T-GCN: A Sampling Based Streaming Graph Neural Network System With Hybrid Architecture

Chengying Huan
Tsinghua University
Baihai Technology Inc.
hcy@baihai.ai

Shuaiwen Leon Song
University of Sydney
leonangel991@gmail.com

Yongchao Liu
Ant Group
yongchao.ly@antgroup.com

Heng Zhang
Chinese Academy of Sciences
zhangheng17@iscas.ac.cn

Hang Liu
Stevens Institute of Technology
hliu77@stevens.edu

Charles He
Ant Group
changhua.hch@antgroup.com

Kang Chen
Tsinghua University
chenkang@tsinghua.edu.cn

Jinlei Jiang
Tsinghua University
jjlei@tsinghua.edu.cn

Yongwei Wu
Tsinghua University
wuyw@tsinghua.edu.cn

ABSTRACT

As many real-world applications are streaming and attached with time instances, a few works have been proposed to learn streaming graph neural networks (GNNs). Unfortunately, current streaming GNNs are observed to have a large training overhead and suffer from bad parallel scalability on multiple GPUs. These drawbacks pose severe challenges to online learning of streaming GNNs and their application to real-time scenarios. To improve training efficiency, one promising solution is to use sampling, a technique widely used in static GNNs. However, to the best of our knowledge, sampling has not been investigated in learning streaming GNNs. Based on these observations, in this paper, we propose T-GCN, the first sampling-based streaming GNN system, which targets temporal-aware streaming graphs and takes advantage of a hybrid CPU-GPU co-processing architecture to achieve high throughput and low latency. T-GCN proposes an efficient sampling method, namely *Segment Its Search*, to offer high sampling speed with respect to three typical types of general graph sampling methods (i.e., node-wise, layer-wise, and subgraph sampling). We propose a locality-aware data partitioning method to reduce CPU-GPU communication latency and data transfer overhead, and an NVLink-specific task schedule to fully exploit NVLink's fast speed and improve GPU-GPU communication efficiency. Besides, we further pipeline the computation and the communication by introducing an efficient memory management mechanism, to improve scalability while hiding data communication. Overall, with respect to end-to-end performance, for single-GPU training, T-GCN achieves up to 7.9× speedup than state-of-the-art works. In terms of scalability, T-GCN runs 5.2× faster on average with 8 GPUs than one GPU. Additionally, in terms of sampling, T-GCN also yields a maximum of 38.8× speedup with our *Segment Its Search* sampling method.

PACT '22, October 10–12, 2022, Chicago, IL, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/10.1145/3559009.3569648

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Graph neural networks, GPU, Sampling

ACM Reference Format:
Chengying Huan, Shuaiwen Leon Song, Yongchao Liu, Heng Zhang, Hang Liu, Charles He, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2022. T-GCN: A Sampling Based Streaming Graph Neural Network System With Hybrid Architecture. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3559009.3569648

1 INTRODUCTION

Graph is an efficient information carrier and is widely used in different real-world applications. Graph embedding has been used in many applications such as graph classification [1, 6] and node classification [8, 15], which are categorized into two types: Skip-Gram models [30] and graph neural networks (GNNs) [8, 15, 34]. Compared with SkipGram, GNNs can efficiently capture nodes' features and structures of graphs by machine learning techniques and thus can get better prediction performance.

However, current works mostly focus on static graph learning methods rather than streaming graph learning. Mostly real-world graphs are evolving over time and are represented as streaming edges. In these cases, edges in graphs will be attached with time instances to show temporal information in real-world applications. Furthermore, edges are in streaming formats and need to train the model online with regard to new edges. For example, citation networks [11] are attached with time instance and grow with time. In this case, there is a necessity to retrain the dynamic parts for some models such as EHNA [11]. In addition, many other domains such as e-commerce [47], education [16], and social networks [26] also evolve with dynamic information. Some streaming graph learning methods are proposed to solve updating problems of dynamic models by means of incremental training with no need of re-training but still suffer from large latency and low throughput problems. Therefore, there are several challenges in training streaming GNNs.

The first challenge is **sampling performance**. Applying sampling methods to GNN is important for performance acceleration [2, 3, 8, 12, 13, 44, 45]. However, conventional sampling algorithms have large sampling complexity. For example, **Bipartite Region Search** (BRS) [27], which is proposed for accelerating subgraph sampling, has to spend 1.5 hours on sampling the Twitter dataset (with 1.46B edges). Therefore, it is an important issue to identify a good sampling method that has a small overhead but does not lower the final accuracy of models.

The second challenge is **training speed**. A few streaming graph learning methods have been proposed for high accuracy but incur high training overhead. For example, DynamicTriad [49], HTNE [52], and EvolveGCN [28] capture temporal information along with local structure by means of training different snapshots. But in processing the arrival of new edges, these works don't have good scalability for training new models and also have high training complexity for new patterns. This performance problem also exists in random walk-based models such as CTDNE [26] and EHNA [11]. This is because random walk sets must be updated for all walkers affected by new edges, thus causing much updating overhead. Along with the updates of random walk sets, the Skip-Gram models have to be fine-tuned with the new sets accordingly and therefore incur extra training overhead. Recently, some recurrent neural networks (RNNs)-based models, e.g., JODIE [16] and DyGNN [23], have been proposed to deal with streaming graphs, as these models can be updated incrementally with modest additional overhead. However, these models have been designed regardless of the overhead of iterative training, which is typically much larger than the inference overhead and should be optimized carefully. In addition, low throughput and high latency are important performance problems faced by RNNs.

The third challenge is **scalability** for multi-GPU training. Many streaming graph learning models intend to achieve good accuracy but seldom pay attention to multi-GPU scalability. We have observed that relatively large real-world datasets consume a lot of memory, caused by a large number of nodes and edges as well as the huge amount of node/edge hidden embeddings (or activations). Especially for node/edge embeddings, they take so large an amount of memory that they cannot be entirely stored on a single GPU. Not only this, a single GPU training will take too much training time. In this case, users have to resort to multiple GPUs to train the model. To achieve high-efficiency multi-GPU training, we need to address two challenges. One is how to partition both node embedding and datasets to multiple GPUs. It is known that inter-device communication (GPU to GPU and CPU to GPU) plays an important role in training, because of the huge size of node features or embeddings. Current GPU-GPU communication via NVLink usually has a much faster speed than the CPU-GPU communication via PCIe. Therefore, partitioning is an important stage to ensure low communication overhead by jointly optimizing PCIe-based CPU-GPU and NVLink-based GPU-GPU communications. The other is how to schedule tasks (keeping edges' training order) without affecting accuracy. Naïve distributed training will affect the temporal information embedded in datasets and thereby reduce the accuracy. In the literature, some works have been conducted to use multi-GPUs to train GNNs but expose different drawbacks. For instance, NeuGraph [22] has much communication overhead caused by the inefficient data partition method. Other works like DGL-KE [48] and PyTorch-BigGraph (PBG) [20] suffer from low GPU utility in distributed training. Overall, the aforementioned three challenges are important issues that need to be addressed by streaming GNN learning.

In this paper, we propose T-GCN, the first streaming GNN system with hybrid architecture for accelerating GNN learning on temporal-aware streaming graphs, while keeping prediction accuracy well. To deal with the sampling performance challenge, T-GCN provides a novel sampling method, namely *Segment Its Search*, which can achieve high performance on three typical types of general graph sampling methods, including node-wise, layer-wise, and subgraph sampling. The evaluation shows that T-GCN can get up to 38.8× speedup in terms of sampling speed over the state-of-the-art sampling algorithms. To cope with the training speed and scalability challenges, T-GCN employs a hybrid CPU-GPU co-processing architecture and puts forward a locality-aware data partitioning method to reduce CPU-GPU data transfer overhead, lower CPU-GPU communication latency, and improve the training throughput. To further reduce GPU-GPU communication, T-GCN proposes an efficient task schedule to maximize the exploitation of NVLink's fast speed. To get better scalability, T-GCN introduces efficient memory management to pipeline the computation and communication. For single V100 GPU training, relying on these optimizations, T-GCN runs up to 7.9× faster than the current works concerning end-to-end performance. When running on an NVIDIA DGX server with eight V100 GPUs, the training speed of T-GCN on 8 GPUs is 5.2× faster than the training speed of T-GCN on one GPU, demonstrating good scalability.

2 BACKGROUND

2.1 Notations of Streaming Graphs

As mentioned in Section 1, T-GCN targets temporal-aware streaming graphs and therefore chooses to use a temporal graph format to represent the underlying graphs for streaming GNN learning. In the following, we will briefly describe the data format of a temporal graph. The main difference between temporal graphs and static graphs is that edges in temporal graphs are attached with time instances. For graph $G = (V, E)$, V is the vertex set and E is the edge set. For each edge $e = (u, v, t)$ in E, $u \in V$ is the source node, $v \in V$ is the destination, and $t \in \mathbb{R}$ refers to the time instance. The time instances of edges between the same vertices can be different. Therefore, each edge has a certain time instance but each vertex is agnostic to time because each vertex may connect multiple edges with different timestamps. An important problem faced by temporal graphs is that each path must satisfy the time constraints. In other words, for a temporal path $P = \{e_1, e_2, \ldots, e_n\}$ with $e_i = (u_i, u_{i+1}, t_i)$ and $e_j = (u_j, u_{j+1}, t_j)$, if $j \leq i$, it satisfies that $t_j \leq t_i$. Constructing paths in chronological order is a widely-used criterion in existing temporal graph models [39, 40]. The time constraints on temporal paths will bring extra computation and space overhead.

In real-world applications, temporal graphs will be created by order of time instance, and this representation of temporal graphs is usually called the edge stream. This means that edges in the edge stream are sorted by increasing order of time. For example, the trade logs of e-commerce are recorded with time increasing. The edge stream is a widely used data model of temporal graphs [10, 16, 26].

2.2 Streaming Graph Neural Network

In general, GNNs work by aggregating embeddings from neighbors on each iteration round. Typical example GNNs include GCN [15], GraphSAGE [8], GAT [34], and GIN [41]. As shown in Equation 1, in each iteration $k+1$, the *Aggregate* function will aggregate embedding of neighbors for each vertex in the previous iteration. Assume that $N(u)$ is the neighbor set of u, h_u^k is the embedding of u at iteration k, and a_u^{k+1} is the aggregation result of iteration $k+1$. After aggregation, neural network functions such as multi-layer perceptrons (MLPs) will be used to combine h_u^k with a_u^{k+1}. As seen in Equation 2, *Combine* function will use neural networks to transform the aggregation result into embedding of u at the iteration $k+1$ as h_u^{k+1}. We take the node classification and link prediction as examples to introduce GNNs. Both of them aggregate embedding from neighbors. The *Combine* can be expressed by $w \cdot a_v^{k+1}$ with MLP operations.

$$a_u^{k+1} = Aggregate(h_v^k | v \in N(u)) \tag{1}$$

$$h_u^{k+1} = Combine(h_u^k, a_u^{k+1}) \tag{2}$$

Different from static versions, streaming GNNs incorporate time instances into machine learning models and capture temporal information along with local structure (e.g., learn embeddings from a sequence of graph snapshots) so as to improve the expressive ability of the models. Some streaming GNNs [16, 23] have been proposed with RNNs or its variants (LSTMs [9] and GRUs [5]) at the core to support time instances. These RNN-based models are inherently capable of processing time instances, dealing with new additions of dynamic edges, and enabling incremental training along with the coming of new edges. DyGNN is a state-of-the-art RNN-based streaming GNN, which has no constraint on the types of streaming graphs and also has good prediction performance. DyGNN uses LSTMs as the core to train streaming graphs. In each round, it passes through the entire graph and uses all edges. For the update of a newly added edge, the computation consists of three units, namely *Interact Unit*, *Update Unit*, and *Merge Unit*. In this paper, we will use DyGNN as a typical use case to demonstrate the power of our framework.

Interact Unit: For an interaction $e = \{v_s, v_g, t\}$, it generates interaction information for e from node information, denoted as $e(t)$ (refer to Equation 3). $E_{v_s}(t-)$ and $E_{v_g}(t-)$ is the feature/embedding of the source node v_s and the destination node v_g at time $t-$, which is right before time t (in other words, $t-$ is infinitely close to t but prior to t), respectively. W_1, W_2, and b_e are the parameters of neural networks. $act(\cdot)$ is an activation function that can be `sigmoid` or `tanh`. The resulting interaction information $e(t)$ is computed as

$$e(t) = act(W_1 \cdot E_{v_s}(t-) + W_2 \cdot E_{v_g}(t-) + b_e) \tag{3}$$

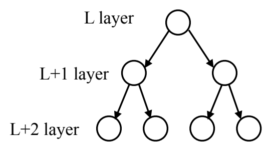


Figure 1: Node-Wise.

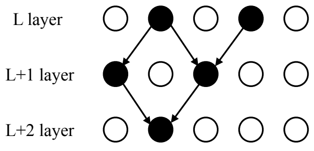


Figure 2: Layer-Wise.

Update Unit: The update unit applies the interaction information $e(t)$ generated from the interacting unit to the nodes participating in the interaction. This unit uses two variants of LSTM, which additionally incorporate time interval information to control the magnitude of forgetting, to update v_s and v_g, respectively. One variant, namely *S-Update*, only updates the source information of a node v, i.e., the cell and hidden states of v, when v is the source node of an iteration $e(t)$. The other variant, namely *G-Update*, only updates the destination information of the same node v when v is the destination node of $e(t)$. *S-Update* and *G-Update* have the same network structure, but with different parameters. Equations 4 and 5 show the update logic of *S-Update* and *G-Update*, respectively. $C_{v_s}^s(t-)$ is the cell state of the source node v_s and $h_{v_s}^s(t-)$ is the hidden state of v_s, at time $t-$. Δ_t is the time interval equal to $t-(t-)$, and $Update(\cdot)$ represents the neural network computation.

$$C_{v_s}^s(t),\ h_{v_s}^s(t) = Update(C_{v_s}^s(t-),\ h_{v_s}^s(t-),\ \Delta_t,\ e(t)) \tag{4}$$

$$C_{v_g}^g(t),\ h_{v_g}^g(t) = Update(C_{v_g}^g(t-),\ h_{v_g}^g(t-),\ \Delta_t,\ e(t)) \tag{5}$$

Merge Unit: For v_s, we have two hidden states $h_{v_s}^s(t)$ and $h_{v_s}^g(t-)$ from the output of *S-Update*. Similarly, for v_g, we have $h_{v_g}^s(t-)$ and $h_{v_g}^g(t)$ from the output of *G-Update*. In this case, the merge unit takes the hidden states from *S-Update* to generate the new embedding $E_{v_s}(t)$ for v_s, and the hidden states from *G-Update* to generate $E_{v_g}(t)$. Equations 6 and 7 show the merge logic for v_s and v_g, respectively, where W_s, W_g and B are trainable parameters.

$$E_{v_s}(t) = W_s \cdot h_{v_s}^s(t) + W_g \cdot h_{v_s}^g(t-) + B \tag{6}$$

$$E_{v_g}(t) = W_s \cdot h_{v_g}^s(t-) + W_g \cdot h_{v_g}^g(t) + B \tag{7}$$

2.3 Graph Sampling

Graph sampling is an important technology for large graphs training to solve the problem that both the graph and the intermediate embeddings (or activations) cannot be entirely stored in GPU memory. This technology can accelerate training on GPUs to some degree but has the major drawbacks of accuracy loss [14, 33] and prediction instability [7]. Popular graph sampling methods include node-wise sampling [8], layer-wise sampling [2] and subgraph sampling [4].

Node-Wise Sampling: GraphSAGE [8] introduces a node-wise random sampling approach to obtaining k-hop neighbors. As seen in Figure 1, for multi-layer GNNs, this method samples a specified number of neighbors of each vertex at each layer. The sampled neighbors for each vertex are independent of each other. The number of sampled nodes grows exponentially with the number of layers. For k layers, the number of sampled nodes in the input layer can explosively increase up to $O(\bar{D}^k)$ with $\bar{D}$ as the average degree per layer.

Algorithm 1 Random walk based subgraph sampling.

```
1:  Input: Original graph G(V,E); Frontier size M; Subgraph size N.
2:  Output: Induced subgraph $G_{sub}(V_{sub}, E_{sub})$.
3:  $FS \leftarrow$ Uniformly select $m$ vertices at random from $V$
4:  for $i$ in range $[1, n]$ do
5:      Select $u \in FS$ with probability $\frac{deg(u)}{\sum_{v \in FS} deg(v)}$
6:      $V_{sub} \leftarrow V_{sub} \cup \{u\}$
7:      Select $(u, u') \in E$ randomly in neighbors of $u$
8:      $FS \leftarrow (FS \setminus u) \cup \{u'\}$
9:  end for
10: $G_{sub} \leftarrow$ Subgraph of $G$ induced by $V_{sub}$ return $G_{sub}$
```

Algorithm 2 Bipartite Region Search.

```
1: Generate a random number $r$
2: Use $r$ to select a vertex in FS. If the vertex has not been selected, done. Otherwise, the region that $r$ falls into corresponds to a pre-selected vertex. Assume the boundary of this region in FS is $(l, h)$.
3: Let $\lambda \leftarrow 1/(1-(h-l))$, $\delta \leftarrow h - l$
4: $r \leftarrow r/\lambda$
5: if $r < l$ then
6:     Search $r$ in $(0, l)$
7: else
8:     $r \leftarrow r + \delta$ and search $r$ in $(h, 1)$
9: end if
```

Layer-Wise Sampling: FastGCN [2] proposes a layer-wise sampling approach, which samples neighbors for each vertex but shares the sampled neighbors among all nodes of the current layer (see Figure 2). The sharing property fits the message passing strategy of GNNs and makes the number of sampled nodes linearly proportional to the number of layers.

Subgraph Sampling: To achieve better performance on large graphs, some works put forward to construct mini-batches from subgraphs. ClusterGCN [4] is proposed to partition the graphs into a set of clusters before training and construct a mini-batch by randomly choosing and merging several clusters. GraphSAINT [45] employs **random walk based samplers** and achieves better accuracy than ClusterGCN. Algorithm 1 shows the random walk-based sampler. To sample a subgraph $G_{sub}(V_{sub}, E_{sub})$ of n vertices, it first provides a frontier FS with m seeding vertices that are randomly selected. We will repeat the following operations until the construction of G_{sub} is completed. Firstly, a vertex u is selected from FS with a probability of $\frac{deg(u)}{\sum_{v \in FS} deg(v)}$ where $deg(u)$ is the degree number of vertex u, and u is added to V_{sub}. Secondly, a neighbor vertex u' of u is randomly chosen to replace u in FS. Finally, after n times repetitive execution of the above operations, we have got G_{sub} constructed. This sampling method is more complex and time-consuming. C-SAW [27] proposes a **Bipartite Region Search** to implement random walk based samplers. As shown in Algorithm 2, Bipartite Region Search re-calculates the edge selection probability on the changed FS. One drawback of this method is that it is designed for individual sampling only. For multiple sampling (i.e., multiple vertex replacements in FS), it needs many efforts. In specific, there will need $O(n)$ time complexity on average for each sampling with n as the size of FS.

2.4 Challenges

In the following, we will elaborate on the three challenges mentioned in Section 1, namely sampling performance challenge, training speed challenge, and scalability challenge on multi-GPUs.

Sampling performance challenge: The goal is to accelerate GNN training without loss of accuracy. For node-wise and layer-wise sampling, to sample a neighbor of each vertex, current works firstly generate a random number and then use the current sampling algorithms such as the alias method [21] and ITS [24] to sample a neighbor of this vertex. If the sampled vertex is already selected, it needs to repeat the selection procedure until getting an unselected neighbor. Apparently, this repetitive sampling behavior will cause heavy sampling overhead for a large number of sample vertices, i.e., each sampling process takes up to $O(n)$ time complexity with n denoting the sampling space. For subgraph sampling, C-SAW [27] uses the **Bipartite Region Search** (BRS) to optimize. However, BRS still needs up to $O(n)$ time complexity for each sampling process.

Training speed challenge: Some works like DynamicTriad [49], HTNE [52], and EvolveGCN [28], train the snapshots of the streaming graph will suffer from the high overhead of incremental neural network updating and high overhead of models training. Some RNN-based works such as DyGNN [23] and JODIE [16] reduce the the former overhead but still do not regard the latter. All these works do not support multi-GPU training, resulting in low throughput of training, e.g., their throughput (calculated by dividing the number of labeled edges by the time per epoch) is as low as only 34K edges/s while the throughput of T-GCN is up to 815.7K edges/s.

Scalability challenge: Current streaming GNN models focus on accuracy other than on speed and scalability of multi-GPU training. More importantly, current multi-GPU training frameworks do not demonstrate good scalability. For instance, NeuGraph [22] proposes to partition graph with a graph schedule strategy, but still has much communication overhead due to the data locality problem and can not make full use of fast GPU-GPU communication. DGL-KE [48] stores parameters in the CPU memory and uses synchronous training on GPUs by mini-batches. However, DGL-KE has low GPU utilization (nearly 10%), caused by the large data transfer between GPU and CPU. PyTorch-BigGraph (PBG) [20] partitions nodes into a set of disjoint parts, and stores them in the disk. During training, it directly loads one or more partitions entirely into GPU to avoid frequent data movement from disk to GPU. Nonetheless, this also results in low GPU utility, only 30%, caused by data swap between disk and GPU. Marius [25] introduces a buffer-aware edge traversal algorithm to reduce disk I/O and improve GPU utilization. However, this additional disk I/O and data movement will lead to longer training time, i.e., 3.5 hours per epoch for the *Twitter* graph training with 1.46 billion edges (throughput is only 115.9K edges/s).

Our objective: T-GCN proposes a novel sampling algorithm *Segment Its Search* to address the sampling performance challenge. We use the hybrid architecture to train the streaming graph neural network models on multi-GPUs and propose a novel locality-aware

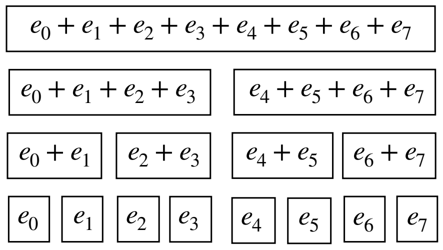

$e_0 + e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7$
$e_0 + e_1 + e_2 + e_3$
$e_4 + e_5 + e_6 + e_7$
$e_0 + e_1$
$e_2 + e_3$
$e_4 + e_5$
$e_6 + e_7$
e_0 e_1 e_2 e_3 e_4 e_5 e_6 e_7

Figure 3: Sampling algorithm example.

data partition, task schedule, and GPU memory management strategy to solve the training speed and scalability challenges.

3 STREAMING GRAPH SAMPLING

To solve the sampling performance challenge, we propose a general sampling method that efficiently deals with node-wise, layer-wise, and subgraph sampling for general graphs. The core idea of our sampling method is to leverage an efficient data structure *Segment Its Search*, which is similar to the segment tree and binary tree, to facilitate fast sampling. Figure 3 describes an example workflow of our *Segment Its Search* sampling. Each e_k represents the weight of the vertex or edge. For example, e_k can represent the degree of each vertex for subgraph sampling. For node-wise and layer-wise sampling, e_k can be 1 for unbiased sampling. Users can customize e_k for their self-defined sampling methods. Specifically, for a fixed array (say the neighbor sets of node-wise sampling, the neighbor sets of layer-wise sampling, or the frontier in subgraph sampling), we process a divide-and-conquer method by first dividing the sum of edge sets $\{e_0, \ldots, e_7\}$ into the sum of two subsets: one is from the subset $\{e_0, \ldots, e_3\}$ and the other from the subset $\{e_4, \ldots, e_7\}$. Subsequently, these two sets can be further partitioned into four parts: $\{\{e_0, e_1\}, \ldots, \{e_6, e_7\}\}$. Having got the sum array determined, replacing edge e_4 will change the sum values of $\{\{e_4\}, \{e_4, e_5\}, \{e_4, e_5, e_6, e_7\}, \{e_0, \ldots, e_7\}\}$. On query, the process will be (1) firstly generating a random number r, and (2) then checking which edge sets it falls into from top to bottom. For example, if r lies in the edges set $\{\{e_i, \ldots, e_j\}\}$, we can figure out that r falls into $\{e_i, \ldots, e_{(i+j)/2}\}$ or $\{e_{(i+j)/2+1}, \ldots, e_j\}$. If r is in the second edge set, we set r to $r - (\sum_{k=i}^{k \leq (i+j)/2} e_k)$ and then search into the set, which will be further split into halves. The search continues until meeting the final edge set that contains only one edge e_w. The data structure *Segment Its Search* needs to maintain the influenced edge sets accordingly. Specifically, the weight of all edge sets containing the sampled edge e_w needs to be updated. For node-wise and layer-wise sampling, we subtract the weight by $Weight(e_w)$. For subgraph sampling, we set $deg(v_w)$ to the corresponding vertices in the frontier, where $deg(v_w)$ is the degree number of vertice v_w.

Algorithm 3 shows the workflow of the sampling process. The **Build** function is to build a binary tree by calculating the range sum array of the candidate vertices set $\mathbb{V}$. For each range $[L, R]$, it will create a node with the sum of vertices' weights in this range which is represented as $\mathbb{V}[L].weight$ and the node will connect to its left node in the range $[L, \frac{L+R}{2}]$ and to the right node in the range $[\frac{L+R}{2} + 1, R]$. The **Search** function is to search the sampled vertex

Algorithm 3 Workflow of *Segment Its Search* sampling.

```
function BUILD($\mathbb{V}$, $L$, $R$)
    if $L == R$ then
        $T = NULL$
        $T.sum = \mathbb{V}[L].weight$
        Return $T$
    end if
    $T = NULL$
    $T.left = Build(\mathbb{V}, L, \frac{L+R}{2})$
    $T.right = Build(\mathbb{V}, \frac{L+R}{2} + 1, R)$
    $T.sum = T.left.sum + T.right.sum$
end function
function SEARCH($root$, $L$, $R$, $g$)
    if $L == R$ then
        Return $L$
    end if
    if $g > root.left.sum$ then
        Return $Search(root.right, \frac{L+R}{2} + 1, R, g - root.left.sum)$
    else
        Return $Search(root.left, L, \frac{L+R}{2}, g)$
    end if
end function
function MODIFY($root$, $L$, $R$, $pos$, $new\_sum$)
    if $L == R$ then
        $root.sum = new\_sum$
    end if
    if $(L + R)/2 < pos$ then
        $Modify(root.right, \frac{L+R}{2} + 1, R, pos, new\_sum)$
    else
        $Modify(root.left, L, \frac{L+R}{2}, pos, new\_sum)$
    end if
    $root.sum = root.left.sum + root.right.sum$
end function
```

in the binary tree by binary search, i.e., if the sampled number g is larger than the left node's sum, g will go to the right node and then be replaced by $g - root.left.sum$. The **Modify** function is to change the value of a certain position of the binary tree and then update all nodes including this position. Actually, there will be up to $log(V)$ nodes needed to be updated with V is the vertex set.

On **Sampling**, the implementation of the sampling will be specialized for each type of sampling method. We take the node-wise sampling as an example which is shown in Algorithm 4. For each sampling on a certain vertex set $\mathbb{V}$, it first creates the binary tree T by **Building** function. For the next n times sampling, it generates a random number g and searches the sampled edge in $\mathbb{V}$ by means of binary searching g in T. The parameter n decides the sampled size which is user-customizable and is set to half of the neighbor size in our evaluations. The sampled vertices are recorded in the vertex set P, same as ITS [24]. For the main sampling process which samples batches for training, **Sample_batch** collects all batches from the current dataset. It first generates a vertex set from a vertex v as the root and samples neighbors of vertices in the previous layers on each layer of GNNs. The network depth in Algorithm 4 is

Algorithm 4 Node-wise sampling.

```
function SAMPLE($\mathbb{V}$, $n$)
    $T = Build(\mathbb{V},\ 1,\ \|\mathbb{V}\|)$
    $P = NULL$
    while $n > 0$ do
        generate a random number $g$
        $L = Search(T,\ 1,\ \|\mathbb{V}\|,\ g)$
        $Modify(T,\ 1,\ \|\mathbb{V}\|,\ L,\ 0)$
        $P = P \cup \mathbb{V}[L]$
        $n = n - 1$
    end while
    Return $P$
end function
function SAMPLE_BATCH($V$, $n$)
    $Batch\_set = Empty$
    for $each\ vertex\ v\ in\ V$ do
        $B = Empty$
        $P = v$
        for $1\ to\ network\ depth$ do
            $\mathbb{V} = Empty$
            for $each\ vertex\ v\ in\ P$ do
                $\mathbb{V} = \mathbb{V} \cup neighbors\ of\ v$
            end for
            $n = sampled\_size$ for the current depth
            $P$ = Sample($\mathbb{V},\ n$)
            $B = B \cup P$
        end for
        $Batch\_set = Batch\_set \cup B$
    end for
    Return $Batch\_set$;
end function
```

the number of graph convolutional layers in GNNs. The sampled vertices and edges will be viewed as a batch B and all batches B constitute the batch set $Batch\_set$. Layer-wise sampling adopts the same **Sampling** function with node-wise sampling.

Algorithm 5 gives the pseudocode of our subgraph sampling. The **Sampling** function will sample a subgraph of n vertices with m being the frontier array size in Section 2.3. Compared with node-wise sampling in Algorithm 4, the main difference is the introduction of the **Modify** function, which replaces the current sampled vertex $FS[L]$ in the frontier FS with the sampled neighbor v' of $FS[L]$. Finally, the *Sample_Batch* function will generate S subgraphs with each inserted into $Batch\_set$.

Sampling Complexity Analysis: The sampling overhead of *Segment Its Search* includes the overhead from the query and modify operations. For a query operation, since the Search function is similar to the binary search, the time complexity of the query operation is $O(log(n))$, where n is the size of the set $\mathbb{V}$. For a modify operation, *Segment Its Search* also needs $O(log(n))$ time complexity. Therefore, the total time complexity of *Segment Its Search* is $O(log(n))$ too. In contrast, both ITS [24] and BRS [27] need nearly $O(n)$ time complexity. Not only this, *Segment Its Search* will not suffer from repetitive sampling problems because *Segment Its Search* can change the probability distribution of sampling by the **Modify**

Algorithm 5 Subgraph sampling.

```
function SAMPLE($V$, $n$, $m$)
    $FS$ = randomly select $m$ vertices from $V$
    $V_{sub} = NULL$
    $T = Build(FS,\ 1,\ \|FS\|)$
    while $n > 0$ do
        generate a random number $g$
        $L = Search(T,\ 1,\ \|FS\|,\ g)$
        $V_{sub} = V_{sub}\ \cup\ FS[L]$
        $v'$ = sample a neighbor of $FS[L]$
        $Modify(T,\ 1,\ \|FS\|,\ L,\ deg(v'))$
        $FS[L] = v'$
        $n = n - 1$
    end while
    Return $V_{sub}$
end function
function SAMPLE_BATCH($V$, $S$, $n$, $m$)
    $Batch\_set = Empty$
    for $i\ in\ range\ [1, S]$ do
        $Batch\_set = Batch\_set\ \cup\ Sample(V,\ n,\ m)$
    end for
    Return $Batch\_set$;
end function
```

function. Thanks to the logarithmic time complexity, our *Segment Its Search* has good scalability.

4 HYBRID ARCHITECTURE OF T-GCN

Some works provide computing frameworks for GNNs such as DGL [36], AliGraph [50], and PyTorch-BigGraph [20]. These works are designed for the distributed GNN environment with a large communication effort. Some works are designed for single-machine GNN systems such as HyGCN [42] which employs a hybrid architecture (GPU HBM+CPU memory). However, HyGCN only implements inference, excluding training, for GNNs, and therefore lacks a foremost functionality of GNN learning systems. Meanwhile, HyGCN merely uses a single GPU and does not provide an efficient data loading strategy that can exploit NVLink, an important communication technique for multi-GPU training. Hence, HyGCN does not take full advantage of hardware benefits.

Similar to HyGCN, T-GCN also uses a hybrid CPU-GPU co-processing architecture but provides support for both training and inference of GNNs. This hybrid architecture combines GPU HBM with CPU memory to address the problem that graph datasets and intermediate embeddings (or activations) cannot be fully contained in a single GPU HBM. The basic idea is to partition the data into several parts and store them in different hardware memory spaces. However, this incurs data transfer overhead, and how to optimize the data locality to reduce the total data loading overhead is a challenge for current researchers. In this paper, we put forward a locality-aware data partitioning method to improve CPU-GPU communication efficiency and introduce an NVLink-specific task schedule to fully exploit the power of NVLink hardware and thereby increase GPU-GPU communication speed. Furthermore, we introduce an efficient memory management mechanism to facilitate pipelining execution of computation and communication, thus

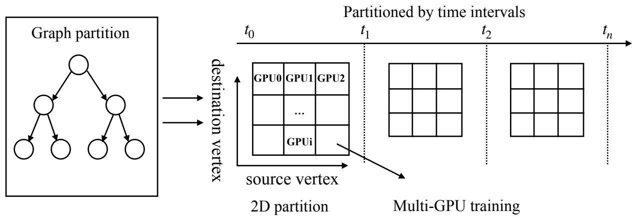


Figure 4: Data partition strategy.

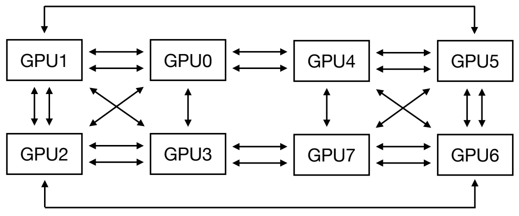


Figure 5: NVLink components.

improving scalability while hiding data communication. Figure 4 illustrates the architecture of T-GCN. Firstly, T-GCN samples the graph in CPU and stores the sampled data in the memory, where the sampled data is partitioned by three dimensions, and the specific partitioning procedure will be discussed in the following. Subsequently, T-GCN loads each partitioned data into GPU for multi-GPU training. On training, the node embedding is partitioned and stored across multiple GPUs.

To deal with the training speed and scalability challenges discussed in Section 2.4, we propose the locality-aware data partitioning, task schedule, and pipeline to improve multi-GPU training performance and achieve good scalability.

4.1 Locality-Aware Data Partition

In T-GCN, we provide an efficient locality-aware data partitioning method, which is based on three dimensions: the first is a time dimension, the second is the source vertex dimension and the third is the destination vertex dimension. In other words, after the time dimension is partitioned, the left is the same as the 2D partition strategy used in traditional graph processing systems [46, 51]. As shown in Figure 4, for each time interval (range of time instance), multiple GPUs are used for training with the edges in the current time interval. The input node embeddings ($E_{v_s}(t-)$ and $E_{v_g}(t-)$ in Section 2.2) are the results from the training procedure corresponding to the previous time interval. A training procedure is conducted based on the edges of the current time interval to compute new embeddings for the nodes affected by these edges and update neural network parameters. With our partitioning method, we do not need to pay attention to the interaction between different time intervals. Instead, we only need to care about multi-GPU training within each time interval. Because three dimensions of the partition strategy will not be affected by degrees, the partition strategy will also not be impacted by the vertex degree distribution of the graph dataset, e.g, power-law degree distribution.

Data Storage: GNN learning stores more data than traditional graph processing systems, including graph data, node embedding, and neural network parameters. The partitioned dataset is stored in CPU DRAM. Each GPU will load the grid, assigned to itself, into its device memory. Some works such as NeuGraph [22] also use the 2D partition but place the node embedding in the CPU memory. In particular, NeuGraph focuses on full-batched training and thus is not suitable for streaming GNNs. To reduce communication, we store the node embedding in GPU HBM and allow each GPU to store an interval of node embedding. In specific, a GPU will deal with all grids with source vertices within the interval assigned to itself. For example, because the nodes $\{u_1, \ldots, u_i\}$ are stored in GPU 0, all edges (u_j, v, t) with $1 \leq j \leq i$ will be trained in GPU 0. Each GPU does not need to store the time instance for each node because the time instances are stored in edges. As mentioned above, edges are organized and trained in grids (refer to Figure 4). Hence, for each GPU, it will access to destination vertices' embedding by on-the-fly transferring from corresponding GPUs via NVLink. As for neural network parameters, they are stored in GPUs and updated in the same way as the traditional synchronized multi-GPU training.

4.2 Task Schedule

We provide an NVLink-specific task schedule to reduce additional NVLink communication caused by data transfer between any pair of GPUs in 8-GPU training. In our 8-GPU computer, the first four GPUs are fully-connected and form a group, while the other four GPUs are fully-connected and form another group. For any pair of GPUs in the same group, the communication overhead is only one NVLink hop. However, for any pair of GPUs from different groups, this overhead will double. In this regard, it would be good to avoid 2-hop inter-group communication during computation. In addition, for eight V100 GPUs, there are two NVLink channels between specific GPUs. This means that we can parallelize node embedding transfer by means of two NVLink channels. In other words, an embedding vector can be transferred by two NVLinks in parallel, e.g., for a 128-length embedding vector, we allow one NVLink channel to process the first 64 elements of the vector and the other to process the remaining. Therefore, besides reducing inter-group communication as mentioned above, it is also important to maximize the use of two NVLinks between GPUs, where some exist in inter-group GPUs and some in intra-group GPUs. Figure 5 illustrates the NVLink connections in our 8-GPU computer, which forms the root cause of our NVLink-specific task schedule.

We can schedule the training task to optimize the communication because each GPU theoretically needs to get all intervals of node embedding for the training procedure corresponding to any time interval. As mentioned above, each GPU has a fixed vertex interval and this GPU will deal with all edges whose source vertices are in this interval but with destination vertices distributed across GPUs. This means, for example, if GPU 0 stores embedding of all nodes in the range of $[0, k]$, it will process all edges with source vertices in $[0, k]$ while the destination nodes are determined by grids. In this regard, we can name a range of destination vertices as a task and schedule the training by rearranging the order of tasks across multiple GPUs. Specifically, we label all eight GPUs from 0 to 7 and allow the i-th GPU to store vertices in the i-th vertex interval.

Algorithm 6 Task Schedule.

```
1:  for $i$ in range $[0, 4)$ do
2:      Swap node embedding of GPU0 and GPU1
3:      Swap node embedding of GPU2 and GPU3
4:      Swap node embedding of GPU4 and GPU5
5:      Swap node embedding of GPU6 and GPU7
6:      Training GPU 0-7
7:      Swap node embedding of GPU0 and GPU4
8:      Swap node embedding of GPU1 and GPU2
9:      Swap node embedding of GPU3 and GPU7
10:     Swap node embedding of GPU5 and GPU6
11:     Training GPU 0-7
12: end for
```

If task j is assigned to i-th GPU, it will deal with the destination vertices in the j-th vertex interval and process the corresponding grid, i.e., the grid with source vertices in the i-th vertex interval and destination vertices in the j-th vertex interval. At initialization, the task assignment vector for GPUs $0 \sim 7$ will be $\{0, \dots, 7\}$. Then, we will change the task assignment to guide the node embedding swapping between inter-group GPUs and intra-group GPUs. The details will be explained below.

As shown in Algorithm 6, firstly, our schedule swaps node embedding (swapping destination vertices embedding, but keeping source vertices embedding unmoved for all GPUs) between intra-group GPUs (lines 3-5) and trains the model with the newly swapped data. Secondly, our schedule swaps node embedding between inter-group GPUs (GPU 0 and GPU 4, GPU 3 and GPU 7, lines 7-10) and then trains the model likewise. After repeating this procedure four times, each GPU has got all node embedding and has trained the model with the corresponding edge data located in its device memory. For eight V100 GPUs, our schedule can not only eliminate 2-hop NVLink communication but also maximize the use of two NVLink channels available between certain GPUs.

4.3 Pipeline and GPU Memory Management

Pipelining is a widely used technique to achieve better performance. In our multi-GPU training, two types of communication are involved, i.e., PCIe and NVLink, as mentioned above. As both types of communication are realized by different hardware interfaces, it is viable and feasible to directly pipeline them. Moreover, communication can also be overlapped with the computation of neural networks. However, there exists data competition for GPU device memory. For example, as mentioned above, neural network parameters are stored in GPU memory. In this regard, the data transferred from CPU to GPU via PCIe will compete for device memory with the neural network parameters. Additionally, node embeddings will also compete with the graph dataset and the parameters. Therefore, there is a necessity to develop a memory management mechanism to properly manage data competition for efficient pipelining.

T-GCN divides the device memory of each GPU into three components. The first component stores neural network parameters and a certain interval of node embedding. The second component stores the current training data, including edges of the current grid and the corresponding destination vertices embedding. Note that according to our data partitioning, a GPU will store a certain interval of source vertices embedding as mentioned above. To train a grid, it will need the embedding of both the source and destination vertices, where the destination vertices embedding will be obtained by data swapping via NVLink. The third component is used to store the next training data, which has the same data format as the second component and is determined by the task assignment vector in our task schedule as described in Section 4.2. The second component is exchangeable with the third one, working similarly to a ping-pong buffer. In other words, after finishing the current step, the second component will point to the storage of the third component and proceed to store the data demanded by the next step, while the third component will point to the storage of the second component and serve as the new current training data. This way, training, and data transfer can be pipelined without any data competition. Not only this, the pipeline is a general approach designed for multi-GPUs and is suitable for different GPU configurations.

Algorithm 7 T-GCN's Workflow.

```
1: function MAIN($E$)
2:     $E'$=Sampling($E$)
3:     $grid_0, \dots, grid_n = Partition(E')$
4:     $Girds = \{grid_0, \dots, grid_n\}$
5:     for each grid in $Grids$ do
6:         $Train(grid, GPU\_n)$
7:     end for
8: end function
```

5 PROGRAMMING INTERFACE

In this section, we will present the programming interface for streaming GNN learning with T-GCN. Different from edge-centric or vertex-centric, which is widely used in traditional graph processing systems [19, 31, 51, 51], our LSTM-based training method can't be easily dealt with by the above programming models because the node embedding is trained by neural networks other than aggregated by edges or vertices as done in traditional graph processing or static GNNs.

The programming interface includes the sampling process and the training process, while sampling has not been investigated by current temporal-aware streaming GNNs [23, 35]. As seen in Algorithm 7, for each training step, it first samples edges from the original dataset and then partitions the sampled dataset into different parts (grids), as described in Section 4.1. Secondly, it trains each partitioned dataset through different GPUs. The main APIs are $Sampling()$, $Partition()$, and $Train()$. $Sampling()$ is used to sample the original dataset to get a sampled dataset (i.e., edges sets for node-wise/layer-wise sampling and subgraph sets for subgraph-based sampling), and users are allowed to rewrite this interface with certain application-specific sampling methods. $Partition()$ is used to partition the sampled dataset into different parts, represented as $Grids$, to match the requirement of our multi-GPU training. Refer to Section 4.1 for the detailed partitioning procedure. Finally, after data partitioning, one grid will be loaded into one GPU to start training the GNN model defined by users with $Train()$. The GPU_n

parameter in $Train()$ represents the setting of GPU numbers for multi-GPU training.

6 EVALUATION

6.1 Experimental Setup

6.1.1 Evaluation Environment. We perform the experiments on an NVIDIA DGX server with eight V100 GPUs. Each GPU has 80 Streaming Multiprocessors, 32GB global memory, and 768KB L2 cache. The system also consists of two 64-core Intel(R) Xeon(R) CPU Platinum 8163 (2.5Hz), and 256GB DDR4 main memory, running with Ubuntu 16.04 (kernel 4.15.0) and CUDA 10.0. Each GPU supports four NVLink link slots, in which two links are connected via NVLink-V2 and the other two links are via NVLink-V1. T-GCN implements sampling methods in C++ and GNNs in PyTorch [29].

Table 1: Real-world temporal graph datasets.

Graph	Vertices ($\|V\|$)	Edges ($\|E\|$)	$Avg(D)$	Feature $\|F\|$
Lastfm	7K	1.3M	364.3	39
Reddit	10K	672K	67	150
Flickr	2.3M	33.1M	28.8	10
Twitter	41.6M	1.47B	74.6	1

6.1.2 Benchmark. We select four popular benchmarks from Koblenz Large Network Collection [17] for our evaluation. Different from the static graph context, we assign each edge from the graph with a random timestamp value from a chosen range of the time period. Same with DyGNN and JODIE, edges with the assigned timestamp value are further categorized into several constant range partitions. The datasets are standard datasets that are used widely in temporal graph engines [39, 40]. Specifically, a temporal graph is usually created by continuously collecting incoming temporal edges, and edges are streaming with the increase of their starting time.

The main attributes of these graphs are listed in Table 1, where $Avg(D)$ represents the average vertex degree. They are especially important attributes for applications whose processing overhead can be significantly affected by the vertex degrees such as the sampling speedup. The degree distribution is power-law for each graph. Since parameter n in Algorithm 4 is set to half of the neighbor size, the sampling of T-GCN does not affect the degree distribution.

The graph data is stored in the CPU memory following the CSR format which is widely used in graph processing engines [18, 32, 51]. The embedding of nodes is partitioned by the locality-aware data partition strategy and stored in GPUs.

6.1.3 Evaluation Methodology. The evaluation uses node classification and link prediction as applications. Both of them are very popular applications for GNNs and other applications are similar to them since the core of them all is generating node or edge embeddings by GNNs.

The learning parameters are set the same as DyGNN [23], e.g., 80% of the edges are used for training, 10% of the edges for validation, and the rest for testing. The parameters of all deep models are set consistently. The number of hidden layers is 2, where the size of each layer is set to 64. The gradients of all trainable parameters are managed by PyTorch and T-GCN stores the global gradient values as additional caching buffers in GPU global memory. Different from static GNN, each streaming update in T-GCN is only updating a small part of the graph. Meanwhile, considering each streaming update is only based on a sampled gradient, T-GCN reduces the GPU memory requirement via utilizing mini-batched SGD, and conducting several updates for each epoch.

Evaluation is conducted from five perspectives: end-to-end performance, sampling performance, runtime piecewise breakdown, GPU device memory overhead, and multi-GPU scalability. The end-to-end performance is indicated by the total execution time including graph sampling and GPU training time. For the runtime piecewise breakdown, we show the corresponding efficiency of our sampling method *Segment Its Search*, task schedule, and memory management (described in Section 4). Note that data partitioning is used to solve the out-of-memory problem encountered by a single GPU and thus its efficiency cannot be evaluated as a single GPU cannot entirely store graph data (including edges data) and node embedding in device memory.

6.2 End-to-End Performance Comparison

Firstly, we conduct an end-to-end performance comparison between two state-of-the-art streaming GNNs, i.e., DyGNN [23] and JODIE [16], and our proposed system T-GCN. Since DyGNN and JODIE only support a single GPU, we benchmark T-GCN on one and eight GPUs to show its good scalability. Experimental results demonstrate that T-GCN significantly outperforms the two counterparts. As shown in Figure 6, on a single GPU supported, T-GCN runs $2.6 \sim 5.1\times$ faster than DyGNN and $2.8 \sim 7.9\times$ faster than JODIE. The geometric average speedup of T-GCN than DyGNN and JODIE are 3.3× and 4.6×, respectively. Note that GPU is not fully utilized on the three relatively small datasets (i.e., Flickr, Lastfm, and Reddit). However, on the largest Twitter dataset, T-GCN finishes training in 9237.4 seconds, whereas both DyGNN and JODIE failed because of the out-of-GPU-memory error caused by their inefficient memory allocation and lack of efficient sampling strategy. For the throughput, DyGNN and JODIE have only 34K edges/s and 60.5K edges/s respectively, while the throughput of T-GCN is 815.7K edges/s. This is because T-GCN can support multi-GPU training with good scalability while DyGNN and JODIE do not.

DyGNN and JODIE are both implemented upon PyTorch [29] through converting irregular graphs to regular tensors, which has to deal with massive padding space. Thus, the tensor conversion operation makes them hard to scale to large temporal graphs, even causing errors on small graphs using the same evaluation configuration. Moreover, they still run multiple times slower than T-GCN as mentioned above. The reasons for achieving efficient large streaming graph training behind T-GCN mainly come from two folds. On one hand, the efficient node-wise, layer-wise, and subgraph sampling methodology for general graphs makes our sampler much faster than others. By introducing *Segment Its Search* data structure, T-GCN facilitates fast sampling and directly converts inputs into a training module. On the other hand, via maximizing memory utilization of GPUs, T-GCN achieves efficient locality-aware streaming data allocation during our streaming GNN training.

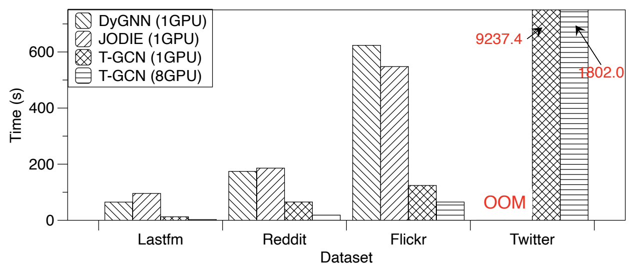
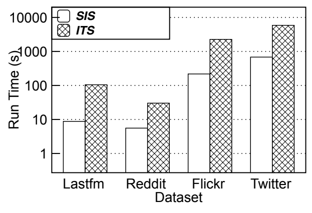
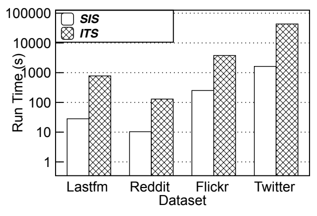


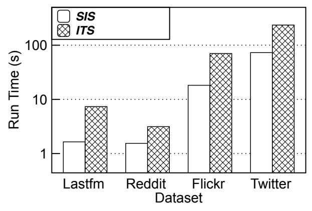
Figure 6: End-to-end performance comparison between DyGNN (1GPU), JODIE (1GPU) and T-GCN (1GPU & 8GPU) on the four datasets. Both DyGNN and JODIE run out-of-GPU-memory on Twitter (marked by red OOM).



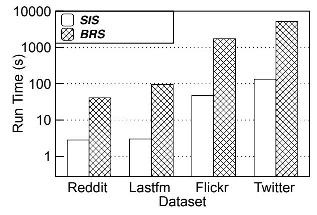
(a) Node-wise sampling (2-layer).



(b) Node-wise sampling (3-layer).



(c) Layer-wise sampling.



(d) Subgraph sampling.

Figure 7: Sampling evaluation on different sampling methods, i.e. node-wise sampling (2-layer and 3-layer), layer-wise sampling, and subgraph sampling.

6.3 Sampling Method Optimization

Secondly, we provide an evaluation of the sampling process. As shown in Section 3, for the node-wise and layer-wise sampling, we compare *Segment Its Search* (**SIS**) to the **ITS** [24] that is a state-of-the-art sampler. For the subgraph sampling, we compare the performance of **SIS** with the Bipartite Region Search [27] (**BRS**) which is designed for subgraph sampling. The evaluation is executed on the four datasets in Table 1. Figure 7 shows the performance evaluation of three sampling algorithms, i.e., node-wise sampling (2-layer and 3-layer), layer-wise sampling, and subgraph sampling.

For the node-wise and layer-wise sampling, we sample 50% edges of the original dataset to form the train set, while current works usually fix the sampling size per layer to a small value, thus not suitable for large power-law graphs. For the subgraph sampling, we set the frontier size as 10^4, the subgraph size as $\frac{|E|}{100}$, and the number of subgraphs as 100 (described in Section 3), similar to the usage in GraphSAINT [45]. For the node-wise sampling, when the layer is two, SIS can achieve 5.4 ∼ 11.9× speedup and is observed to yield higher speedups on the datasets with larger degree numbers. The speedup comes from the sampling complexity improvement as shown in Section 7. When the layer is three, the running time is large because the total sampling overhead is exponential to the 2-layer sampling. SIS can achieve 12.1 ∼ 26.7× speedup on the 3-layer node-wise sampling. Layer-wise sampling has a small sampling overhead because it only needs sampling neighbors of each vertex. Although the total run time is small, SIS can still achieve 2 ∼ 4.5× speedup. The subgraph sampling also takes a smaller sampling overhead than the node-wise sampling but has a larger sampling overhead than layer-wise. For subgraph sampling, SIS can achieve up to 7.1 ∼ 38.8× speedup in the four datasets.

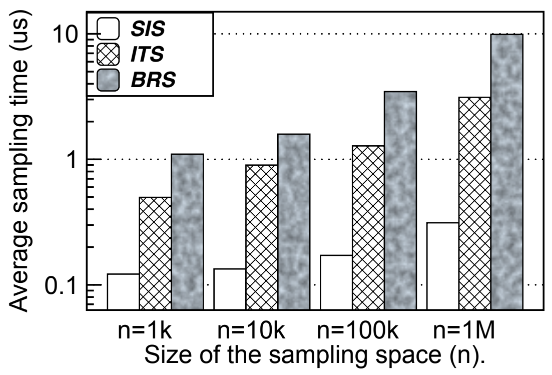


Figure 8: Scalability of different sampling algorithms.

Figure 8 shows the scalability of SIS, ITS, and BRS under different sampling space sizes (n) which represents the size of $\mathbb{V}$ in Algorithm 3. Figure 8 reports the average sampling time of a single sampling operation. We can see that SIS has better scalability than other sampling algorithms. When the sampling space becomes larger, the average sampling time of SIS merely increases a little. However, the average sampling time of ITS and BRS grows a lot with the increase of the sampling space.

6.4 Accuracy Comparison

Thirdly, we further compare T-GCN and DyGNN [23] with respect to node classification accuracy. Since DyGNN is more accurate than JODIE [16], we exclude JODIE from this comparison. The accuracy is computed by dividing the number of correct predictions by the total number of predictions. The test datasets give the ground truth and the best accuracy is obtained if the predicted class of each node equals its true class. Given a dataset, the test dataset is created by randomly sampling a fraction of nodes and splitting them from it. For the configuration, we divide the entire timestamp duration of datasets to a constant (default to 11) number of pieces according to the timestamp distribution. Figure 9 illustrates the accuracy comparison in the function of continuous-time duration steps. From the results, T-GCN maintains good accuracy results over time duration on each dataset. Even with the sampling, T-GCN does not lose any accuracy and even has better accuracy, because the used sampling models such as the node-wise sampling and the layer-wise sampling are widely used in GNNs such as GraphSAGE [8] and FastGCN [2] with good accuracy. Moreover, DyGNN cannot handle the large Twitter dataset, whereas T-GCN can also achieve good accuracy on it, thanks to our sampling method. We can conclude that with the sampling algorithm and system-level optimizations, T-GCN can handle large datasets without loss of accuracy.

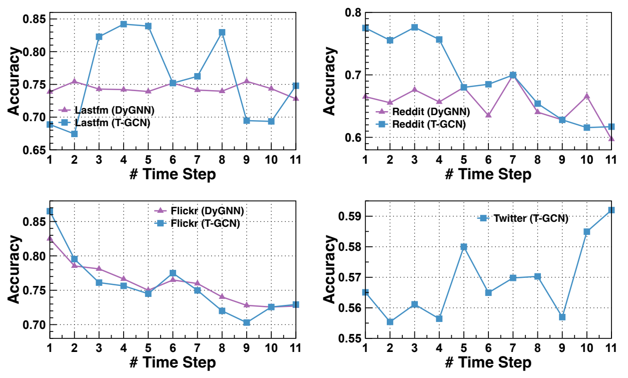


Figure 9: Accuracy evaluation.

6.5 Piecewise Breakdown Analysis of T-GCN

Table 2: Comparisons of IO and computation overhead.

Time(s)	DyGNN			T-GCN		
Dataset	IO	Comp.	Time	IO	Comp.	Time
Lastfm	5.53	64.89	70.42	5.24	32.89	36.42
Reddit	24.30	143.93	168.23	25.80	83.53	99.84
Flickr	63.18	560.41	623.59	58.52	105.98	124.10
Twitter	OOM	OOM	OOM	1603.90	9025.20	9273.05

Fourthly, T-GCN enables both sampling and training kernel optimizations, which play important roles in achieving good end-to-end performance as shown above, and it is of significance to analyze how much each optimization contributes to the end-to-end time of streaming GNN learning. For this ablation study, we first disable SIS-based temporal graph sampling (Section 3), as well as Locality-Aware Data Partition (Section 4.1) and pipeline task scheduling described in Section 4.2. Then, we turn on these optimizations one by one and measure the resulting speedups they brought. Meanwhile, we also profile the streaming GNN execution on GPUs with *nvprof* to better understand the improvements.

Our ablation study shows that GNN training takes a large portion of the total workload compared to host-device IO overhead. As shown in Table 2, 83% ∼ 88% of the end-to-end time of streaming GNN is spent on graph-structured training operations for DyGNN and T-GCN (even with GPU accelerations). Compared to DyGNN, the optimizations in T-GCN benefit from good task overlapping with better scheduling and reduce both I/O and computation overhead significantly. Thus, the overall runtime of T-GCN is less than the I/O time plus the computation time. Moreover, DyGNN's incapability of supporting relatively large graphs like Twitter highlights T-GCN's better applicability to a wider range of real-world GNN applications. Finally, the graph partition stage takes a very small part of the total time, e.g., nearly 0.3% for Twitter.

Table 3: Comparisons of memory usages.

Dataset	DyGNN	JODIE	T-GCN
Lastfm	1529MB	1837MB	822MB
Reddit	1461MB	1201MB	826MB
Flickr	13562MB	17246MB	6580MB

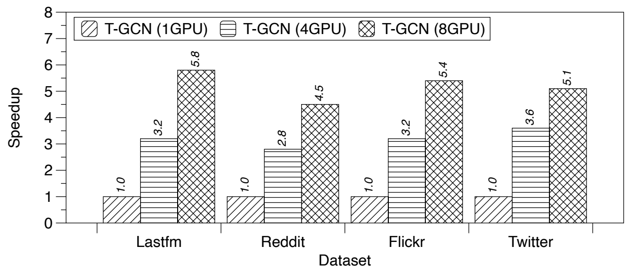


Figure 10: Scaling out streaming GNN with T-GCN.

6.6 Device Memory Overhead on a Single GPU

Fifthly, for training large-scale streaming GNNs, besides the performance of training, the total memory usage is often necessary and more important than the runtime, which will restrict the scale of training graphs and also the system scalability. The memory usage includes the memory needed for caching all the structural datasets and the hidden units and values for many epochs. For the sake of speedup training, T-GCN exploits data sampling for reducing caching space and locality-aware data allocation to enhance the GPU memory utilization. Table 3 compares the memory consumption of the three systems: DyGNN, JODIE, and T-GCN. Considering that both DyGNN and JODIE are unable to run on Twitter due to OOM, we have excluded Twitter in this comparison. T-GCN is much more memory efficient than the other two systems and achieves up to 2.58× space-saving. In particular, on Lastfm DyGNN and JODIE need 1529MB and 1837MB GPU device memory, respectively, but T-GCN only utilizes 822MB memory.

6.7 Scalability on Multiple GPUs

Finally, we evaluate the scalability of T-GCN by varying the number of GPUs in order to understand the performance of our scheduling strategy. From our evaluation, T-GCN achieves efficient scalability from one GPU to multiple GPUs by allowing each GPU to process a streaming subgraph, fully considering NVLink-based peer-to-peer (P2P) data communication and reducing the bandwidth contention (detailed in Section 4.2). To evaluate the scale-up scalability over 8 GPUs, we conduct separate experiments by setting P2P disabled or enabled, in order to understand the performance of our local-aware data scheduling.

Figure 10 illustrates the evaluation results of T-GCN with P2P enabled, where the speedup of T-GCN is 4.5 ∼ 5.8× in the function of 8 GPUs used compared to a single GPU. The average speedup is 5.2×. The benefit of P2P-based data movement is obvious. If P2P is disabled, the speed even decreases when scaling from 1 GPU to 4 GPUs, and otherwise the average speedup improves to 2.8 ∼ 3.6×. This is mainly because, when P2P is disabled, two GPUs within the same PCIe switch need to load input edge/vertex data through a shared link concurrently, which may easily cause bandwidth contention and become the bottleneck. In contrast, enabling P2P allows the second GPU to load vertex data directly from the first one, alleviating the pressure on the shared PCIe link.

7 RELATED WORK

Streaming graph embedding on large-scale graphs has become an important research direction. However, most existing works have not paid more attention to the performance issues. For the random walks models such as CTDNE [26], EHNA [11], and CAW [38], they have large overhead for processing new edges with incremental updating especially on large graphs. For snapshot-based GNNs such as DynamicTriad [49] and EvolveGCN [28], they have high training complexity for training new edges with new time stamps. For LSTM-based GNNs such as DyGNN and JODIE, they can train new edges easier but still suffer from high training complexity. T-GCN not only can provide new sampling methods with high efficiency to reduce training overhead but also provide high-performance multi-GPU training architecture with maximum use of inter-GPU networks.

There are some engines that have been proposed for static graph embedding. For the random walk engines, KnightKing [43] introduces a rejection sampling into high-order random walks, while GraphWalker [37] proposes efficient random walks management and graph loading strategy for out-of-core sampling. Unfortunately, these random walk engines can not support streaming graph random walks well because streaming graph random walks have higher sampling dimensions and traditional sampling methods can not provide efficient performance. For static GNN engines such as NeuGraph [22] and DGL-KE [48], they can deal with static graphs with high performance but are not suitable to streaming graphs because they do not support incremental learning and the new edges can cause an overhead of re-training the model.

8 CONCLUSION

In this paper, T-GCN proposes an efficient sampling method *Segment Its Search* together with a locality-aware data partitioning method, and an NVLink-specific task schedule to accelerate streaming GNN learning. Moreover, we further pipeline the computation and the communication (i.e., both CPU-GPU and GPU-GPU communications) by introducing an efficient memory management mechanism, to improve scalability while hiding data communication. In conclusion, on the same hardware, T-GCN achieves up to 7.9× speedup than state-of-the-art works as for end-to-end performance comparison. In addition, T-GCN achieves a maximum of 38.8× speedup on sampling through our *Segment Its Search* sampling method.

9 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. The authors from Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2020YFC1522702), Natural Science Foundation of China (61877035, 62141216), National Science Foundation CRII Award 2000722, CAREER Award 2046102, SOAR fellowship, University of Sydney Faculty Startup funding, Australia Research Council (ARC) Discovery Project DP210101984, and Ant Group through Ant Research Intern Program. We specifically thank the excellent support we have received from Ant Group and Beijing Baihai Technology Inc. Yongchao Liu and Yongwei Wu are joint corresponding authors.

REFERENCES

[1] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.6203

[2] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rytstxWAW

[3] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 942–950.

[4] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 257–266. https://doi.org/10.1145/3292500.3330925

[5] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. In *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014*, Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi (Eds.). Association for Computational Linguistics, 103–111. https://doi.org/10.3115/v1/W14-4012

[6] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 3837–3845. https://proceedings.neurips.cc/paper/2016/hash/04df4d434d481c5bb723be1b6df1ee65-Abstract.html

[7] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 world wide web conference*. 1775–1784.

[8] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 1024–1034. https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[9] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[10] Chengying Huan, Hang Liu, Mengxing Liu, Yongchao Liu, Changhua He, Kang Chen, Jinlei Jiang, Yongwei Wu, and Shuaiwen Leon Song. 2022. TeGraph: A Novel General-Purpose Temporal Graph Computing Engine. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 578–592. https://doi.org/10.1109/ICDE53745.2022.00048

[11] Shixun Huang, Zhifeng Bao, Guoliang Li, Yanghao Zhou, and J. Shane Culpepper. 2020. Temporal Network Representation Learning via Historical Neighborhoods Aggregation. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1117–1128. https://doi.org/10.1109/ICDE48307.2020.00101

[12] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive sampling towards fast graph representation learning. In *Advances in neural*

information processing systems. 4558–4567.

[13] Yugang Ji, Mingyang Yin, Hongxia Yang, Jingren Zhou, Vincent W Zheng, Chuan Shi, and Yuan Fang. 2020. Accelerating Large-Scale Heterogeneous Interaction Graph Embedding Learning via Importance Sampling. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 1 (2020), 1–23.

[14] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[15] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[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, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 1269–1278. https://doi.org/10.1145/3292500.3330895

[17] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, Leslie Carr, Alberto H. F. Laender, Bernadette Farias Lóscio, Irwin King, Marcus Fontoura, Denny Vrandecic, Lora Aroyo, José Palazzo M. de Oliveira, Fernanda Lima, and Erik Wilde (Eds.). International World Wide Web Conferences Steering Committee / ACM, 1343–1350. https://doi.org/10.1145/2487788.2488173

[18] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

[19] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[20] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/book/282.pdf

[21] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. 2014. Reducing the sampling complexity of topic models. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. 891–900. https://doi.org/10.1145/2623330.2623756

[22] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[23] Yao Ma, Ziyi Guo, Zhaochun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming Graph Neural Networks. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, Jimmy Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu (Eds.). ACM, 719–728. https://doi.org/10.1145/3397271.3401092

[24] Frederic P. Miller, Agnes F. Vandome, and John Mcbrewster. 2010. *Inverse Transform Sampling*.

[25] Jason Mohoney, Roger Waleffe, Yiheng Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Learning Massive Graph Embeddings on a Single Machine. *CoRR* abs/2101.08358 (2021). arXiv:2101.08358 https://arxiv.org/abs/2101.08358

[26] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-Time Dynamic Network Embeddings. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 969–976. https://doi.org/10.1145/3184558.3191526

[27] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. 2020. C-SAW: a framework for graph sampling and random walk on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 56. https://doi.org/10.1109/SC41405.2020.00060

[28] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2020. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 5363–5370. https://aaai.org/ojs/index.php/AAAI/article/view/5984

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[30] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani (Eds.). ACM, 701–710. https://doi.org/10.1145/2623330.2623732

[31] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 472–488. https://doi.org/10.1145/2517349.2522740

[32] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 472–488.

[33] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing communication in graph neural network training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[34] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. *CoRR* abs/1710.10903 (2017). arXiv:1710.10903 http://arxiv.org/abs/1710.10903

[35] Junshan Wang, Guojie Song, Yi Wu, and Liang Wang. 2020. Streaming Graph Neural Networks via Continual Learning. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). ACM, 1515–1524. https://doi.org/10.1145/3340531.3411963

[36] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[37] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. 2020. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 559–571. https://www.usenix.org/conference/atc20/presentation/wang-rui

[38] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=KYPz4YsCPj

[39] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *Proc. VLDB Endow.* 7, 9 (2014), 721–732. https://doi.org/10.14778/2732939.2732945

[40] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 145–156. https://doi.org/10.1109/ICDE.2016.7498236

[41] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ryGs6iA5Km

[42] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 15–29. https://doi.org/10.1109/HPCA47549.2020.00012

[43] Ke Yang, Mingxing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. 524–537. https://doi.org/10.1145/3341301.3359634

[44] Minji Yoon, Théophile Gervet, Baoxu Shi, Sufeng Niu, Qi He, and Jaewon Yang. 2021. Performance-Adaptive Sampling Strategy Towards Fast and Accurate Graph Neural Networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2046–2056.

[45] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=BJe8pkHFwS

[46] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 608–621. https://doi.org/10.1145/3173162.3173208

[47] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep Learning Based Recommender System: A Survey and New Perspectives. *ACM Comput. Surv.* 52, 1 (2019), 5:1–5:38. https://doi.org/10.1145/3285029

[48] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. DGL-KE: Training Knowledge Graph Embeddings at Scale. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, Jimmy Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen, and Yiqun Liu (Eds.). ACM, 739–748. https://doi.org/10.1145/3397271.3401172

[49] Le-kui Zhou, Yang Yang, Xiang Ren, Fei Wu, and Yueting Zhuang. 2018. Dynamic Network Embedding by Modeling Triadic Closure Process. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 571–578. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16572

[50] 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. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

[51] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, Shan Lu and Erik Riedel (Eds.). USENIX Association, 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

[52] Yuan Zuo, Guannan Liu, Hao Lin, Jia Guo, Xiaoqian Hu, and Junjie Wu. 2018. Embedding Temporal Network via Neighborhood Formation. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 2857–2866. https://doi.org/10.1145/3219819.3220054