TGLite: A Lightweight Programming Framework for Continuous-Time Temporal Graph Neural Networks

Yufeng Wang
University of Illinois at Urbana-Champaign, USA
yufengw2@illinois.edu

Charith Mendis
University of Illinois at Urbana-Champaign, USA
charithm@illinois.edu

Abstract

In recent years, Temporal Graph Neural Networks (TGNNs) have achieved great success in learning tasks for graphs that change over time. These dynamic/temporal graphs represent topology changes as either discrete static graph snapshots (called DTDGs), or a continuous stream of timestamped edges (called CTDGs). Because continuous-time graphs have richer time information, it will be crucial to have abstractions for programming CTDG-based models so that practitioners can easily explore new designs and optimizations in this space. A few recent frameworks have been proposed for programming and accelerating TGNN models, but these either do not support continuous-time graphs, lack easy composability, and/or do not facilitate CTDG-specific optimizations.

In this paper, we propose a lightweight framework called TGLite to fill this apparent gap in the status quo. It provides abstractions that serve as composable building blocks for implementing TGNN models for CTDGs. It introduces a novel `TBlock` representation for capturing message-flow dependencies between nodes, with explicit support for temporal-related attributes, which is well-suited for common TGNN computation patterns. `TBlocks` serve as a central representation on which many different operators can be defined, such as temporal neighborhood sampling, scatter/segmented computations, as well as optimizations tailored to CTDGs. We use TGLite to implement four existing TGNN models. Compared to the TGL framework, TGLite is able to accelerate runtime performance of training (1.06 – 3.43×) and inference (1.09 – 4.65×) of these models on V100 and A100 GPUs across different experimental settings. Notably, when scaling to larger datasets, TGL runs out-of-memory in some cases on the V100 while TGLite is able to run successfully.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software libraries and repositories**.

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640414

Keywords: Temporal Graph Neural Networks, Dynamic Graphs, Programming Framework, Data Abstractions

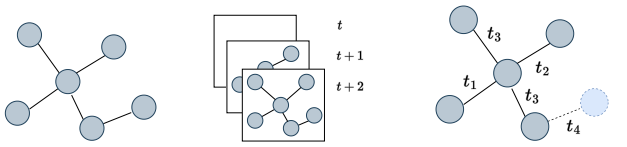
ACM Reference Format:
Yufeng Wang and Charith Mendis. 2024. TGLite: A Lightweight Programming Framework for Continuous-Time Temporal Graph Neural Networks. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640414

1 Introduction

Graph Neural Networks (GNNs) [22] have emerged as a powerful paradigm for representation learning and predictive modeling on graph-structured data across various problem domains [2, 8, 25, 27, 35]. Following its rise in the research community, several frameworks have been established for programming and optimizing GNN models, such as DGL [28], PyG [3], and NeuGraph [16]. These frameworks are a boon to developers and researchers for implementing and exploring different GNN architectures and optimizations. However, these existing works are designed solely for GNNs that operate on static graphs, while real-world graphs are often dynamic and change their topologies over time (e.g. social networks with new users [19], financial networks with new transactions [30], knowledge graphs with temporally evolving facts [29]).

Recently, Temporal Graph Neural Networks (Temporal GNNs, TGNNs) have been proposed by researchers for dynamic/temporal graph learning, such as TGAT [34] and TGN [19], among others [11, 17, 21, 30]. Like static GNNs, a major computation that these models perform is aggregating information from neighbors. But unlike their counterparts, TGNNs jointly learn on structural graph information (e.g. neighbor node features) and temporal information (e.g. edge timestamps) to encode the dynamic nature of the graph into time-dependent node embedding vectors [37]. The additional time information allows TGNNs to outperform static GNNs on many temporal graph prediction and classification tasks.

As Figure 1 shows, temporal graphs are categorized into two common types [9]: discrete-time graphs with static graph snapshots (DTDGs), and continuous-time graphs with edge timestamps (CTDGs). Compared to DTDGs, CTDGs capture richer time information and are suitable for real-world settings where edges and nodes can continuously appear at



(a) Static. **(b)** Discrete-time. **(c)** Continuous-time.

Figure 1. Representation of the same graph: (a) without temporal information, (b) as discrete snapshots over several time steps, or (c) with continuous-time edge timestamps.

any time [19]. This makes CTDGs crucial for certain applications, such as real-time fraud detection [30] and time-aware recommendation systems [36]. Hence, it is important to have abstractions for programming CTDG-based models so that developers and researchers can easily explore new designs, applications, and optimizations in this space.

A few early frameworks have emerged to meet the rising interest in TGNNs, such as PyG Temporal [20], DynaGraph [6], and TGL [37]. These frameworks provide a simple API for implementing TGNN models, and/or introduce novel techniques for scaling up the training of these models. However, some drawbacks prevent them from being an efficient, easy-to-use interface for exploring TGNN models for continuous-time graphs.

First, the frameworks do not have programming abstractions for continuous-time TGNN models. PyG Temporal mainly provides implementations of spatio-temporal discrete-time models, while DynaGraph targets implementing and optimizing discrete-time models that use Recurrent Neural Networks (RNNs) in a particular way. Although TGL does support models for CTDGs, the framework mainly provides a general approach to structuring TGNN training rather than providing a programming interface. In fact, users interact with the framework via configuration files[1].

Second, the framework interfaces are not designed for easy composability, which is important to allow researchers to explore new TGNN designs. Despite the lack of a programming interface, users of TGL can clone the open-sourced code and add new components, but they must follow its proposed structure which discourages novel compositions and orderings. Meanwhile, the modules and layers in PyG Temporal could be composed, but it lacks composable primitives for these layers themselves which impedes exploration of new layers, as well as being tailored to DTDGs.

Lastly, the frameworks do not facilitate optimizations tailored to continuous-time models, which are crucial to allow for efficient implementations. Optimizing continuous-time TGNNs is an under-explored area, but prior work does exist such as TGOpt [32] which proposes redundancy-aware optimizations for TGAT based on deduplication, memoization, and precomputation. Aside from a temporal CSR graph format and a parallel graph sampler, TGL does not provide much in the way of optimizing based on CTDG characteristics. Naturally, PyG Temporal and DynaGraph do not offer optimizations that are applicable to models for CTDGs.

Thus, the current status quo has a gap for a framework that can enable practitioners to implement continuous-time models while facilitating for their optimization. In this paper, we introduce TGLite, a lightweight framework designed for programming TGNN models for CTDGs. By *lightweight*, we mean that TGLite only supplies a few key abstractions and a set of composable operators. The user can use these as building blocks to orchestrate computations common to TGNNs, while we rely on a tensor backend to provide the user with canonical tensor and neural network operations.

To facilitate composition, we introduce a novel data abstraction called a `TBlock` to capture message-flow dependencies, which is commonly required for computations like neighborhood aggregation. `TBlocks` have explicit support for temporal-related attributes that are missing from the message-flow graphs (MFGs) used with static GNNs. It also differs from MFGs in a few key ways: 1) its doubly-linked list design allows explicit representation and better support for multi-hop aggregations, 2) it provides easier manipulation of message-flow dependencies by making neighbor information optional, and 3) it has a flexible *hooks* mechanism that optimizations can use to schedule post-processing operations without imposing the burden on the user.

Given these design choices, `TBlocks` serve as a central representation that different operators can be applied to, leading to easy composition. With TGLite's Python interface, users can define new block operators for their needs or explore applying the operators in new ways. The framework currently provides operators for temporal neighborhood sampling, scatter/segmented computations, and redundancy-based optimizations (which are semantic-preserving transformations and does not affect model accuracy). It also provides a few other data representations useful for working with CTDG data during the offline training scenario. More details are discussed in §3.

We subsequently apply the TGLite framework to implement a collection of four existing TGNN models (see §4). All these models work with continuous-time graphs, but use different techniques to capture the temporal information. Their implementations make heavy use of the data representations and operators in TGLite. This exercise demonstrates that TGLite has the necessary generality and expressiveness for programming this class of TGNN models.

To evaluate our framework, we compared these implementations to TGL as a strong baseline, using standard CTDG datasets on two different GPU machines. Our results show that TGLite-based models can achieve considerable speedups for both training and inference. When optimization operators are enabled, TGLite can achieve speedups of 1.06−3.43× for training and 1.09 − 4.65× for inference on these standard

[1] https://github.com/amazon-science/tgl/blob/main/config/TGAT.yml

benchmarks. We further evaluate on two larger-scale benchmarks and obtain similar or better speedups, particularly for TGAT with up to 9.02× on training and 15.63× on inference.

In summary, this paper makes the following contributions:

- We propose a framework centered on lightweight abstractions and composable operators for programming TGNNs for continuous-time graphs. In particular, we introduce a novel `TBlock` abstraction for capturing the message-flow computation dependencies commonly seen in TGNN models, enabling easy manipulations and optimizations.
- We identify and provide a set of `TBlock`-based operators, such as temporal neighborhood sampling, scatter/segmented computations, and redundancy-aware optimizations, that users can compose in flexible ways to fit their needs.
- We conducted an exercise of implementing different TGNN models using TGLite to demonstrate its expressiveness, and evaluated these on a variety of CTDG datasets. Our results show that TGLite is capable of yielding substantial speedups compared to TGL, reaching 1.06 − 3.43× for training and 1.09 − 4.65× for inference when using semantic-preserving optimization operators. On large-scale datasets, TGLite (with optimizations) can reach up to 9.02× for training and up to 15.63× for inference, while TGL can run out of GPU memory for some cases.

2 Background

We review a few key concepts relevant to Temporal GNNs.

Graph Neural Networks. GNNs act as graph operators that aggregate node features with local neighborhood information. They compute d_v-dimensional vectors $h \in \mathbb{R}^{d_v}$, called node embeddings, that are used for downstream tasks such as node classification and edge prediction. GNNs typically assume the graph is static and temporal changes have no affect on node embeddings, often leading to poorer predictive performance on time-sensitive tasks. Temporal GNNs are designed to extend static GNNs to learn on temporal graphs. Computations for continuous-time graphs are mainly based on temporal neighborhood aggregations as well as encoding time information (i.e. edge timestamps, time deltas, etc) using either RNNs or other time-encoding techniques.

Temporal Message Passing. Neighborhood aggregation is often expressed in the message-passing style [4], which splits the process into three key steps:

$$m_j(t) = \text{msg}\left(h_i(t),\, h_j(t_j),\, e_{ij}(t_j)\right) \tag{1}$$

$$r_i(t) = \text{agg}\left(\{m_j(t) : j \in \mathcal{N}(i,t)\}\right) \tag{2}$$

$$h_i(t) = \text{upd}\left(h_i(t),\, r_i(t)\right) \tag{3}$$

where i, j are nodes, $e_{ij}(t_j)$ is edge features, and t is a time parameter, indicating that embeddings are now a function of the node and time [32]. In brief, Eq. (1) creates a *message* vector for each neighbor j, Eq. (2) reduces them into a single vector, and Eq. (3) combines this with the node features. $\mathcal{N}(i,t)$ denotes the set of neighbors of node i restricted to those with edge timestamps less than t, and further sampled based on a strategy such as most-recent neighbors.

TGNN models often perform message-passing in several layers, thereby recursively aggregating neighbor information from multiple hops away. MFGs from DGL (a static GNN library) are often used to represent this computation, where each MFG captures the 1-hop relationships and performs the three steps mentioned above. Figure 2 shows a 2-hop aggregation example, where the result of `mfg0` is passed to `mfg1` and the final result is stored in `dstdata['h']`. More importantly, note that for TGNNs the user has to manage time-related properties themselves, as well as orchestrating how data is passed between the individual MFG objects.

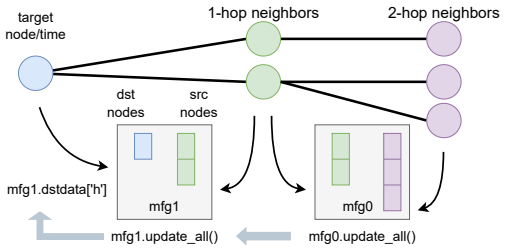


Figure 2. An example of 2-hop aggregation using MFG objects and operators.

Temporal Learning Techniques. Existing TGNN models leverage two main techniques for learning temporal patterns. TGAT [34] is a representative model using the time-encoding technique, where it uses a *time-encoder* function $\Phi : T \longrightarrow \mathbb{R}^{d_t}$ that maps a time value to a d_t-dimensional vector. TGAT performs the following computation for each layer [32]:

$$z_i(t) = h_i(t) \,\|\, \Phi(0) \tag{4}$$

$$z_j(t) = h_j(t_j) \,\|\, e_{ij}(t_j) \,\|\, \Phi(t - t_j) \tag{5}$$

$$r_i(t) = \text{Attn}\left(z_i(t),\, \{z_j(t) : j \in \mathcal{N}(i,t)\}\right) \tag{6}$$

$$h_i(t) = \text{FFN}\left(r_i(t) \,\|\, h_i(t)\right) \tag{7}$$

where the time vector is injected into message-passing by concatenating with node/edge features in Eq. (4) and (5), then performs aggregation using self-attention [26] in Eq. (6), and lastly a feed-forward network in Eq. (7). Time-encoding is often computed using weight (ω) and bias (ϕ) vectors, and a time delta Δt scalar value as input [32]:

$$\Phi(\Delta t) = \cos\left(\omega \cdot \Delta t + \phi\right). \tag{8}$$

Meanwhile, other TGNNs can be classified as memory-based models, where the key idea is to encode a node's historical interactions in long-term memory (as a vector). Abstractly, these models update the memory in a similar fashion

as message-passing [19]:

$$m_i(t) = \text{msg}\left(s_i(t^-), s_j(t^-), \Delta t, e_{ij}(t)\right) \quad (9)$$

$$\bar{m}_i(t) = \text{agg}\left(m_i(t_1), m_i(t_2), \ldots, m_i(t_b)\right) \quad (10)$$

$$s_i(t) = \text{mem}\left(\bar{m}_i(t), s_i(t^-)\right) \quad (11)$$

where for a node i it create messages using its memory $s_i(t^-)$ and neighbor j's memory, aggregating across t_b number of edges in a batch, then applies a memory update function.

Redundancy-Aware Optimizations. Techniques based on redundancy exploit observations that CTDG-based models (like TGAT) often perform redundant computations. Optimizations such as deduplication, memoization, and time-precomputation eliminate these redundancies without changing model semantics or accuracy. Deduplication filters out duplicates to ensure embeddings are only computed for unique node-time pairs, while memoization exploits the observation that previously computed embeddings could be reused and cached, thus avoids running computations altogether. For time-precomputation, the time-encoder often produces the same time vectors, so those can be precomputed ahead-of-time and reused. For more details, refer to [32].

Model Training. The training scheme is more complicated for memory-based models. In order for memory-related modules to receive a gradient, the memory vectors will need to be involved in the loss calculation. But care must be taken to avoid the issue of information leakage, where information about the current batch is *leaked* to the model when it makes predictions for that same batch [19]. The strategy adopted by most TGNN models is to store raw messages in a *mailbox* and use those in a later batch (thereby effectively delaying the memory update caused by the current batch).

3 The TGLite Framework

First we provide an overview of TGLite, before diving into the core abstractions and operators.

3.1 Overview

TGLite is designed to be a *lightweight* framework that supports the offline training and inference scenarios. By lightweight, we mean that TGLite provides a few core abstractions and operators that users can use to compose together a TGNN model. However, TGLite is not complete on its own since these core abstractions mainly deal with the temporal graph, graph operations, as well as common TGNN computation patterns, but does not provide other operators such as tensor math. As such, users can pair TGLite with the PyTorch deep learning library [18]. This design allows us to focus on providing TGNN abstractions while reusing and integrating with many of the facilities provided by PyTorch, such as tensor operations and automatic differentiation.

The abstractions provided by TGLite are designed to make it easier to implement various TGNN models (which often requires tedious and error-prone programming efforts), as well as to easily apply optimizations on these models. There are two main groups of abstractions that TGLite provides: data representations/objects and compute operators. The data objects serve as containers for graph and tensor data needed during training and inference, with easy-to-use interfaces. Meanwhile, the compute operators capture common computation patterns on these objects, optimization techniques, as well as data loading/management functions.

To demonstrate the programmability and ease-of-use of TGLite, we compare implementations of TGAT with and without our framework. Listing 1 shows an example without our framework and with manually applied optimizations. The forward pass of the model (not shown) computes time-aware node embeddings for both the source and destination nodes of edges in a batch by calling `compute()`, then calculates a score using these embeddings to predict edges. The code in region Ⓗ implements Eqs. (4-7), and Ⓘ implements Eq. (8). A few key observations:

- **Manual Optimizations:** Applying redundancy-aware optimizations like deduplication requires applying pre- and post-processing which the programmer needs to manually manage (Ⓐ). For the caching optimization, the programmer needs to manually check for cache hits/misses (Ⓒ), run computation for the misses, and then populating the cache afterwards. Because these optimizations operate in-between layers it is difficult to pair these with MFG objects since they lack facilities for these kinds of in-between processing.
- **C++ Extension:** Applying optimizations is feasible but an efficient implementation requires writing a C++ "extension", which demands systems-level programming knowledge and bindings to integrate with Python (Ⓕ). These codes are not presented in Listing 1 for brevity.
- **Recursive Flow:** The recursive nature of temporal message-passing results in a tricky recursive implementation. In the base case (before any computations), node features are retrieved (Ⓑ). For the actual computations, it requires sampling temporal neighbors and recursively computing their node embeddings (Ⓓ).
- **Ad-hoc Data Structures:** A way to store the temporal graph and sampling the neighbors is needed. Consequently, implementations often have one-off data structures (e.g. `NeighborFinder` in Ⓖ) that has to be repeated for other implementations and projects.

In contrast, when compared to an implementation of TGAT using our TGLite framework shown in Listing 2, we observe the following benefits:

- **Reusable Library:** TGLite provides a Python library (Ⓙ) as the main interface for users to use and interact

```
class TGAT(nn.Module):
  def compute(self, nids, ts, layer, n_nbr):  Ⓐ
    nids,ts,inv = self.opt.dedup_filter(nids,ts)
    embs = self.embeds(nids, ts, layer, n_nbr)
    return self.opt.dedup_invert(embs, inv)

  def embeds(self, nids, ts, layer, n_nbr):
    if is_last(layer):
      return self.lookup_nfeats(nids...)  Ⓑ
    idx,embs = self.opt.cache_lookup(nids,ts...)
    if not is_all_hits(idx):
      nids = nids[misses(idx)]  Ⓒ
      ...
Ⓓ  nbr,nbr_ts,... = self.sample(nids,ts,n_nbr)
    nbr_ft = self.compute(nbr...,layer - 1...)
    feats = self.embeds(nids,ts,layer - 1,n_nbr)
    ...
    deltas = ts - nbr_ts
Ⓔ  nbr_tfeat = self.opt.time_embs(deltas, ...)
    tfeats = self.opt.time_zeros(batch_size)
    ...
    res = attn(feats,tfeats,nbr_ft,nbr_tfeat...)
    self.opt.cache_store(layer, res, nids, ts)
    embs[misses(idx)] = res
    return embs

class Optimizer:  Ⓕ
  def dedup_filter(self, nids, ts):
    return cpp_ext.dedup_node_time(nids, ts)
  ...
class NeighborFinder:  Ⓖ
  def sample(self, nids, ts, n_nbr):
    return cpp_ext.sample_recent(n_nbr, ts, ...)

class TemporalAttnLayer(nn.Module):
  def forward(self,feat,feat_t,nbr,nbr_t,...):
    Q = torch.cat([feat, feat_t], ...)
    Z = torch.cat([nbr, nbr_e, nbr_t], ...)
    ...
    attn = torch.bmm(Q, K.transpose(...))  Ⓗ
    attn = attn.masked_fill(mask, -1e10)
    attn = self.softmax(attn)
    out = torch.bmm(attn, V)
    return ...
class TimeEncode(nn.Module):  Ⓘ
  def forward(self, ts):
    return torch.cos(ts*self.weight + self.bias)
```

Listing 1. Truncated TGAT implementation with manual optimizations, derived from [32]. See descriptions in §3.1.

with the framework. This allows users to reuse constructs and benefit from the framework across projects.

- **Built-in Optimizations:** Optimizations can be applied by calling operators on the `TBlock` objects (Ⓛ), allowing users to easily experiment with different ones. Post-processing steps are scheduled by TGLite without manual intervention from the user, compared to the tedious bookkeeping required in Listing 1. Because `TBlocks` can capture multi-hop relationships (see §3.2), built-in operators can apply these processing steps and bookkeeping in-between layers. Also, all the C++ extension code in Ⓕ are built into the framework.
- **Built-in Operators:** Feature data necessary for the computations can be efficiently loaded using an operator from the framework (Ⓜ), and additional tensor data can be attached using a simple dictionary-like interface. For temporal self-attention, Listing 1 requires intricate tensor manipulations like batched matrix-multiply and masked softmax (Ⓗ), while TGLite allows the user to express these more naturally with "edge-wise" computation operators on `TBlocks` (Ⓠ). Moreover, time feature computations and lookups are manually orchestrated by the programmer in Ⓔ of Listing 1, while in Listing 2 these can be queried from precomputed time vectors managed by the framework (Ⓟ).
- **Iterative Flow:** Users can easily organize the recursive computation by creating `TBlocks` and sampling neighbors in a more succinct and iterative manner (Ⓚ). It is also more straightforward for a user to see which optimizations are applied (Ⓛ). Additionally, the node embedding computation using self-attention across multiple blocks (i.e. multiple layers) is simply encapsulated by an operator from the framework (Ⓝ).
- **Domain-Specific Data Structures:** TGLite provides built-in objects for the temporal graph, temporal sampling, and the `TimeEncode` module that users can reuse (Ⓞ). See §3.4 for more details.

```
import tglite as tg  Ⓙ

class TGAT(nn.Module):
  def __init__(self, ctx: tg.TContext, ...):
    self.sampler = tg.TSampler(n_nbr, 'recent')
    ...
  def forward(self, batch: tg.TBatch):
    head = batch.block(self.ctx)  Ⓚ
    for i in range(self.n_layers):
      tail = head if i == 0 \
             else tail.next_block(...)
      tail = tg.op.dedup(tail)  Ⓛ
      tail = tg.op.cache(self.ctx, tail, ...)
      tail = self.sampler.sample(tail)
    tg.op.preload(head, use_pin=True)  Ⓜ
    tail.dstdata['h'] = tail.dstfeat()
    tail.srcdata['h'] = tail.srcfeat()
    embs = tg.op.aggregate(
           head, self.attn_layers, key='h') Ⓝ
    return self.edge_predictor(embs)

class TemporalAttnLayer(nn.Module):
  def __init__(self, ctx: tg.TContext, ...): Ⓞ
    self.time_encoder = tg.nn.TimeEncode(...)
    ...
  def forward(self, blk: tg.TBlock):
    tfeats = tg.op.precomputed_zeros(...)  Ⓟ
    nbr_t = tg.op.precomputed_times(...)
    Q = torch.cat([blk.dstdata['h'],tfeats],...)
    Z = torch.cat([blk.srcdata['h'],
                   blk.efeat(), nbr_t], ...)
    ...
    attn = torch.sum(Q * K, ...)
    attn = tg.op.edge_softmax(blk, attn)  Ⓠ
    out = torch.reshape(V * attn...)
    out = tg.op.edge_reduce(blk, out, op='sum')
    return ...
```

Listing 2. Example TGAT implementation using our TGLite framework. See descriptions in §3.1 for details.

In summary, TGLite provides core abstractions for users to succinctly express computation patterns of TGNN models and applying relevant optimizations.

3.2 The TBlock Data Abstraction

TBlock, or temporal block, is an important centerpiece of our framework. A TBlock essentially captures the 1-hop message-flow dependencies between target node-time pairs (i.e. destination nodes) and their temporally sampled neighbors (i.e. source nodes), along with their respective edges. Blocks are motivated by the MFG objects available from the DGL library. Although MFGs work well for static GNNs and can be leveraged for TGNNs (as is done by TGL), it lacks explicit timestamps which are important for CTDG models. Following this point, we devised a novel block representation to better allow us to perform manipulations and optimizations for CTDGs. Figure 3 illustrates the high-level design and internal structure of a TBlock. There are three key design choices that distinguish these blocks from MFGs.

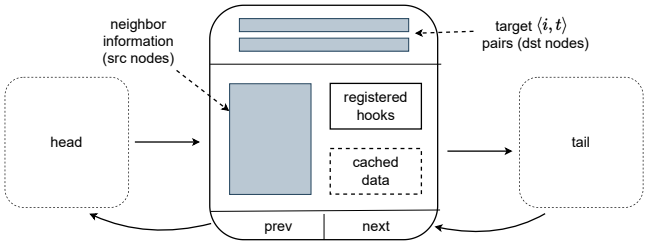


Figure 3. Diagram of the doubly-linked list design and internal structure of a TBlock (target $\langle i, t \rangle$ denotes the destination node-timestamp pairs).

First, one distinguishing factor is that we use a doubly-linked list structure for the blocks. A single TBlock captures 1-hop for one layer, and multiple blocks are chained together with previous and next links. This helps to explicitly capture the multi-hop neighbor sampling/aggregation relationship that blocks are used for, whereas the MFGs in DGL/TGL are standalone objects without these links. One advantage of this design is that operators on a TBlock can figure out what other blocks are related to the one it's working on. For example, this is helpful when we need to pass computed data across a multi-hop aggregation operation. From the perspective of aggregation, the computation starts at the "tail" which can be found by traversing next links to successor blocks, and then passing data to predecessor blocks.

Second, TBlock objects only require information on the target destination nodes, and optionally the neighbor source nodes. By making the neighbor information optional to start with, this allows us to manipulate the target node information more easily. This is necessary to make certain optimizations (like deduplication and caching from TGOpt) more effective, since these should be applied on the destination nodes before sampling for the neighbors. In contrast, MFGs require both destination and source node information upfront. Although a user can deal with these information separately before creating an MFG, the block representation provides a central abstraction for doing these manipulations.

Third, TBlock provides a *hooks* mechanism for running any post-processing procedures. These hooks are callable functions that are invoked after computations are performed on the block. This mechanism is useful for scheduling certain transformations on the computed output. For example, when applying deduplication there is an extra step after computations to revert filtering and preserve output semantics. Rather than putting the burden on the user to call this post-processing step (which they may forget to do), we can register a hook with the block and the TGLite runtime will handle running it automatically at the appropriate computation stage (usually in-between layers during aggregation). The user may also leverage this hooks mechanism for any post-processing that they will like to perform.

A block object also contains cached data that it manages internally. TBlock allows for a simple interface for accessing feature data specific to the nodes and edges in the block (e.g. `dstfeat()`, `efeat()`). These data are stored in the block's cached area so we avoid fetching them a second time and incurring data movement costs. If desired, these cached data can be flushed by the user and the TBlock will gracefully reload them when needed. Aside from these, a TBlock also provides methods for creating and accessing other blocks for multi-hop operations.

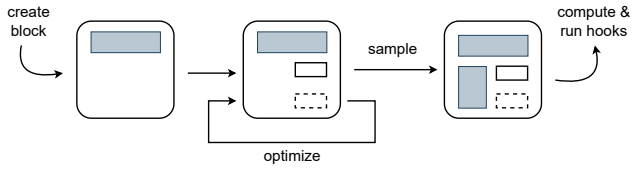


Figure 4. Typical workflow of constructing and using a TBlock object, and applying optimizations on it.

Lifecycle of a Block. To put the pieces together, Figure 4 illustrates the typical lifecycle that a block object goes through. Listing 2 demonstrates how blocks are created and used in practice. TBlocks can be created via several methods (lines 8 and 11), or constructed directly by the user. Once created, users can apply optimizations to it (lines 12 and 13) before sampling its neighbors (line 14) so to minimize the size of the following subgraphs and thus minimize potential computations. TBlock operators can manipulate the block in-place, register hooks, or cache data in the block. Then, we can use the block for computations (as in code region Ⓠ).

3.3 TBlock-based Operators

The block representation provides a powerful and general abstraction for defining various graph operations (e.g. sampling) and computations (e.g. segmented softmax). Optimizations can also be framed in terms of blocks. Table 1 describes a subset of these operators. We can categorize TBlock-based operators as whether they perform an optimization or not.

Table 1. A subset of operators currently provided by TGLite. Brackets in top-right corner of each row indicates whether its a *single*-block, *multi*-block, or *opt*imization operator.

`TSampler.sample(blk)` Update block with sampled 1-hop source neighbors.	[single]
`edge_softmax(blk, data)` Computes segmented softmax on given data using edge information from the block.	[single]
`edge_reduce(blk, data, op='sum')` Computes segmented reduction (e.g. sum or mean) on given data using edge information from the block.	[single]
`coalesce(blk, by='latest')` Segmented operation to reduce source nodes for each destination node by a certain property, such as selecting node with the latest timestamp.	[single]
`aggregate(blk, fn_or_list, key)` Performs pull-style multi-hop aggregation from the tail block back towards the given block by applying function to each block, using the `key` to pass along results.	[multi]
`propagate(blk, fn_or_list)` Performs push-style multi-hop propagation from given block to the tail block by applying function to each block.	[multi]
`preload(blk, use_pin=True)` Prefetch data (e.g. features, memory, mails) needed by the `TBlock` and its subsequent blocks for computations.	[opt]
`dedup(blk)` Applies the deduplication optimization to the `TBlock` by rewriting the destination nodes.	[opt]
`cache(ctx, blk, ...)` Applies the caching optimization to the `TBlock` by rewriting the destination nodes and using `ctx` as scratch space.	[opt]

Optimization Operators. The framework provides several redundancy-aware optimizations, such as deduplication and memoization. Their corresponding operators in TGLite, `dedup()` and `cache()`, will manipulate the block's destination node-time pairs in-place and register post-processing hooks on the block. For instance, `cache()` will filter destination node-time pairs to ones that have not been cached and register a hook to combine computed outputs with cached results, thus avoiding repeated computations for cached embeddings and retaining expected output semantics.

`preload()` is a block operator that manages loading data for all the blocks in the linked list, and focuses on optimizing data movements. During training, data are often stored on CPU host memory and transferred to GPU device memory, which is costly. Therefore, one technique is to use pinned memory to minimize data transfer costs by allowing the GPU to directly access host memory for copying. `preload()` uses this by default, and TGLite manages a pool of pre-allocated pinned memory so no manual user intervention is required.

Computation Operators. Other operators are designed to perform some computation-related tasks. For example, `edge_reduce()` performs *segmented* reduction, where for each destination node it applies a reduce operation to its group of source nodes to combine their data (thus being segmented). Meanwhile, `coalesce()` re-arranges and reduces the source nodes for each destination node based on some property, such as latest edge timestamp. In general, these *single-block operators* can be applied during any stage of the block lifecycle to manipulate/optimize the block or to compute an output.

Furthermore, *multi-block operators* can apply some computation across multiple blocks, where the doubly-linked structure of blocks represents a multi-hop subgraph. A key computation pattern used in TGNN models is operating on neighborhood information. We can distill this pattern into two types based on the direction of data flow (push versus pull). One is *aggregation*, which pulls information from neighbors in a multi-hop subgraph. This is the typical message-passing paradigm as used by models like TGAT. The `aggregate()` operator implements the pull-style of neighborhood aggregation, where given a block it will traverse the linked list to the tail and apply a function provided by the user to each block all the way back up to the starting block. It also handles some tedious bookkeeping that is necessary when passing information across blocks, such as assigning the correct data to the destination and source nodes. On the other hand, the `propagate()` operator does the push-style where it starts at the given block and works its way toward the tail of the list. This propagation pattern is useful for the APAN model (as seen in Appendix A).

3.4 Other Abstractions and Operators

Graphs and Batches. Table 2 summarizes the core data objects provided by TGLite. These data objects act as containers for graph/tensor data needed for computations, and present users with a central way to access related data. These also allow TGLite to perform more efficient data storage and movement, without imposing the burden on the user.

In particular, a `TGraph` is the central hub for all data related to a CTDG dataset. Temporal graphs are sparse in nature and thus benefits from the compact storage footprint of formats such as coordinate (COO) and compressed sparse row (CSR). TGLite initially stores temporal edges in COO format, sorting based on timestamp so that the common case of iterating through the edges chronologically will be fast. When a model needs to perform neighborhood sampling, such as TGAT, it is best to use a CSR format for faster lookups. TGLite automatically handles the construction and management of these graph formats without intervention from the user.

Table 2. Summary of the core data representations/objects currently supported by TGLite.

`TContext` - Settings and scratch space used by the TGLite runtime, such as for caching values.
`TGraph` - Manages storage of the temporal graph topology, serves as a container for node and edge tensor data.
`TBatch` - Represents a batch of temporal edges to process.
`TBlock` - Captures 1-hop relationships between node/time pairs and their neighbors for doing computations, such as segmented softmax and message-passing aggregation.
`TSampler` - Parallel temporal neighborhood sampling, using either uniform or most-recent sampling strategies.
`Memory` - Storage for node memory vectors and their last updated timestamps.
`Mailbox` - Storage for node mailbox message vectors and delivery timestamps.

Meanwhile, a `TBatch` represents a set of edges to be processed. In some implementations, batches are haphazardly represented as several arrays containing the nodes and timestamps, whereas a `TBatch` in TGLite is a thin wrapper with a `TGraph` reference and without actually materializing any arrays until they are needed.

Memory and Mailbox. For memory-based TGNN models, it is useful to have support for node memory and mailbox. TGL provides a nice representation for these, which we also use in TGLite. However, one key difference is that the `Memory` and `Mailbox` storage components are made part of the `TGraph` interface so that users can access these data in a central place and it allows TGLite to better manage and optimize for them, such as preloading and caching the data.

Temporal Sampling. Neighborhood sampling is naturally implemented as a block operator. TGLite provides a `TSampler` module that exposes 1-hop temporal sampling via its `sample()` method, which can be used as a block operator.

Non-block Operators. The standalone functions called `precomputed_zeros()` and `precomputed_times()` implements the time-precomputation optimization. The first is specialized to the case when a user knows that they have time deltas of zeros, while the latter is the more general version. These *precomputed-time* operators will compute time vectors ahead-of-time for the time-encoder modules and reuse them as much as possible.

4 Case Study: TGN

Here we describe applying TGLite to implement a different TGNN model called TGN. We also applied TGLite to implement memory-based models called JODIE and APAN. See Appendix A for more details on these latter two.

```
class GRUMemoryUpdater(nn.Module):
  def forward(self, mfg):  Ⓡ
    d = mfg.srcdata['ts'] - mfg.srcdata['memts']
    tfeat = self.time_encoder(d)
    mail = cat([mfg.srcdata['mail'],tfeat], ...)
    mem = self.gru_cell(mail,mfg.srcdata['mem'])
    self.last_updated_ts = ...
    self.last_updated_mem = ...
    self.last_updated_nids = ...
    mfg.srcdata['h'] =
        mem + self.linear(mfg.srcdata['h'])
    ...

class MailBox():
  def update_mailbox(self, ...):  Ⓢ
    src_mail = cat([mem_src,mem_dst,efeats],...)
    dst_mail = cat([mem_dst,mem_src,efeats],...)
    mail = torch.cat([src_mail, dst_mail], ...)
    ...
    uniq, inv = torch.unique(nids, ...)  Ⓣ
    perm = torch.arange(inv.size(0), ...)
    perm = inv.new_empty(uniq.size(0))
           .scatter_(..., inv, perm)
    mail = mail[perm]
    self.mailbox[nids..] = mail
    ...
```

Listing 3. Memory-related modules in the TGL framework for the TGN model, derived from [37].

```
class TGN(nn.Module):
  def forward(self, batch: tg.TBatch):
    ...
    mem = self.update_memory(tail)
    nfeat = self.linear(tail.nfeat())
    tail.dstdata['h'] = nfeat[:...] + mem[:...]
    tail.srcdata['h'] = nfeat[...:] + mem[...:]
    embeds = tg.op.aggregate(head, ...) Ⓤ
    self.save_raw_msgs(batch)
    return self.edge_predictor(embeds)
  def update_memory(self, blk: tg.TBlock):
    ...
    delta = mail_ts - blk.g.mem.time[nodes]
    tfeat = tg.opt.precomputed_times(..., delta)
    mail = torch.cat([blk.mail(), tfeat], ...)
    mem = self.gru_cell(mail,blk.mem_data())  Ⓥ
    blk.g.mem.update(nodes, mem, mail_ts)
    return mem
  def save_raw_msgs(self, batch: tg.TBatch):
    blk = batch.block_adj(self.ctx)
    blk = tg.op.coalesce(blk, by='latest')  Ⓦ
    ...
    uniq, nbrs = blk.dstnodes, blk.srcnodes
    mail = cat([mem[uniq],mem[nbrs],efeats],...)
    blk.g.mailbox.store(uniq, mail, mail_ts)
```

Listing 4. Relevant model implementation and memory-related code for TGN using our TGLite framework.

TGN [19] is a model that combines techniques from TGAT with memory-based learning techniques. For node memory updates, it uses a GRU cell and only retains the latest message in the batch for each node. Node memory is then merged with node features before feeding them into message-passing.

Listing 3 shows the relevant memory-related modules provided by the TGL framework, which implements Eqs. (9-11). Memory is updated using mailbox messages and time features in Ⓡ. Various time- and mail-related data must be present in the MFG object, which is error-prone as these

are string-based mappings that can be easily altered by end-users or other processes. Additionally, information about "last updated" are needed for later processing. In Ⓢ, mailbox messages are created by combining node memory with other data. A complex code sequence is needed to find the unique nodes and to select their latest messages to be stored (Ⓣ).

Comparatively, Listing 4 shows how TGN is handled using our TGLite framework. For the forward pass, TGN borrows the time-encoder and self-attention from TGAT, so we can use code similar to Listing 2 for those components (Ⓤ). For node memory updates, relevant memory and mailbox data for the update operation can be safely and efficiently retrieved from the `TBlock` in Ⓥ. Additionally, in Ⓦ the block abstraction and the `coalesce()` operator can be used to express the reduction operation needed to extract the latest message from the batch.

Overall, this example (along with others in §A) illustrates the generality of TGLite, and as we will see in §5.2 we do not lose out on performance. What's more is that users can easily apply optimization operators with a single line of code (e.g. Listing 2 line 12) to gain speedups as compared to frameworks like TGL. In contrast, users of TGL must write their own implementations and expose new configuration settings. In fact, we observe that TGL's design is not general enough for all models, such as JODIE, and that the configuration must expose settings specific to JODIE to accommodate[2].

5 Evaluation

We evaluated the effectiveness of our TGLite framework against TGL as a strong baseline on a set of widely-used CTDG datasets, four different TGNN models, and across two GPUs (V100 and A100). In summary, the results show that an implementation using TGLite is on-par or outperforms TGL in terms of running time. By enabling semantic-preserving optimization operators, TGLite can achieve speedups of $1.06-3.43\times$ for training and $1.09-4.65\times$ for inference (across machines and experimental settings, see §5.2). To scale our evaluation, we further examine two larger datasets and obtain speedups of $1.15-9.02\times$ for training and $1.15-15.63\times$ for inference (see §5.5). For larger datasets and more complex models like TGAT and TGN, the TGL framework runs out-of-memory while TGLite is able to finish the experiments.

5.1 Experimental Setup

We closely follow the training setup of TGL's experiments for a fairer comparison. The specific models we evaluated on are: JODIE, APAN (1 layer, mailbox of size 10), TGAT (2 layers, 10 neighbors), and TGN (2 layers, 10 neighbors). Unless otherwise noted, we use a batch size of 600, train for 10 epochs, using recent sampling, and with default values from TGL for the rest of the model hyperparameters.

[2] https://github.com/amazon-science/tgl/blob/main/config/JODIE.yml

Benchmark Selection. We use standard datasets as seen from the literature [11, 32, 37]: Wiki, MOOC, Reddit, and LastFM (see Table 3). For larger-scale performance evaluation, we use two additional datasets: WikiTalk and GDELT. WikiTalk is a temporal graph from the SNAP repository [13] (with name wiki-talk-temporal) representing users editing each other's Talk pages on Wikipedia. GDELT is a temporal knowledge graph of events happening around the world as reported by news outlets [37]. We use the GDELT dataset as prepared by TGL. These are considerably larger than the standard benchmarks in terms of nodes and edges, with edges in GDELT being two orders of magnitude larger.

Table 3. Benchmark datasets. * denotes randomly generated node feature vectors (d_v), while † denotes randomly generated edge features (d_e).

Dataset	$\|V\|$	$\|E\|$	d_v	d_e	$\max(t)$
Wiki	9,227	157,474	172*	172	2.7e6
MOOC	7,144	411,749	128*	128†	2.6e6
Reddit	10,984	672,447	172*	172	2.7e6
LastFM	1,980	1,293,103	128*	128†	1.4e8
WikiTalk	1,140,149	7,833,140	128*	128†	1.2e9
GDELT	16,682	191,290,882	413	186	1.8e5

Testbed Machines. We conducted our experiments on two different machines. One is an AWS p3.8xlarge machine instance provisioned with 32 vCPUs @ 2.3GHz, 244GB memory, and using a single Nvidia Tesla V100 16GB GPU (henceforth the V100 machine). Another is a compute cluster node allocated with 2x Intel Xeon Platinum 8358 @ 2.6GHz, total of 64 cores, 251GB main memory, and a single Nvidia A100 80GB GPU (henceforth the A100 machine).

Implementation Details. The 4 model implementations are described in Appendix A, and use `preload()` and other optimization operators where applicable. The temporal sampler in both frameworks uses 32 threads on the V100 machine and 64 on A100. We note that TGL has multi-GPU support, but with TGLite being a lightweight framework our objective is to target programmability and single-GPU optimizations, with multi-GPU support left as future work.

5.2 Training Performance

We evaluate training performance under different experimental settings, using the total time to run a single epoch in seconds as the main metric. One crucial factor that affects the running time is where the training data (i.e. node/edge features, memory, mailbox) are stored. We first examine the scenario where all data resides on GPU device memory (all-on-GPU case, §5.2.1). In practice, it is unlikely that real-world datasets will fit entirely in GPU memory (see §5.5). Therefore, we examine the second case where data resides on

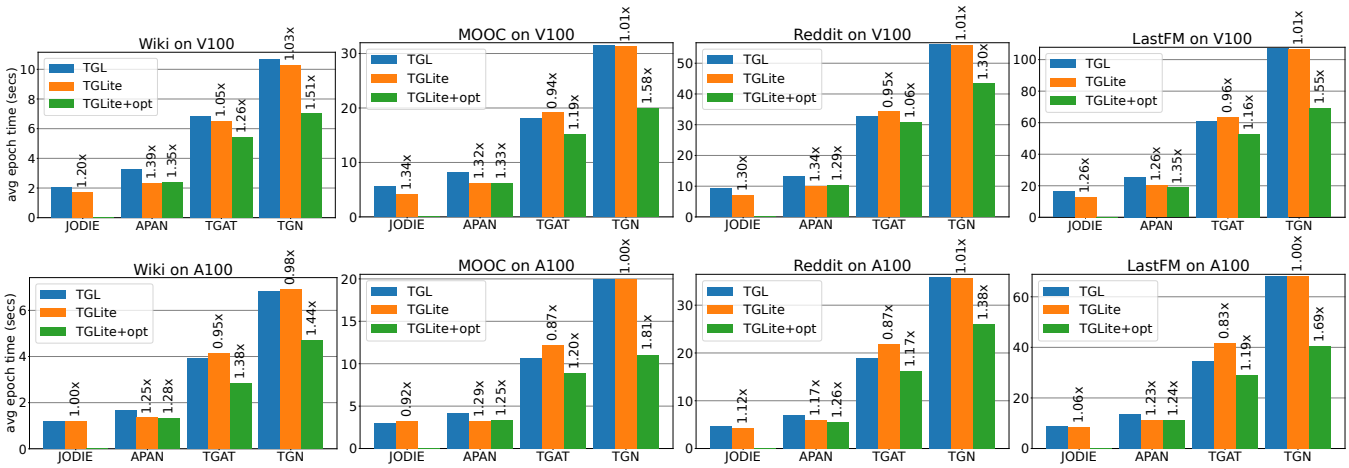


Figure 5. Training time per epoch (seconds) for different datasets with data residing on GPU device memory (all-on-GPU case). Top row is V100 and bottom is A100. Bar labels are speedups against TGL. TGLite+opt for JODIE is same as TGLite setting.

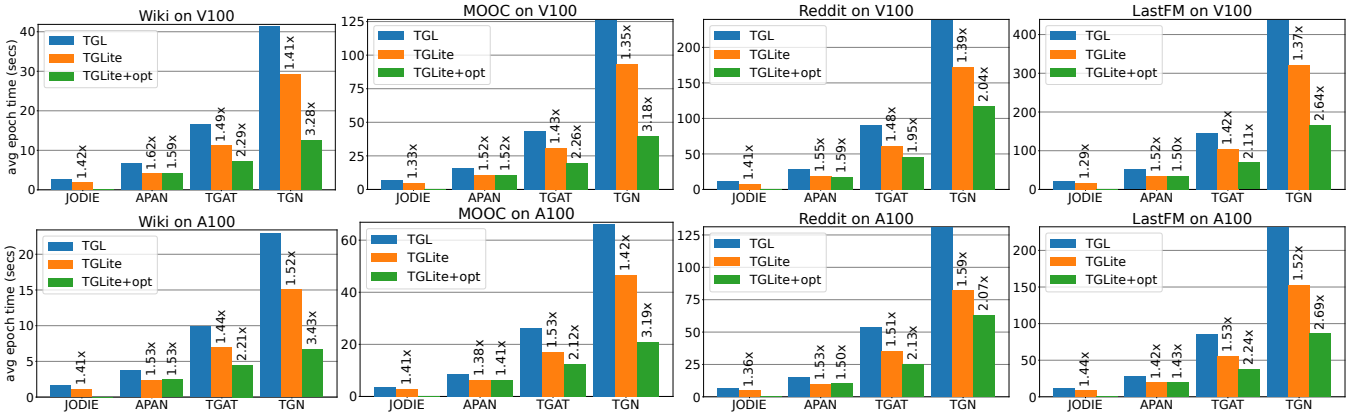


Figure 6. Training time per epoch (seconds) for different datasets with data residing on CPU host memory (CPU-to-GPU case). Top row is V100 and bottom is A100. Bar labels are speedups against TGL. TGLite+opt for JODIE is same as TGLite setting.

CPU host (CPU-to-GPU case, §5.2.2). To show the impact of optimizations, we evaluate the models with optimization operators (i.e. `preload()`, `dedup()`, `cache()`, and the two precomputed-time operators) applied as mentioned in §A (henceforth labeled TGLite+opt). As an ablation variant, we also present results with only the `preload()` operator and no other optimizations (simply labeled TGLite). Note that no further optimization operators are applied for the JODIE model due to its simplicity, so we skip the TGLite+opt setting for JODIE. We then present a breakdown analysis in §5.2.3. Overall, we observe that our framework is able to match or outperform TGL across datasets, models, and GPUs.

5.2.1 All-on-GPU Training Case. In Figure 5, we observe that optimizations used in TGLite+opt can lead to speedups against TGL, with 1.06 – 1.81×. Although the `preload()` operator in TGLite has no effect in this scenario since all data are already on GPU, TGLite is able to perform on-par with TGL as the similar epoch times show. However, TGL seems to have an advantage in some cases, such as for the TGAT model on A100 where TGLite (without other optimizations) has poorer performance of 0.83 – 0.95×. We note that applying optimization operators as in the case of TGLite+opt helps to offset these slowdowns.

Table 4 lists the training accuracy results on the A100 machine, using the average precision (AP) scoring metric on the evaluation set. We observe that across models and datasets, implementations using our framework achieves similar levels of model accuracy as TGL. More importantly, the optimizations we provide in the framework (and applied in the TGLite+opt setting) are semantic-preserving and does not affect the accuracy of the model. The minor differences measured are due to stochasticity in training, such as nondeterministic algorithms used in the PyTorch/CUDA libraries. Similar trends can be observed for the V100 machine and the CPU-to-GPU scenario, and are not shown here for brevity.

Table 4. Training evaluation AP scores for best epoch in the all-on-GPU case on A100 machine. TGLite+opt for JODIE is same as TGLite setting and is skipped (denoted with -).

Data	Model	A100 Machine		
		TGL	TGLite	TGLite+opt
Wiki	JODIE	94.99	96.80	-
	APAN	95.43	95.09	94.80
	TGAT	98.77	98.66	98.74
	TGN	99.03	99.48	99.52
MOOC	JODIE	99.17	100.0	-
	APAN	99.00	99.01	98.99
	TGAT	99.38	99.38	99.39
	TGN	99.40	99.42	99.42
Reddit	JODIE	99.57	99.10	-
	APAN	99.61	99.47	99.41
	TGAT	99.66	99.65	99.65
	TGN	99.67	99.69	99.69
LastFM	JODIE	72.24	83.43	-
	APAN	77.80	75.75	75.32
	TGAT	89.42	89.17	89.17
	TGN	87.42	89.63	88.67

5.2.2 CPU-to-GPU Training Case. First, between Figures 5 and 6 we observe that TGL generally takes 4× longer to run (e.g. 107 vs 438 seconds for the TGN/LastFM/V100 case), indicating the high cost of data transfers. Next, in Figure 6 we observe that TGLite+opt achieves the best performance, with speedups of 1.41 – 3.43×. Lastly, we observe that TGLite is able to benefit from using pinned memory to outperform TGL, with speedups of 1.29 – 1.62×, indicating that using data movement optimizations like `preload()` is worthwhile.

Comparing TGLite with TGLite+opt in both cases, we see that performance is relatively the same for APAN, which is unsurprising since no further optimization operators are applied during training. We skip JODIE because it has no further optimizations applied. For TGAT and TGN, we observe that the `dedup()` operator brings significant improvements to the speedup across the datasets, especially for TGN since deduplication helps avoid costly memory update operations.

5.2.3 Breakdown Analysis. To better understand the results in the all-on-GPU case, we focus on the TGAT model with the LastFM dataset on the A100 machine. Figure 7 shows a cost breakdown of each major operation of interest for the TGL, TGLite, and TGLite+opt settings.

Both the backward pass and evaluation times are similar between TGL and TGLite, but TGLite is more efficient with preparing the batch/input data, which we attribute to the better data management in our framework. However, it incurs higher cost compared to TGL for time-encoding of neighbor time delta values (2.88 vs 1.76 seconds). This is because an additional step is needed to compute the deltas, while TGL is calculating these as it samples the neighbors during which it already has access to the time values. This is a trade-off between generality and efficiency, where we choose to provide more general abstractions and operations at the cost of slightly more overhead, and TGL vise-versa.

Next, we see that the attention operation has a much higher cost than in TGL (14.39 vs 9.20 seconds). Upon further investigation, we attribute this to the `edge_softmax()` operator used in self-attention, where it currently uses an external kernel library under-the-hood. This results in additional costs for transforming data for library calls and overhead of crossing the library boundary. This can be addressed within the framework as part of future work and all users will benefit from it simply by upgrading to a newer version of TGLite. Compared to the other two, TGLite+opt has slightly more overhead for the precomputed-time operators (labeled as `time_zero` and `time_nbrs` in Figure 7). But otherwise, the redundancy optimizations applied in TGLite+opt help reduce the cost of all the other operations, most notably the expensive self-attention operation, and thus gain speedups.

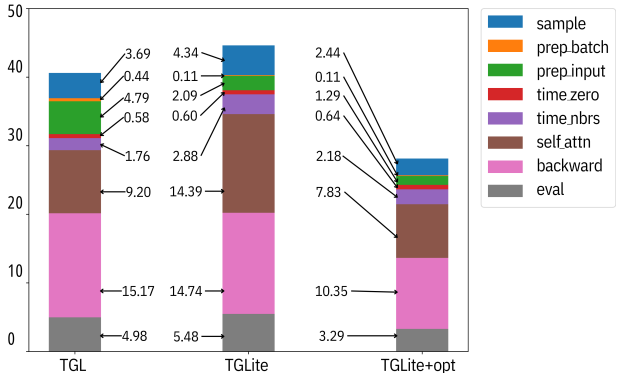


Figure 7. Breakdown of major operations in TGAT training epoch runtime (in seconds) with LastFM dataset on A100.

5.3 Inference Performance

We examined inference runtime (in seconds) for the testing set using the same experimental settings. In the all-on-GPU case, TGLite+opt achieves speedups of 1.09 – 1.54× and TGLite is 0.85 – 1.61× (slowdowns due to reasons mentioned above). For the CPU-to-GPU case, TGLite+opt achieves better speedups of 1.28 – 4.65× while TGLite is 1.17 – 1.75×. For the sake of space, Table 5 only shows the all-on-GPU case.

For APAN, TGLite+opt tends to result in lower speedups than TGLite, indicating that applying the precomputed-time operators alone might not be worthwhile. There is a wider margin for the LastFM dataset accuracy due to TGL's training script not saving node memories after training but instead recreates it before inference from training data. For TGAT and TGN with the TGLite+opt setting, TGAT tends to achieve higher speedups than TGN (on both the all-on-GPU and CPU-to-GPU cases), suggesting that the addition of the `cache()` operator during inference can bring noticeable benefits.

Table 5. Testing set inference times (in seconds) and AP (average precision) scores for standard benchmarks in the all-on-GPU case. Speedups against TGL are in parenthesis. TGLite+opt for JODIE is same as TGLite setting and is skipped.

Data	Model	V100 Machine						A100 Machine					
		TGL	AP	TGLite	AP	TGLite+opt	AP	TGL	AP	TGLite	AP	TGLite+opt	AP
Wiki	JODIE	0.28	96.15	0.20 (1.40×)	96.49	-	-	0.17	95.47	0.11 (1.55×)	97.25	-	-
	APAN	0.45	95.26	0.29 (1.55×)	95.17	0.34 (1.32×)	94.67	0.24	95.44	0.17 (1.41×)	95.37	0.19 (1.26×)	95.02
	TGAT	0.94	98.35	0.96 (0.98×)	98.36	0.74 (1.27×)	98.38	0.54	98.39	0.56 (0.96×)	98.23	0.38 (1.42×)	98.36
	TGN	1.29	98.70	1.36 (0.95×)	99.47	1.04 (1.24×)	99.43	0.78	98.75	0.79 (0.99×)	99.41	0.55 (1.42×)	99.42
MOOC	JODIE	0.79	99.14	0.49 (1.61×)	99.34	-	-	0.44	99.12	0.28 (1.57×)	99.69	-	-
	APAN	1.17	98.91	0.77 (1.52×)	98.68	0.86 (1.36×)	98.68	0.61	98.89	0.45 (1.36×)	98.72	0.50 (1.22×)	98.74
	TGAT	2.53	99.22	2.80 (0.90×)	99.23	1.98 (1.28×)	99.26	1.49	99.22	1.54 (0.97×)	99.25	0.97 (1.54×)	99.23
	TGN	3.97	99.43	4.01 (0.99×)	99.31	2.97 (1.34×)	99.29	2.25	99.38	2.27 (0.99×)	99.34	1.59 (1.42×)	99.31
Reddit	JODIE	1.29	99.20	0.82 (1.57×)	99.36	-	-	0.72	99.56	0.59 (1.22×)	99.39	-	-
	APAN	1.82	99.36	1.30 (1.40×)	99.26	1.44 (1.26×)	99.42	1.01	99.00	0.88 (1.15×)	99.39	0.83 (1.22×)	99.31
	TGAT	4.70	99.66	4.92 (0.96×)	99.65	3.66 (1.28×)	99.65	2.61	99.66	2.80 (0.93×)	99.64	1.86 (1.40×)	99.64
	TGN	6.93	99.65	7.00 (0.99×)	99.69	6.35 (1.09×)	99.70	3.92	99.67	4.13 (0.95×)	99.69	3.39 (1.16×)	99.69
LastFM	JODIE	2.32	66.63	1.54 (1.51×)	83.25	-	-	1.32	66.85	0.85 (1.55×)	83.69	-	-
	APAN	3.46	68.21	2.50 (1.38×)	76.31	2.61 (1.33×)	77.00	1.99	63.27	1.50 (1.33×)	75.99	1.67 (1.19×)	77.17
	TGAT	8.14	86.56	9.27 (0.88×)	87.30	6.52 (1.25×)	87.06	4.73	87.50	5.54 (0.85×)	85.46	3.18 (1.49×)	86.29
	TGN	12.37	87.38	12.83 (0.96×)	85.74	10.27 (1.20×)	87.06	7.33	87.59	7.19 (1.02×)	87.22	5.64 (1.30×)	86.42

5.4 Ablation Studies

To further investigate the benefits of our design and the semantic-preserving optimizations that it provides, we conducted several ablation studies. For consistency, we use the experimental setting of TGAT with LastFM dataset on A100.

Optimizations. This ablation examines the gain from individual optimizations by enabling only one at a time. We only look at the inference runtime since this is where all 3 optimizations are applicable for TGAT. In Table 6 the first column shows speedup of TGLite without these optimizations. We see that optimizing it with the time-precompute operators (+time) help to increase the speedup in both the CPU-to-GPU and all-on-GPU cases, showing that reusing the time vectors for this model and dataset is beneficial. The `dedup()` and `cache()` operators (+dedup and +cache, respectively) each bring even more speedups, as those help to reduce expensive computations such as self-attention.

Table 6. Inference runtime speedups compared to the TGL baseline for the TGAT/LastFM/A100 setting, with one optimization at a time.

	TGLite	+dedup	+cache	+time
CPU-to-GPU	1.72×	2.94×	3.23×	1.77×
All-on-GPU	0.85×	1.29×	1.15×	0.94×

TBlock-vs-MFG. This ablation examines the impact of our `TBlock` abstraction by removing it from our framework and instead use MFGs. We found that an implementation for TGAT with MFGs leads to noticeable slowdowns (i.e. about 9% and 3% slower for the CPU-to-GPU and all-on-GPU training cases, respectively). One contributing factor is higher data movement costs since MFGs require all data associated with the MFG to be stored on the same device, but this is not always necessary and TGLite can better control this with the `TBlock` abstraction. The user also needs to reimplement several operators, such as multi-hop operators like `aggregate()`, since those are no longer provided by the framework with the use of MFGs. Along with various helper functions, a user has to write an additional 200 lines of user-level application code. These will need to be repeatedly written by other users for other projects.

Hooks Mechanism. This ablation examines the impact of the hooks feature by removing it from TGLite. With a few changes in TGLite to provide users with callable post-processing functions that they can run themselves, we found that users can emulate this feature in their applications without incurring noticeable performance regressions. We note that what the user implements here is effectively what TGLite provides via the hooks mechanism. Additionally, the user needs to reimplement operators like `aggregate()` themselves, since the framework will no longer be scheduling the necessary post-processing steps in this case. Overall, this results in an additional 49 lines of user-level code.

5.5 Large-Scale Benchmarks

Due to the scale of the larger datasets, we only evaluate cases where all data resides on CPU host memory. For instance, GDELT has 191M edges and each edge has a 186-dimensional feature vector of 4-byte floats, so it is well over 130GB of data, which cannot fit on most modern GPUs (including the A100

Table 7. Training and inference times (in seconds) for larger benchmarks on both V100 and A100 machines. Speedups against TGL in parenthesis. OOM indicates GPU out-of-memory.

Data	Model	V100 Machine				A100 Machine			
		TGL		TGLite+opt		TGL		TGLite+opt	
		Train	Test	Train	Test	Train	Test	Train	Test
WikiTalk	JODIE	120.66	17.71	23.07 (5.23×)	3.79 (4.67×)	72.02	10.66	51.39 (1.40×)	6.93 (1.54×)
	APAN	302.19	46.97	133.55 (2.26×)	22.05 (2.13×)	157.22	23.72	114.78 (1.37×)	17.08 (1.39×)
	TGAT	639.95	83.52	154.93 (4.13×)	26.40 (3.16×)	318.96	37.37	264.95 (1.20×)	27.43 (1.36×)
	TGN	1603.84	199.63	420.53 (3.81×)	69.88 (2.86×)	716.09	83.70	621.91 (1.15×)	72.65 (1.15×)
GDELT	JODIE	837.39	146.25	716.08 (1.17×)	121.23 (1.21×)	538.27	96.46	425.58 (1.26×)	78.27 (1.23×)
	APAN	5891.39	988.89	3703.84 (1.59×)	644.34 (1.53×)	2761.27	451.27	2123.00 (1.30×)	367.37 (1.23×)
	TGAT	OOM	OOM	4686.81	490.75	20569.59	3454.65	2280.99 (9.02×)	220.98 (15.63×)
	TGN	OOM	OOM	10958.43	2192.70	33937.89	7359.72	5105.93 (6.65×)	1003.29 (7.34×)

that we test with). So we store data on CPU for both GDELT and WikiTalk for consistency. We also only compare the baseline with TGLite+opt where all available optimizations for the model are enabled. Lastly, to make experimentation more tractable, we use a batch size of 4000 (as suggested by TGL experiments) and only 3 training epochs for GDELT. Table 7 lists the training and testing inference times. Refer to Appendix B for training and inference accuracy results.

We observe that TGLite+opt achieves reasonable speedups against TGL for larger datasets, at least 1.15× for both training and inference. For TGAT/TGN, speedups are much more amplified for the GDELT dataset. In fact, TGAT achieves up to 9.02× for training and 15.63× for inference on the A100 machine, indicating that the effectiveness of the optimization operators is more beneficial as the temporal graph size grows. Interestingly, TGL runs out of GPU memory for TGAT and TGN on the V100 machine, despite all feature data being stored on CPU host, whereas TGLite+opt is able to finish running the experiment. This suggests that our TGLite framework (with optimizations) can be space-efficient while being faster in terms of training and inference.

6 Related Work

Lightweight Frameworks and Abstractions. This work is inspired by the approach of designing lightweight frameworks and abstractions for domain-specific problems, particularly for graph analytics workloads. In particular, our work is influenced by Ligra [24], which is a lightweight framework for implementing graph processing algorithms such as BFS and PageRank. Its interface consists of a vertex subset data abstraction, and exposes only three core operators that are applied on this vertex subset. Flash [14] builds upon Ligra's interface to support more advanced distributed graph algorithms, while Gunrock [31] provides abstractions for GPUs. Many other works exist to provide abstractions to ease the programming/optimizing of graph processing solutions [1, 5, 10, 12, 15, 23]. However, these systems lack tensor operators, neural network primitives, and gradient-based training support, thus not applicable to our domain.

In the GNN machine learning space, FeatGraph [7] provides a programming abstraction for GNN kernels that combines sparse templates with user-defined functions. Meanwhile, uGrapher [38] presents a unified abstraction that decouples the computation and scheduling of GNN operators, particularly CUDA kernels. Seastar [33] is another work that proposes a vertex-centric programming model for GNNs that generates efficient kernels from user-provided Python code. From the perspective of TGLite, these are orthogonal works that can be viewed as lower-level kernel libraries, which it can potentially incorporate internally or expose as block operators. However, it will require more investigation into how these can be generalized to the Temporal GNN space.

GNN Frameworks. DGL [28] is a popular framework for programming static GNNs. It provides a flexible graph abstraction and operators for message-passing. Our interfaces take inspiration from DGL so to lower the learning curve for developers. PyTorch Geometric (PyG) [3] is another well-established framework for GNNs. However, these frameworks are often larger in scope, such as providing implementations of full GNN models or layers as compared to the lightweight building blocks provided by TGLite.

7 Future Work

While this work focuses on continuous-time TGNN models, it will be important to explore extending support for discrete-time models as a future direction. Rather than simply taking inspiration from discrete-time frameworks like DynaGraph, this exploration will be done in accordance with TGLite's design approach of providing core data abstractions and composable operators. An attractive avenue to explore is incorporating optimizations from these other frameworks, perhaps as composable operators on a graph snapshot abstraction. Another prospective future direction

is adding support for multi-GPU and distributed training, which TGLite currently lacks. This will be helpful as users work with ever-larger temporal graph datasets. These efforts and directions will guide TGLite towards being a more full-featured and unified framework for programming both current and subsequent TGNN models.

8 Conclusion

We have described TGLite, a lightweight framework for programming TGNNs for CTDGs. TGLite is designed to provide a core set of data abstractions and composable operators. Our novel TBlock representation for capturing the 1-hop message-flow dependencies allows for many operators to be defined on it, ranging from temporal-related computations to optimizations. We also discussed using TGLite for various model implementations to demonstrate its expressiveness and applicability. Our experiments show that TGLite-based implementations can achieve speedups of $1.06-3.43\times$ for training and $1.09-4.65\times$ for inference against the TGL framework across different experimental settings.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and our shepherd for their guidance. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and by NSF under grant CCF-2316233.

Appendix A Implementation Details

For the TGL baseline, we used the open-sourced code from the authors[3]. We updated this code to use newer versions of Python (v3.7), PyTorch (v1.12.1), and DGL (v1.0.1). We also fixed an issue with recent sampling, did a bit of cleanup, and added timing code. For TGLite, we implemented it using the same version of Python and PyTorch.

TGAT. Listing 2 illustrates the main code for TGAT, where it iteratively creates and operates on TBlocks. As for further optimizations, we enable the dedup() operator during training, and additionally the precomputed-time and cache() operators during inference (following the work in [32]).

TGN. Listing 4 shows code relevant to TGN. Similar to TGAT, we apply the same optimization operators in the same fashion, except we do not apply the cache() operator during inference since it does not seem effective as node memory will be updated thus invalidating cached results.

JODIE. JODIE [11] focuses on interactions between nodes in a bipartite graph and predicting future interactions. For example, an e-commerce network can be represented as a bipartite graph with user and item nodes, and an interaction is a user buying an item. JODIE does not perform neighbor sampling or aggregation, but rather mainly updates node memory using RNNs (as the update_memory() function in Listing 5 shows). It also stores messages in a mailbox for later batches in save_raw_msgs(). We do not apply any further optimization operators for JODIE due to its simplicity.

```
class JODIE(nn.Module):
  def update_memory(self, batch: tg.TBatch):
    ...
    mem_ts = batch.g.mem.time[nodes]
    mail_ts = batch.g.mailbox.time[nodes]
    tfeat = self.time_encode(mail_ts - mem_ts..)
    input = batch.g.mailbox.mail[nodes]
    input = torch.cat([input, tfeat], ...)
    mem = batch.g.mem.data[nodes]
    mem = self.rnn_cell(input, mem)
    return mem, mail_ts
  def save_raw_msgs(self, batch: tg.TBatch):
    ...
    blk = batch.block_adj(self.ctx)
    mail = batch.g.mem.data[blk.srcnodes]
    mail = torch.cat([mail, blk.efeat()], ...)
    ..mailbox.store(blk.dstnodes,mail,mail_ts)
```

Listing 5. Truncated example of memory-related code for JODIE model implementation with TGLite.

APAN. APAN [30] has all the same components as TGN: attention-based aggregation, time-encoding, and node memory update, but is centered around propagating mailbox messages. While other models first samples the neighbors and then generate embeddings, APAN reorders and swaps this around by first performing embedding generation using stored messages, then propagating messages to neighbors. Listing 6 illustrates this using the propagate() operator and other single-block operators (such as sample() and a scatter-based operator). Comparatively, TGL has special handling code for this in the mailbox/memory-related modules. For further optimizations, the only relevant ones that we apply are the precomputed-time operators during inference.

```
class APAN(nn.Module):
  def forward(self, batch: tg.TBatch):
    embeds = self.attention(batch)
    ...
    blk = tg.TBlock(self.ctx, 0, nodes, times)
    blk = self.sampler.sample(blk)
    self.create_mails(batch, blk, ...)
    tg.op.propagate(blk, self.send_mails)
    return self.edge_predictor(embeds)
  def create_mails(self, batch, blk, ...):
    ...
    mail_s = cat([mem_src, mem_dst, ...], ...)
    mail_d = cat([mem_dst, mem_src, ...], ...)
    blk.dstdata['mail'] = cat([mail_s,mail_d]..)
  def send_mails(self, blk: tg.TBlock):
    ...
    mail = blk.dstdata['mail'][blk.dstindex]
    mail = tg.op.src_scatter(blk,mail,op='mean')
    m_ts = ...src_scatter(blk,mail_ts,op='mean')
    ..mailbox.store(blk.uniq_src()[0],mail,m_ts)
```

Listing 6. Truncated example of memory-related code for APAN model implementation with TGLite.

[3] https://github.com/amazon-science/tgl

Appendix B Additional Results

Table 8 lists AP scores for the larger datasets. We observe that TGLite (with optimizations) is comparable to TGL in terms of predictive performance, in both training and inference, since our optimizations are semantic-preserving. Similar trends are observed for the V100 machine.

Table 8. Training and inference AP (average precision) scores for larger benchmarks on the A100 machine.

Data	Model	A100 Machine			
		TGL		TGLite+opt	
		Train	Test	Train	Test
WikiTalk	JODIE	88.96	84.40	91.87	93.77
	APAN	95.88	83.06	95.90	93.32
	TGAT	87.58	85.69	90.73	87.18
	TGN	94.02	95.41	98.24	98.44
GDELT	JODIE	98.59	98.66	98.83	98.95
	APAN	97.49	97.45	96.80	97.21
	TGAT	98.70	98.77	98.70	98.77
	TGN	98.40	98.17	98.52	98.18

Appendix C Artifact Description

C.1 Abstract

The artifact packages together source code for our TGLite framework, the TGL framework, as well as other scripts for replicating the results in this paper. The four smaller standard benchmark datasets are also bundled with the artifact so it will be easier to get started with the experiments.

Our evaluation results were collected on two different machines: one contains an Nvidia V100 GPU (AWS p3.8xlarge instance, 32 vCPUs, 244GB RAM) and another with an A100 GPU (2x Intel Xeon CPUs, 251GB RAM). Running the full set of experimental settings for both machines takes approximately 5 days. You may choose to run a subset of the experiments for one of the GPUs.

C.2 Artifact check-list (meta-information)

- **Compilation:** GCC >= 11.4.
- **Model:** TGNN models are included in the artifact.
- **Data set:** Smaller datasets included (1GB), larger datasets can be downloaded via provided script (additional 53GB).
- **Run-time environment:** Linux, CUDA 11.8, Python 3.7, PyTorch 1.12, DGL 1.0. See `README.md` file for more details.
- **Hardware:** x86 CPU, Nvidia V100 GPU and/or A100 GPU.
- **Metrics:** Execution time, model average precision accuracy.
- **Output:** Console text and CSV files.
- **Experiments:** Model training and inference, ablation studies. See `README.md` file and provided scripts for details. To save time, run for only one of the GPU settings.
- **How much disk space required (approximately)?:** 60GB.
- **How much time is needed to prepare workflow (approximately)?:** 1-2 hours.
- **How much time is needed to complete experiments (approximately)?:** Up to 12 hours for smaller experiments, additionally up to 3 days for larger experiments.
- **Archived (provide DOI)?:** 10.5281/zenodo.10504480

C.3 Description

C.3.1 How to access. Download artifact from Zenodo: https://doi.org/10.5281/zenodo.10504480.

C.3.2 Hardware dependencies. A Linux-based system with a modern x86 CPU (ideally a 2-node NUMA machine with Intel Xeon). Smaller experiments require at least 6GB RAM and GPU device with 16GB. Larger experiments require at least 160GB RAM and GPU with at least 64GB. The artifact file is about 200MB and unpacks to 1GB of disk space.

C.3.3 Software dependencies. Need CUDA 11.8 toolkit, Python 3.7, PyTorch 1.12, and related Python packages. GCC 11.4 or higher compiler is needed to compile C++ extensions. Conda is used to setup Python environment.

C.3.4 Data sets. Artifact includes four datasets, and the two larger ones can be downloaded via included script.

C.3.5 Models. Both the TGL- and TGLite-based models used in our experiments are included in the artifact.

C.4 Installation

Obtain the artifact, extract the files, and follow the instructions in the `README.md` file. To kick-the-tires, run the following from the `tglite/examples` directory (see the readme):

```
$ ./exp/tgat.sh -d wiki --epochs 3 --move --opt-all
```

C.5 Experiment workflow

The bulk of the experiments is running model training and inference. The artifact contains the necessary training scripts, which by default will run several epochs of training and then inference on the testing portion of the dataset. The code and scripts for TGL and TGLite are in separate directories: `tgl/` and `tglite/`, respectively. Two of the ablation studies make modifications to TGLite's code, which are provided as separate directories in the artifact.

C.6 Evaluation and expected results

The `README.md` file contains detailed step-by-step instructions to reproduce the results shown in our paper's Evaluation section (§5). When executing the commands and scripts, they will write output text to the console and timing data to CSV files. For some experiments, these CSV files need to be manually consolidated, but then the provided Python scripts can be used to plot the relevant figures from our paper.

C.7 Experiment customization

The training scripts provide several command-line flags to allow for running customized experiments and workflows (use `--help` for details).

References

[1] Ilya V. Afanasyev, Vladimir V. Voevodin, Kazuhiko Komatsu, and Hiroaki Kobayashi. Vgl: a high-performance graph processing framework for the nec sx-aurora tsubasa vector architecture. *The Journal of Supercomputing*, 77(8):8694–8715, Aug 2021. https://doi.org/10.1007/s11227-020-03564-9.

[2] Dawei Cheng, Xiaoyang Wang, Ying Zhang, and Liqing Zhang. Graph neural network for fraud detection via spatial-temporal attention. *IEEE Transactions on Knowledge and Data Engineering*, 34(8):3800–3813, 2022. https://doi.org/10.1109/TKDE.2020.3025588.

[3] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. https://arxiv.org/abs/1903.02428.

[4] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1263–1272. JMLR.org, 2017. https://dl.acm.org/doi/10.5555/3305381.3305512.

[5] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association. https://dl.acm.org/doi/10.5555/2387880.2387883.

[6] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. Dynagraph: Dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '22, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3534540.3534691.

[7] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. Featgraph: A flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. https://dl.acm.org/doi/abs/10.5555/3433701.3433795.

[8] Wengong Jin, Kevin Yang, Regina Barzilay, and Tommi Jaakkola. Learning multimodal graph-to-graph translation for molecular optimization. *ICLR*, 2019. https://arxiv.org/abs/1812.01070.

[9] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *J. Mach. Learn. Res.*, 21(1), jan 2020. https://www.jmlr.org/papers/volume21/19-447/19-447.pdf.

[10] Sai Charan Koduru, Rajiv Gupta, and Iulian Neamtiu. Size oblivious programming with infinimem. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, page 3–19, Berlin, Heidelberg, 2015. Springer-Verlag. https://doi.org/10.1007/978-3-319-29778-1_1.

[11] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 1269–1278, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3292500.3330895.

[12] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 31–46, USA, 2012. USENIX Association. https://dl.acm.org/doi/10.5555/2387880.2387884.

[13] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014. http://snap.stanford.edu/data.

[14] Xue Li, Ke Meng, Lu Qin, Longbin Lai, Wenyuan Yu, Zhengping Qian, Xuemin Lin, and Jingren Zhou. Flash: A framework for programming distributed graph processing algorithms. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 232–244, 2023. https://doi.org/10.1109/ICDE55515.2023.00025.

[15] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, page 340–349, Arlington, Virginia, USA, 2010. AUAI Press. https://dl.acm.org/doi/10.5555/3023549.3023589.

[16] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 443–457, USA, 2019. USENIX Association. https://www.usenix.org/system/files/atc19-ma_0.pdf.

[17] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. EvolveGCN: Evolving graph convolutional networks for dynamic graphs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 5363–5370. AAAI Press, 2020. https://arxiv.org/abs/1902.10191.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., Red Hook, NY, USA, 2019. https://dl.acm.org/doi/10.5555/3454287.3455008.

[19] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. In *ICML 2020 Workshop on Graph Representation Learning*, 2020. https://arxiv.org/abs/2006.10637.

[20] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzmán López, Nicolas Collignon, and Rik Sarkar. Pytorch geometric temporal: Spatiotemporal signal processing with neural machine learning models. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, page 4564–4573, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3459637.3482014.

[21] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, WSDM '20, page 519–527, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3336191.3371845.

[22] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *Trans. Neur. Netw.*, 20(1):61–80, jan 2009. https://doi.org/10.1109/TNN.2008.2005605.

[23] Zechao Shang, Jeffrey Xu Yu, and Zhiwei Zhang. Tufast: A lightweight parallelization library for graph analytics. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 710–721, 2019. https://doi.org/10.1109/ICDE.2019.00069.

[24] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2442516.2442530.

[25] Wen Torng and Russ B. Altman. Graph convolutional neural networks for predicting drug-target interactions. *Journal of Chemical Information and Modeling*, 59(10):4131–4149, 2019. https://pubs.acs.org/doi/10.1021/acs.jcim.9b00628.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. https://dl.acm.org/doi/10.5555/3295222.3295349.

[27] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. A semi-supervised graph attentive network for financial fraud detection. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 598–607, 2019. https://arxiv.org/abs/2003.01171.

[28] 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. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. https://arxiv.org/abs/1909.01315.

[29] Ruijie Wang, Zheng Li, Dachun Sun, Shengzhong Liu, Jinning Li, Bing Yin, and Tarek Abdelzaher. Learning to sample and aggregate: Few-shot reasoning over temporal knowledge graphs. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16863–16876. Curran Associates, Inc., 2022. https://arxiv.org/abs/2210.08654.

[30] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2628–2638, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3448016.3457564.

[31] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2851141.2851145.

[32] Yufeng Wang and Charith Mendis. Tgopt: Redundancy-aware optimizations for temporal graph attention networks. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 354–368, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3572848.3577490.

[33] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. Seastar: Vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 359–375, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3447786.3456247.

[34] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. In *International Conference on Learning Representations (ICLR)*, 2020. https://arxiv.org/abs/2002.07962.

[35] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 5171–5181, Red Hook, NY, USA, 2018. Curran Associates Inc. https://dl.acm.org/doi/10.5555/3327345.3327423.

[36] Yuyue Zhao, Xiang Wang, Jiawei Chen, Yashen Wang, Wei Tang, Xiangnan He, and Haiyong Xie. Time-aware path reasoning on knowledge graph for recommendation. *ACM Trans. Inf. Syst.*, 41(2), dec 2022. https://doi.org/10.1145/3531267.

[37] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. Tgl: A general framework for temporal gnn training on billion-scale graphs. *Proc. VLDB Endow.*, 15(8):1572–1580, apr 2022. https://doi.org/10.14778/3529337.3529342.

[38] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, Xiangwen Liu, and Hanqing Wu. Ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 878–891, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3575693.3575723.