PiPAD: Pipelined and Parallel Dynamic GNN Training on GPUs

Chunyang Wang*
Beihang University
Beijing, China
wangchunyang@buaa.edu.cn

Desen Sun*
Beihang University
Beijing, China
sy2006344@buaa.edu.cn

Yuebin Bai
Beihang University
Beijing, China
byb@buaa.edu.cn

Abstract

Dynamic Graph Neural Networks (DGNNs) have been widely applied in various real-life applications, such as link prediction and pandemic forecast, to capture both static structural information and temporal characteristics from dynamic graphs. Combining both time-dependent and -independent components, DGNNs manifest substantial parallel computation and data reuse potentials, but suffer from severe memory access inefficiency and data transfer overhead under the canonical one-graph-at-a-time training pattern. To tackle these challenges, we propose PiPAD, a **Pi**pelined and **PA**rallel **D**GNN training framework for the end-to-end performance optimization on GPUs. From both algorithm and runtime level, PiPAD holistically reconstructs the overall training paradigm from the data organization to computation manner. Capable of processing multiple graph snapshots in parallel, PiPAD eliminates unnecessary data transmission and alleviates memory access inefficiency to improve the overall performance. Our evaluation across various datasets shows PiPAD achieves 1.22× −9.57× speedup over the state-of-the-art DGNN frameworks on three representative models.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; **Machine learning**; • **Computer systems organization** → **Single instruction, multiple data**.

Keywords: Dynamic GNN, GPU, Parallelism

1 Introduction

Graph Neural Networks (GNNs) have been broadly adopted to process graph-structured data and extract the underlying dependencies [1] in varieties of graph-related applications ranging from node classification [10], recommendation [25] to link prediction [44]. Among numerous variants of GNNs, such as Graph Convolution Network (GCN) [20], a common and prevalent method is combining both graph (aggregation) and neural (update) operations. The aggregation function collects and aggregates feature vectors from the neighbors of each node while the update phase utilizes neural network operations, such as a fully connected (FC) layer, to update each vertex with the aggregated information. With retrieving the sparse adjacent matrices that renders irregular memory accesses and sparse computation, the Sparse-Dense Matrix-Matrix-Multiplication-like (SpMM-like) aggregation operation [15] is generally considered as the main bottleneck of GNN and attracts lots of research attentions [9, 15, 16, 48].

In many real-world scenarios, such as financial transaction [5], social media [22] and molecular biology [8], the topology and node features of graphs may dynamically evolve over time. According to the partition principle and granularity, the dynamic graph representation can be categorized into two classes [5, 19, 51]: Continuous Time Dynamic Graphs (CTDGs) and Discrete Time Dynamic Graphs (DTDGs). And the GNN research community proposes various dynamic GNNs (DGNNs) to capture both temporal and structural information in the mobility graph. For DTDGs that represent the dynamic graph as a sequence of *snapshots* sampled at regular intervals, a general method is to use static GNNs (e.g., GCN) for spatial graph learning on individual snapshots at all timesteps while deploying Recurrent Neural Networks (RNNs) to obtain temporal characteristics among different snapshots [5, 12, 39]. Besides, the *sliding window* mechanism that feeds multiple continuous snapshots to the model simultaneously, is widely adopted to better capture the temporal dependence and improve the accuracy [12, 32, 33, 39]. In this paper, we focus on **DTDG-based** DGNNs and refer to the sliding window as *frame* for simplicity.

Compared to traditional GNNs, DGNNs integrate time-series components operating a mass of snapshots, which leads to **two main performance issues**. First, since DGNN training requires updating graph snapshots continuously along the timeline, the data transfer overhead dominates overall training time and further exacerbates GPU under-utilization that already exists due to the memory-intensive GNN aggregation operation. Our preliminary experiments (§ 3.1) based on the state-of-the-art DGNN framework, PyTorch Geometric Temporal (PyGT) [38], show the data transmission via PCIe occupies nearly 39% of total execution

*Both authors contributed equally to this work.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577487

and GPU utilization is lower than 42% on average. Second, as introduced detailedly in § 3.3, the combination of time-independent (GNN) and time-dependent (RNN) components produces substantial parallelism opportunities (e.g., executing those independent operations from multiple snapshots concurrently) while the snapshot overlap among adjacent frames offers plenty of data reuse chances. These acceleration potentials desperately need to be sufficiently exploited.

Over the most recent years, certain research efforts have been made to realize efficient DGNN computation on GPUs [5, 12, 22, 38, 51] and some of them involve the above issues. However, the related solutions either mainly focus on the scaling of large graphs and distributed training [5] or restrict their input scenarios by assuming the graph topology (or node features) not changing along the time [12, 22], which weakens the applicability. Furthermore, they all fail to tackle both aforementioned issues at the same time for maximal end-to-end training performance optimizations.

Instead of analyzing the two problems separately, we address them from a more holistic angle and in a collaborative way. The idea is motivated by two key discoveries. First, real-world dynamic graphs normally change at a slow pace rendering massive topology overlaps among the adjacent snapshots [5]. Second, except the irregular access pattern of the SpMM-like aggregation, GNN suffers from other kinds of memory inefficiency stemming from the diverse node feature dimensions. Performing the aggregation over single graph tends to incur the *bandwidth unsaturation*, *low thread utilization* or *request burst* problems (§ 3.2). In summary, the canonical one-snapshot-at-a-time training paradigm inevitably causes not only substantial redundant data transmission but also inefficient memory accesses. Therefore, our insight is that the GPU underutilization and parallelism neglect issues could be alleviated simultaneously. Specifically, the expensive aggregation operates on the adjacent matrices that are exactly the main source of redundant data transfer. Extracting the overlapped topology and conducting single time-irrelevant aggregation operation for multiple snapshots can not only reduce the communication volume but also have more opportunities to enable coalescent memory accesses and address the low bandwidth utilization issue.

To this end, we propose PiPAD, a **Pi**pelined and **PA**rallel **D**GNN training framework to optimize the end-to-end performance on single GPU from both algorithm and runtime level. Without making any assumption on the input scenarios, PiPAD enables pipelined and parallel execution via reconstructing the overall DGNN training process from the data organization and transfer method to computation manner. **First**, capitalizing on the gradual changing characteristic of dynamic graphs, we propose a slice-based graph representation format and overlap-aware multi-snapshot transfer method. The design objective is to reduce the data loading time from CPU to GPU and facilitate the efficient parallel computation. **Second**, we devise the intra-frame parallelism and inter-frame reuse mechanism to sufficiently unleash the acceleration potential in DGNN. The former realizes a dimension-aware parallel GNN with thread-aware slice coalescing to boost the aggregation and locality-optimized weight reuse for the update. Our parallel GNN supports processing multiple snapshots simultaneously for better computation performance and alleviating the memory access inefficiency. Inter-frame reuse can eliminate the redundant transfer and computation overhead to the maximum extent. **Third**, to optimize the overall training performance, we implement a pipeline execution framework to orchestrate CPU-side operations, data transmission over PCIe and GPU computation. Besides, PiPAD integrates a runtime tuner to dynamically adjust the parallelism- and reuse-level of DGNN training for better adaptation to various datasets.

Since our objective is to reduce the end-to-end execution time, we conduct our additional data processing (graph slicing) online and avoid any offline model-scale profiling. Specifically, PiPAD only previously profiles and analyzes our parallel GNN kernel to guide the online tuning. As GCN is the most popular GNN employed in DGNNs nowadays [39], this paper targets GCN-based models. But with the SpMM-like aggregation being the foundation of mainstream GNNs [15, 16] (e.g., Graph Attention Network [44]), our methodology thus can be applied to various types of DGNNs.

Overall, this paper makes three contributions:

- We conduct a comprehensive analysis of the execution and performance characteristics for three typical DGNN models (§ 2.1 & § 3). Several key discoveries of DGNN training are revealed and motivate our design.
- We propose PiPAD to optimize the end-to-end performance for single-GPU DGNN training. Incorporating the overlap-aware data organization (§ 4.1), intra-frame parallelism (§ 4.2), pipeline execution mechanism (§ 4.3) and inter-frame reuse with dynamic tuning technique (§ 4.4), PiPAD holistically enables pipelined and parallel training in a multi-snapshot manner.
- Our evaluation across various datasets shows PiPAD achieves $1.22\times - 9.57\times$ speedup over the state-of-the-art DGNN frameworks on three representative models.

2 Background

2.1 Dynamic Graph Neural Networks

A DTDG is a ordered set of snapshots $\{\mathcal{G}^1, \mathcal{G}^2 \cdots \mathcal{G}^t\}$, where $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t\}$ represents the snapshot with vertices $\mathcal{V}^t$ and edges $\mathcal{E}^t$ at the timestep t. For one individual snapshot $\mathcal{G}$ and a GNN with K layers, let h_v^{k-1} denotes the feature vector of vertex v at layer $k-1$ and $\mathcal{N}(v)$ denotes all one-hop neighbors of vertex v. The computing pattern of GNN on layer k can be represented in Equation 1. After K layers of propagation, we will get the final features for each vertex.

$$h_v^k = \mathbf{Update}^k(\mathbf{Aggregation}^k(h_u^{k-1} | \forall u \in \mathcal{N}(v) \cup \{v\})) \quad (1)$$

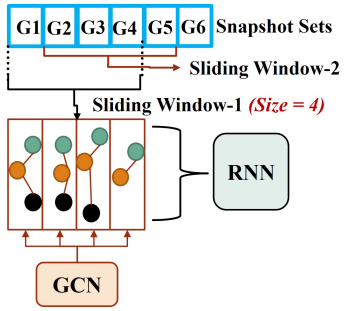
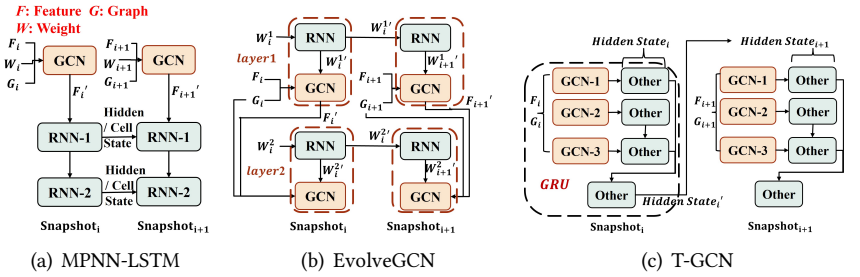


Figure 1. DGNN Execution Flow.



(a) MPNN-LSTM (b) EvolveGCN (c) T-GCN

Figure 2. DGNN Model Structure. →: Data Propagation.

Different GNNs differ in the specific functions used for the two key phases. In GCN, the aggregation processes the gathered features with *mean* function to obtain the aggregation results a_v^k (namely hidden representations) for each vertex while the FC-layer-based update computes h_v^k using the weights w^k and bias parameters b^k.

DTDG-based DGNNs follow the sliding-window/frame processing pattern and may encompass multiple layers. As shown in Figure 1, inside each layer, GNN components learn the structural information from each snapshot while RNN modules capture the temporal characteristics along the timeline. DGNNs can be classified into two categories [39]: integrated and stacked. We choose three representative models for the analysis and evaluation in the rest of this paper.

MPNN-LSTM [32] (Figure 2(a)) stacks one 2-layer GCN and two LSTM [14] models in a plain manner. LSTM operations are applied over the features produced by GCN. The only data dependence across multiple snapshots exists in the hidden state updating process for LSTM.

EvolveGCN [33] (Figure 2(b)) involves a 1-layer GCN and one GRU [6] in each of its two layers. The GCN in the second layer takes outputs from the first layer as input features. Different from MPNN-LSTM, EvolveGCN enables the weights evolving by applying the RNN module over GCN weight matrices, which generates cross-snapshot dependence. The survey [39] categories EvolveGCN into the integrated type.

T-GCN [50] (Figure 2(c)) integrates several 1-layer GCNs into GRU by replacing the original general matrix-matrix multiplication (GEMM). The hidden state propagation over the timeline creates the dependence similar to MPNN-LSTM.

2.2 GNN Acceleration

Static GNN. Over the last few years, there have been substantial research achievements for static GNN acceleration on GPUs covering general runtime frameworks [7, 26, 47, 52], the SpMM-like aggregation optimization [9, 15, 16, 48] and the scaling of distributed training [18, 23, 41, 43, 45, 46, 49].

DGNN. PyGT [38] and TGL [51] are two general frameworks aiming to implement the ubiquitous support for as many DGNN models as possible. By contrast, DynaGraph [12] and CacheG [22] focus on optimizing a certain group of models. Also discovering the parallelism/reuse opportunities (§ 3.3) in DGNN, they leverage the timestep fusion and intermediate results cache to improve the performance, respectively. However, DynaGraph only targets integrated DGNNs and assumes the graph topology not evolving over time while CacheG demands the node features remaining unchanged. Cambricon-G [40] and ESDG [5] both involve DGNN's data transfer overhead issue (§ 3.1). The hardware accelerator solution Cambricon-G, devises a cuboid-based processing architecture that supports the fine-grained data transmission to avoid unnecessary snapshot topology updating. ESDG employs a similar graph-difference based transfer method to reduce the communication volume for the scaling of large graphs and multi-GPU training. But those two studies neglect the intrinsic parallelism potentials in DGNN and blunder away the chance of fulfilling further acceleration.

Pipeline/parallel training for GNN or traditional DNN is widely adopted in the distributed environment [17, 27, 28, 41, 45, 49] and concurrent multi-model computation [3, 42]. OOB [31] and nimble [21] are two similar studies to us enabling parallel training for single DNN on single GPU. But the former's motivation lies on the backward propagation while the latter focuses on using CUDA Graph [11] and dependency pre-analysis mechanisms to remove the multi-stream overhead. Their work are orthogonal to our concern.

Summary. Works about the SpMM-like aggregation optimization for static GNNs are more general and can be applied to the DGNN domain. But some studies [15, 48] rely on the onerous node reordering (up to seconds per snapshot) that could badly damage the end-to-end performance. Besides, these solutions and the existing DGNN optimization methods cannot address both key issues simultaneously.

3 Motivation

This section first reveals the bottleneck and underlying execution inefficiency of DGNN training via preliminary experiments with our evaluation setting (§ 5.1). Then we dissect the data reuse and parallel computation potentials, which can be leveraged to resolve these issues simultaneously.

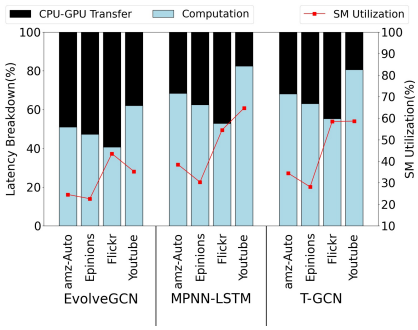


Figure 3. Latency Breakdown and SM Utilization of DGNN Training.

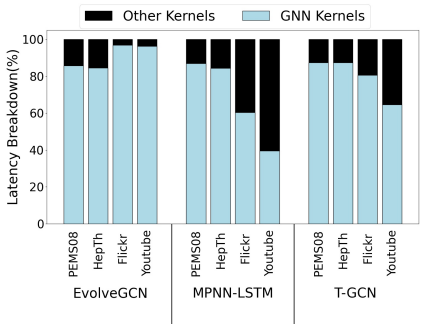


Figure 4. Breakdown of GPU Computation Time in DGNN Training.

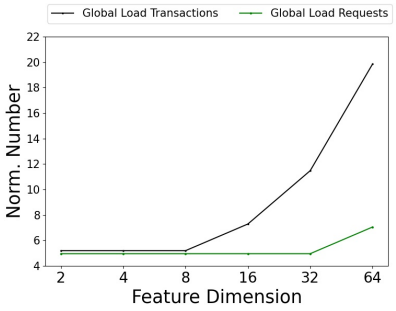


Figure 5. The number of Global Memory Requests and Transactions.

3.1 Performance Bottlenecks of DGNN Training

Data transfer overhead. We analyze the overall performance characteristic of DGNN training using PyGT, which is built on top of the popular GNN computing framework PyTorch Geometric (PyG) [7]. Compared to the static GNN targeting single graph, DGNN operates on a snapshot sequence and needs to constantly load new graph data from CPU to GPU. Figure 3 depicts the breakdown of GPU-related training time (left-axis) and SM utilization (right-axis). As presented, the arduous data transfer occupies the total execution time with an average 38.7% proportion. Subsequently, this leads to severe GPU underutilization where Streaming Multiprocessor (SM) utilization measured by PyTorch Profiler [35] is lower than 41.2% on average. Leveraging CUDA streams to enable asynchronous data transmission is a straightforward and elementary way to ease the problem.

Topology overlap. The changing rate of the topology among adjacent snapshots in real-life dynamic graphs is generally limited (nearly 10% on average across our datasets). ESDG [5] utilizes this fact to employ a graph-difference transfer method that only updates changed parts of the topology, but still follows the one-snapshot-at-a-time training manner.

3.2 Memory Access Inefficiency in GNN

Still being the major computation burden in DGNN models (Figure 4), GNN suffers from the low memory access efficiency. The access irregularity in the aggregation, brought by the sparsity of input graphs, is revealed by previous GNN studies [9, 15, 16, 48]. They refactor access patterns to the sparse adjacent matrices and accordingly implement the locality-aware computation to increase the parallelism. However, there are other types of inefficiency stemming from the dimension of dense matrices (input or aggregated feature vectors), neglected by those existing solutions. In a common implementation of SpMM, one warp takes charge of processing single element from the sparse matrix and one entire row from the dense matrix for each outer iteration. To leverage the locality and reduce expensive off-chip accesses, GPU cores load the features from global memory and store the intermediate results of the current iteration on shared memory for reuse in the next iteration. Note that mainstream GPUs (e.g., NVIDIA) normally employ a minimum 32-byte granularity for global memory access. Besides, following the Single Instruction Multiple Thread (SIMT) fashion, a warp consisting of 32 threads can fetch 128 (32×4) bytes at most with one request. Considering the row length in bytes of the feature matrix equal to 4 times of its feature dimension (F), there are two types of memory access inefficiency:

- **Bandwidth unsaturation** with F less than 32/4, which means that the size of useful data is smaller than the minimum access granularity of single transaction.
- **Request burst** with F larger than 128/4, which means that a warp accessing one row needs multiple requests to both shared and global memory to finish.

We further demonstrate the above insights via experiments of running GCN with GNNAdvisor [48], a cutting-edge GNN aggregation optimization work. As shown in Figure 5, when the feature length lower than 8, the number of global memory requests (#R) and transactions (#T) are nearly the same and both barely change. Then #T increases as the dimension exceeds 8 while #R begins to rise when the dimension larger than 32. Considering the diversity of dimension features used in real-life graph-based applications, these issues definitely need to be resolved. Moreover, assigning one row to one warp when processing the feature matrix with F less than 32, means many threads inside the warp stay idle during the execution. The **low thread utilization** issue can be reflected by the *warp_execution_efficiency* metric (§ 5.3).

3.3 Opportunities for Data Reuse and Parallelism

Data Reuse. The forward stride size of the frame mechanism (Figure 1) is normally set to 1 for sufficient temporal interaction capture among all snapshots. Therefore, there are plenty of overlaps among contiguous frames, which means some related redundant data transfer and computation can be avoided. Specifically, the aggregation operation in the first GCN layer operates over the adjacent matrix of the graph

topology and the original node features, which is independent of the parameter updating along the timeline. We can cache the related aggregation results (a_v^k in § 2.1) and reuse them in the next frame or training epoch.

Parallelism. As depicted in Figure 2, the cross-snapshot dependence of three various DGNNs all exists in the time-dependent RNN components. It means that the same operation of GNN for different snapshots can be conducted in parallel for better performance. With careful design and optimizations, the multi-snapshot processing manner may hit two birds with one stone. Specifically, we first take *topology overlap* into consideration cooperatively and design a new graph format that can extract the overlaps efficiently to eliminate the redundant transfer. Then we **devise a "parallel" GNN computation pattern that can process multiple graphs simultaneously. The basic idea is to directly perform the aggregation on the overlap topology of one snapshot group with all their feature matrices**. In this way, we can enable coalesced memory accesses to multiple features and alleviate the bandwidth unsaturation issue. Further, **employing flexible access granularities for huge dimension situations can avoid the request burst**.

4 PiPAD

We holistically design and implement PiPAD to facilitate the end-to-end DGNN training performance. Our optimization is three-fold: reducing data transfer volume via the overlap-aware data organization and inter-frame reuse, accelerating GNN computation with intra-frame parallelism and maximizing the overlap among different operations through the pipeline. This section presents those mechanisms in detail.

4.1 Overlap-aware Data Organization

By extracting the overlap topology among adjacent snapshots and constructing a new individual adjacent matrix for it, we can avoid the needless transmission and further realize the parallel computation idea. Due to the frame mechanism, one snapshot may be coalesced with separate snapshots for processing in different frames. Therefore, the overlap analysis and extraction will be conducted constantly throughout the opening training epochs. The widely-used format to store the sparse adjacent matrix, Compressed Sparse Row (CSR) manages the graph using a coarse row granularity and a tightly ordered layout (Figure 6). These characteristics directly restrict the flexibility of overlap extraction leading to poor efficiency. Besides, the sparsity in real-world graphs could easily cause load imbalance in SpMM computation since a warp is normally set to handle one or several rows of the adjacent matrix based on the CSR representation way.

Slice-based graph representation. There are two choices to support a finer storing granularity and tackle the above challenges: two-dimensional *tile* and one-dimensional *slice*. The former (e.g., Block CSR [4]) splits the original matrix into multiple smaller matrices while the latter divides each row and transforms the whole matrix into many slices. Since the non-zero elements (nnz) in each tiled sub-matrix are discrete leading to the more uneven sparsity, the slice way not only incurs less space overhead but also offers better load balance with the compression. Therefore, we devise a sliced CSR format shown in Figure 6. As single slice is within one row, we modify the original *Row Offsets* array in CSR to *Row Indices* (RI) and add a new array *Slice Offsets* (SO). RI indicates the row index of every slice and SO stores the offset of the first element of each slice in *Column Indices* field. Our method manages the whole graph at a slice granularity and each slice stores certain nnz with a upper bound (2 in Figure 6). This can enable quick slice-grained overlap construction and assist to achieve better load balance in the computation.

Overlap-aware data transfer. Introduced later, PiPAD performs the parallel training by dividing each frame into several *partitions* that each contain continuous snapshots. We thus adjust the data transfer into a partition-grained pattern. The topology data (adjacent matrices) are regrouped as one overlap part for all snapshots and several exclusive parts for each to reduce the transmission volume. Then we utilize separate GPU streams and pinned memory to enable asynchronous data transfer in the on-demand manner.

Space overhead. The main cost comes from the modified RI and newly added SO. Let $\#nnz$ denotes the number of nnz in the matrix. Our sliced CSR requires $(2\times\#nnz+2\times\#Slice+1)$ space while CSR needs $(2\times\#nnz+\#vertices+1)$. Another popular graph representation way Coordinate Format (COO), employed by PyG, uses three arrays to respectively store all nnz's values, the row and column indices with the $(3\times\#nnz)$ space usage. We set single slice can hold up to 32 nnz and our spatial cost thus normally falls in between CSR and COO.

4.2 Intra-frame Parallelism

We implement intra-frame parallelism based on our *dimension-aware parallel GNN* that performs the aggregation and update operations over multiple snapshots simultaneously (❶ and ❷ in Figure 6). We further devise three key optimizations: vector-memory-instruction based memory access for the large-dimension situation, *thread-aware slice coalescing* (❸) to resolve the low thread utilization issue for the small-dimension case and *Locality-optimized weight reuse* (❹) to enable the efficient parallel update.

Dimension-aware parallel GNN. As analyzed in § 3.3, two types of memory access inefficiency can be alleviated through the multi-snapshot computation manner and flexible access granularities. First, coalescing the feature matrices from different snapshots to perform one aggregation can more efficiently utilize the bandwidth especially for those graphs with node features in the small dimension. Following the overlap-aware data organization, we split the original aggregation into two parts: the parallel aggregation on adjacent matrices of the overlap with all features of one snapshot

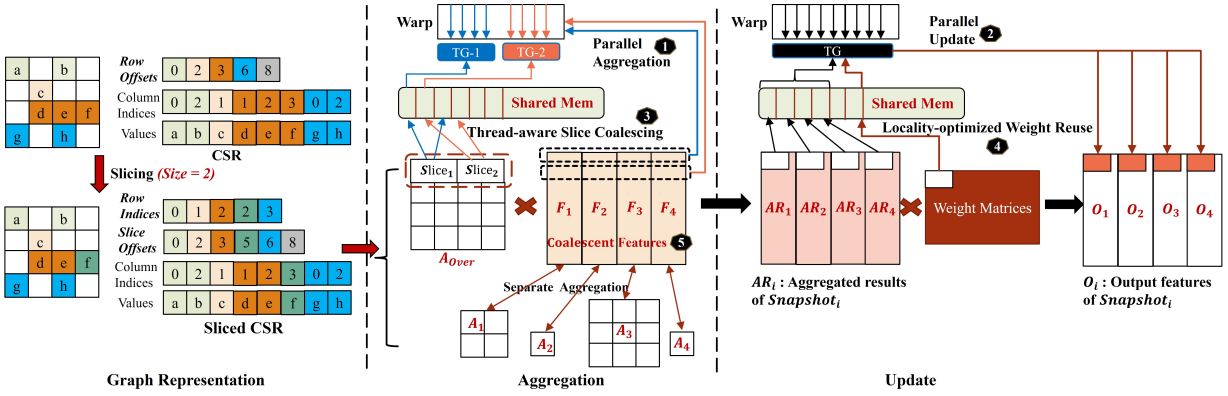


Figure 6. An Example of the Sliced Graph Representation and the Parallel GNN Computation. F_i: Input Features of $Snapshot_i$. A_i: Adjacent Matrix of the Exclusive Parts of $Snapshot_i$. A_{Over}: Adjacent Matrix of the Overlap Part. TG: Thread Group.

group (referred to as the *coalescent* features: ❺ in Figure 6), and the non-parallel aggregations on the exclusive topology parts with respective features from each snapshot. Second, leveraging the vector memory instructions [24] that support loading 32/64/128 floats for one request is a feasible option for the request burst problem. Based on those vector instructions with different data-fetching widths, we realize loading the data in an efficient larger-grained way when processing graphs with features in the large dimension (larger than 32). We introduce our aggregation implementation with *slice coalescing* together later.

Thread-aware slice coalescing. Even if merging multiple feature matrices to a coalescent one, the total row length (dimension) may still be less than 32 and the low thread utilization problem remains. In this situation, to further increase the number of active threads per warp, we coalesce adjacent slices in the adjacent matrix as a group and set it as the basic processing unit for each warp. Correspondingly, threads inside the same warp are uniformly divided into several thread groups (TGs) and each TG is assigned to exclusively operate on one slice. Then we implement the slice-group data layout on shared memory in a interleave pattern (❸ in Figure 6) so that the access address of single data load is continuous from the warp view. Each TG operates on one element from the corresponding slice and one row from the coalescent feature matrices at each computing iteration. The maximal size of the slice group (*coalesce_num*), namely the number of TGs per warp, is set as 4 to ensure each TG's data access granularity not exceed single memory transaction length (32 bytes). Algorithm 1 presents the detailed procedure of our parallel aggregation (for the small dimension cases) with the inputs: the coalescent features, the max number of nnz each slice owns (32) and *coalesce_num*. In the implementation, each thread is first assigned to load elements in one particular slice from global memory to shared memory with a fixed stride and the slice index is directly corresponding to the thread index (Line 3 and 9-20). This design is based on

Algorithm 1: Parallel aggregation & Slice coalescing

```
   Input: features, slice_size, coalesce_num
   Output: aggregation_result
1  lane_id = getIndexInWarp()
2  /* Compute the index of the specific slice distributed to current thread for
     load and computation, respectively*/
3  load_index = lane_id % coalesce_num
4  comp_index = lane_id / features.dim
5  /* Compute the specific dimension of the coalescent features distributed to
     current thread for load and computation */
6  comp_dim = lane_id % features.dim
7  /* Check whether the current thread is distributed with work*/
8  if comp_index < coalesce_num then
9      /*Compute the specific workload size (within a slice) for the thread*/
10     load_size, comp_size = compWork(load_index, comp_index)
11     /*Load the target elements in the slice*/
12     for nid ← lane_id to (slice_size * coalesce_num) do
13         /*Compute the element offset in the slice for load*/
14         load_offset = nid / coalesce_num
15         if load_offset > load_size then
16             break
17         end
18         LoadSliceToSharedMem(load_offset)
19         nid += coalesce_num * features.dim
20     end
21     /*Load the target dimension in the features and then compute*/
22     for idx ← 0 to comp_size do
23         /*Compute the element offset in the slice for computation*/
24         comp_offset = idx * coalesce_num + comp_index
25         Slice = LoadSliceFromSharedMemToReg(comp_offset)
26         Feature = LoadFeatFromGlobalMemToReg(comp_dim)
27         MultiplyandACcumulate(Slice, Feature, partial_result)
28         idx ++
29     end
30     actomicAdd(aggregation_result, partial_result)
31 end
```

the aforementioned interleaving data layout for each warp's continuous memory access. Then the thread fetches the data corresponding to one specific dimension of the features that is distributed to it (Line 6 and 26), loads the elements of one target slice on shared memory with the same stride pattern and finally performs the computation (Line 4 and Line 21-30).

Locality-optimized weight reuse. To cooperate with the parallel aggregation, we utilize the weight reuse principle

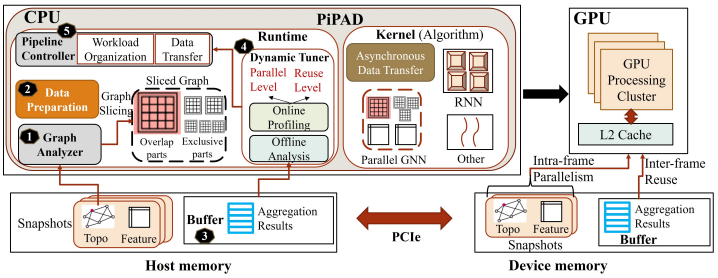


Figure 7. The Overall Architecture of PiPAD.

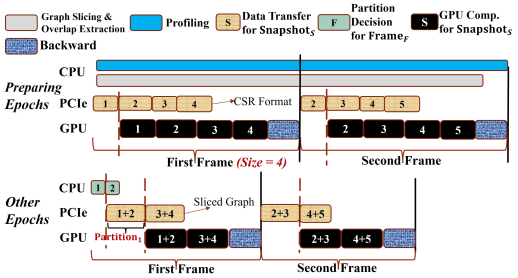


Figure 8. A Pipelined Execution Example.

to enable the parallel update with better locality. Instead of naively conducting GEMM for one snapshot and then moving to the next, we choose to keep one weight tile staying on shared memory, and compute with the features of all snapshots successively (❹) and then move to the next weight tile for the same iteration. This locality-optimized computation paradigm can reuse the weights to the maximal extent. Note that this technique can not be applied to EvolveGCN since it updates the weights along the timeline.

For other kernels in DGNN, we execute them sequentially in the order of computation and data dependence. Besides, we reference OOB [31] and leverage CUDA Graph API [11] to launch these kernels together and reduce the issue overhead.

4.3 Pipeline Execution Framework

As illustrated in Figure 7, PiPAD consists of the **algorithm-level** optimization for the efficient multi-snapshot computation and several **runtime-level** components to coordinate the general training process.

Specifically, in the algorithm level, we devise the sliced CSR for convenient extraction of graph overlaps and reconstruct GNN to enable parallel computation over multiple snapshots within a frame. In the runtime level, to fit with the new graph format, we first realize a low-overhead online *graph analyzer* (❶ in Figure 7) to efficiently transform the format of all snapshots from the original CSR to ours. The adjacent matrices data for each partition includes the common overlap and exclusives for separate snapshots. A *data preparation module* ❷ is used to conduct the overlap extraction among adjacent snapshots and prepare for the partition-wise computation manner. The above two components only need to function once during the first several epochs (namely *preparing* epochs). In the following epochs, PiPAD performs the parallel training at the partition granularity for each frame. As for supporting data reuse among different frames, we maintain both *CPU- and GPU-side buffers* ❸ to cache the aggregation results. On the upper tier of our runtime, we implement a *dynamic tuner* ❹ to online adjust the parallelism-level (the number of snapshots per partition inside a frame) and reuse-level (the size of aggregation results to cache in GPU-side), based on the offline analysis of our parallel GNN kernel and online profiling in the preparing epochs. Note that the preliminary offline experiments (§ 4.4) only involve the GNN part for sensitivity analysis. Finally, our *pipeline controller* ❺ regulates the overall training procedure.

Figure 8 provides a PiPAD's pipelined execution example with the setting of two snapshots per partition and the frame size equal to 4. In the **preparing** epochs, the training abides by the traditional one-snapshot fashion with asynchronous data transfer. In the host, we collect the necessary statistics including the input data size, execution time and maximal memory consumption of each snapshot/frame. This profiling executes online to avoid early offline model-scale test and amortize the overhead for end-to-end performance improvements. These data will be leveraged to guide our dynamic tuner to choose the optimized parallel options without triggering the out-of-memory (OOM) exception (§ 4.4). Besides, since extracting the graph overlaps and preparing the partition-wise adjacent matrices would also take time, we perform the graph slicing and subsequent extractions in the beginning once for all. The configurations of the snapshot amount per partition is a finite set (e.g., 2, 4 & 8), thus the extraction overhead is controllable. Meanwhile, we also store the graph evolving rates among snapshots within each frame for the online tuning decision in the future. Then during **the following** epochs, the training proceeds in a partition manner with reduced data loading overhead and high computation parallelism. Compared to the canonical pattern, we first determine the parallelism-level for current frame (§ 4.4), and then transport the partition-wise training data. Due to the performance similarity among different epochs in the training, we only perform this procedure once and stick to the generated configurations for each subsequent epoch.

Overhead analysis. PiPAD introduces certain time and space overheads. First, our slice-based graph format requires extra space (§ 4.1) and an auxiliary graph slicing process in the preparing epochs. We also need to extract the overlaps to build the overlap-based adjacent matrices for each partition previously. Note that our slicing strategy manages graphs at a fine granularity and can accelerate the extraction. By the actual measurements under our evaluation settings, those preprocessing can be accomplished within the first two

epochs. Second, the data transfer for each partition can only proceed after the partition decision completes. The cost can also be amortized since this one-off operation is conducted asynchronously in CPU and overlap with other operations of the former partition/frame (Figure 8).

4.4 Inter-frame Reuse and Dynamic Tuning

This section presents our design of the data reuse across different frames and dynamic tuning on two key parameters.

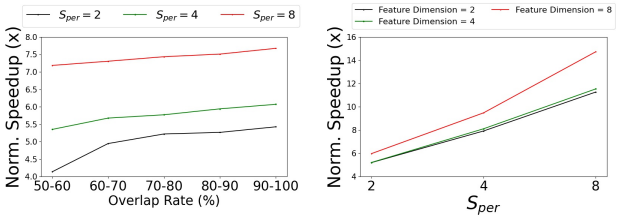
Inter-frame reuse. As summarized in § 3.3, the aggregation operation in the first GCN layer is unaffected by the parameter updating along the timeline. Therefore, a straightforward way is to store all relative aggregation results derived in the preparing epochs to CPU memory and reuse them for subsequent epochs. This can avoid repeatedly transmitting the adjacent matrices of those overlapped snapshots (for the models with only one single-layer GCN) and eliminate the redundant aggregation computation. But those aggregation results still need to be transferred to GPU for the next frame.

To further facilitate the reuse level, we maintain a GPU-side buffer to cache some aggregation results based on the used order in the next frame. The maximal buffer size is limited by the memory consumption of DGNN training process and GPU memory capacity. According to the frame-wise memory usage statistics provided by our profiling in the preparing epochs, we dynamically allocate the buffer with different sizes for each frame according to the demand. Note that, since the *malloc/free* function execution is time-consuming, we only reallocate the buffer when its previous size is too small to accommodate the new intermediates.

Dynamic Tuning. To reduce the complexity of online decision-making, we uniformly distribute the snapshots in single frame to each partition. Then the dynamic tuner needs to determine two key parameters for each frame: the number of snapshots per partition (S_{per}) and the size of aggregation results to cache in GPU for the next frame reuse. The latter is decided by the memory usage conditions as explained above. With regard to S_{per}, there are three impact factors:

- **Memory consumption**: Multiple-snapshot computing would increase the GPU memory usage at a certain stage compared to the one-snapshot pattern since processing each partition needs GPU to hold all related snapshots' data. We should avoid the OOM exception.
- **Computation speedup of the parallel GNN**: The acceleration effects of our intra-frame parallelism may vary over different input graphs and values of S_{per}.
- **Overlap level between the computation and data transfer**: Multi-snapshot processing not only promotes performance in the computing phase but also augments the data size of single transmission, which may stall the pipeline and lower GPU utilization instead.

We decide the value of S_{per} with the following procedure.



(a) Speedup of different S_{per} settings as OR changes normalized to the one-snapshot execution.

(b) Normalized speedup of different S_{per} settings as the feature dimension changes.

Figure 9. The Offline Analysis Results of the Parallel GNN.

First, since we employ the overlap-aware data organization method and the training data size gets reduced, the maximal memory consumption in the N-snapshot mode would not exceed N times of that in the one-snapshot computation. Leveraging our online profiling statistics, we can derive an upper bound U that assures our option not to trigger OOM.

Second, the computing speedup is mainly relevant to three elements: S_{per}, the node feature dimension of single graph and topology overlap rate (OR) among all snapshots within the partition. Simultaneously operating more snapshots normally means more parallelism but lower OR. Since the main optimization of our intra-frame parallelism lies in utilizing the overlaps to enable the efficient aggregation, the damage from low OR is not negligible. Therefore, we perform the offline analysis regarding our parallel GNN to guide the online decision. Figure 9(a) and 9(b) show the average speedup that different S_{per} settings achieve across our datasets under various OR and feature dimension conditions, respectively. We construct the OR configurations with randomly selecting snapshot groups that satisfy the target overlap requirements. The results clearly show larger S_{per} is preferred under the same OR or feature dimension settings. More importantly, the sketchy speedup distribution for separate OR sections can assist us to estimate the accelerating effects of different S_{per} settings at runtime. We leverage results from the offline analysis and actual frame-wise topology changing rate statistics obtained online in the preparing epochs to derive the estimated execution latency of all available S_{per} options.

Third, we previously analyze the data transfer latency for each option with the profiled data size information. Then, we reject those schemes rendering the pipeline stall and choose the one with the highest computing speedup from leftovers. With a specific parallelism and reuse option for each frame, our pipeline controller orchestrates the overall training.

4.5 Discussion

We implement PiPAD on top of PyTorch [34] with C++, CUDA C and Python. But our methodology is general and independent of the specific deep learning or GNN frameworks.

Table 1. Graph Datasets for Evaluation. The number of vertices (#N), edges (#E), feature dimension (D) and edges after graph smoothening with edge-life [5] (#E-S) across all snapshots as well as the number of snapshots (#S) are shown.

Dataset	#N	#E	D	#S	#E-S
Social Network [36, 37]					
Flickr	2.3 M	33.1 M	2	132	480 M
Youtube	3.2 M	602 K	2	198	11 M
E-commerce [36, 37]					
amz-Automotive	1.1 M	1.3 M	2	524	55 M
Epinions	727 K	13.6 M	2	99	78 M
Citation Network [36, 37]					
HepTh	22 K	2.6 M	16	214	18 M
Traffic Network [2]					
PEMS08	170	7202	16	90	7202
Disease Transmission [32]					
Covid19-England	130	82 K	16	61	108 K

Limitation. Multi-snapshot parallel training inevitably increases the phase-based memory consumption and causes the inherent scaling issue of large graphs. This limitation can be resolved through extending PiPAD to support multi-GPU training since our sliced CSR offers the convenience to further split the graphs. We leave it as our future work.

5 Evaluation

5.1 Experiment Setup

Models & Datasets. We choose three representative DGNN models (§ 2.1): MPNN-LSTM [32], EvolveGCN [33] and T-GCN [50]. Shown in Table 1, the datasets for our evaluation cover various real-world applications and are obtained from Network Repository [36, 37] (used by ESDG [5]), ASTGNN [13] and MPNN-LSTM [32]. Considering the memory capacity limit of single GPU, we set the input feature dimension of small and large-scale graph datasets to 16 and 2, respectively. And the hidden dimension is set to 32 and 6, respectively.

Baselines. Since DynaGraph [12] and CacheG [22] restrict the input scenarios while ESDG [5] targets the distributed training, we compare PiPAD with the baseline **PyGT** and its three variants: (1) **PyGT-A**: an enhanced version of PyGT enabling asynchronous data transfer; (2) **PyGT-R**: a solution that integrates our inter-frame reuse mechanism into PyGT-A; (3) **PyGT-G**: compared to PyGT-R, PyGT-G replaces the original PyG-version of GCN module with GE-SpMM [15], a state-of-the-art aggregation optimization work without previous node reordering over the graphs like [48] and [16]. GE-SpMM leverages shared memory to cache rows from sparse matrices and increases the locality for the CSR-based aggregation computation. We utilize this incremental comparison design to evaluate the overall performance and our individual optimization techniques simultaneously.

Environment settings & Metrics. Our hardware platform is a 24-core Intel(R) Xeon(R) E5-2680 v3 CPU with 128GB host memory and one NVIDIA Tesla V100 GPU with 16GB HBM. We set the frame size to 16 and conduct all experiments on Ubuntu 18.04 with CUDA 11.3. And the version of PyGT and PyTorch is 0.53.0 and 1.10 respectively. In the overall performance comparison, we train for 200 epochs and measure the end-to-end training time including the graph slicing, data loading and GPU computation as well as the hardware utilization. We also perform the detailed analysis of our parallel GNN and sliced CSR.

5.2 Overall Performance

Figure 10 shows the end-to-end training time speedup of three models over PyGT. In general, with the holistic design covering data organization, transfer and computation manner, PiPAD outperforms all compared methods and achieves $1.54\times - 9.57\times$ improvements over the baseline PyGT (4.71×, 3.98× and 5.18× on average for EvolveGCN, MPNN-LSTM and T-GCN, respectively). As the small-scale datasets (HepTh, PEMS08 and Covid19) have less vertices and smaller feature vector size, our topology-wise data transfer reduction and computation optimizations can make a larger impact. Besides, due to the huge node amount and total memory usage, PiPAD can only enable 2-snapshot parallelism in the evaluation for the large datasets, which restricts our acceleration space. Thus our speedup is generally higher in the small-scale datasets while PyGT-A presents the opposite characteristic since the asynchronous transfer is its only optimization.

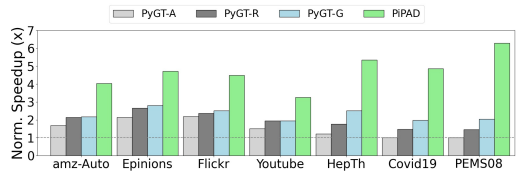
PyGT-G obtains the second-best performance for almost all scenarios due to the three-fold optimizations: asynchronous data transfer to reduce transmission overhead, inter-frame reuse to eliminate the redundancy and GE-SpMM to accelerate the expensive aggregation. Compared with PyGT-G, our main advantage is the intra-frame parallelism that can not only process multiple snapshots for larger throughput but also facilitate the update function with the weight reuse. PyGT-R performs very close to PyGT-G or even better in some cases. First, employing inter-frame reuse would directly skip the aggregation in the first GCN layer and GE-SpMM targeting the aggregation acceleration thus turns nearly useless in T-GCN that executes multiple GCNs simultaneously and behaves like only owning one GCN. Second, GE-SpMM requires both CSR and Compressed Sparse Column (CSC) format to store the graphs for backward propagation in its implementation. This incurs more transfer overhead and damages the performance in large-scale datasets like Youtube.

The specific speedups vary greatly in different situations. We next perform the detailed analysis from the model aspect with the help of our experiments in Figure 3 and Figure 4.

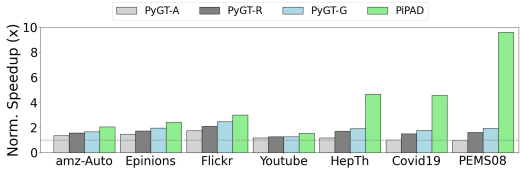
EvolveGCN. Incorporating two layers that each contain one GCN, EvolveGCN still needs to perform the aggregation in the GCN of the 2nd layer even if deployed with data reuse optimizations. This offers more acceleration space to

Table 2. GPU Utilization (%) of Different Methods. AA: amz-Automotive; EP: Epinions; FL: Flickr; YT: Youtube; HT: HepTh; CE: Covid19-England; PE: PEMS08. The low values under small-scale datasets arise from the relatively larger CPU-side latency.

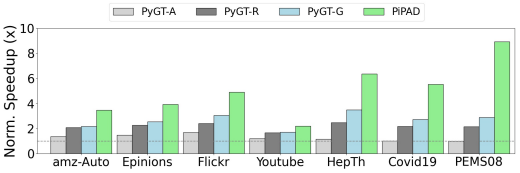
Method	EvolveGCN							MPNN-LSTM							T-GCN						
	AA	EP	FL	YT	HT	CE	PE	AA	EP	FL	YT	HT	CE	PE	AA	EP	FL	YT	HT	CE	PE
PyGT	75.18	70.75	76.47	78.85	31.9	12.04	10.87	84.81	80.44	81.36	89.91	36.01	11.89	10.99	83.8	79.56	81.93	87.67	35.41	13.15	11.92
PyGT-A	93.45	92.47	99.08	94.94	35.67	12.16	10.95	96.42	96.31	**99.38**	97.94	39.14	11.92	11.0	94.44	93.45	99.08	95.92	37.43	13.19	11.98
PyGT-R	**94.78**	**94.45**	**99.08**	**96.03**	**45.81**	**13.6**	**11.52**	**97.3**	**96.91**	99.16	**98.65**	50.17	**13.8**	12.51	**95.39**	94.5	**99.13**	97.72	**50.96**	**15.36**	13.59
PyGT-G	78.86	91.33	88.16	91.98	34.86	8.81	8.44	91.05	94.28	93.5	94.27	42.57	8.97	8.9	89.5	93.55	92.25	93.46	40.64	10.47	11.23
PiPAD	87.75	91.8	92.28	94.12	36.25	11.75	10.91	96.31	95.13	93.96	98.55	**67.92**	12.6	**14.25**	94.39	**94.68**	92.33	**98.07**	50.61	12.18	**14.23**



(a) EvolveGCN



(b) MPNN-LSTM



(c) T-GCN

Figure 10. Training Speedup over PyGT.

our intra-frame parallelism. In addition, the other kernels besides GNN module in EvolveGCN only take small portions of total computation time (Figure 4), which further amplifies the effects of the parallel GNN. Therefore, PiPAD achieves the highest average speedup (2.13×) against the second-best PyGT-G here among three models.

MPNN-LSTM. Contrast to other two models, MPNN-LSTM has the higher computation proportion and more time-consuming RNN components under the larger datasets with massive vertices (e.g., Youtube in Figure 3 and 4). Therefore, when dealing with the large-scale datasets, PiPAD achieves the lowest speedup here among all cases (down to 1.22× over PyGT-G). But the RNN-related execution latency of MPNN-LSTM is much smaller for small datasets (Figure 4). And MPNN-LSTM deploys a 2-layer GCN but dose not update weights along the timeline like EvolveGCN. The acceleration in both aggregation and update operation of our parallel GNN can fully function under those small-scale datasets (up to 9.57× over PyGT).

T-GCN. With multiple GCNs executing in parallel, all aggregation operations are eliminated in T-GCN with the inter-frame reuse. Therefore, PyGT-R, PyGT-G and PiPAD all gain considerable speedups. But as mentioned above, the effects of GE-SpMM in PyGT-G are also removed with inter-frame reuse while PiPAD still can leverage the locality-optimized weight reuse to accelerate the update phase.

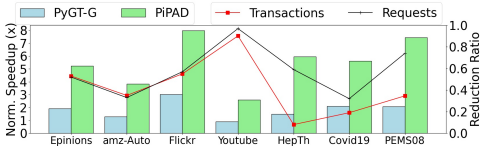
With PyTorch Profiler not support CUDA Graph, we utilize NVIDIA System Management Interface [30] to collect the GPU utilization data where running memory copy kernels also counts towards the final utilization (Table 2). Therefore, PyGT-R and PyGT-A counter-intuitively perform better because PiPAD and PyGT-G greatly reduce the computation time (§ 5.3) and amplify the percentage of total time taken by CPU-side operations. But we still outperform PyGT-G with the parallel GNN decreasing the number of issued kernels.

5.3 Analysis of Parallel GNN

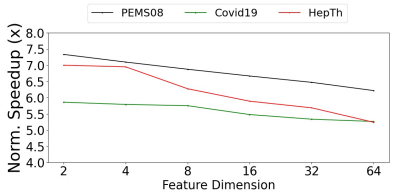
Since the data transfer greatly impacts the end-to-end training time especially in large-scale datasets, this section specially analyzes our algorithm-level optimization: intra-frame parallelism. We profile the execution time and kernel-level global memory access statistics of GNN (1-layer) module during the overall DGNN training process. PyGT and PyGT-G are chosen for comparison. We disable inter-frame reuse to thoroughly reveal the characteristics of GNN.

Speedup. Figure 11(a) (left-axis) presents the GNN execution time speedup over PyGT. Compared to GE-SpMM following the inefficient one-snapshot manner and only targeting the SpMM-like aggregation, our parallel GNN processes multiple snapshots simultaneously and optimizes both aggregation and update phases. We achieve an average 5.6× and 3.1× improvements over PyGT and PyGT-G, respectively. From the dataset respective, since PiPAD and GE-SpMM both use shared memory to cache elements from the adjacent matrix in the aggregation, our higher speedups locate at the denser graph datasets with better locality in the graph structure (e.g., Flickr, HepTh and Epinions) similar to PyGT-G.

Memory efficiency. Unlike GE-SpMM only leveraging shared memory to optimize the access to the adjacent matrix, our parallel execution can enable coalescent memory access to multiple feature matrices. For the large datasets with 2-dimensional features, the parallel GNN can load the 16-byte feature with one transaction for two graphs simultaneously



(a) Speedup of GNN execution time (left-axis) and Memory access performance (right-axis)



(b) Normalized speedup over PyGT as the feature dimension changes.

Figure 11. Detailed Analysis.

while GE-SpMM needs two individual transactions to do this. For the small-scale datasets with 16-dimensional features, single 64-byte data load generates one memory request and two transactions for one graph under GE-SpMM. But with the vector memory instructions, PiPAD can load the 256-byte data with one memory request and eight transactions for four graphs simultaneously. We access memory with the larger granularity while avoiding the *request burst*. Hence, we can decrease the total number of global memory requests and transactions against PyGT-G by average 57% and 45% respectively (right-axis in Figure 11(a)). The improvements under Youtube stand out due to the excessive sparsity of this dataset. Numerous empty rows in CSR lead to the vast redundant memory accesses under GE-SpMM while our sliced CSR avoids this problem via the finer managing granularity.

Thread utilization. The warp_execution_efficiency metric measured via NVIDIA profiler [29] directly shows the ratio of the average active threads per warp. To better demonstrate the low thread utilization issue, we set the input and hidden dimension of all seven datasets as 2 and 6. Our evaluation shows that the average thread utilization of the GNN-related kernels under PyGT-G is 57.2% while PiPAD achieves 64.9% based on our thread-aware slice coalescing.

Sensitivity. We perform the dimension sensitivity test to prove the dimension-awareness of our parallel GNN with the results shown in Figure 11(b). Due to the memory capacity limit of our testbed GPU, we only test three small-scale datasets. Employing both vector memory instructions and thread-aware slice coalescing, our parallel GNN can achieve considerable speedups (at least 5.2×) for various feature dimension settings. But due to the memory consumption factor, we can enable the higher parallelism-level (larger S_{per} in § 4.4) in the small-dimensional case for higher speedups.

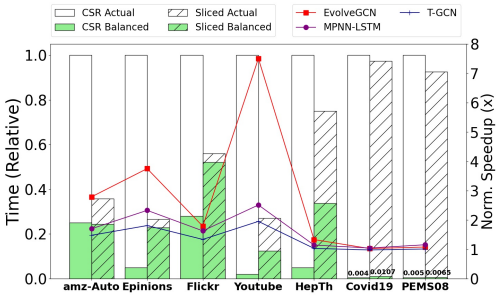


Figure 12. Load Balance Analysis and Overall Performance Comparison for the Sliced CSR.

5.4 Analysis of Sliced CSR

We further implement the variant of PiPAD using the original CSR and conduct comparison experiments to analyze our sliced CSR. Referencing the methodology in [16], we plot our effects of promoting load balance for the GNN kernels in Figure 12 (left-axis). The Balanced bars (green-colored portion) represent the ideal execution latency in perfect load balance, derived through dividing the total time of all thread blocks by the maximal number of active thread blocks a GPU can accommodate [16]. The gap between this ideal value and the actual execution time (Actual bars) reflects the degree of load imbalance, which is reduced by our sliced CSR across all datasets. The improvements are less significant under the small-scale datasets since they are more dense and easy to achieve load balance in the computation even with the original CSR. Besides, the speedups of overall training time from the model perspective (right-axis) show the similar trend to the results of load balance. Specially, the prominent improvement of EvolveGCN under Youtube is brought together by the large execution portion of GNN kernels in EvolveGCN (computation) and sliced CSR's less space usage than CSR due to the extreme sparsity of Youtube (data transfer).

6 Conclusion

This paper proposes PiPAD, a pipelined and parallel DGNN training framework to optimize the end-to-end performance on GPUs. With efficient parallel multi-snapshot processing and the runtime-level pipeline orchestration, PiPAD addresses the challenges of excessive data transmission, unexploited parallelism and memory access inefficiency in DGNN computing. Our evaluation shows that PiPAD achieves 1.22× −9.57× speedup over the state-of-the-art DGNN frameworks.

Acknowledgments

We would like to thank all reviewers for their valuable feedback. This work was supported by National Natural Science Foundation of China under Grant 51877004, 61732002 and 61572062. Yuebin Bai is the corresponding author.

References

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.

[2] Lei Bai, Lina Yao, Can Li, Xianzhi Wang, and Can Wang. 2020. Adaptive graph convolutional recurrent network for traffic forecasting. *Advances in neural information processing systems* 33 (2020), 17804–17815.

[3] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. {PipeSwitch}: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, USA, 499–514.

[4] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. Association for Computing Machinery, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/1654059.1654078

[5] Venkatesan T Chakaravarthy, Shivmaran S Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, and Shashanka Ubaru. 2021. Efficient scaling of dynamic graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1–15.

[6] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[7] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019), arXiv–1903. arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[8] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein Interface Prediction using Graph Convolutional Networks. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., Long Beach, CA, USA, 6530–6539. https://proceedings.neurips.cc/paper/2017/file/f507783927f2ec2737ba40afbd17efb5-Paper.pdf

[9] Qiang Fu, Yuede Ji, and H. Howie Huang. 2022. TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 122–134. https://doi.org/10.1145/3502181.3531467

[10] Alberto García-Durán and Mathias Niepert. 2017. Learning Graph Representations with Embedding Propagation. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Curran Associates, Inc., USA, 5119–5130. https://proceedings.neurips.cc/paper/2017/hash/e0688d13958a19e087e123148555e4b4-Abstract.html

[11] Alan Gray. 2019. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/.

[12] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: Dynamic Graph Neural Networks at Scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Philadelphia, Pennsylvania) *(GRADES-NDA '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages. https://doi.org/10.1145/3534540.3534691

[13] Shengnan Guo, Youfang Lin, Huaiyu Wan, Xiucheng Li, and Gao Cong. 2022. Learning Dynamics and Heterogeneity of Spatial-Temporal Graph Data for Traffic Forecasting. *IEEE Trans. Knowl. Data Eng.* 34, 11 (2022), 5415–5428. https://doi.org/10.1109/TKDE.2021.3056502

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

[15] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: general-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. 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, USA, 72. https://doi.org/10.1109/SC41405.2020.00076

[16] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/3437801.3441585

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., USA, 103–112. https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[18] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. 2022. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems* 4 (2022), 172–189.

[19] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *J. Mach. Learn. Res.* 21, 70 (2020), 1–73.

[20] 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, USA, 1–14. https://openreview.net/forum?id=SJU4ayYgl

[21] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems* 33 (2020), 8343–8354.

[22] Haoyang Li and Lei Chen. 2021. Cache-Based GNN System for Dynamic Graphs. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (Virtual Event, Queensland, Australia) *(CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 937–946. https://doi.org/10.1145/3459637.3482237

[23] Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan, Heng Liu, Linyong Huang, Zhaoyang Du, Fei Xue, Yuanwei Fang, Hongzhong Zheng, and Yuan Xie. 2022. Hyperscale FPGA-as-a-Service Architecture for Large-Scale Distributed Graph Neural Network. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 946–961. https://doi.org/10.1145/3470496.3527439

[24] Justin Luitjens. 2013. CUDA Pro Tip: Increase Performance with Vectorized Memory Access. https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/.

[25] Chen Ma, Liheng Ma, Yingxue Zhang, Jianing Sun, Xue Liu, and Mark Coates. 2020. Memory Augmented Graph Neural Networks for Sequential Recommendation. 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, USA, 5045–5052. https://ojs.aaai.org/index.php/AAAI/article/view/5945

[26] 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*. USENIX Association, USA, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[27] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[28] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021 (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, USA, 7937–7947. http://proceedings.mlr.press/v139/narayanan21a.html

[29] NVIDIA. 2022. Nvidia: Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/.

[30] NVIDIA. 2022. Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface.

[31] Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. 2022. Out-of-Order Backprop: An Effective Scheduling Technique for Deep Learning. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 435–452. https://doi.org/10.1145/3492321.3519563

[32] George Panagopoulos, Giannis Nikolentzos, and Michalis Vazirgiannis. 2021. Transfer Graph Neural Networks for Pandemic Forecasting. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, USA, 4838–4845. https://ojs.aaai.org/index.php/AAAI/article/view/16616

[33] 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, February 7-12, 2020*. AAAI Press, New York, NY, USA, 5363–5370. https://ojs.aaai.org/index.php/AAAI/article/view/5984

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc., USA, 8024–8035. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[35] PyTorch. 2022. PyTorch: Pytorch profiler. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[36] Ryan A. Rossi and Nesreen K. Ahmed. 2014. NetworkRepository: A Graph Data Repository with Visual Interactive Analytics. *CoRR* abs/1410.3560 (2014), arxiv–1410. arXiv:1410.3560 http://arxiv.org/abs/1410.3560

[37] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, Blai Bonet and Sven Koenig (Eds.). AAAI Press, USA, 4292–4293. http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9553

[38] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzmán López, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management* (Virtual Event, Queensland, Australia) *(CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 4564–4573. https://doi.org/10.1145/3459637.3482014

[39] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2021. Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey. *IEEE Access* 9 (2021), 79143–79168.

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

[41] Qidong Su, Minjie Wang, Da Zheng, and Zheng Zhang. 2021. Adaptive Load Balancing for Parallel GNN Training. In *Proceedings of MLSys Workshop on Graph Neural Networks and Systems (GNNSys)*. mlsys.org, San Jose, CA, USA, 1–8.

[42] Qingxiao Sun, Yi Liu, Hailong Yang, Ruizhe Zhang, Ming Dun, Mingzhen Li, Xiaoyan Liu, Wencong Xiao, Yong Li, Zhongzhi Luan, and Depei Qian. 2022. CoGNN: Efficient Scheduling for Concurrent GNN Training on GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Los Alamitos, CA, USA, Article 39, 15 pages.

[43] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, USA, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[44] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, USA, 1–12. https://openreview.net/forum?id=rJXMpikCZ

[45] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, USA, 1–24. https://openreview.net/forum?id=kSwqMH0zn1F

[46] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3447786.3456229

[47] 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. arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[48] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, USA, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

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

[50] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. 2019. T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE Transactions on Intelligent Transportation Systems* 21, 9 (2019), 3848–3858.

[51] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A General Framework for Temporal GNN Training on Billion-Scale Graphs. *CoRR* abs/2203.14883 (2022), arxiv–2203. https://doi.org/10.48550/arXiv.2203.14883 arXiv:2203.14883

[52] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.