FreshGNN: Reducing Memory Access via Stable Historical Embeddings for Graph Neural Network Training

Kezhao Huang*
Tsinghua University
hkz20@mails.tsinghua.edu.cn

Haitian Jiang*
New York University
haitian.jiang@nyu.edu

Minjie Wang†
Amazon
minjiw@amazon.com

Guangxuan Xiao
MIT
xgx@mit.edu

David Wipf
Amazon
daviwipf@amazon.com

Xiang Song
Amazon
xiangsx@amazon.com

Quan Gan
Amazon
quagan@amazon.com

Zengfeng Huang
Fudan University
huangzf@fudan.edu.cn

Jidong Zhai
Tsinghua University
zhaijidong@tsinghua.edu.cn

Zheng Zhang
Amazon
zhaz@amazon.com

ABSTRACT

A key performance bottleneck when training graph neural network (GNN) models on large, real-world graphs is loading node features onto a GPU. Due to limited GPU memory, expensive data movement is necessary to facilitate the storage of these features on alternative devices with slower access (e.g. CPU memory). Moreover, the irregularity of graph structures contributes to poor data locality which further exacerbates the problem. Consequently, existing frameworks capable of efficiently training large GNN models usually incur a significant accuracy degradation because of the currently-available shortcuts involved. To address these limitations, we instead propose FreshGNN, a general-purpose GNN mini-batch training framework that leverages a historical cache for storing and reusing GNN node embeddings instead of re-computing them through fetching raw features at every iteration. Critical to its success, the corresponding cache policy is designed, using a combination of gradient-based and staleness criteria, to selectively screen those embeddings which are relatively stable and can be cached, from those that need to be re-computed to reduce estimation errors and subsequent downstream accuracy loss. When paired with complementary system enhancements to support this selective historical cache, FreshGNN is able to accelerate the training speed on large graph datasets such as `ogbn-papers100M` and `MAG240M` by 3.4× up to 20.5× and reduce the memory access by 59%, with less than 1% influence on test accuracy.

PVLDB Reference Format:
Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. FreshGNN: Reducing Memory Access via Stable Historical Embeddings for Graph Neural Network Training. PVLDB, 17(6): 1473 - 1486, 2024.
doi:10.14778/3648160.3648184

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xxcclong/history-cache.

Proceedings of the VLDB Endowment, Vol. 17, No. 6 ISSN 2150-8097.
doi:10.14778/3648160.3648184

1 INTRODUCTION

Graphs serve as a ubiquitous abstraction for representing relations between entities of interest. Linked web pages, paper citations, molecule interactions, purchase behaviors, etc., can all be modeled as graphs, and hence, real-world applications involving non-i.i.d. instances are frequently based on learning from graph data. To instantiate this learning process, graph neural networks (GNN) have emerged as a powerful family of trainable architectures with successful deployment spanning a wide range of graph applications, including community detection [6], recommender systems [47], fraud detection [9], drug discovery [14] and more. The predictive performance of GNNs is largely attributed to their ability to exploit both entity-level features as well as complementary structural information or network effects via *message passing* schemes [15], whereby updating any particular node embedding requires collecting and aggregating the embeddings of its neighbors. Repeatedly applying this procedure by stacking multiple layers allows GNN models to produce node embeddings that capture local topology (with extent determined by model depth) and are useful for downstream tasks such as node classification or link prediction.

Not surprisingly, the scale of these tasks is rapidly expanding as larger and larger graph datasets are collected. As such, when the problem size exceeds the memory capacity of hardware such as GPUs, a workaround is required, with some form of mini-batch training being the most common [7, 16, 53, 54]. Similar to the mini-batch training of canonical i.i.d. datasets involving images or text, one full training epoch is composed of many constituent iterations, each optimizing a loss function using gradient descent w.r.t. a small batch of nodes/edges. In doing so, mini-batch training reduces memory requirements on massive graphs but with the added burden of frequent data movement from CPU to GPU. The latter is a natural consequence of GNN message passing, which for an L-layer model requires loading the features of the L-hop neighbors of each node in a mini-batch. The central challenge of efficient GNN mini-batch training then becomes the mitigation of this **data loading bottleneck**, which otherwise scales exponentially with L; even for moderately-sized graphs this quickly becomes infeasible.

Substantial effort has been made to address the data loading challenges posed by large graphs using system-level optimizations,

* Contributed equally. Work done during internship at AWS Shanghai AI lab.
† Corresponding author.

algorithmic approximations, or some combination thereof. For example, on the system side, GPU kernels are used to efficiently load features in parallel or store hot features in a GPU cache [32, 33, 50]; however, these approaches cannot avoid memory access to the potentially large number of nodes that are visited less frequently.

On the other hand, there are generally speaking two lines of work on the algorithm front. The first is based on devising sampling methods to reduce the computational footprint and the required features within each mini-batch. Notable strategies of this genre include neighbor sampling [16], layer-wise sampling [5, 57], and graph-wise sampling [7, 53]. However, neighbor sampling does not solve the problem of exponential growth mentioned previously, and the others may converge slower or to a solution with lower accuracy [56]. Meanwhile, the second line of work [4, 12, 27] stores intermediate node representations computed for each GNN layer during training as *historical embeddings* and reuses them later to reduce the need for recursively collecting messages from neighbors. Though conceptually promising and foundational to our work, as we will later show in Section 2.3, these solutions presently struggle to simultaneously achieve *both* high training efficiency *and* high model accuracy when scaling to large graphs, e.g., those with more than 10^8 nodes and 10^9 edges.

To this end, we propose a new mini-batch GNN training solution with system and algorithm co-design for efficiently handling large graphs while preserving predictive performance. As our starting point, we narrow the root cause of accuracy degradation when using historical embeddings to the non-negligible accumulation of estimation error between true and approximate representations computed using the history. As prior related work has no practical mechanism for controlling this error, we equip mini-batch training with a *historical embedding cache* whose purpose is to *selectively* admit accurate historical embeddings while evicting those likely to be harmful to model performance. In support of this cache and its attendant admission/eviction policy, we design a prototype system called FreshGNN: Reducing mEmory access via Stable Historical embeddings, which efficiently trains GNN models on large-scale graphs with high accuracy. In realizing FreshGNN, our primary contributions are as follows:

- We propose a novel mini-batch training algorithm for GNNs that achieves scalability without compromising model accuracy. This is accomplished through the use of a historical embedding cache, with a corresponding cache policy that adaptively maintains node representations (via gradient and staleness criteria to be introduced later) that are likely to be stable across training iterations. Moreover, by design our algorithm judiciously balances the caching on GPU of both embeddings and raw node features to reduce IO costs. In this way, we can economize GNN mini-batch training while largely avoiding the reuse of embeddings that are likely to lead to large approximation errors and subsequently, poor predictive accuracy.
- We create the prototype FreshGNN system to realize the above training algorithm with efficient implementation of subgraph pruning and data loading for both single-GPU and multi-GPU hardware settings.
- We provide a comprehensive empirical evaluation of FreshGNN across common baseline GNN architectures, large-scale graph datasets, and hardware configurations. Among other things, the results demonstrate that FreshGNN can closely maintain the accuracy of non-approximate neighbor sampling (within 1%) while training 3.4× up to 20.5× faster than state-of-the-art baselines.

2 BACKGROUND AND MOTIVATION

2.1 Graph Neural Networks

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with node set $\mathcal{V}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, where $n = |\mathcal{V}|$. Furthermore, let $A \in \{0,1\}^{n \times n}$ be the graph adjacency matrix such that $A_{uv} = 1$ if and only if there exists an edge between nodes u and v. Finally, $X \in \mathbb{R}^{n \times d}$ denotes the matrix of d-dimensional node features (i.e., each row is formed by the feature vector for a single node) while $Y \in \mathbb{R}^{n \times c}$ refers to the corresponding node labels with c classes.

Given an input graph defined as above, the goal of a GNN model is to learn a representation h_v for each node v, which can be used for downstream tasks such as node classification or link prediction. This is typically accomplished via a so-called *message passing* scheme [15]. For the $(l+1)$-th GNN layer, this involves computing the hidden/intermediate representation

$$\begin{aligned} h_v^{(l+1)} &= f_W^{(l+1)}\left(h_v^{(l)}, \left\{h_u^{(l)} : u \in \mathcal{N}(v)\right\}\right) \\ &= \phi_W^{(l+1)}\left(h_v^{(l)}, \operatorname*{AGG}_{u \in \mathcal{N}(v)}\left(\psi_W^{(l+1)}\left(h_u^{(l)}, h_v^{(l)}\right)\right)\right), \end{aligned} \tag{1}$$

where $h_v^{(l)}$ and $\mathcal{N}(v)$ are the embedding and neighbors of node v respectively, with $h_v^{(0)}$ equal to the v-th row of X. Additionally, ψ computes messages between adjacent nodes while the operator AGG is a permutation-invariant function (like sum, mean, or element-wise maximum) designed to aggregate these messages. Lastly, ϕ represents an update function that computes each layer-wise embedding. Note that both ϕ and ψ are parameterized by a learnable set of weights W. Within this setting, the goal of training an L-layer GNN is to minimize an application-specific loss function $\mathcal{L}(H^{(L)}, Y)$ with respect to W. This can be accomplished via gradient descent as $W \leftarrow W - \eta \nabla_W \mathcal{L}$, where η is the step size.

2.2 Difficulty in Training Large-Scale GNNs

Graphs used in GNN training can have a large number of nodes containing high-dimensional features (e.g., $d \in \{100, \ldots, 1000\}$) [19, 51]. As a representative example, within the widely-adopted Open Graph Benchmark (OGB) [18, 19], the largest graph `MAG240M` has 240M nodes with 768-dimensional 16-bit float vectors as features (i.e. 350GB total size); real-world industry graphs can be much larger still. On the other hand, as nodes are dependent on each other, full graph training requires the features and intermediate embeddings of all nodes to be simultaneously available for computation, which goes beyond the memory capacity of a single GPU.

In light of this difficulty with full graph training, the most widely accepted workaround is to instead train with stochastic mini-batches [13, 16, 41, 50, 51]. At each iteration, instead of training all the nodes, a subset/batch are first selected from the training set. Then, the multiple-hop neighbors of these selected nodes (one hop for each network layer) are formed into the subgraph needed to compute a forward pass through the GNN and later to back-propagate gradients for training. Additionally, further reduction in the working memory requirement is possible by sampling these multi-hop nodes

as opposed to using the entire neighborhood [16]. The size of the resulting mini-batch subgraph with sampled neighborhood is much smaller relative to the full graph, and for some graphs can be trained using a single device achieving competitive accuracy [52] and generality [16]. Hence mini-batch training with *neighbor sampling* has become a **standard paradigm** for large-scale GNN training.

Yet even this standard paradigm is limited by a significant bottleneck: loading the features of the sampled multiple-hop neighbors in each mini-batch, which still involves data movement growing exponentially with the number of GNN layers. In many cases, data movement can occupy more than 85% of the total training time.

2.3 Existing Mini-Batch Training Overhauls

Both system-level and algorithmic approaches have been pursued in an attempt to alleviate the limitations of mini-batch training.

System Optimizations. GNNLab [50] and GNNTier [32] cache the raw features of frequently visited nodes to GPU using metrics such as node degree, weighted reverse PageRank, and profiling. However, for commonly-encountered graphs exhibiting a power-law distribution [10], most of the nodes will experience a moderate visiting frequency and hence, the feature cache is unlikely to reduce memory access to them. More generally, because the overall effectiveness of this approach largely depends on graph structure, consistent improvement across different graph datasets is difficult to guarantee. PyTorch-Direct [33] proposes to store node features in CUDA Universal Virtual Addressing (UVA) memory so that feature loading can be accelerated by GPU kernels. Even so, data loading remains a bottle-neck for PyTorch-Direct, occupying 66% of the total execution time, as the data transfer bandwidth is still limited by CPU-GPU bandwidth.

Broader Sampling Methods. While neighbor sampling reduces the size of each mini-batch subgraph, it does not completely resolve recursive, exponential neighbor expansion. Consequently, alternative sampling strategies have been proposed such as layer-wise [5, 57] and graph-wise [7, 53] sampling. However, the resulting impact on model accuracy is graph-dependent and prior evaluations [50, 56] on large graphs like `ogbn-papers100M` indicate that a significant degradation (over 10% accuracy drop) may occur.

Reusing Historical Intermediate Embeddings. The other line of algorithmic work [4, 12, 27] for revamping mini-batch training approximates the embeddings of some node set $\mathcal{S}$ using their historical embeddings from previous training iterations. This involves modifying the original message passing scheme from 1 to become

$$\begin{aligned} h_v^{(l+1)} &= f_W^{(l+1)}\left(h_v^{(l)}, \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)}\right) \\ &= f_W^{(l+1)}\left(h_v^{(l)}, \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)\setminus\mathcal{S}} \cup \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)\cap\mathcal{S}}\right) \\ &\approx f_W^{(l+1)}\Bigg(h_v^{(l)}, \left\{h_u^{(l)}\right\}_{u\in\mathcal{N}(v)\setminus\mathcal{S}} \cup \underbrace{\left\{\bar{h}_u^{(l)}\right\}_{u\in\mathcal{N}(v)\cap\mathcal{S}}}_{\text{Historical embeddings}}\Bigg). \end{aligned} \tag{2}$$

The above computation is mostly the same as the original, except that now the node embeddings from $\mathcal{S}$ are replaced with their historical embeddings $\bar{h}_u^{(l)}$. A typical choice for $\mathcal{S}$ is any node not included within the selected seed nodes, and after each training step, the algorithm will refresh $\bar{h}_v^{(l+1)}$ with the newly generated embedding $h_v^{(l+1)}$ (authentic embedding). Using historical embeddings avoids recursive visits to neighbor node features and the aggregation of neighbor embeddings as well as the corresponding backward propagation. This reduces not only the number of raw features to load but also the computation associated with neighbor expansion. Moreover, the underlying training methodology is compatible with arbitrary message passing architectures.

While promising, there remain two major unresolved issues with existing methods that utilize historical embeddings. First, by unselectively recording the history, they all require an extra storage of size $O(Lnd)$ for an L-layer GNN, an amount which can be even larger than the total size of node features ($O(nd)$), a significant limitation. Secondly, historical embeddings as currently used may introduce impactful estimation errors during training. To help illustrate this point, let $\tilde{h}_v^{(l+1)}$ denote the approximated node embedding computed using Equation (2). The estimation error can be quantified by $\|\tilde{h}_v^{(l+1)} - h_v^{(l+1)}\|$, meaning the difference between the approximated embeddings and the authentic embeddings computed via an exact message passing scheme.

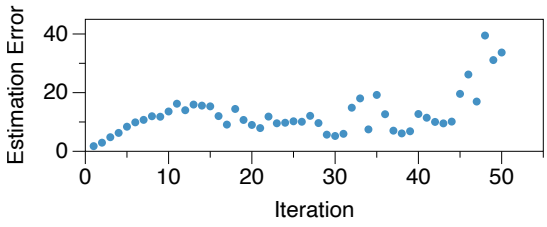


Figure 1: Average estimation error in one training epoch for GAS [12] on `ogbn-products`.

Figure 1 shows that the estimation error of each mini-batch increases considerably with more training iterations on the `ogbn-products` graph from OGB [19] when using GNNAutoScale (GAS) [12], a representative system based on historical embeddings. The root cause of this problem is that existing methods lack a mechanism for controlling the quality of the cached embeddings used for replacing message passing. Since the model parameters are updated by gradient descent after each iteration, the un-refreshed embeddings may gradually drift away from their authentic values resulting in a precipitous accuracy drop compared with the target accuracy from mini-batch training with vanilla/canonical neighbor sampling.

For the above reasons there remain ample room for new system designs that exploit historical embeddings for the setting of mini-batch training in a more nuanced way so as to maintain accuracy. This is especially true as graph size grows larger and existing methods begin to exhibit accuracy degradation or run out of GPU memory as shown in Figure 2 (see also Section 7.3 for more in-depth evaluation of these approaches).

2.4 Historical Embeddings in Full Graph Training

We note that the SANCUS algorithm from [36] also utilizes historical embeddings; however, the purpose is to reduce the communication overhead of multi-GPU *full graph training* by refreshing the remotely cached embeddings when they drift away. Even so,

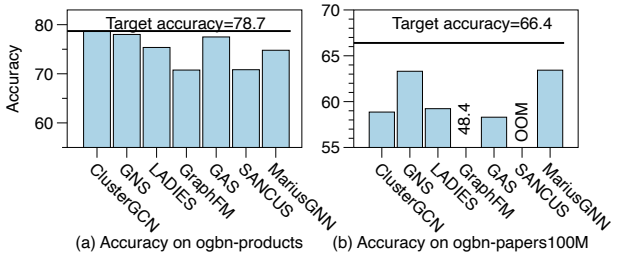


Figure 2: Test accuracy of mini-batch training algorithms (ClusterGCN [7], GNS [8], LADIES [57], GraphFM [27], GAS [12], and MariusGNN [41]) and full graph training with historical embedding (SANCUS [36]) compared with the target accuracy achieved by (expensive) neighbor sampling on: (a) relatively small `ogbn-products` graph where the gap is modest for most algorithms, and (b) larger `ogbn-papers100M` graph where the gap grows significantly.

SANCUS still relies on an $O(Lnd)$ storage for all the historical embeddings, making it run out of memory on large graphs like ogbn-papers100M, or else compromising the model capacity by the necessity of a small hidden size and/or lower floating precision. See Figure 2 for representative examples and Section 7.3 for further details. More critically, this approach requires access to the authentic embeddings themselves, which are naturally obtainable only in full graph training; in the setting of more scalable mini-batch training (our focus), computing them requires exact message passing, *the very process we are trying to avoid.*

3 DESIGN OF FRESHGNN

In this work, we propose a new strategy for utilizing historical embeddings, with targeted control of the resulting estimation error to economize GNN mini-batch training on large graphs without compromising accuracy. Intuitively, this strategy is designed to favor historical embeddings with small $\|\bar{h}_u^{(l)} - h_u^{(l)}\|$ when processing each mini-batch, while avoiding the use of those that have drifted away. However, directly computing this error requires the authentic embeddings $h_u^{(l)}$, which are only obtainable via the exact/expensive message passing we are trying to avoid as mentioned previously. We therefore adopt an alternative strategy based on a key observation about the **stability** of the node embeddings during GNN mini-batch training: *most of the node embeddings experience only modest change across the majority of iterations.*

Embedding Stability Illustration. Figure 3 showcases this phenomenon using a GCN [24] model trained on the `ogbn-arxiv` dataset. We measure the cosine similarity of the node embeddings at mini-batch iteration t with the corresponding embeddings at $t-s$ for lag $s = 20$ and plot the resulting distribution across varying t. After iteration 100 (the model converges with more than 500 iterations), most of the node embeddings exhibit a cosine similarity greater than 0.8. This provides evidence of temporal stability in many node embeddings during GNN mini-batch training, which motivates a refined historical embedding cache to selectively detect, store, and reuse such stable node embeddings.

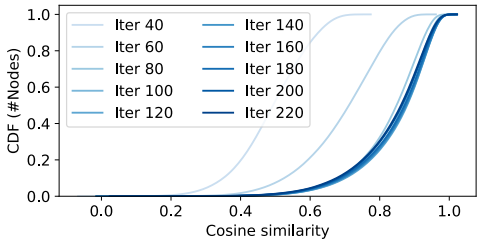


Figure 3: Distribution of cosine similarity between embeddings at iteration t and embeddings at iteration $t-s$ during the training of a GCN model on `ogbn-arxiv`. Here $s = 20$.

Algorithm 1 Mini-Batch Training w/ Historical Emb. Cache

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, input node features $H^{(0)}$, number of batches B, number of layers L, historical embedding cache C, rate for check-out using gradient p_{grad}, the maximum staleness threshold t_{stale}
$i \leftarrow 0$ ▷ Iteration ID
$\{\mathcal{B}_1, \cdots, \mathcal{B}_B\} \leftarrow Split(\mathcal{G}, B)$ ▷ Mini-batches of training nodes
for $\mathcal{B}_b \in \{\mathcal{B}_1, \cdots, \mathcal{B}_B\}$ **do**
  ▷ Subgraph generator
  $\mathcal{G}_b \leftarrow sample(\mathcal{G}, \cup_{v \in \mathcal{B}_b})$
  for $l \in \{L-1, \cdots, 1\}$ **do**
    $\mathcal{V}_{cache}^{(l)} \leftarrow \mathcal{V}_{\mathcal{G}_b}^{(l)} \cap C^{(l)}$ ▷ History index lookup
    $\mathcal{G}_b.remove\_subgraph(root = \mathcal{V}_{cache}^{(l)})$
    ▷ Remove nodes for the calculation of $\mathcal{V}_{cache}^{(l)}$
    $\mathcal{V}_{normal}^{(l)} \leftarrow \mathcal{V}_{\mathcal{G}_b}^{(l)} \setminus \mathcal{V}_{cache}^{(l)}$
  end for
  ▷ Data loader
  $H_{normal}^{(0)} \leftarrow Load\_feature(\mathcal{V}_{normal}^{(0)})$
  for $l \in \{1, ..., L\}$ **do**
    ▷ Historical embedding cache
    $h_v^{(l)} \leftarrow C^{(l)}[v], \forall v \in \mathcal{V}_{cache}^{(l)}$
    $h_v^{(l)} \leftarrow f_W^{(l)}\left(h_v^{(l-1)}, \left\{h_u^{(l-1)}\right\}_{u \in \mathcal{N}_{\mathcal{G}_b}(v)}\right), \forall v \in \mathcal{V}_{normal}^{(l)}$
  end for
  $loss = \mathcal{L}(h_{\mathcal{B}_b}^{(L)}, label_{\mathcal{B}_b})$
  $loss.backward()$
  $i \leftarrow i + 1$
  for $l \in \{1, \cdots, L\}$ **do**
    UPDATE$(C^{(l)}, \mathcal{V}_{normal}^{(l)}, \mathcal{V}_{\mathcal{G}_b}^{(l)}, h^{(l)}, i, t_{stale}, p_{grad})$
  end for
end for

FreshGNN Training Algorithm. To leverage the aforementioned embedding stability, Algorithm 1 introduces our mini-batch training process using the historical embedding cache. For each batch, we first generate a subgraph according to a user-defined sampling

method (Line 5). A subgraph consisting of the sampled L-hop neighbors for the training nodes is returned, where L is defined by whatever GNN model is chosen. Then for each layer, the nodes in the subgraph are divided into two types: normal nodes ($\mathcal{V}_{normal}$ on Line 9) and those nodes whose embeddings can be found in the historical cache ($\mathcal{V}_{cache}$ on Line 7). For the latter, embeddings are directly read from the cache (Line 13), while for the normal nodes, embeddings are computed by the neighbor aggregation assumed by the GNN (Line 14). Finally, at the end of each iteration, the algorithm will utilize information from the forward and backward propagation steps to update the historical embedding cache. The goal is to check in stable embeddings that are more reliable for future reuse while checking out those that are not (Line 20).

Figure 4 further elucidates Algorithm 1 using a toy example. Here node $v1$ is selected as the seed node of the current mini-batch as in Figure 4(a). Computing $v1$'s embedding requires recursively collecting information from multi-hop neighbors as illustrated by the subgraph in Figure 4(b). As mentioned previously, neighbor sampling can reduce this subgraph size as shown in Figure 4(c). Figure 4(d) then depicts how the historical embedding cache can be applied to further prune the required computation and memory access. The cache contains node embeddings recorded from previous iterations as well as some auxiliary data related to staleness and gradient magnitudes as needed to estimate embedding stability. In this example, the embeddings of node $v3$ are found in the cache, hence its neighbor expansion is no longer needed and is pruned from the graph. Additionally, after this training iteration, some newly generated embeddings (e.g., node $v2$) will be pushed to the cache for later reuse. Existing cached embeddings may also be evicted based on the updated metadata. In this example, both $v11$ (by staleness) and $v3$ (by gradient magnitude criteria to be detailed later) are evicted from the cache.

Main FreshGNN Components. To instantiate Algorithm 1, and enable accurate, efficient GNN training over large graphs in CPU-GPU or multi-GPU scenarios, we require three system components, namely, (i) the historical embedding cache, (ii) the subgraph generator, and (iii) the data loader; each of these correspond with a colored block in Algorithm 1). Figure 5 situates these components within a typical GNN training workflow, while supporting summaries are as follows (with subsequent sections filling in the full details):

(1) The **historical embedding cache** is the central component of our FreshGNN design, selectively storing node embeddings in GPU and providing efficient operations for fetching or updating its contents as will be detailed in Section 4. Note that we also couple historical embeddings with frequently-visited raw features in the cache to reduce memory access.
(2) The **subgraph generator** is responsible for producing a pruned subgraph given the current mini-batch and cached historical embeddings as will be discussed further in Section 5. Compared with other GNN systems, a unique characteristic of our approach is that each subgraph structure is dependent on the nodes stored in the historical embedding cache, and the latter is dynamically updated at each training iteration. This essentially creates a reversed data dependency between the stages of mini-batch preparation and mini-batch training, making it difficult to apply pipelining to overlap the two stages like in other GNN systems [34, 41, 50]. To address this challenge, we further partition the workload into two steps: graph sampling and graph pruning, where only the latter step depends on the historical embedding cache. We then adopt a mixed CPU-GPU design that samples graphs in CPU while pruning graphs in GPU, where CPU sampling can be overlapped by training time; the pruning is complemented by a GPU-friendly data structure for fast graph pruning.
(3) The **data loader** is in charge of loading the relevant node features or historical embeddings given a generated subgraph. For each node that is not pruned by historical embeddings, the data loader fetches its raw features. Since these features are typically stored in a slower but larger memory device, FreshGNN further optimizes the data transmission for three scenarios: (1) fetching features from CPU to a single GPU, (2) fetching features from CPU to multiple GPUs, and (3) fetching features from other GPUs. See Section 6 for further details.

4 HISTORICAL EMBEDDING CACHE

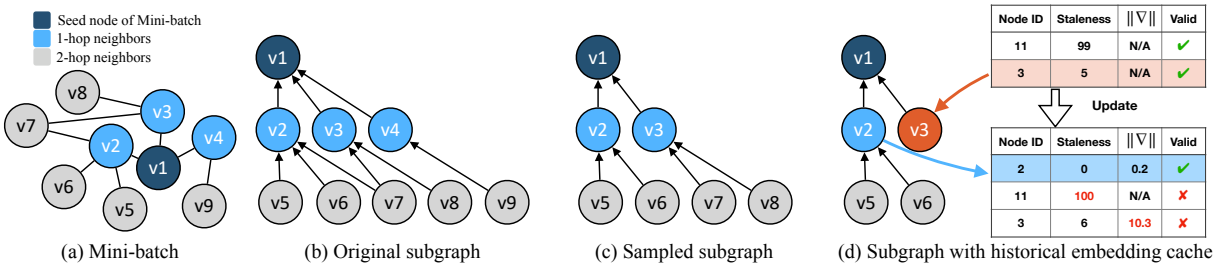
The historical embedding cache design is informed by the following: (1) *What is a suitable cache policy for selecting **stable** node embeddings that favor high accuracy?* and (2) *On top of this, how can we simultaneously take advantage of both embeddings and raw features to optimize system performance?* We address each in turn below.

4.1 Cache Policy for Accuracy

Caching intermediate node embeddings is fundamentally different with caching raw node features. Unlike the raw node features staying unchanged, the embeddings are constantly updated during model training, meaning the quality of cached embedding will influence the accuracy of trained model. As a result, we have to selectively cache and reuse the stable embeddings.

4.1.1 Caching Stable Embeddings. In order to selectively cache historical embeddings, it is crucial to identify the stable ones. However, quantifying the stability of embeddings poses a significant challenge. In the context of GNN training, stability is measured by the disparity between a true node embedding and its corresponding cached version. A naive approach would involve recomputing all embeddings in the cache after each training iteration and removing those that have deviated significantly. However, this solution is impractical due to its reliance on computationally expensive data loading and computation, which contradicts the purpose of caching aimed at reducing costs.

FreshGNN introduces a lightweight approach utilizing *gradient-based criteria* to identify stable embeddings. During training, the gradients of node embeddings are naturally computed to update the weight parameters, resulting in zero-cost acquisition of embedding gradients. These gradients serve as feedback from the model training process and can effectively indicate the stability of the embeddings. Specifically, a near-zero gradient magnitude suggests that the embedding contributes to accurate predictions and requires minimal adjustments during the current training iteration. In contrast, embeddings with large gradient magnitudes are considered less stable. By comparing the absolute values of embedding gradients, FreshGNN is capable of assessing the stability of embeddings at each iteration, enabling the storage of newly produced stable embeddings and invalidating unstable ones in the cache.



(a) Mini-batch (b) Original subgraph (c) Sampled subgraph (d) Subgraph with historical embedding cache

Figure 4: Illustration of historical embedding cache using an example mini-batch graph.

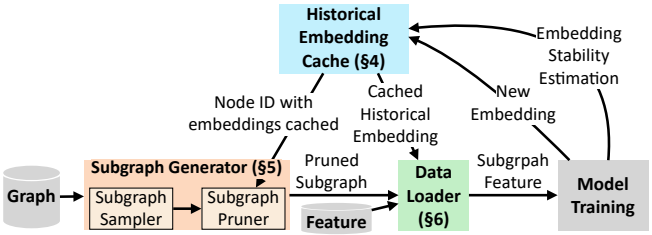


Figure 5: FreshGNN system workflow.

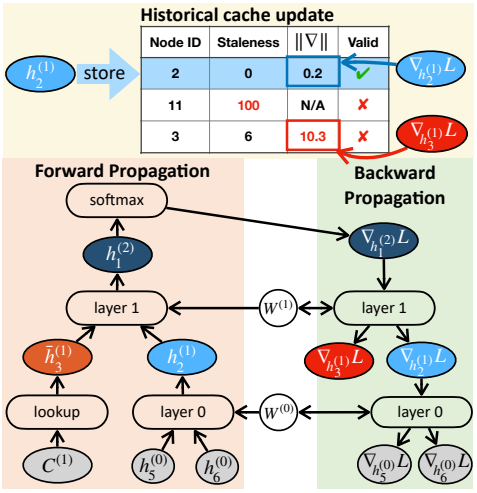


Figure 6: Illustration of the admission/eviction policy of the historical embedding cache. After a training iteration, the cache admits the embedding of Node 2 but evicts Node 3 (which was originally in the cache at the start of the iteration) by the gradient-based criteria. It also evicts Node 11 according to the staleness criteria.

Based on the concept of using gradient as the indicator of embedding stability, we formulate the embedding cache policy for accuracy as follows. Given a mini-batch graph, denote the set of nodes at layer l as $\mathcal{V}^{(l)}$ and the set of cached nodes as $\mathcal{V}_{cache}$. For nodes $v \in \mathcal{V}^{(l)}$, we use the magnitude of embedding gradients w.r.t. the training loss as a proxy for node stability at each layer. FreshGNN admits nodes with small absolute values of gradients to the cache, with the rate controlled by p_{grad}, the fraction of newly generated embeddings to be admitted. Of the remaining $(1-p_{grad})$ fraction of the nodes in the mini-batch, if any of these are already present in the cache, they will now be evicted.

In the example shown in Figure 6, at layer 1, Node 3 fetches its embedding from the cache while Node 2 computes its embedding faithfully by aggregating from neighbors. During backward propagation, FreshGNN calculates both gradients $\nabla_{h_3^{(1)}}\mathcal{L}$ and $\nabla_{h_2^{(1)}}\mathcal{L}$, and compares their norms to decide which to admit or evict; here the embedding cache decides to admit Node 2 and evict Node 3.

4.1.2 Evicting Stale Embeddings. To ensure model accuracy, it is essential to address the issue of stale embeddings that may arise due to the continuous weight parameter updates during each iteration. In addition to the gradient-based criteria, FreshGNN bounds the *staleness* of the embeddings. The staleness is initially set to zero when an embedding is admitted to the cache. With each subsequent iteration, the staleness increases by one. FreshGNN considers embeddings with staleness exceeding a predefined threshold t_{stale} as outdated and consequently evicts them from the cache. As illustrated in Figure 6, Node 11 is evicted based on this criterion. More broadly, utilizing and limiting staleness is a commonly used technique in training neural networks; the difference between FreshGNN and prior work that incorporates staleness criteria during training is discussed in Section 8.

4.1.3 Resulting Adaptive Cache Size. A notable difference from typical cache usage, which emerges from the above gradient-based and staleness-based criteria, is that our historical embedding cache size is not static or preset, but rather implicitly controlled by the two thresholds p_{grad} and t_{stale}. Larger p_{grad} or t_{stale} means more embeddings to be cached but also requires the model to tolerate larger approximation errors introduced by historical embeddings. Moreover, setting $p_{grad} = 0$ or $t_{stale} = 0$ degrades the algorithm to normal neighbor sampling without caching historical embeddings. In contrast, setting $p_{grad} = 100\%$ and $t_{stale} = \infty$ results in a policy that is conceptually equivalent to that used by GAS [12] and VR-GCN [4]. So in this sense FreshGNN is a more versatile paradigm w.r.t. previous historical embedding based methods. Later in Section 7 we demonstrate that finding suitable values of p_{grad} and t_{stale} is relatively easy in practice; however, we leave open to future research on exploiting this flexible dimension within the historical cache design space.

4.1.4 Avoiding Initial Instability. According to Figure 3, we remark that many embeddings may be unstable at the beginning of training. To address this, FreshGNN can incorporate a *stabilization period* when training begins. This involves initially disabling the historical cache and instead using regular mini-batch training. After the stabilization period, the historical cache is activated to reuse the stable embeddings. Experimental results in Section 7 demonstrate that introducing a short stabilization period (less than half of an epoch) can at times lead to improved model quality.

4.2 Cache Policy for System Performance

The cache policy of FreshGNN, aimed at ensuring model accuracy, incorporates gradient and staleness criteria to maintain stable and up-to-date embeddings. However, it is also crucial to consider the cache's impact on system performance. Therefore, in addition to accuracy considerations, we refine the cache policy by integrating both embeddings and raw features to minimize memory access and improve system performance.

The design of a runtime cache should be capable of effectively managing both embeddings and features on the GPU becomes imperative. However, the dynamic nature of embeddings presents two distinct challenges. The first challenge involves the generation of features and embeddings, noting that raw features remain static whereas embeddings are dynamically generated during training. Consequently, it is crucial to efficiently cache the embeddings while leveraging the static property of features, thereby optimizing resource utilization. The second challenge revolves around the eviction of features and embeddings. Raw features can be retained in the cache permanently, whereas embeddings necessitate dynamic eviction to maintain accuracy. Therefore, it is important to invalidate evicted embeddings with minimal overhead while simultaneously freeing up space to accommodate new embeddings.

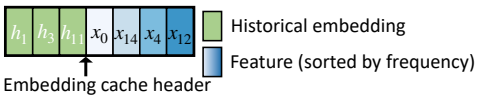


Figure 7: Bidirectional cache implementation.

To tackle these challenges, we propose a novel solution called the *bidirectional cache* to effectively manage both embeddings and features. The fundamental concept is to statically cache frequently visited features on one side of the buffer, while dynamically caching the embeddings generated during training on the other side. At the junction between embeddings and features in the buffer, features are less frequently visited and will be replaced by newly generated embeddings. The detailed design of the bidirectional cache is presented in three distinct parts: the feature side, the embedding side, and the junction where they converge.

Before the training process, FreshGNN populates the available GPU buffer with hot raw features (right hand side in Figure 7). These features are ordered based on their estimated saved memory access, ensuring that those with larger saved memory access are positioned closer to the feature side of the buffer. This placement reduces the likelihood of these features being replaced by newly generated embeddings present on the other side of the buffer.

During the training phase, FreshGNN dynamically stores the newly generated embeddings on the embedding side of the buffer (left hand side in Figure 7). To optimize runtime efficiency, GPU threads are utilized to effectively manage embeddings in parallel. To facilitate this parallel management, FreshGNN maintains a node ID mapping array with a length of $O(|\mathcal{V}|)$, where each entry stores the index of the cache that holds the embedding of the corresponding node. The additional storage required for this mapping array is affordable in comparison to the storage of the embeddings themselves. Moreover, this mapping array ensures that the compute complexity of each fetching operation is $O(1)$. Since there are no dependencies between different entries, FreshGNN can fully exploit the parallelism capabilities of the GPU.

At the junction of cached features and embeddings, FreshGNN employs a ring buffer design that effectively admits new embeddings while evicting old ones. This design ensures that features are not unnecessarily displaced by newly generated embeddings. As illustrated in Figure 7, FreshGNN maintains an embedding cache header. It initially points to the first item of the cache and moves forward when new embeddings are added. At every t_{stale} iteration, FreshGNN resets the embedding cache header to the beginning. Consequently, newly added embeddings overwrite outdated ones, naturally evicting them from the cache. To evict embeddings with significant gradient magnitudes, FreshGNN employs an approach where it invalidates the corresponding entries in the node ID mapping array, instead of physically deleting them. These invalidated slots are naturally recycled within the ring buffer design.

5 CACHE-AWARE SUBGRAPH GENERATOR

In FreshGNN, the mini-batch subgraphs are generated adaptively according to the node embeddings stored in the cache. As the cache is in turn updated at the end of each iteration, this prevents FreshGNN from adopting a naive pipelining strategy to parallelize subgraph generation and model training as in other systems. To address this challenge, we decompose subgraph generation into two steps: graph sampling and graph pruning, where only the latter depends on the historical embedding cache. We then adopt a mixed CPU-GPU design to further accelerate them.

Asynchronous CPU Graph Sampling. This step first extracts/samples subgraphs normally for the given mini-batch and then moves them to GPU without querying information from the cache. As a result, graph sampling can be conducted asynchronously with the later GPU computation. We further utilize multithreading instead of multiprocessing to produce multiple subgraphs concurrently in contrast to the existing systems like DGL [44] and PyG [11]. Additionally, we use a task queue to control the production of subgraphs and avoid overflowing the limited GPU memory.

GPU Graph Pruning. The graph pruning step scans the mini-batch graph from the seed node layer to the input node layer. For any cached node, it recursively removes all the multiple-hop neighbors so that the corresponding computation is no longer needed for model training. The remaining challenge is that traditional sparse formats are not suitable for parallel modification in GPU. As shown in Figure 8, the Coordinate (COO) format represents a graph using two arrays containing the source and destination node IDs of each edge. Pruning incoming neighbors of a node in COO requires first locating neighbors using a binary search and then deleting them from both arrays. The prune complexity is $O(\log(|\mathcal{E}_{sample}|) + N_{neighbors})$, in which $|\mathcal{E}_{sample}|$ is the number of edges in the sampled graph and $N_{neighbors}$ is the number of neighbors to be pruned. Somewhat differently, for the Compressed Sparse Row (CSR) format, after deleting the edges, the row index arrays need to be adjusted accordingly, with prune complexity of $O(|\mathcal{V}_{sample}|+N_{neighbors})$, where $|\mathcal{V}_{sample}|$ is the number of nodes in the sampled graph.

To reduce these graph pruning costs on GPU, we designed a novel data structure called CSR2 for FreshGNN. CSR2 uses two arrays to represent row indices – the first array records the starting offset

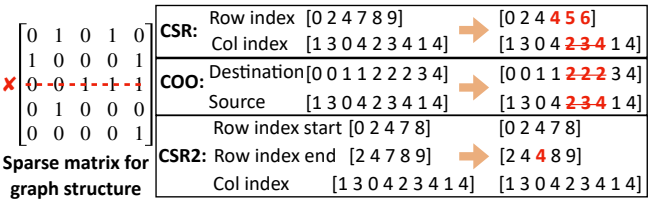


Figure 8: Removing a center node's neighbors using different graph data structures

of a node's neighbors to the column index array while the second array records its ending offset. As illustrated in Figure 8, for node i, its neighbors are stored in the column index segment starting from $start[i]$ to $end[i]$. To remove a node's neighbors, we can simply set the corresponding $end[i] = start[i]$ without any changes to the column index array, resulting in an $O(1)$ prune complexity. The data structure is also suitable for parallel processing on GPU because there is no data race condition.

6 DATA LOADER

Once a subgraph has been pruned, FreshGNN needs to load the historical embeddings for cached nodes and the features for unpruned nodes. Unlike historical embeddings stored in local GPU memory, features are stored on CPU (single-GPU training) or remote GPUs (multi-GPU training). Therefore, the loading process is still critical to performance.

However, the indices for unpruned nodes and their features reside on different devices. The index is calculated in the GPU (computation device) while the needed features are stored on the CPU or remote GPUs (storage device). Under such conditions, the naive way to fetch the features is via **two-sided** communication. This involves first transferring node indices from the computation device to the storage device, compacting the corresponding features on the storage device, and then sending the packed features back to the computation device. This process introduces extra communication for transferring indices and synchronization between the computation and storage devices. The FreshGNN data loader can initialize memory access from a GPU side computation device (*one-sided communication*) and change the communication schedule to fully utilize the bandwidth (*multi-round communication*). As a result, it can efficiently load the features of the unpruned nodes.

One-sided Communication. FreshGNN employs one-sided communication to address this problem. Based on Unified Virtual Addressing (UVA) [33], the CPU and GPU memories are mapped to a unified address. This enables the computation device to directly fetch features from a mapped buffer of the storage device using the node index. As shown in Figure 9(a), nodes needed for training are partly pruned by the cache (green color), and for the unpruned ones (orange color), the GPU fetches features using their node ID directly from node features mapped with UVA. In Figure 9(b), multiple GPUs can concurrently fetch data using UVA for parallel training.

Multi-round Communication. With a larger number of GPUs available during training, all node features can be partitioned and stored across multiple machines such that GPUs serve as both computation and storage devices. Therefore, each GPU fetches the features of the relevant unpruned nodes from other GPUs, resulting in all-to-all communication between every pair of GPUs. UVA can still be used to perform one-sided memory access in this scenario. However, as GPUs are connected asymmetrically, link congestion could badly degrade the overall bandwidth. To address this problem, in addition to one-sided communication, FreshGNN breaks cross-GPU communication into multiple rounds to avoid congestion and fully utilize the bi-directional bandwidth on links. Figure 9(c) shows a typical interconnection among four GPUs, where GPUs are first connected via PCIe and then bridged via a host. For this topology, FreshGNN will decompose the all-to-all communication into five rounds. In round one, data is exchanged only between GPUs under the same PCIe switch, while the remaining four rounds are for exchanging data between GPUs across the host bridge. The data transmission at each round is bi-directional to fully utilize the bandwidth of the underlying hardware. This multi-round communication can effectively avoid congestion in all-to-all communication.

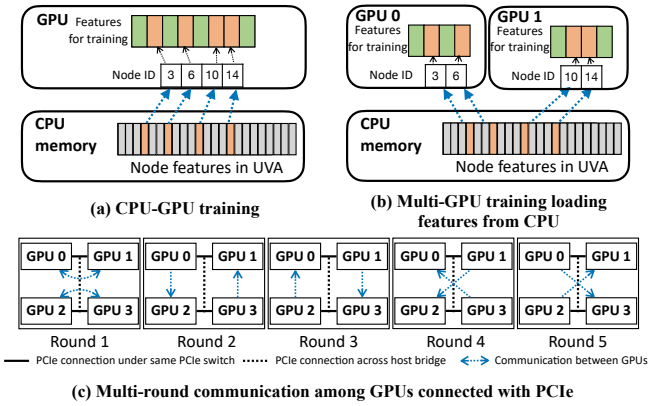


Figure 9: Feature data loading for (a) CPU-GPU training and (b) multi-GPU training.

7 EVALUATION

In this section, we first describe our experimental setup, followed by results covering system efficiency and model accuracy. We conclude with an empirical study of our cache effectiveness, ablations over our system optimizations, and heterogeneous extensions.

7.1 Experimental Setup

Table 1: Graph dataset details, including input node feature dimension (Dim.) and number of classes (#Class).

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	Dim.[1]	#Class
ogbn-arxiv [19]	2.9M	30.4M	128	40
ogbn-products [19]	2.4M	123M	100	47
ogbn-papers100M [19]	111M	1.6B	128	172
MAG240M [18]	244.2M	1.7B	768	153
Twitter [48]	41.7M	1.5B	768	64
Friendster [49]	65.6M	1.8B	768	64

Environments. Experiments were conducted on servers each with two AMD EPYC 7742 CPUs (2×64 cores in total) and four NVIDIA

[1]The first three datasets use float32 while the latter three use float16 [18, 19].

A100 (40GB) GPUs connected via PCIe 4.0. The software environment on this machine is configured with Python v3.9, PyTorch v1.10, CUDA v11.3, DGL v0.9.1, and PyG v2.2.0.

Datasets. The dataset statistics are listed in Table 1. Among them, the two smallest datasets, `ogbn-arxiv` and `ogbn-products`, are included only for model accuracy comparisons and to provide contrast with much larger datasets, including `ogbn-papers100M` and `MAG240M` [18] that are used to test both accuracy and speed. Following the common practice of previous work [13, 50], we also use the graph structure from `Twitter` [48] and `FriendSter` [49] with artificial features to test the speed of different systems.

GNN models & Training details. We employed three widely-used GNN architectures for our experiments: GraphSAGE [16], GCN [24], and GAT [39]. All models have 3 layers and 256 as the hidden size. The base sampling method for mini-batch training is neighbor sampling, and we follow the setting of OGB leaderboard [19] to set the neighbor sampling fan-out as 20, 15, and 10. To measure their baseline model accuracy, we train the models using mini-batch neighbor sampling in DGL. The batch size is chosen to be 1000. We set $p_{grad} = 0.9$ and $t_{stale} = 200$ for FreshGNN for all experiments (with the exception of Section 7.4, where we study the impact of these thresholds). For speed tests, as the performance bottleneck is data loading, we only measure the performance of different systems on GraphSAGE; similar performance occurs with other models.

7.2 System Efficiency

We first demonstrate the system advantage of FreshGNN against state-of-the-art/representative alternatives for mini-batch training, including PyG [11], GAS [12], ClusterGCN [7], DGL [44] (as well as DistDGL [56]), PyTorch-Direct [33], MariusGNN [41], and GNNLab [50].

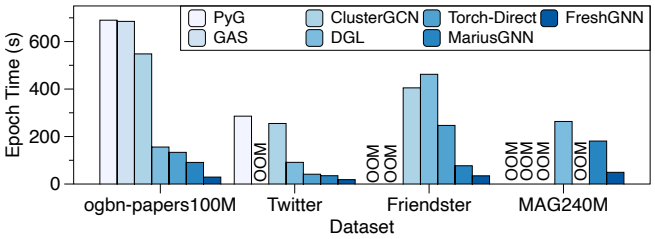


Figure 10: Epoch time comparisons training a GraphSAGE model using a single GPU.

Figure 10 compares the time for training a GraphSAGE model for one epoch on the four large-scale graph datasets using a single GPU. FreshGNN significantly outperforms all the other baselines across all the datasets. On `ogbn-papers100M` and `MAG240M`, FreshGNN has average speedup of 3.4× over the best of other baselines (MariusGNN). Compared with the widely-used GNN systems DGL and PyG, FreshGNN is 5.3× and 23.6× faster respectively on `ogbn-papers100M`. Both PyTorch-Direct and FreshGNN utilize CUDA UVA memory to accelerate feature loading, but FreshGNN is still 4.6× faster because it can reduce the number of features to load by a large margin. With respect to other mini-batch training algorithms, FreshGNN is orders of magnitude faster than GAS and ClusterGCN on `ogbn-papers100M`. GAS also runs out of memory on graphs that have either more nodes/edges or larger feature dimensions due to the need to store the historical embeddings of all the nodes. Incidentally, on the largest dataset `MAG240M` only DGL, MariusGNN, and FreshGNN avoid OOM, but with FreshGNN executing 5.3× faster than DGL and 3.7× faster than MariusGNN.

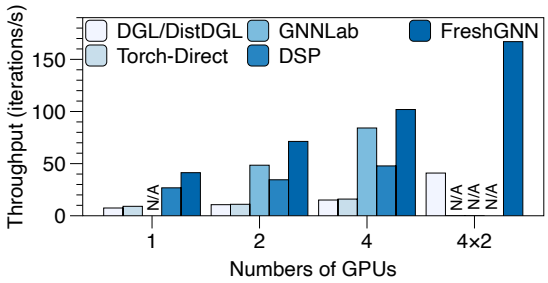


Figure 11: Scalability comparison for training a GraphSAGE model on `obgn-papers100M` using multiple GPUs.

FreshGNN can also scale to multiple GPUs and machines. In this setting, we include new baselines DistDGL [56] for distributed training, DSP [2] for fast multi-GPU sampling and training, and GNNLab [50] which partitions GPUs to sampling or training workers; hence GNNLab does not have single-GPU performance. Figure 11 compares the throughput (measured as the number of iterations computed per second) when training GraphSAGE on `ogbn-papers100M`. Both DGL and PyTorch-Direct deliver almost no speedup because of the data loading bottleneck that cannot be parallelized via addition of more GPUs. DSP utilizes GPU resources for graph sampling. However, as its graph structure is stored in a distributed manner on multiple GPUs, extra communication is needed among GPUs, which leads to inferior performance on our testing server equipped with PCIe connections. Meanwhile, FreshGNN enjoys good scalability from 1 to 4 GPUs and is up to 1.49× faster than GNNLab. When scaling to two machines each with four GPUs, the scalability of FreshGNN becomes better, as it has more CPU resource for graph sampling, and is 4.07× faster than DistDGL.

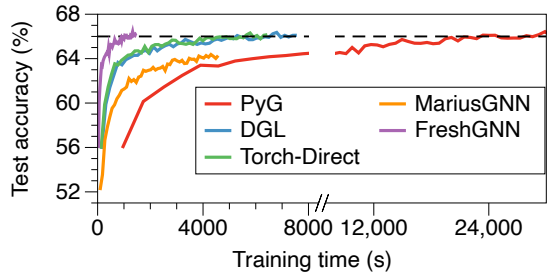


Figure 12: Test accuracy versus training time comparisons for GraphSAGE on `ogbn-papers100M`.

To further showcase FreshGNN speed advantages, Figure 12 plots the time-to-accuracy curve of different training systems across 50 epochs. Except MariusGNN, all the baselines here are using mini-batch neighbor sampling without any further approximation so they converge to the same accuracy of ~66%; MariusGNN obtains a lower accuracy of ~63% and further epochs did not improve performance. FreshGNN can reach this same accuracy in 25 minutes; the slowest baseline (PyG) takes more than 6 hours.

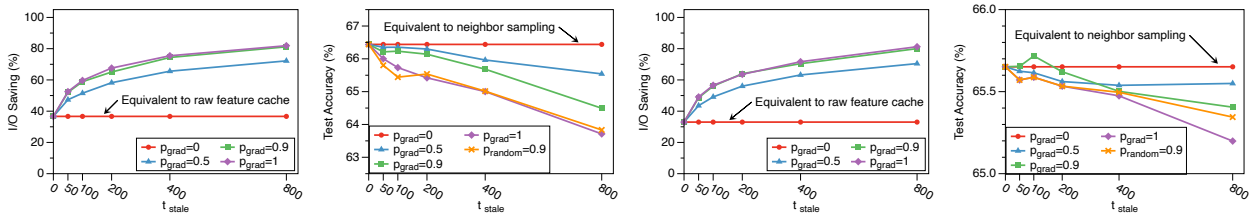


Figure 13: (a) (c) The percentage saving of I/O for loading node features and (b) (d) the test accuracy on `ogbn-papers100M` and `MAG240M` under different choices of p_{grad} and t_{stale}.

7.3 Model Accuracy

Table 2 compares the test accuracy of FreshGNN with other mini-batch training algorithms GAS [12], ClusterGCN [7], GraphFM [27], and MariusGNN [41], as well as the full-graph training algorithm SANCUS [36] (which also uses historical-embeddings as described previously). Following the common practice in [8, 13], the target accuracy is obtained from training the base models (GraphSAGE, GAT, GCN) using neighbor sampling.

In general, nearly all algorithms perform relatively well on small graphs such as `ogbn-arxiv` and `ogbn-products` with only a few exceptions. However, when scaling to larger graphs such as `ogbn-papers100M`, most of the baselines experience a substantial accuracy drop (from 7% to 18%) while running out of memory on `MAG240M`. MariusGNN can run on all datasets with GraphSAGE and GCN (but not GAT), albeit with lower accuracy (over 2% drop) and longer epoch time (from Figure 10). We remark that SANCUS, as the representative of scalable full graph training, has low accuracy or suffers from OOM on large graphs due to the required storage overhead mentioned in Section 2.4. By contrast, FreshGNN only experiences a less than 1% accuracy difference across all datasets and base models.

Table 2: Test accuracy of different training algorithm minus the target accuracy obtained by neighbor sampling (larger is better). Bold numbers denote the best performing method.

		Small datasets		Large datasets	
	Methods	`arxiv`	`products`	`papers100M`	`MAG240M` [2]
GraphSAGE	Target Accuracy	70.91	78.66	66.43	66.14
	GAS	+0.44	-1.19	-8.17	OOM
	ClusterGCN	-3.10	+0.00	-7.57	OOM
	GraphFM	**+0.62**	-7.90	-18.40	OOM
	MariusGNN	-3.91	-4.33	-3.43	-2.97
	SANCUS	-4.42	-7.83	OOM	OOM
	FreshGNN	+0.60	**+0.38**	**-0.15**	**-0.51**
GAT	Target Accuracy	70.93	79.41	66.13	65.16
	GAS	**-0.04**	-2.23	-8.67	OOM
	ClusterGCN	-3.91	-2.92	-8.08	OOM
	GraphFM	-22.67	-16.54	OOM	OOM
	MariusGNN	-1.37	OOM	OOM	OOM
	SANCUS	-2.47	OOM	OOM	OOM
	FreshGNN	-0.50	**-0.54**	**-0.71**	**-0.36**
GCN	Target Accuracy	71.24	78.57	65.78	65.24
	GAS	+0.44	-1.91	-12.29	OOM
	ClusterGCN	-3.13	**+0.40**	-12.43	OOM
	GraphFM	**+0.47**	-15.70	-18.70	OOM
	MariusGNN	-0.55	-1.10	-2.04	-2.37
	SANCUS	-3.04	-9.44	OOM	OOM
	FreshGNN	-0.71	-0.31	**-0.16**	**-0.29**

7.4 Cache Effectiveness

Recall that p_{grad} and t_{stale} are the two thresholds that control the admission and eviction criteria of the historical embedding cache. In this section, we study their impact on system performance and model accuracy. We will show that a straightforward choice of the two thresholds can lead to the aforementioned competitive system speed as well as model accuracy.

Impact on System Performance. In typical cache systems, the I/O saving percentage is equal to the cache hit rate. However, hitting the cached historical embeddings of a node will prune away all the I/O operations that would otherwise be needed to load the features of multi-hop neighbors, meaning the I/O savings can potentially be much larger than the cache hit rate. We plot this saving percentage w.r.t. neighbor sampling (i.e., no caching) under different p_{grad} and t_{stale} in Figure 13 (a) and (c). Because the historical embedding cache in FreshGNN is initialized via raw node features (Section 4.2), the red $p_{grad} = 0$ line reflects the performance of neighbor sampling with a raw feature cache. As expected, larger p_{grad} or t_{stale} results in more I/O reduction. On both graph datasets, a raw feature cache can only reduce I/O by $< 40\%$ but the historical embedding cache can reduce I/O by more than 60% when choosing $t_{stale} > 200$.

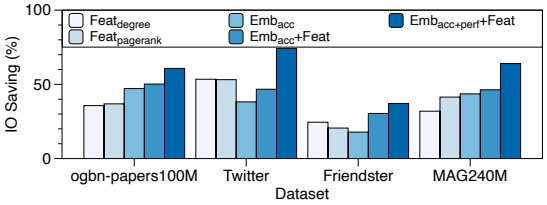


Figure 14: IO saving using different caching policies

We further present a detailed comparison of IO saving among various cache policies in Figure 14. Specifically, we evaluate the degree-based and PageRank-based cache policies [32] for feature cache and investigate three caching policies for historical embedding cache: cache policy for accuracy (Emb_{acc}), cache policy for accuracy with features ($Emb_{acc} + Feat$), and the cache policy for accuracy and performance with features ($Emb_{acc+perf} + Feat$) that utilizes bidirectional cache and prioritizes embeddings with high memory access savings. Our findings suggest that the IO saving achieved through feature caching is similar across different policies (on average 36.4% vs. 38.0%), as there are only a few hot nodes. On the other hand, caching historical embeddings alone does not result

[2]We report validation accuracy for MAG240M due to the absence of labels for test set.

Table 3: Accuracy with historical cache introduced at different iterations. INF refers to never starting historical cache, which reproduces the original target accuracy.

Iteration ID	0	200	400	800	INF/Target accuracy
papers100M	-0.15	-0.29	-0.06	-0.27	66.43
MAG240M	-0.51	-0.12	-0.18	-0.12	66.14

in significant IO saving (36.7%), as the embedding cache policy is limited to stable embeddings. However, by combining feature and embedding caching, we observe a substantial improvement (43.4%) compared to using either of the two caching schemes individually. Furthermore, by leveraging cache policy for performance to select embeddings that save a larger amount of memory access, FreshGNN achieves even higher IO savings (59.0%).

Impact on Model Accuracy. We plot the corresponding accuracy curves under different configurations in Figure 13 (b) and (d). As expected, larger p_{grad} or t_{stale} means more relaxed control on the embedding errors which consequently results in lower test accuracy. Beyond this, there are two interesting findings with practical significance. First, we can set p_{grad} very close to one without an appreciable impact on the model accuracy. This aligns well with the observation in Figure 3 that most of the node embeddings are temporally stable and can be safely admitted to the cache. Secondly, with a proper p_{grad} value, GNN models can tolerate node embeddings that were last updated hundreds of iterations ago. By cross-checking the I/O saving figures, we find that this is also a sweet spot in terms of system performance, which eventually led us to choose $p_{grad} = 0.9$ and $t_{stale} = 200$ for all the experiments (obviously excluding the Figure 13 results).

The accuracy of FreshGNN can be further improved using the stabilization period from Section 4.1.4. Table 3 shows the accuracy of GraphSAGE with the historical cache introduced at different iterations (for reference, there are 1200 iterations in an epoch for papers100M). Here 0 means starting the historical cache at the beginning of training, and *INF* means never starting historical cache, which is equivalent to regular mini-batch training. For papers100M, the effect is inconsequential, as the accuracy gap is small even without a stabilization period. However, for mag240M, the accuracy gap can be reduced from -0.51 to -0.12 with the stabilization period.

7.5 Ablation Study of System Optimizations

In this section, we study the system performance of individual components. All the results are tested on ogbn-papers100M with a three-layer GraphSAGE model.

Subgraph Generator. Figure 15(a) shows sampling time per epoch of FreshGNN and DGL, as well as the training epoch time achieved by FreshGNN. FreshGNN sampler shows good scalability with more CPU threads. With 32 CPU threads, FreshGNN is able to reduce the sampling time per epoch to 11 seconds, which can be overlapped by training epoch time (~30s). While DGL takes more than 70 seconds to finish graph sampling.

Figure 15(b) measures the time to prune the cached nodes and their neighbors from a subgraph. Overall, CSR2 is orders-of-magnitude faster than CSR and COO regardless of the batch size in use. Specifically, CSR requires frequent CPU-GPU synchronization when invalidating the neighbors of the pruned nodes in the column index.

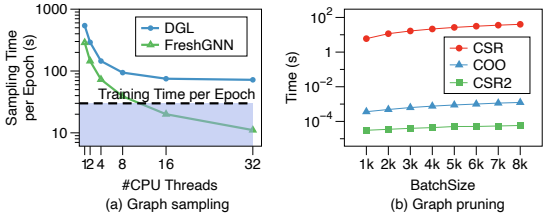


Figure 15: Effectiveness of FreshGNN's subgraph generator (measured on ogbn-papers100M).

As a result, graph pruning in total takes 99% of the iteration time. Subgraph pruning using COO is faster, which reduces the pruning overhead to 4.5% of the iteration time, but is still much slower than CSR2. The subgraph pruning time using CSR2 is negligible – it only occupies 26µs per iteration.

Data Loader. Figure 18 shows the improvement afforded by the optimizations of FreshGNN's data loader for multi-GPU communication, on the aforementioned PCIe server and a server equipped with NVLink. Compared with the communication utilizing NCCL all-to-all operations, the one-sided communication is 23% faster on average on PCIe and NVLink GPUs. After scheduling using the multi-round communication pattern, the bandwidth is increased by 145% and 85%, respectively.

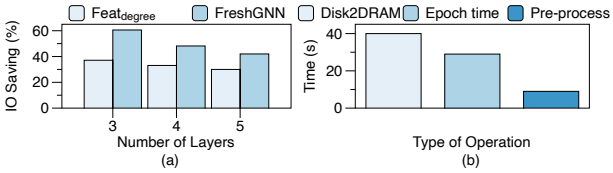


Figure 16: (a) IO savings for papers100M with varying model depth; (b) Overhead analysis for training on papers100M

IO Savings with Varying Model Depth. Figure 16(a) shows how IO savings change with GNN depth. For both feature cache and historical caches adopted by FreshGNN, the IO saving decreases with the increased number of layers and accessed nodes. However, FreshGNN consistently maintains higher IO saving of 1.50× compared to a standard feature cache based on node degree.

Pre-processing Overhead. In Figure 16(b), we compared the pre-processing overhead with other necessary steps during training, such as loading node features from the disk (Disk2DRAM) and the training time for each epoch (Epoch time). The pre-processing overhead is smaller compared to these steps. Additionally, for the same dataset, the pre-processing results can be reused multiple times. Therefore, the pre-processing overhead of FreshGNN is not a significant factor.

IO Savings with Different Graph Topologies. Figure 17 shows the IO saving from historical cache on graph structures generated with different topology. By changing the average node degree and diameters, we find that historical cache used by FreshGNN can always lead to more IO saving than degree-based feature cache (1.56× and 1.43×).

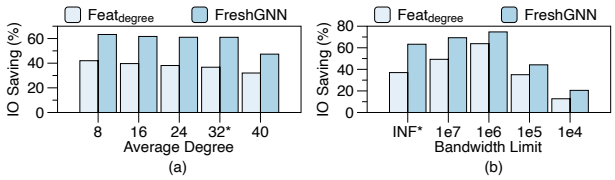


Figure 17: IO saving for (a). graphs with different average degrees; (b). graphs with different bandwidth limits.

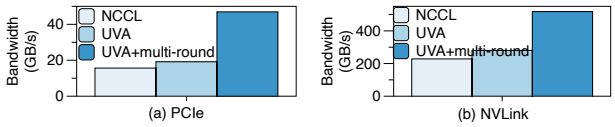


Figure 18: Optimizations for multi-GPU communication

7.6 Extension to Heterogeneous Graphs

While our primary focus has been on the most popular large-scale benchmark settings, FreshGNN can also be naturally extended to heterogeneous graphs. We now show one such use case on `MAG240M`, which was homogenized in the previous experiments. Here we instead use the heterogeneous form, along with R-GCN [37], which is arguably the most popular heterogeneous GNN architecture. In Figure 19, we compare FreshGNN with the neighbor sampling DGL baseline, which is the only related work capable of running in this setting; other baselines either do not support heterogeneous graphs or else suffer from OOM. From these results we observe that while the accuracy is almost the same, FreshGNN is 20.5× faster.

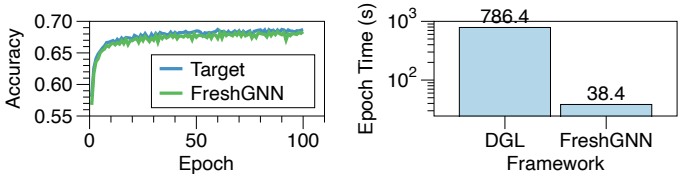


Figure 19: R-GCN evaluation on `MAG240M`.

8 RELATED WORK

Full-Graph GNN Systems. There exists a rich line of work on scaling full-graph GNN training, where the general idea is to split the graph into multiple partitions that can fit into the computing devices. Unlike mini-batch training, full-graph training updates all the nodes and edges simultaneously. Notable systems include NeuGraph [30], ROC [21], and DistGNN [31], which design smart data partitioning or data swap strategies to reduce the memory consumption and I/O cost of full-graph training. DGCL [1] proposes a new communication pattern to reduce network congestion. Dorylus [38] and GNNAdvisor [45] optimize the computation of GNN training by exploring properties of the graph structure. BNS-GCN [42] proposes boundary-node-sampling to reduce the communication incurred by boundary nodes during full graph training. Beyond these, PipeGCN [43] and DIGEST [3] utilize stale representations of neighbor nodes to reduce communication overhead, and MGG [46] applies pipelined communication for overhead hiding. Finally, as discussed in Section 2.4, SANCUS [36] uses historical embeddings to reduce communication overhead among GPUs, and compares them with authentic embeddings to limit bias.

System Optimizations for GNNs. Previous work has shown that data movement is the bottleneck for GNN mini-batch training on large graphs. By using smart graph partitioning [22, 29, 32, 56] and data placement [2, 13, 33, 50], this cost can be reduced under various training settings. For example, DistDGL [56] replicates high-degree nodes, together with sparse embedding updates, to reduce the communication workload for distributed training. DSP [2] stores node features and graph structure in a distributed way on GPUs to fully utilize NVlink bandwidth. MariusGNN [41] addresses the scenario of out-of-core training, reducing data swaps between disk and CPU memory by reordering training samples for better locality. FreshGNN's selective historical embedding method is orthogonal to both DistDGL and MariusGNN, and can potentially be combined with them. Other systems, such as GNNLab [50] and NextDoor [20], utilize GPUs to accelerate the graph sampling process. These techniques are complementary to FreshGNN which optimizes feature loading.

Training with Staleness. Staleness criteria have long been adopted for efficiency purposes when training deep neural networks. Systems including SSP [17], MXNet [25], and Poseidon [55] update model parameters asynchronously across different devices, helping to reduce global synchronization among devices and improve parallelism. Another branch of work, including PipeDream [35] and PipeSGD [26], pipeline DNN training and utilize stale parameters to reduce pipeline stall. They limit the staleness by periodically performing global synchronization. FreshGNN, however, selectively stores and reuses stale node embeddings instead of parameters which is unique to GNN models.

Cache for GNNs. There are two ways to cache for GNN training. The first is based on finding and caching node features that are frequently visited to save memory access. Previous methods utilize node degree [28, 40, 45], PageRank [23, 32], or profiling [50] to estimate the frequency. The second is based on historical embeddings. Previous work [4, 12] caches either the historical embeddings of all nodes, or boundary nodes [36]. Such methods may experience out-of-memory or low accuracy on large graphs as we have shown.

9 CONCLUSION

In this paper, we propose FreshGNN, a general framework for training GNNs on large, real-world graphs. At the core of our design is a new mini-batch training algorithm that leverages a historical cache for storing and reusing GNN node embeddings to avoid re-computation from raw features. To identify stable embeddings that can be cached, FreshGNN designates a cache policy using a combination of gradient-based and staleness criteria. Accompanied with other system optimizations, FreshGNN is able to accelerate the training speed of GNNs on large graphs by 3.4× over state-of-the-art systems, with less than 1% influence on model accuracy.

ACKNOWLEDGMENTS

This work is partially supported by National Key R&D Program of China under Grant 2021YFB0300300, National Natural Science Foundation of China (U20A20226), NSFC for Distinguished Young Scholar (62225206).

REFERENCES

[1] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. DGCL: An efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 130–144, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. Dsp: Efficient gnn training with multiple gpus. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 392–404, 2023.

[3] Zheng Chai, Guangji Bai, Liang Zhao, and Yue Cheng. Distributed graph neural network training with periodic stale representation synchronization, 2022.

[4] Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

[5] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[6] Zhengdao Chen, Xiang Li, and Joan Bruna. Supervised community detection with line graph neural networks. *arXiv preprint arXiv:1705.08415*, 2017.

[7] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 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*, pages 257–266, 2019.

[8] Jialin Dong, Da Zheng, Lin F Yang, and Geroge Karypis. Global neighbor sampling for mixed cpu-gpu training on giant graphs. *arXiv:2106.06150*, 2021.

[9] Yingtong Dou, Zhiwei Liu, Li Sun, Yutong Deng, Hao Peng, and Philip S Yu. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 315–324, 2020.

[10] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 251–262, New York, NY, USA, 1999. Association for Computing Machinery.

[11] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[12] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. GNNAutoScale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning*, pages 3294–3304. PMLR, 2021.

[13] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 551–568, 2021.

[14] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics*, 22(6):bbab159, 2021.

[15] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017.

[16] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

[17] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in Neural Information Processing Systems*, 26, 2013.

[18] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. OGB-LSC: A large-scale challenge for machine learning on graphs. *arXiv:2103.09430*, 2021.

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in Neural Information Processing Systems*, 33:22118–22133, 2020.

[20] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 311–326, 2021.

[21] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, March 2-4, 2020*, pages 187–198, Austin, TX, USA, 2020. mlsys.org.

[22] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[23] Taehyun Kim, Changho Hwang, KyoungSoo Park, Zhiqi Lin, Peng Cheng, Youshan Miao, Lingxiao Ma, and Yongqiang Xiong. Accelerating gnn training with locality-aware partial execution. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 34–41, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR '17, Palais des Congrès Neptune, Toulon, France, 2017.

[25] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27, 2014.

[26] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-sgd: A decentralized pipelined sgd framework for distributed deep net training. *Advances in Neural Information Processing Systems*, 31, 2018.

[27] Zekun Li, Shu Wu, Zeyu Cui, and Xiaoyu Zhang. GraphFM: Graph factorization machines for feature interaction modeling. *arXiv:2105.11866*, 2021.

[28] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 401–415, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. {BGL}:{GPU-Efficient}{GNN} training by optimizing graph data {I/O} and preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 103–118, 2023.

[30] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 443–458, Renton, WA, July 2019. USENIX Association.

[31] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. DistGNN: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[32] Seung Won Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-mei Hwu. Graph neural network training and data tiering. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 3555–3565, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. *arXiv:2101.07956*, 2021.

[34] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 533–549, 2021.

[35] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment*, 15(9):1937–1950, 2022.

[37] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[38] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 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 21)*, pages 495–514, 2021.

[39] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, Vancouver, BC, Canada, 2018. OpenReview.net.

[40] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius++: Large-scale training of graph neural networks on a single machine. *arXiv preprint arXiv:2202.02365*, 2022.

[41] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. MariusGNN: Resource-efficient out-of-core training of graph neural networks. In *Eighteenth European Conference on Computer Systems (EuroSys' 23)*, 2023.

[42] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. BNS-GCN: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. *Proceedings of Machine Learning and Systems*, 4:673–693, 2022.

[43] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. PipeGCN: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428*, 2022.

[44] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[45] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 515–531, 2021.

[46] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. MGG: Accelerating graph neural networks with Fine-Grained Intra-Kernel Communication-Computation pipelining on Multi-GPU platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 779–795, Boston, MA, July 2023. USENIX Association.

[47] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: a survey. *ACM Computing Surveys (CSUR)*, 2020.

[48] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, page 177–186, New York, NY, USA, 2011. Association for Computing Machinery.

[49] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[50] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, 2022.

[51] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[52] Jiaxuan You, Zhitao Ying, and Jure Leskovec. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33:17009–17021, 2020.

[53] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. Decoupling the depth and scope of graph neural networks. *Advances in Neural Information Processing Systems*, 34, 2021.

[54] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. GraphSAINT: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[55] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 181–193, USA, 2017. USENIX Association.

[56] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. DistDGL: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.

[57] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems*, 32, 2019.