hook's acquire call until it succeeds.

Table 15 compares this guard-wait time with the serialized dynamic-state stage times reported in Table 13. These values are measured as per-epoch times averaged over five epochs. For non-pipelined execution, the fractions represent the combined time of *H2D: DynState* and *Write-back* relative to the non-pipelined epoch time. For FROST, they represent the guard-wait time relative to the steady-state pipelined epoch time. We focus on $d_{\text{mem}} = 128$ as a representative setting, since memory dimensions of 100 [9], 128 [16], and 172 [22] are widely used in memory-based TGNNs. In this setting, the P2P thread releases the semaphore before the hook attempts to acquire it, making the guard-wait time negligible. With larger memory dimensions, however, the hook can reach the

Table 15. Serialized state handling vs. P2P guard waiting.

| | Metric | $d_{\text{mem}}$ | | |
|---|---|---|---|---|
| | | 128 | 256 | 512 |
| Non-pipelined | H2D: DynState + Write-back | 5.40 s | 10.25 s | 24.60 s |
| | Fraction of epoch time | 58.80% | 67.20% | 74.50% |
| FROST | Guard wait | 0.75 ms | 0.71 s | 2.37 s |
| | Fraction of epoch time | 0.04% | 20.93% | 29.92% |

guard before all six calls are posted, increasing the guard latency. Nonetheless, for every memory dimension, the guard-wait time accounts for a substantially smaller fraction of the end-to-end training time than the serialized H2D dynamic-state fetching and CPU write-back. Therefore, FROST successfully hides a large fraction of the original serialized latency.

In addition, to examine how workload imbalance across GPUs affects the guard-wait overhead, we profile the same two-GPU TGN-LastFM configuration with $d_{\text{mem}} = 128$. For each routing round, we quantify workload imbalance as $S = \max(N_{0\to1}, N_{1\to0})/(N_{0\to1} + N_{1\to0})$, where $N_{i\to j}$ is the number of node-state entries routed from GPU $i$ to GPU $j$. While perfect balance corresponds to $S = 50\%$, we observe that $S$ has a mean of 81.7%, a 95th percentile of 86.0%, and a maximum of 90.1% over five epochs. Despite this pronounced routing skew, the measured guard-wait time remains negligible at this small memory dimension, showing that FROST hides the P2P overhead in the current setting. It is worth noting that for larger datasets or communication volumes, skew-aware reordering of events within each macro-batch before per-GPU assignment could distribute routed states more evenly across GPU pairs and potentially hide a greater fraction of the P2P overhead. We leave such in-macro-batch event reordering to future work.

## D Tensor-Based Execution and Data Transfer

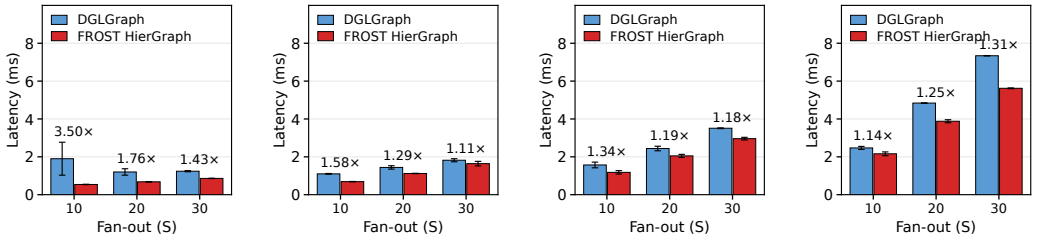
In this section, we examine how FROST's tensor-based graph representations reduce computation and data-transfer overheads. We first compare dense-tensor execution with DGL-style sparse message passing (§D.1). We then quantify the inter-process communication (IPC) scatter savings enabled by the shared-memory FlatGraph layout (§D.2) and discuss compiler compatibility (§D.3).

### D.1 Dense Tensor Execution vs. DGL-Style Sparse Message Passing

To separate framework overhead from model computation, we benchmark two *functionally equivalent* implementations of the TGN temporal attention layer: (i) the TGL version built on the Deep Graph Library (DGL)'s sparse graph blocks [26], and (ii) the FROST version built on dense tensors. The dense layout uses tensors of shape $[N, S, D]$—where $N$ is the number of target nodes, $S$ is the per-target fan-out, and $D$ is the combined feature width (node + edge + time encoding)—and computes attention with standard tensor operations.

*Setup.* Both layers use identical hyperparameters: $d_{\text{node}} = d_{\text{edge}} = 172$, $d_{\text{time}} = 100$, $d_{\text{out}} = 100$, 2 attention heads, and dropout 0.1. For the DGL variant, we use the standard bipartite blocks whose source set contains the $N$ destination nodes together with their $N \cdot S$ sampled neighbors. For the dense variant, we pack the same data into contiguous tensors of $[N, S, D]$. We sweep twelve $(N, S)$ configurations with $N \in \{500, 1000, 2000, 4000\}$ and $S \in \{10, 20, 30\}$. Each configuration is warmed up for 50 iterations. We then report the mean and standard deviation of the combined forward and backward latency over 100 timed iterations.

*Results.* Fig. 12 summarizes the measurements. Across all configurations, the dense-tensor path is consistently faster than the DGL sparse path, with speedups ranging from 1.1× to 3.5×. The




(a) $N = 500$ (b) $N = 1000$ (c) $N = 2000$ (d) $N = 4000$

Fig. 12. Forward and backward latency of DGLGraph vs. FROST HierGraph.

Table 16. Host-side IPC scatter time on TGN-LastFM.

| BS | #Batches | TGL (s) | FROST (s) | Reduction |
|---|---|---|---|---|
| 500 | 2587 | 9.25 | 3.57 | 61% |
| 1000 | 1294 | 5.41 | 1.80 | 67% |
| 2000 | 647 | 3.31 | 0.88 | 73% |
| 4000 | 325 | 2.41 | 0.46 | 81% |

largest gains appear at smaller fan-outs ($S$), where graph-framework dispatch overhead dominates actual computation. As $S$ grows, both implementations spend a larger fraction of time on useful arithmetic operations, and the gap narrows. Nevertheless, the dense representation remains faster in all twelve settings. Note that the DGL sparse implementation exhibits non-monotonic latency at small $(N, S)$, which we attribute to kernel-launch and memory-allocator warm-up effects in DGL's C++/FFI layer that are amortized at larger problem sizes.

### D.2 IPC Scatter

General-purpose graph frameworks such as DGL [26] and PyG [6, 7] expose rich graph objects for model construction. For cross-process batch dispatch, however, these abstractions are heavier than a flat tensor buffer. The graph topology and attached data must move as structured objects rather than as a single contiguous byte layout. Transferring such objects across process boundaries therefore relies on generic Python object serialization and incurs substantial CPU overhead. FROST's FlatGraph is instead a contiguous byte buffer that can be written directly into shared memory and read by the consumer without serialization or deserialization.

We benchmark the host-side *object-scatter* overhead when using TGN-LastFM on an NVLink-connected A5000 server with two GPUs for one epoch, with $n_{\text{neg}}$=1 and fanout=10. FROST scatters only lightweight per-worker metadata and execution plans, while graph data stays in shared memory. TGL must scatter serialized DGL blocks plus auxiliary root/timestamp/edge-id arrays. Table 16 reports total scatter time over one epoch. Here, batch sizes 500–4000 correspond to per-worker mini-batches 250–2000. FROST consistently reduces scatter time, saving 61–81% against the full TGL payload. The advantage widens at larger batch sizes because TGL's serialized payload grows with the sampled graph, whereas FROST's metadata payload stays compact and the bulk data bypasses Python serialization entirely through shared memory.

### D.3 Compiler Compatibility

*DGL-based frameworks.* We attempted to compile a DGL-based TGN implementation with the Inductor backend (PyTorch 2.2, Triton 2.2) under both DGL 1.x and DGL 2.x. However, end-to-end compiler capture failed on the first training batch in both cases. DGL's topology-dependent

Table 17. Batch-Size and Multi-GPU Scaling Settings.

| Setting | $n_{\text{GPU}}$ | $b_{\text{mini}}$ | $B_{\text{macro}} = n_{\text{GPU}} \times b_{\text{mini}}$ |
|---|---|---|---|
| S1 | **2 (fixed)** | $1000 \rightarrow 2000 \rightarrow 4000$ | $2000 \rightarrow 4000 \rightarrow 8000$ |
| S2 | $1 \rightarrow 2 \rightarrow 4$ | **2000 (fixed)** | $2000 \rightarrow 4000 \rightarrow 8000$ |
| S3 | $1 \rightarrow 2 \rightarrow 4$ | $4000 \rightarrow 2000 \rightarrow 1000$ | **4000 (fixed)** |

operations run through opaque C++/FFI calls, preventing `torch.compile` from determining their output tensor shapes and capturing the entire forward pass for compilation. Most multi-GPU TGNN acceleration frameworks (TGL [34], DistTGL [35], MSPipe [23], and PipeTGL [19]) are built on DGL and inherit the same incompatibility. Although recent PyTorch Geometric releases add support for this `torch.compile` path, none of these distributed TGNN runtimes use it as their graph backend.

*FROST's HierGraph path.* FROST's HierGraph path uses only native PyTorch dense-tensor operators, with no graph-framework primitives in the forward pass, so the compiler can capture the entire trace. This compatibility follows directly from the dense-tensor design validated in Section D.1. Because FROST never invokes graph-framework primitives during training, the full forward and backward passes can be captured and compiled end to end without trace interruptions.

## E Batch-Size and Multi-GPU Scaling Analysis

To further evaluate the scalability of FROST with respect to batch size and GPU count, we consider the three settings summarized in Table 17. These settings systematically vary the per-GPU mini-batch size, macro-batch size, and GPU count, while all intersecting at a baseline configuration of two GPUs with a 2,000-event per-GPU mini-batch. We measure GPU utilization, GPU memory usage, throughput, and test AP on H200 GPUs. Specifically, utilization is measured from samples taken every 50 ms over 500 post-warm-up intervals, averaged across participating GPUs and five seeds. Memory usage is reported as the mean PyTorch reserved memory per GPU (GiB). Tables 18 and 19 report the full results. We evaluate TGN (§E.1) and DyGFormer (§E.2), and then analyze temporal fidelity under S1 and S3 (§E.3).

Two-GPU TGN throughput entries are three-run medians from a separate H200 SXM server; all other entries retain the original measurements.

### E.1 Multi-GPU Scaling: Memory-Based Models

In S1, increasing the per-GPU mini-batch size from 1,000 to 4,000 events improves per-GPU throughput from 108.0k to 303.3k ev/s/GPU (2.81×) and GPU utilization from 70.9% to 79.7%. This gain comes with a larger memory footprint (0.96 to 3.16 GiB/GPU) and a decrease of 1.76 percentage points in test AP (86.25% to 84.49%) as the macro-batch grows from 2,000 to 8,000 events.

In S2, increasing the GPU count from one to four at a fixed 2,000-event per-GPU mini-batch raises aggregate throughput from 212.5k to 778.1k ev/s, a 3.66× increase (91.5% of ideal four-GPU scaling). Per-GPU throughput nevertheless decreases by 8.5%, from 212.5k to 194.5k ev/s/GPU, while GPU utilization decreases from 78.4% to 72.3%. As the macro-batch simultaneously grows from 2,000 to 8,000 events, test AP decreases by 1.87 percentage points, from 86.55% to 84.68%.

S3 isolates the effect of adding GPUs by keeping the macro-batch fixed at 4,000 events. Aggregate throughput increases from 305.2k to 431.2k ev/s (1.41×) from one to four GPUs, but per-GPU throughput falls from 305.2k to 107.8k ev/s/GPU as the per-GPU mini-batch shrinks. Test AP remains within a narrow 85.17%–85.32% range.

Overall, a larger per-GPU mini-batch improves individual GPU throughput, while S2 and S3 demonstrate that scaling the GPU count boosts aggregate throughput. When the macro-batch size is fixed, varying the GPU count has little effect on test AP. Thus, the observed AP decrease is

Table 18. Multi-GPU Scaling Analysis on TGN.

| Setting | GPU count | Batch size | | GPU | | Throughput | | Test AP (%) |
|---|---|---|---|---|---|---|---|---|
| | | Per-GPU | Total | Utilization | Memory | Per-GPU | Total | |
| S1 | 2 | 1,000 | 2,000 | 70.9% | 0.96 GiB | 108.0 k ev/s | 216.1 k ev/s | 86.25 ± 0.52 |
| | 2 | 2,000 | 4,000 | 77.9% | 1.72 GiB | 195.9 k ev/s | 391.8 k ev/s | 85.32 ± 0.60 |
| | 2 | 4,000 | 8,000 | 79.7% | 3.16 GiB | 303.3 k ev/s | 606.5 k ev/s | 84.49 ± 0.79 |
| S2 | 1 | 2,000 | 2,000 | 78.4% | 1.61 GiB | 212.5 k ev/s | 212.5 k ev/s | 86.55 ± 0.86 |
| | 2 | 2,000 | 4,000 | 77.9% | 1.72 GiB | 195.9 k ev/s | 391.8 k ev/s | 85.32 ± 0.60 |
| | 4 | 2,000 | 8,000 | 72.3% | 1.76 GiB | 194.5 k ev/s | 778.1 k ev/s | 84.68 ± 0.76 |
| S3 | 1 | 4,000 | 4,000 | 84.1% | 3.02 GiB | 305.2 k ev/s | 305.2 k ev/s | 85.28 ± 0.42 |
| | 2 | 2,000 | 4,000 | 77.9% | 1.72 GiB | 195.9 k ev/s | 391.8 k ev/s | 85.32 ± 0.60 |
| | 4 | 1,000 | 4,000 | 68.1% | 0.97 GiB | 107.8 k ev/s | 431.2 k ev/s | 85.17 ± 0.64 |

Table 19. Multi-GPU Scaling Analysis on DyGFormer.

| Setting | GPU count | Batch size | | GPU | | Throughput | | Test AP (%) |
|---|---|---|---|---|---|---|---|---|
| | | Per-GPU | Total | Utilization | Memory | Per-GPU | Total | |
| S1 | 2 | 1,000 | 2,000 | 98.4% | 7.39 GiB | 10.7 k ev/s | 21.4 k ev/s | 94.05 ± 0.06 |
| | 2 | 2,000 | 4,000 | 98.9% | 14.59 GiB | 11.2 k ev/s | 22.4 k ev/s | 93.92 ± 0.02 |
| | 2 | 4,000 | 8,000 | 99.0% | 29.16 GiB | 11.3 k ev/s | 22.6 k ev/s | 93.61 ± 0.06 |
| S2 | 1 | 2,000 | 2,000 | 98.9% | 14.59 GiB | 11.3 k ev/s | 11.3 k ev/s | 94.05 ± 0.05 |
| | 2 | 2,000 | 4,000 | 98.9% | 14.59 GiB | 11.2 k ev/s | 22.4 k ev/s | 93.92 ± 0.02 |
| | 4 | 2,000 | 8,000 | 98.3% | 14.59 GiB | 11.1 k ev/s | 44.5 k ev/s | 93.60 ± 0.07 |
| S3 | 1 | 4,000 | 4,000 | 98.5% | 29.16 GiB | 11.2 k ev/s | 11.2 k ev/s | 93.93 ± 0.05 |
| | 2 | 2,000 | 4,000 | 98.9% | 14.59 GiB | 11.2 k ev/s | 22.4 k ev/s | 93.92 ± 0.02 |
| | 4 | 1,000 | 4,000 | 98.0% | 7.39 GiB | 10.6 k ev/s | 42.5 k ev/s | 93.92 ± 0.04 |

associated with macro-batch growth rather than GPU count alone. A larger macro-batch places more temporally dependent events within the same macro-batch, where intermediate memory updates are not visible. This prevents the model from capturing those in-batch memory dependencies and therefore reduces test AP.

### E.2 Multi-GPU Scaling: Memoryless Models

In S1, DyGFormer is already compute-saturated at the smallest configuration. GPU utilization remains above 98% as the per-GPU mini-batch grows from 1,000 to 4,000 events. Aggregate throughput therefore improves by only 5.6%, from 21.4k to 22.6k ev/s, while GPU memory usage grows from 7.39 to 29.16 GiB/GPU and test AP decreases by 0.44 percentage points.

In S2, increasing the GPU count from one to four at a fixed 2,000-event per-GPU mini-batch raises aggregate throughput from 11.3k to 44.5k ev/s (3.94×, or 98.5% of ideal four-GPU scaling). Per-GPU throughput, GPU utilization, and GPU memory usage remain nearly constant, while test AP decreases by 0.45 percentage points as the macro-batch grows from 2,000 to 8,000 events.

In S3, the macro-batch size remains fixed as the GPU count increases from one to four. Aggregate throughput increases from 11.2k to 42.5k ev/s (3.79×), while test AP remains within 93.92%–93.93%. Dividing the fixed workload across GPUs also reduces per-GPU memory usage from 29.16 to 7.39 GiB.

Overall, DyGFormer is compute-saturated, so larger per-GPU mini-batches offer little throughput benefit and primarily increase GPU memory usage. Adding GPUs provides near-linear aggregate scaling in both S2 and S3. At a fixed macro-batch size, GPU count has negligible effect on test AP. When the macro-batch grows, test AP decreases only slightly, indicating lower sensitivity than TGN.

### E.3 Macro-Batch Size and Temporal Fidelity

For memory-based models, FROST's $\sigma(v, b) = 0$ guarantee ensures that macro-batch $b$ reads the latest dynamic state committed by all preceding macro-batches, but it does not make intermediate memory updates visible among events within the same macro-batch. With a larger macro-batch, updates are committed less frequently, resulting in coarser temporal granularity. This loss of granularity arises from macro-batched training rather than multi-GPU execution. A single-GPU run with the same macro-batch size has the same limitation.

S1 quantifies the resulting accuracy sensitivity. With two GPUs, increasing the per-GPU mini-batch size from 1,000 to 4,000 events grows the macro-batch from 2,000 to 8,000 events, while TGN's test AP decreases from 86.25% to 84.49%. In contrast, S3 keeps the macro-batch size fixed while varying the GPU count. At a matched global macro-batch size, FROST provides the same temporal fidelity as single-GPU execution, and test AP remains nearly unchanged. Together, S1 and S3 show that temporal fidelity depends on the global macro-batch size rather than how each macro-batch is partitioned across GPUs.

## F TGB Benchmark and MRR

The Temporal Graph Benchmark (TGB) was introduced by Huang et al. [11] as an open-source evaluation suite that provides a collection of temporal graph datasets with new evaluation protocols. To demonstrate that FROST is also compatible with TGB's mean reciprocal rank (MRR) evaluation protocol, we evaluate it under TGB's official one-vs-many link-prediction setting. We first introduce the datasets and evaluation setup (§F.1). We then compare the test MRR of TGL, TGB, and FROST using TGN, and discuss their performance differences (§F.2).

### F.1 Datasets and Evaluation Protocol

We use three TGB datasets: tgbl-wiki-v2, tgbl-review-v2, and tgbl-coin. The former two datasets are bipartite user-item interaction datasets, whereas the last one is a cryptocurrency transaction dataset. Note that tgbl-wiki-v2 contains the same interaction history and features as the Wiki dataset used in the main experiments (Table 5). However, they differ in their evaluation protocols. In the main experiments, the positive-to-negative destination ratio is one-to-one and AP is used as the metric, while here we follow TGB's official one-vs-many candidate sets and report MRR.

Table 21 summarizes the evaluation protocols and models supported in existing single-server systems. We select TGL, TGB, and FROST for the MRR evaluation of TGN on TGB datasets, as the other systems support neither one-positive-to-many-negative destination sampling nor MRR reporting. TGN is the only model supported by all three systems. To remain consistent with the original TGB setup, all three systems use a batch size of 200 and one sampled negative per positive during training. They also use identical TGN dimensions, the edge features provided by each dataset, TGB's fixed candidate sets, and its tie-aware MRR definition. However, since the official TGB implementation has no multi-GPU support, we evaluate all three systems on a single GPU, with the edge features and dynamic memory resident in GPU memory. Note that for TGL, since it natively reports MRR with randomly sampled negatives, we modified it to support TGB's fixed-negative candidates.

Table 20. Test MRR of TGN on TGB datasets.

| Dataset | TGB | TGL | FROST |
|---|---|---|---|
| tgbl-wiki-v2 | 0.517 ± 0.023 | 0.684 ± 0.011 | 0.697 ± 0.017 |
| tgbl-review-v2 | 0.397 ± 0.007 | 0.403 ± 0.005 | 0.403 ± 0.007 |
| tgbl-coin | 0.760 ± 0.007 | 0.764 ± 0.008 | 0.768 ± 0.004 |

Table 21. Metrics and model support of TGNN systems.

| Paper | System | 1:*N* pos.:neg sampling | Metric | | Supported TGNN models |
|---|---|---|---|---|---|
| | | | AP | MRR | |
| [34] | TGL | ✓ | ✓ | ✓ | TGN, JODIE, APAN, TGAT |
| [30] | DyGLib | ✗ | ✓ | ✗ | TGAT, CAWN, TCL, GraphMixer, DyGFormer |
| [11] | TGB | ✓ | ✗ | ✓ | TGN, DyRep |
| [23] | MSPipe | ✗ | ✓ | ✗ | TGN, JODIE, APAN |
| [19] | PipeTGL | ✗ | ✓ | ✗ | TGN |
| | **FROST** | ✓ | ✓ | ✓ | TGN, JODIE, APAN, TGAT, CAWN, TCL, GraphMixer, DyGFormer |

### F.2 MRR Results and Analysis

Table 20 reports the test MRR of TGL, TGB, and FROST using TGN on the three TGB datasets. During testing, each positive is ranked against TGB's fixed negatives. The three datasets, in table order, use 999, 100, and 20 negatives, respectively. All results are computed over five seeds, with up to 30 epochs and early stopping. As shown in Table 20, FROST and TGL obtain similar MRR on all three datasets, whereas TGB's MRR is consistently lower. The difference is small on tgbl-review-v2 and tgbl-coin but unusually large on tgbl-wiki-v2.

The performance gap exposes an important implementation distinction. The original TGN implementation [22] queries each root's temporal neighborhood at its own event timestamp, allowing each query to access strictly earlier interactions within the same batch. In contrast, in TGB's implementation [11], occurrences of the same node at different event timestamps within a batch share the same temporal neighborhood. During training, the TGB implementation deduplicates roots solely by node ID, so these occurrences also share one batch-start embedding. However, like the original TGN implementation, FROST and TGL retain the timestamp of each query occurrence and separately sample and encode its temporal neighborhood.

This distinction is especially consequential on tgbl-wiki-v2. Each full 200-query batch contains about 200,200 root occurrences after expanding 999 negatives per positive. The median counts are only 1,082 unique node IDs but 195,200 unique pairs of node ID and timestamp across the full test batches. Thus, TGB's node-based deduplication reduces the number of computed temporal embeddings by approximately 185×, whereas TGN, TGL, and FROST separately compute embeddings for distinct combinations of node ID and timestamp. This loss of within-batch temporal variation may contribute to TGB's lower MRR. In contrast, TGL and FROST preserve this temporal variation, which helps maintain higher MRR scores.

## G TPNet Compatibility

While TPNet [20] achieves state-of-the-art performance on TGB link prediction datasets, it has a unique architecture, which is substantially different from memory-based as well as memory-less models. Nonetheless, to extend FROST's model coverage, we integrate TPNet through FROST's

Table 22. TPNet Speedup Linearity and Test AP on FROST.

| System | GPU count | Metric | Mini-batch size | | | |
|---|---|---|---|---|---|---|
| | | | 200 | 600 | 1,200 | 2,000 |
| Original | 1 | Throughput | 1.17 k ev/s | 2.47 k ev/s | 2.98 k ev/s | 3.27 k ev/s |
| | | Test AP (%) | 93.13 | 92.68 | 91.64 | 90.47 |
| FROST | 1 | Throughput | 1.41 k ev/s (1.21×) | 2.39 k ev/s (0.97×) | 2.99 k ev/s (1.00×) | 3.12 k ev/s (0.95×) |
| | | Test AP (%) | 93.13 | 92.68 | 91.64 | 90.47 |
| FROST | 2 | Throughput | 2.38 k ev/s (2.03×) | 4.38 k ev/s (1.77×) | 5.52 k ev/s (1.85×) | 6.06 k ev/s (1.85×) |
| | | Test AP (%) | 92.95 | 91.65 | 89.94 | 87.82 |
| FROST | 4 | Throughput | 3.68 k ev/s (3.14×) | 7.25 k ev/s (2.93×) | 9.49 k ev/s (3.18×) | 10.41 k ev/s (3.18×) |
| | | Test AP (%) | 92.27 | 89.89 | 87.29 | 84.96 |

unified external-model launch interface. It is worth noting that TPNet maintains layered random-projection matrices whose update rules cannot be directly represented by FROST's current node-memory abstraction. For each batch, TPNet first decays all rows in every higher-layer projection matrix. It then updates each endpoint of an interaction using the other endpoint's lower-layer projection. However, FROST's memory updater supports memory-based models, which directly update only root nodes, while other nodes receive information through the message mechanism. Thus, supporting TPNet natively within FROST's routing mechanism would require extending FROST's memory-update abstraction to efficiently handle global decay.

Because of this, while our current implementation preserves TPNet's original state transitions, it does not use FROST's pipelined execution but instead enables TPNet to run across multiple GPUs. The resulting aggregate-throughput gains therefore come from multi-GPU scaling rather than pipeline overlap. We leave native integration into FROST's routing mechanism to future work.

Table 22 reports end-to-end total throughput and test AP on LastFM for the original TPNet codebase and FROST's integration on one, two, and four H200 GPUs. Total throughput is summed across all participating GPUs. The throughput runs have random projection enabled and use up to 100 full training batches. Test AP is evaluated using TPNet's protocol with a fixed 2,000-event evaluation batch. Speedups reported in Table 22 are relative to the original TPNet codebase at the same mini-batch size.

For throughput, FROST's two-GPU aggregate throughput is 1.77×–2.03× that of the original TPNet codebase across the four batch sizes. Four GPUs reach 3.68k–10.41k ev/s, or 2.93×–3.18× the original TPNet codebase throughput. Relative to two GPUs, this is a 1.54×–1.72× gain, corresponding to 77.2%–86.0% incremental efficiency. Thus, FROST achieves multi-GPU throughput scaling for TPNet, especially at larger batch sizes, although four-GPU scaling remains sublinear.

For test AP, performance decreases for both the original TPNet codebase and FROST's integration as the macro-batch size (i.e., mini-batch size × GPU count) increases. At matched macro-batch sizes, the two implementations achieve nearly identical AP. Within FROST, changing the GPU count while holding the macro-batch size fixed also leaves AP nearly unchanged. These results indicate that AP is governed primarily by the macro-batch size rather than the GPU count.