FGNN: A Factored System for Sample-based GNN Training over GPUs

Paper #45

Abstract

We propose FGNN, a sample-based GNN training system in a single machine multi-GPU setup. FGNN adopts a factored design for multiple GPUs, where each GPU is dedicated to the task of graph sampling or model training. It accelerates both tasks by eliminating GPU memory contention. To balance GPU workloads, FGNN applies a global queue to bridge GPUs asynchronously and adopts an effective method to adaptively allocate GPUs for different tasks. Furthermore, FGNN embodies a new pre-sampling based caching policy (PreSC) that takes both sampling algorithms and GNN datasets into account, and shows an efficient and robust caching performance. Evaluations on three representative GNN models and four real-life graphs show that FGNN outperforms DGL and PyG by up to $9.1\times$ (from $2.2\times$) and $74.3\times$ (from $10.2\times$), respectively. In addition, PreSC achieves $90\% \sim 99\%$ of the optimal cache hit rate in all tests.

1 Introduction

Graph neural networks (GNNs) are an emerging family of neural networks that operate on graph-structured data [53], and have demonstrated convincing performance on many applications in recommendation systems [26, 55], molecule analysis [16], social network mining [30, 51, 58], to name a few. GNNs aim to learn a low-dimensional feature representation (i.e., embedding) for each vertex in a graph, which can be further fed into various downstream graph-related tasks like vertex classification [22, 45] and link prediction [42, 58]. Different from traditional deep learning models, in GNN models, each vertex *recursively* updates its feature by aggregating features from its neighbors in the input graph [18], causing both implementation and performance challenges for existing tensor-oriented deep learning frameworks like TensorFlow [6], PyTorch [32] and MXNet [13]. Many GNN systems [5, 14, 17, 28, 35, 36, 44, 47, 48, 52, 61] have been developed in the recent past to address these challenges.

Many real-life graphs are large-scale and associated with rich vertex attributes (i.e., features), sometimes with highly skewed power-law degree distributions [7, 12, 19]. Under such conditions, it is inefficient, even unfeasible, to *simultaneously* consider all neighbors within L hops for each training vertex when training a GNN model with L layers [33, 61]. A practical solution is *sample-based* GNN training, which adopts various graph sampling algorithms to sample a fixed size of neighbors within L hops for each training vertex. Each training vertex and its corresponding sampled neighbors constitute a sample, and all samples are processed in a *mini-batch* manner [22, 55, 59]. In this way, the computation of each mini-batch can be largely reduced. The whole training process conducts such *iteration* (i.e., epoch) multiple times until the GNN model converges to expected accuracy [17, 57, 60].

Recently, GPUs have been widely exploited to accelerate GNN training [24, 28]. A typical scenario is a single machine equipped with *multiple* GPUs. Since large-scale graphs (including both topological and feature data) exceed limited GPU memory capacity (i.e., 32GB or smaller), most of the existing GNN systems store graph topological and feature data in the main memory of a machine [60, 61]. Given a batch size, CPUs repeatedly conduct graph sampling and extract features of sampled vertices to produce training samples; meanwhile, generated samples are *continuously* copied to GPUs to trigger model training. As also observed in §3, recent work [17, 33] has reported that under such a setting, conducting graph sampling and transferring feature data from host memory to GPU memory dominate the end-to-end training process, leading to GPUs being heavily underutilized.

Two optimizations from separated perspectives have been proposed to alleviate the above two bottlenecks of training GNNs in a single machine multi-GPU setup. In many cases, a major reason for such a large size of GNN datasets is not the graph topology in itself, but features of vertices. Based on this observation, some existing studies [27, 38] transfer graph topology into the GPU memory and apply GPUs to accelerate graph sampling. In addition, prior work [33] found that some vertices are more frequently sampled than others. A static caching strategy is introduced [33] to reduce data movement from the host memory to the GPU memory. Features of high out-degree vertices are cached in the GPU memory in advance, assuming that vertices with high out-degrees are more likely to be frequently sampled.

However, the benefits of these two optimizations cannot be simultaneously achieved in a conventional *time sharing* design, i.e., one GPU performs both graph sampling and model training, since the limited GPU memory capacity cannot cope with both topological data and cached features at the same time (see detailed analysis in §3). In addition, we find existing degree-based caching policy only works well on the k-hop random neighborhood sampling algorithm [22] and graphs with skewed degree distribution, and cannot cover the diversity of sampling algorithms and GNN datasets.

Based on the above analysis, we propose FGNN, a factored system for sample-based GNN training in a single machine multi-GPU setup. FGNN adopts a *space sharing* strategy for multiple GPUs, i.e., one GPU loads either graph topology or cached features in its memory, and only conducts either graph sampling or model training based on its stored data. It eliminates GPU memory contention by leaving more GPU memory for both graph topological data and cached features. In this way, graph sampling and model training can be accelerated at the same time. However, this factored design may suffer from imbalanced loads between GPUs for graph sampling and model training. To solve this problem, FGNN divides the GNN training pipeline into two kinds of executors, namely Sampler and Trainer, and bridges two kinds of executors *asynchronously*. A simple yet effective method is proposed to adaptively determine the appropriate GPU numbers for Samplers and Trainers. FGNN further leverages dynamic switching from Samplers to Trainers to avoid idle waiting on GPUs if needed.

To improve the efficiency of GPU-based caching policies for diverse GNN datasets and sampling algorithms, we propose a general caching scheme. It consists of two parameters, a *hotness metric* that estimates the frequency of a vertex being sampled and a cache ratio that determines how many vertices can be cached in GPUs. Existing caching policies can be represented in this caching scheme naturally. However, the hotness metrics adopted by prior work, e.g., vertex out-degree [33], fail to capture the diversity of sampling algorithms and GNN datasets. To address this issue, we propose a pre-sampling based caching policy PreSC, which is inspired by the observation that the most frequently sampled vertices overlap a lot among different epochs. PreSC tries out a few rounds of sampling phases and uses the average visit count as the vertex hotness metric. We find that PreSC achieves a high cache hit rate even with a small cache ratio and is robust to diverse datasets and sampling algorithms.

We have implemented FGNN that adopts all optimization strategies mentioned above. We evaluated FGNN on three GNN models (i.e., GCN [30], GraphSAGE [22], and PinSAGE [55]) over four real-life graphs, and compared it with the state-of-the-art GNN systems (i.e., DGL [48] and PyG [14]). Experimental results show that FGNN outperforms DGL and PyG by up to $9.1\times$ (from $2.2\times$) and $74.3\times$ (from $10.2\times$), respectively. In addition, PreSC is able to achieve $90\% \sim 99\%$ of the optimal cache hit rate in all tests.

Contributions. We summarize our contributions as follows.

(1) An in-depth analysis of the performance issues and challenges of sample-based GNN systems with the conventional design over GPUs (§3).

(2) A new factored design for sample-based GNN training that eliminates intra-task resource contention and unleashes inter-task data locality (§4), and solutions to tackling the imbalanced load issues introduced by the factored design (§5).

(3) A general GPU-based feature caching scheme, as well as an adaptive policy based on pre-sampling that is robust to diverse sampling algorithms and GNN datasets (§6).

(4) An evaluation with various GNN datasets and models that demonstrates the advantage and efficacy of FGNN (§7).

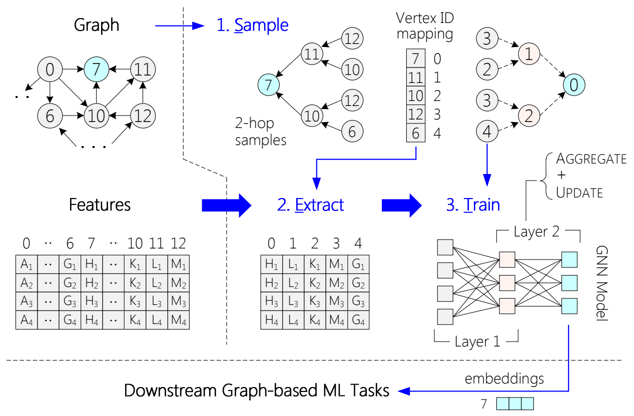

Fig. 1. An example of the SET model for sample-based training in a 2-layer GNN on V_7.

2 GNNs and Sample-based Training

Given a graph $G = (V, E)$, where each vertex is associated with a vector of data as its feature, a GNN learns a low-dimensional embedding for each vertex by stacking multiple GNN layers. For a GNN layer, vertex v updates its feature by aggregating features of its neighbors $N(v)$. A training epoch represents that all training vertices are processed. To train a GNN model, a natural way is *whole-graph training*, i.e., each vertex considers its all neighbors when aggregating features. However, whole-graph training is hard to scale [43]. First, it is increasingly common for GNNs to encounter a large-scale graph with high-dimensional features [17, 33]. Second, many graphs follow a highly skewed degree distribution [12]. A well-connected vertex will aggregate information from a large fraction of the graph with just a few hops (e.g., two), leading to substantial work imbalance.

Sample-based GNN training. Due to the above issues, many emerging GNNs [10, 22, 25, 56] adopt sample-based training. This approach splits the training vertices into multiple mini-batches and conducts GNN training on mini-batches iteratively by following the **SET** model. The model is split into three stages: Sample, Extract, and Train. First, starting from each vertex in a mini-batch, the input graph is sampled according to a user-defined algorithm; the output contains all vertices sampled (also referred to as *samples*).[1] Next, the features of sampled vertices are extracted into an individual buffer. Finally, GNN training is conducted on the samples with their features. Fig. 1 illustrates an example of the SET model (left-right) for training a 2-layer GNN on V_7. The graph sampling uniformly selects two neighbors for each vertex. Note that the sampled vertices are deduplicated (e.g., V_{12}) and reassigned with consecutive IDs (starting from 0).

Prior work has shown that the sample-based approach can achieve almost the same training accuracy but with much less computational cost and scales well for large graphs [11, 34,

[1]There exist various sampling algorithms, such as k-hop random/weighted neighborhood sampling [22, 40, 54]. The sampling probability could be uniform [22] or non-uniform (e.g., in proportion to the edge weight [40]).

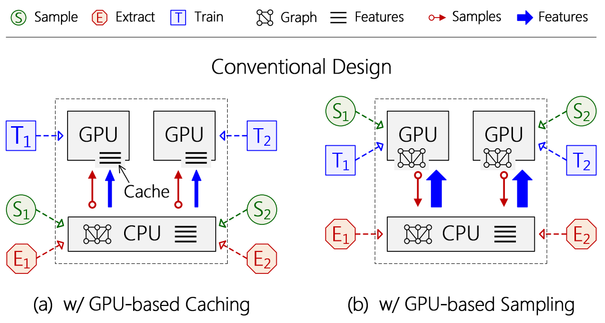

Fig. 2. The conventional design for sample-based GNN training with two optimizations.

43]. Hence, it has been widely adopted by existing GNN systems [1, 5, 17, 33, 61]. Furthermore, since the sample-based approach forms the grid-structured data with fixed-size (i.e., sampled vertices and their features), GPUs are becoming popular in GNN training [9, 14, 17, 33, 35, 36, 52]. Fig. 2(a) shows a conventional design for sample-based GNN training over two GPUs. All graph topological data and features are kept in the host memory. For each mini-batch, graph sampling and feature extracting are performed on CPUs sequentially; then the samples and their features are transferred to GPU memory for model training. Further, GNNs use data parallelism by default to enable multiple GPUs, as they commonly employ simple models with only 2 or 3 layers [22, 30, 45]. Each GPU trains mini-batches independently and exchanges gradients among GPUs synchronously to update model parameters.

However, with the increase of input data—large-scale graphs and high-dimensional features, sampling the graph on CPUs and transferring features to GPU memory become two main performance bottlenecks. Table 1 reports the runtime breakdown of representative GNN systems for training a 3-layer GCN [30] on OGB-Papers [2].[2] After enabling GPU-based training, the Sample and Extract stages dominate the end-to-end GNN training time, accounting for 24% and 54%, respectively, on DGL [1]. It will get worse when using multiple GPUs, becoming 38% and 49% for 8 GPUs. Consequently, this motivates recent research efforts to further improve GNN training in these two aspects.

GPU-based feature caching. The running time of the Extract stage is mainly dominated by loading features from host memory to GPU memory due to the limited PCIe bandwidth (normally less than 16GB/s) [29, 33]. Thus, prior work (e.g., PaGraph [33]) proposed to selectively cache the features associated with frequently sampled vertices in GPU memory (see Cache in Fig. 2(a)). Further, a static caching strategy is adopted to avoid the overhead of dynamic data tracking and swapping. It pre-sorts all vertices by their out-degrees and fills up the GPU cache with the features of the top-ranked vertices. As shown in Table 1, by caching 21% features in

[2]Detailed experimental setup can be found in §7.

Table 1: The runtime breakdown (in seconds) of a training epoch on GNN systems with key optimizations. **GNN**: A 3-layer GCN [30] with random neighborhood sampling. **Dataset**: OGB-Papers [2]. **Testbed**: The server has two 24-core Intel Xeon CPUs and one NVIDIA Tesla V100 GPU with 16GB memory.

GNN Systems	Sample	Extract	Train	Total
DGL [1]	4.91	11.32	4.00	20.78
w/ GPU-base Sampling	1.21	10.87	3.97	16.18
SGNN	2.93	5.55	4.00	12.50
w/ GPU-base Caching [33]	2.88	1.73	4.00	8.62
w/ GPU-base Sampling	0.70	5.46	4.01	10.21
w/ Both	0.70	3.62	4.00	8.37

GPU memory (11.1GB), SGNN[3] reduces 62.8% data transfer during a training epoch (from 25.3GB to 9.4GB), resulting in 1.45× end-to-end time improvement (12.5s vs. 8.62s).

GPU-based graph sampling. Graph sampling also takes a substantial portion of the end-to-end GNN training time [27, 43]. Thus, leveraging GPUs to accelerate graph sampling has appeared in both academic and open-sourced projects, like NEXTDOOR [27] and DGL [4]. As shown in Fig. 2(b), for each mini-batch, graph topological data is first loaded into GPU memory and then sampled on the GPU. Next, the samples are returned to CPUs for extracting the features of sampled vertices to GPU memory. Finally, the GPU trains a GNN model with the samples and their features. Generally, the graph topology is preloaded and kept in the GPU memory if possible [4]. For example, the end-to-end speedup by using one GPU for graph sampling reaches 1.28× in DGL and 1.22× in SGNN, as shown in Table 1.

3 Analysis of Sample-based GNN Training

Although the aforementioned two optimizations can individually improve the performance of GNN systems, the benefits of them cannot be achieved in one system of the conventional design, resulting in suboptimal performance. As shown in Table 1, SGNN just achieves 1.49× performance speedup by enabling both of two optimizations.[4] Therefore, we present an in-depth analysis of sample-based GNN training to reveal fundamental performance issues and pinpoint main challenges to unleash the full power of optimizations.

Capacity. The conventional design follows *time sharing* to perform a sequence of Sample, Extract and Train stages on one GPU, as shown horizontally in Fig. 3. We observe that different GNN stages operate on different input data (namely graph topology for sampling and features for extracting), and

[3]Since DGL does not support GPU-based feature caching, we implemented SGNN, a state-of-the-art GNN system based on the conventional design in Fig. 2(b), which extends DGL with a static GPU cache [33] and a fast GPU-based sampler from scratch.

[4]Prior work [27, 33] on these two optimizations did not consider each other at all. In addition, existing GNN systems also support at most one of them, like DGL [1] and PaGraph.

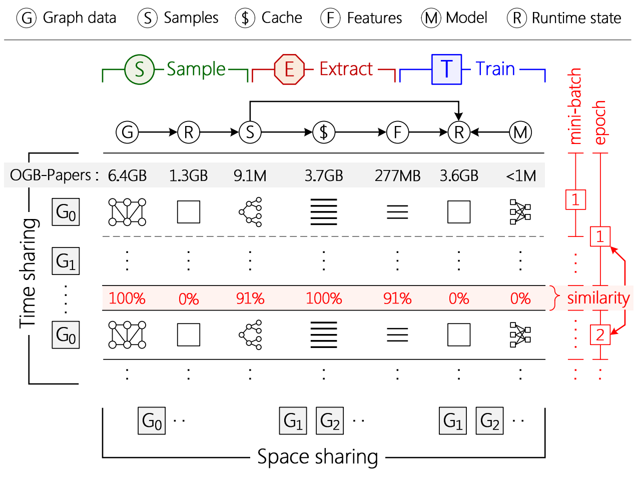

Fig. 3. A breakdown of memory usage and data similarity for different stages of the SET model when training OGB-Papers over multiple GPUs (G_0, G_1, ...) with 16GB of memory each.

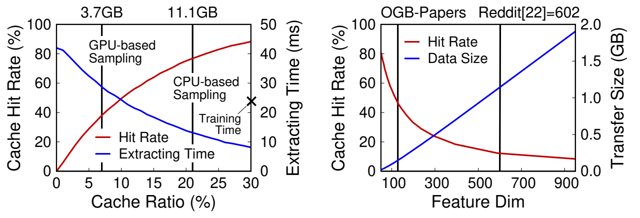

Fig. 4. (a) The cache hit rate and the extracting time for OGB-Papers with the increase of cache ratio. (b) The cache hit rate and the size of transferred data with the increase of feature dimensions.

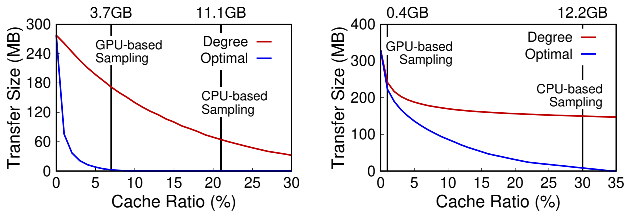

Fig. 5. The size of transferred data of degree-based and optimal caching policies with the increase of cache ratio for (a) OGB-Papers with uniform sampling and (b) Twitter with weighted sampling.

only share a small amount of data (like the samples) within a mini-batch, which means extremely poor *intra-task* data locality. Unfortunately, it is common that the memory of a single GPU (e.g., 16GB) cannot store all input data—namely graph topology and vertex features, especially for large-scale graphs with high-dimensional features. Thus, although GPU memory is usually able to keep all graph topology, it will greatly limit the available memory capacity for feature cache. As an example, the OGB-Papers [2] dataset includes 6.4GB graph topological data and 53GB features. When training a GNN model on a GPU with16GB memory, enabling GPU-based sampling decreases the cache ratio of features from 21% to 7% (11.1GB to 3.7GB), due to keeping graph topology in GPU memory as well as the runtime cost of graph sampling and model training. Note that graph sampling and training a mini-batch also consume considerable GPU memory at runtime (e.g., about 1.3GB and 3.6GB for DGL).

Due to limited cache size, the improvement of the extracting time by caching features becomes trivial as it is positively correlated with the cache ratio of features in GPU memory, as shown in Fig. 4(a). As a result of degraded cache ratio, the extracting time significantly increases 2.19× (from 13.1ms to 28.7ms for a mini-batch), even surpassing the training time (24.4ms), since the cache hit rate drops from 76.8% to 38.0%. More importantly, it will get worse when the topological data and feature dimensions increase—this aligns well with the trends in GNN workloads [17, 22]. Fig. 4(b) shows that the cache hit rate of a 5GB cache rapidly drops to 15% when increasing the feature dimension of OGB-Papers to 600; it will take about 126ms to load more than 1GB features to GPUs.

The first challenge is *how to eliminate contention on GPU memory between different stages of the SET model.*

Efficiency. The efficacy of a static cache depends on not only the memory capacity available for feature cache, but also the policy of how to select features to be cached. To the best of our knowledge, the only caching policy for sample-based GNN training on GPUs is based on the out-degrees of vertices, which selects the features of high out-degree vertices to fill up the cache [33]. It assumes that the input graph has a highly skewed out-degree distribution (e.g., power-law). Meanwhile, the sampling algorithm should select neighbors uniformly. Under such a condition, the vertex with a higher out-degree has a higher probability of being sampled.

However, as shown in Fig. 5, a significant gap exists in the size of transferred data between the existing (degree-based) policy [33] and the optimal results[5] for diverse GNN datasets and sampling algorithms, especially at a small cache ratio of features (e.g., less than 10%). The reasons are two-fold.

First, the out-degree distributions of many GNN datasets are not highly skewed, such as citation networks (e.g., OGB-Papers [2]) and web graphs (e.g., UK-2006 [8]), which violates the narrow assumption of graph structures in prior work [33]. Further, the sampling algorithm is only conducted on the training set—usually a small fraction of vertices (e.g., 1%)—and just aggregates their 3-hop neighborhoods [22, 30, 45]. However, the existing policy takes all vertices of a graph into consideration. For example, in Fig. 5(a), on a non-power-law graph OGB-Papers whose training vertices only account for 1.1% of total vertices, compared with the optimal policy, the degree-based policy needs to transfer 69.0× data to GPUs when the cache ratio is 7%.

Second, the caching policy should also take access patterns of the sampling algorithm into account. The algorithm for weighted graph might change the probability of picking a neighbor much [40, 45, 55], which is overlooked by

[5] Given a cache ratio, to obtain the optimal cache performance (transferred data size/cache hit rate), all sample footprints are recorded. After training, we calculate the corresponding metric if we cache the most visited vertices.

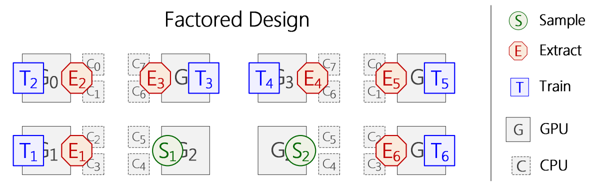

Fig. 6. An example of the factored design for sample-based GNN training over 8 GPUs and two 8-core CPUs.

the existing policy. As shown in Fig. 5(b), even if the Twitter [31] dataset has a highly skewed out-degree distribution, the amount of transferred data to GPUs is still far from optimal when using a 3-hop weighted neighborhood sampling. The weight of a vertex represents the year registered, and the sampling algorithm prefers to select the newer neighbors.

The second challenge is *how to achieve optimal cache efficiency for diverse GNN datasets and sampling algorithms*.

Discussion. Recently another architecture that adopts *batch-mode* for GNN training has been proposed in AGL [57]. Specially, at the beginning of one epoch, all GPUs load graph topology into memory and conduct graph sampling. After that, all GPUs swap graph topology out, and load feature cache for effective feature extraction and model training. We find this design unsuitable for our setting, as graph topology is swapped between host and GPUs back and forth at the beginning/end of each epoch. As we can see in §7.6, it may take a few seconds to load graph topology and large feature cache, while during the same time interval, tens of epochs can be finished. Thus we omit this design in this work.

4 Approach and Overview

Opportunity: inter-task locality. Our work is motivated by an attractive observation that different training epochs in the same stage share a large amount or even all of the data, which means that sample-based GNN training has extremely good *inter-task* data locality. As shown in Fig. 3, graph topology and feature cache, occupying more than 64% of the total 16GB GPU memory, are fully shared by the Sample and Extract stages in different epochs, respectively. This means that leveraging *space sharing* in the stage level, as shown vertically in Fig. 3, can significantly reduce the cost of data transfer, which is the major obstacle to optimizing sample-based GNN training over GPUs.

Our approach: a factored design. Inspired by the *factored* operating system (fos) [50], the key idea behind FGNN is to perform each stage of the SET model (e.g., Sample) on dedicated processors (GPUs and/or CPUs) for different mini-batches. FGNN is a new factored system for sample-based GNN training over GPUs, in which space sharing replaces time sharing to improve performance significantly. Fig. 6 illustrates a brief example of FGNN that conducts two samplers, six extractors and six trainers on a machine with 8

```
class KHOP (...):  # K-hop neighborhood sampling
  def __init__(graph, n_hops=3, sizes=[...], ...):
    ...
  def sample(self, minibatch, ...):
    nbrs = samples.push(minibatch)
    for i in range(n_hops)
      # select neighbors for each vertex
      nbrs = uniform_select(nbrs, sizes[i])
      samples.push(nbrs)
    return samples
class GCN (...):   # graph convolutional network
  def __init__(n_layers=3, ...):
    for i in range(n_layers)
      layers[i] = GraphConv(...)  # set NN layers
    ...
  def forward(self, samples, in_feats, ...):
    out_feats = in_feats
    for i in range(n_layers)
      out_feats = layers[i](samples.pop(), out_feats)
    return out_feats
```

Fig. 7. Example of GCN model and k-hop sampling in FGNN.

GPUs and two 8-core CPUs.

The factored design can naturally eliminate resource contention on GPU memory between different GNN stages—namely the first challenge in §3. Specifically, FGNN keeps graph topology and cached features in the memory of different GPUs. It brings two advantages by leaving more GPU memory space for both topological data and feature cache. First, FGNN can sample larger graph data using a single GPU without additional data loading. Meanwhile, FGNN can significantly reduce the extracting time by caching more features in the GPU memory (see Fig. 4(a)).

Challenge: load imbalance. The aforementioned optimizations aim to unleash the power of GPUs by shifting workloads from CPUs to GPUs. CPUs are thereby no longer the main performance bottleneck, even facing more GPUs (e.g., 8 GPUs in our testbed). However, the factored design may suffer from load imbalance across GPUs due to the coarse-grained stage-level workload partitioning (space sharing). Specifically, both the Sample and Train stages of the SET model perform on dedicated GPUs, while the execution time might be significantly different, even up to 10×, as the datasets (graph topology and feature dimensions) and workloads (sampling algorithms and GNN models) are diverse. Therefore, FGNN needs to flexibly assign GPUs to different stages and make them work together efficiently. In some cases, FGNN should further dynamically change the stage to avoid idle on GPUs—for instance, running two stages with a 10× difference in execution time on a host with two GPUs.

5 FGNN Architecture

FGNN is a new factored GNN system that performs different stages of the SET model on different dedicated processors (GPUs and CPUs). Hence FGNN can support two key optimizations gracefully—namely sampling graph data and caching features over GPUs—and gain the benefits of both by eliminating contention on the GPU memory. In this sec-

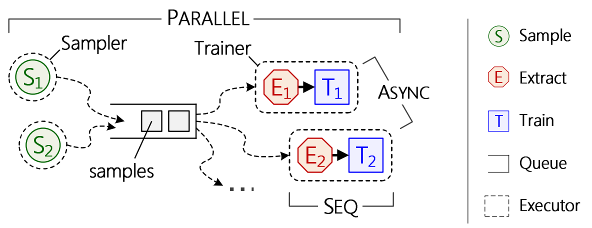

Fig. 8. The execution flow of FGNN.

```
class Sampler(...):
  def run(self, dev, q, graph, ...):
    load(dev, graph)  # load graph to GPU memory
    khop3 = KHOP(graph, ...)  # define 3-hop sampling
    ...
    while (minibatch = get_minibatch())
      samples = khop3(minibatch)
      remap(dedup(samples))
      q.enque(samples)  # (async) send task
class Trainer(...):
  def run(self, dev, q, features, ...):
    model = GCN(...)  # define a 3-layer GCN
    loss_func = ...   # define a loss function
    ...
    while (samples = q.deque())  # (async) recv task
      in_feats = extract(features, samples)
      loss = loss_func(model(samples, in_feats), ...)
      loss.backward()
```

Fig. 9. Two kinds of executors in FGNN.

tion, we first describe the programming model in FGNN and then introduce key designs to tackle load imbalance across GPUs, such as hybrid execution and flexible scheduling.

5.1 Programming Model

FGNN provides a simple data-parallel programming model for various sampling algorithms and GNN models, similar to existing GNN systems (e.g., DGL [1]). Fig. 7 outlines the implementation of the GCN model and 3-hop random neighborhood sampling in FGNN. For sampling algorithms (see *class* KHOP), FGNN's API performs a user-provided function on each mini-batch and returns a set of sampled vertices (i.e., *samples*). The API can capture many different sampling schemes such as k-hop random/weighted neighborhood [22] and random walk [20]. For GNN models (see *class* GCN), FGNN's API defines a model by stacking multiple GNN layers.

5.2 Hybrid Execution

To adapt to diverse workloads, FGNN flexibly assigns GPUs to different stages and runs them in *parallel*. We observe that the Sample and Extract stages only share a small amount of data (i.e., samples). Thus FGNN divides the SET model into two kinds of individual executors, named Sampler and Trainer respectively, as shown in Fig. 8. FGNN uses a global queue in the host memory to link two kinds of executors asynchronously, which is flexible in supporting different numbers of executors. The concurrent queue would not be the bottleneck since the updates are infrequent. Fig. 9 outlines the implementation of executors in FGNN.

FGNN binds each Sampler to a GPU, and it will load graph topology into the GPU memory. The Sampler iteratively generates the samples for each mini-batch following a certain graph sampling scheme (e.g., k-hop random neighborhood sampling). To accelerate the pace of feature extraction, the sampled vertices will be deduplicated and reassigned with consecutive IDs (starting from 0). Finally, the samples will be sent to the Trainer *asynchronously* via a global queue. For multiple Samplers, a global scheduler assigns tasks (i.e., mini-batches) dynamically across them in order to achieve load balance without synchronization [41]. For larger graphs that cannot fit in GPU memory, a simple approach is to divide the whole graph into multiple partitions and iteratively load a partition to the GPU memory for graph sampling [27]. We leave this as future work.

On the other hand, FGNN binds each Trainer to a GPU and several CPU cores. The Trainer *sequentially* executes the Extract and Train stages for each mini-batch. After receiving samples of a mini-batch, the Trainer will simultaneously extract their features from host memory and the cache in GPU memory (if any). Note that FGNN adopts a static caching scheme, so each sampled vertex can be marked in the Sample stage whether its feature is cached in GPU memory or not (see §6 for more details). The Trainer then runs a forward pass that computes the output based on a certain GNN model (e.g., GCN), followed by a backward pass that uses a loss function to compute parameter updates. Moreover, FGNN employs a simple pipelining mechanism in the Trainer to overlap the Extract and Train stages. Note that existing GNN systems (e.g., DGL [1]) already leverage the pipelining mechanism during model training, which is also enabled within the Train stage of FGNN. For multiple Trainers, they do not interact with each other except by exchanging locally produced gradients to update GNN model parameters. To support pipelining, FGNN updates gradients asynchronously with bounded staleness, similar to prior work [17, 36, 37, 39, 44], which effectively mitigates the convergence problem.

5.3 Flexible Scheduling

FGNN can assign multiple GPUs to different executors on demand for load balance—all GPUs are fully utilized. However, it is not obvious to set the number of different executors, i.e., Samplers and Trainers, for a given GNN workload. Fortunately, we observe that the performance of executors in FGNN is quite stable for sample-based GNN training. For the Sampler, the input (i.e., mini-batches) and the output (i.e., samples) of sampling algorithms are commonly regular and highly similar. For the Trainer, the runtime is dominated by the Train stage since the extracting time is trivial due to using GPU-based cache and is easy to be hidden by the training time with pipelining. The inputs of the Train stage become regular after graph sampling, and the GNN computations on the GPU have negligible performance variability [21]. Finally, both kinds of executors are parallel in the mini-batch level without synchronization.

Given a GNN workload, we assume that the processing time of Sampler (T_s) and Trainer (T_t) for a mini-batch can be estimated by training an epoch in advance. Then, the number of GPUs allocated to Samplers (N_s) is calculated as

$$\frac{T_t}{T_s} = K \quad and \quad \left\lceil \frac{N_g}{K+1} \right\rceil = N_s,$$

where K is the ratio of the training time to the sampling time, and N_g is the total number of available GPUs. FGNN prefers to allocate GPUs to Samplers, because temporarily switching from Samplers to Trainers can be quickly fulfilled, but not vice versa. Specifically, the Sampler may have to take a few seconds, which can sample hundreds of mini-batches, to load the whole graph topology into GPU memory before execution (see §7.6). On the contrary, the Trainer can immediately run on the GPU, and the size of feature cache in GPU memory only affects the performance. Therefore, N_s is set to the ceiling of the fraction of GPUs allocated, and the rest of the GPUs are allocated to Trainers ($N_t = N_g - N_s$).

Dynamic executor switching. Our approach can always choose the optimal GPU allocation scheme under the given condition. However, in some severe cases, static scheduling still leaves large room for improvement. For example, FGNN might encounter highly skewed workloads and a machine with limited GPUs (such as two or even one). Then, no matter which executor the GPU is allocated to, the GPU will be idle for a long time, even forever.

To remedy this, we propose dynamic executor switching. As aforementioned, FGNN only needs to consider switching from Samplers to Trainers, and the switching is temporary. FGNN switches Samplers to Trainers after sampling all mini-batches of the current epoch, and then switches back at the end of the current epoch. Thus, the graph topology should always be kept in GPUs, even though it would limit the size of feature cache for Trainers. To enable fast switching, FGNN launches a standby Trainer on each GPU allocated to the Sampler, so that the Trainer can immediately replace the Sampler and fetch tasks (samples) from the global queue.

To determine whether to switch, FGNN will check the number of remaining tasks in the global queue (M_r) and calculate a profit metric

$$\mathcal{P} = \begin{cases} \frac{M_r \times T_t}{N_t} - T_{t'} & \text{if } N_t > 0 \\ +\infty & \text{if } N_t = 0 \end{cases},$$

where $T_{t'}$ is the processing time of the standby Trainer (with limited feature cache) for a mini-batch. The profit metric ($\mathcal{P}$) measures whether the standby Trainer can complete one task before existing (normal) Trainers complete all remaining tasks. If the profit is greater than zero (i.e., $\mathcal{P}$>0), FGNN will wake the standby Trainer to process training task. If there is no Trainer (i.e., N_t=0), the profit is infinity, and FGNN should always switch Samplers to Trainers. Note that each standby Trainer should calculate the profit metric separately before each time it fetches a task from the global queue.

Although FGNN adopts a space sharing strategy for sample-based GNN training over multiple GPUs, it can still run on a single GPU efficiently. This could be seen as a special case of dynamic switching, where the solo GPU is used by alternating between graph sampling (Sampler) and model training (Trainer), switching once an epoch. Storing all samples of an epoch in the global queue located at host memory is affordable, e.g., from 200MB to 1.4GB in our experiments.

6 GPU-based Feature Caching

As discussed in §3, the existing degree-based caching policy only works well under certain assumptions. Thus a caching policy that is efficient and robust to diverse GNN datasets and sampling algorithms is highly favored.

6.1 A General Caching Scheme

We start with a general GPU-based caching scheme. The scheme has two parameters, a hotness metric h_v and a cache ratio α, which are defined as follows.

Hotness metric. A *hotness metric* h_v aims to estimate the frequency that a vertex v is sampled in the graph sampling stages. Intuitively, we prefer to cache vertices with larger h_v values to improve the cache hit rate and reduce the data movement cost. Different caching policies use different hotness metrics, e.g., PaGraph [33] utilizes vertex out-degree as the hotness metric for power-law graphs.

Cache ratio. The *cache ratio* α determines how many vertices can be cached in GPUs. Observe that a larger α usually implies a higher cache hit rate. However, due to the limited GPU memory, it is unfeasible to cache all vertex features. We also need to reserve enough memory for GNN model training. In general, the value of α for a given training task can be determined by two factors, the available GPU memory amount for caching and the vertex feature dimension. To determine GPU memory capacity allocated for caching, we adopt the method proposed in PaGraph [33], where we simulate one-time model training for a mini-batch and record the peak memory usage for model training. Then the rest of the available GPU memory is allocated for feature cache.

Following this general scheme, FGNN provides a built-in procedure `load_cache(hotness_map,`α`)` to enable GPU-based feature cache. Here `hotness_map` is a data structure that stores the *hotness* value of each vertex, and α is the cache ratio which can either be specified by users manually or determined as we have discussed above. The procedure identifies and loads the features of the top-ranked $\alpha|V|$ vertices w.r.t. h_v into the GPU memory. It also builds a hash table to indicate the location in feature cache of a given vertex. We can easily implement existing caching policies in FGNN with this caching scheme. For example, to implement

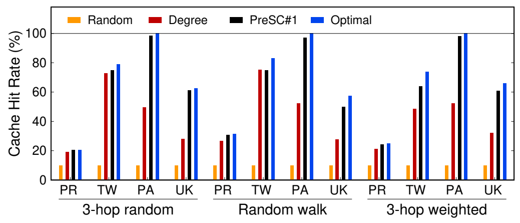

Fig. 10. The comparison of the cache hit rate for various workloads by using different caching policies. The cache ratio of features is set to 10%.

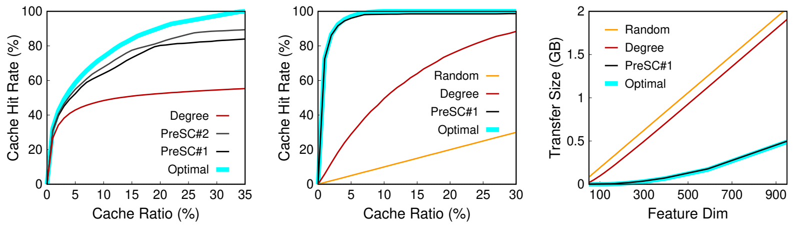

Fig. 11. The comparison between different caching policies for (a) Twitter with weighted sampling, (b) OGB-Papers with 3-hop neighborhood, and (c) OGB-Papers with the increase of feature dimensions. PreSC#K conducts K sampling stages.

Table 2: The similarity (in percentage) of access footprint between two epochs for various datasets and sampling algorithms.

Sampling algorithms	**PR** [3]	**TW** [31]	**PA** [2]	**UK** [8]
3-hop random	73.97	78.89	91.29	77.46
Random walk	78.16	72.68	87.14	64.40
3-hop weighted	77.69	66.64	89.57	72.96

the degree-based caching policy adopted by PaGraph [33] in FGNN, it suffices to compute the out-degree of each vertex v as h_v and construct the data structure `hotness_map`.

6.2 Analysis of Caching Policies

Most GNN models shuffle the training set T at the beginning of each epoch and divide T into multiple mini-batches. Thus, it is hard to predict the vertices sampled in each mini-batch. Nevertheless, observe that all the sampling operations of one epoch are started from the same T. That is, the vertices sampled in one epoch are the results of a stochastic process defined by the sampling algorithm A, the input graph G and training set T. Let $\hat{h}_v$ be the number of times of vertex v being sampled in *one* epoch. The expectation $\mathbb{E}[\hat{h}_v]$ of $\hat{h}_v$ can be served as an ideal hotness metric for each vertex v in *all* epochs. Note that sampling operations of different epochs are *independent*, and thus we can use $\mathbb{E}[\hat{h}_v]$ of one epoch to reflect the expected visit frequency in all epochs for vertex v.

It is theoretically possible to compute $\mathbb{E}[\hat{h}_v]$ for each vertex v, but this would incur high pre-processing overheads. Observe that we are only interested in the vertices with top-ranked $\mathbb{E}[\hat{h}_v]$'s. Thus, it suffices to approximate $\mathbb{E}[\hat{h}_v]$. For example, we can try out a few epochs with the Sample stage alone and use the average visit count h_v of each vertex as an approximation of $\mathbb{E}[\hat{h}_v]$. We find that the vertices with top-ranked $\mathbb{E}[\hat{h}_v]$ and top-ranked h_v overlap with a high probability. To see this, we conducted 100 iterations of graph sampling for all training vertices on three sampling algorithms and three real-life graphs, and compared the similarity of frequency among the top 10% frequently sampled vertices between each pair of adjacent epochs. We define the similarity of i-th epoch to j-th epoch by $\sum_{v\in T_i\cap T_j} \min(f_i(v), f_j(v)) / \sum_{v\in T_j} f_j(v)$, where T_i and T_j are the sets of top 10% accessed vertices in epochs i and j, and $f_i(v)$ and $f_j(v)$ are the sampled frequency of v in epochs i and j. As shown in Table 2, for the top-ranked vertices, on average over 75% of the access footprint overlaps between two iterations. This indicates that it is feasible to pre-sample a few rounds to estimate vertex hotness.

6.3 A Pre-sampling Based Caching Policy

Based on above theoretical analysis, we propose a *pre-sampling based* feature caching policy PreSC. More specifically, given a dataset G, a sampling algorithm A and a training set T, PreSC conducts K sampling stages, starting from the vertices in T. Here K is a user-defined parameter. It records the visit count of the sampled vertices and uses the average count as the hotness metric h_v. We use PreSC#K to denote the variant of PreSC that conducts K sampling stages.

We find that a small number of sampling stages, i.e., $K \le 2$, already produce a decent hotness estimation and suffice for most training tasks (see Fig. 11(a)). Thus it is feasible to compute the h_v's of PreSC *online*, since (i) GPU-based graph sampling is lightweight, e.g., on average it only takes $1.4\times$ time of one epoch (see §7.6), and (ii) a typical GNN training pipeline usually has over 100 epochs. Specifically, we run the first K epochs of an end-to-end training pipeline for pre-sampling without features cache and determine which vertices should be cached. Then features of selected vertices are loaded into GPUs, and the rest of epochs can benefit from reduced data movement to GPUs. This pre-sampling process can also be dealt with in an *offline* manner.

The benefits of the PreSC caching policy are two-fold.

Efficiency. PreSC is very efficient in terms of cache hit rate. This is because the ideal hotness estimation metric $\mathbb{E}[\hat{h}_v]$ captures the sampled frequency of a vertex in all epochs, and the hotness metric h_v of PreSC provides a good approximation of $\mathbb{E}[\hat{h}_v]$. As shown in Fig. 10, fixing cache ratio $\alpha = 10\%$, the cache hit rate of PreSC is almost as good as the Optimal caching policy, and is on average $1.5\times$ (up to $2.2\times$) higher than that of the Degree policy. Recall that the Optimal policy defines an upper bound of cache hit rate for an fixed cache ratio, since it assumes that we can cache the actual most frequently sampled vertices in all epochs in advance.

Robustness. PreSC is robust to diverse datasets and sampling algorithms. As shown in Fig. 10, on four GNN datasets and three sampling algorithms, PreSC constantly beats other

Table 3: Datasets used in evaluation. Volume$_G$ (resp. Volume$_F$) is the data volume of graph topology (resp. features) in host memory.

Dataset	#Vertex	#Edge	Dim.	Volume$_G$	Volume$_F$
PR [3]	2.4 M	124 M	100	481 MB	934 MB
TW [31]	41.7 M	1.5 B	256	5.6 GB	40 GB
PA [2]	111 M	1.6 B	128	6.4 GB	53 GB
UK [8]	77.7 M	3.0 B	256	11.3 GB	74 GB

baselines, including the Random policy and Degree policy adopted by PaGraph [33]. This is because, as oppose to prior work, the hotness metric h_v of PreSC is computed by taking both the GNN dataset G and sampling algorithm A into account. Observe that the performance of Degree policy is unstable. For example, for the 3-hop random neighborhood and random walk sampling, the Degree policy has a similar cache hit rate as PreSC on the power-law graph TW [31]. However, if we either use a non-power-law graph, e.g., PA [2], or use the weighted sampling, the cache hit rate of Degree drops very quickly, i.e., on average below 51%. Instead, the performance of PreSC is stable and very close to Optimal in all 12 cases. These verify the robustness of PreSC.

The high efficiency and robustness of PreSC bring substantial advantages in processing large-scale graphs and high-dimensional features. To see this, in Fig. 11(b) we first plot the cache hit rate to the cache ratio α on dataset OGB-Papers with 3-hop random neighborhood sampling. The cache hit rate of PreSC increases fast and reaches 96% when $\alpha = 5\%$. This verifies the effectiveness of PreSC to process large-scale graphs, i.e., PreSC is able to achieve a decent cache hit rate even with a very small α. In contrast, the cache hit rates of Random and Degree policies are below 5% and 29% when $\alpha = 5\%$, respectively. They increase much slower than PreSC. Observe that the hotness metric of PreSC takes the training set T into account, while the Random policy and Degree policy overlook the impact of T. For example, the Degree policy uses the static vertex out-degree as the hotness metric, which essentially assumes that the sampling operations are started from all vertices in the dataset. This largely reduces the cache utilization since some high-degree vertices may never be sampled from a given T. Fixing 5GB cache size, Fig. 11(c) further shows the size of data moved from the host memory to the GPU memory in one epoch as the feature dimension increases. We can see that as the dimension increases from 100 to 900, the transferred data size of PreSC increases much slower than Random and Degree. The transferred data size of PreSC is less than 500MB when the dimension is 900. Instead, Degree and Random need to move nearly 2GB data, which is 4× of that of PreSC.

7 Evaluation

We implemented FGNN starting from scratch, with about 15,200 lines of C++ and CUDA codes. The build-in graph sampling algorithms include k-hop random/weighted neighborhood sampling and random walk. In addition, FGNN utilizes DGL as the GNN execution runtime (the Train stage).

Table 4: The runtime (in seconds) of one epoch on different GNN systems. (nS) indicates n GPUs allocated to Samplers. "×" indicates that the target GNN model is not supported.

GNN Model	Dataset	DGL	PyG	SGNN	FGNN
GCN	**PR**	1.33	11.91	0.22	0.33 (3S)
	TW	3.86	12.15	1.80	0.47 (2S)
	PA	4.56	14.82	2.46	0.84 (2S)
	UK	OOM	15.04	OOM	1.47 (2S)
GraphSAGE	**PR**	0.79	8.17	0.07	0.11 (4S)
	TW	1.81	8.18	0.35	0.20 (2S)
	PA	2.47	9.68	0.85	0.30 (2S)
	UK	OOM	9.86	2.01	0.61 (1S)
PinSAGE	**PR**	0.86	×	0.30	0.40 (1S)
	TW	2.38	×	0.98	0.58 (1S)
	PA	2.79	×	1.65	1.05 (1S)
	UK	OOM	×	OOM	1.81 (1S)

7.1 Experimental Setup

Environments. The experiments were conducted on a GPU server that consists of two Intel Xeon Platinum 8163 CPUs (total 2∗24 cores), 512GB RAM, and eight NVIDIA Tesla V100 (16GB memory, SXM2) GPUs. The software environment of the server was configured with Python v3.8, PyTorch v1.7, CUDA v10.1, DGL v0.7.1, and PyG v1.7.0.

GNNs. We used three representative GNN models, namely GCN [30], GraphSAGE [22], and PinSAGE [55]. GCN (resp. GraphSAGE) adopts 3-hop (resp. 2-hop) random neighborhood sampling [60]. Starting from a training vertex, the numbers of sampled neighbors for different layers of GCN (resp. GraphSAGE) were 15, 10, and 5 (resp. 10 and 25). PinSAGE has 3 layers and in each layer it adopts random walk sampling to select 5 neighbors from 4 paths of length of 3. We determine the above settings (e.g., size of sampled neighbors and sampling algorithms) by comprehensively referring to the reported details in original papers [22, 30, 55] together with official examples and guidelines of DGL. The dimension of the hidden layers of all models is set as 256, and the batch size is 8000, the same as prior work [28, 60].

Datasets. We used 4 datasets as listed in Table 3 for evaluation, including a social graph Twitter [31] (TW), a web graph UK-2006 [8] (UK) and two GNN datasets from Open Graph Benchmark [23]—a co-purchasing network OGB-Products (PR), and a citation network OGB-Papers (PA). Similar to prior work [17], we generated random features and labels for TW and UK, as they originally have no features and labels.

Baselines. We compared FGNN with DGL [48], PyG [14] and SGNN[6]. GPU-based sampling is enabled in DGL to accelerate graph sampling, while PyG conducts graph sam-

[6]Since PaGraph is built with DGL v0.4.1 which only supports sampling by CPUs, SGNN has clearly outperformed PaGraph and also adopted degree-based caching. Therefore, we do not consider PaGraph in our experiments.

Table 5: The runtime breakdown (in seconds) of one epoch for DGL and FGNN. S, M, and C represent graph sampling, marking cached vertices, and copying samples to the host memory in the Sample stage, respectively. GSG and PSG are short for GraphSAGE and PinSAGE.

GNN	Data set	DGL			PyG			FGNN		
		Sample	Extract	Train	Sample	Extract	Train	Sample = S + M + C	Extract (Ratio, Hit%)	Train
GCN	PR	0.35	2.81	1.22	7.15	3.19	2.14	0.39 = 0.29 + 0.01 + 0.09	0.19 (100%,100%)	1.18
	TW	0.74	9.44	1.48	6.25	9.52	2.51	0.37 = 0.26 + 0.03 + 0.08	0.80 (25%, 89%)	1.50
	PA	1.20	10.70	4.00	9.08	10.27	5.91	0.96 = 0.68 + 0.10 + 0.18	0.61 (21%, 99%)	3.81
	UK	OOM	OOM	OOM	7.19	16.69	4.83	0.56 = 0.38 + 0.03 + 0.14	3.08 (14%, 70%)	3.12
GSG	PR	0.13	1.92	0.23	3.89	2.06	0.23	0.20 = 0.15 + 0.01 + 0.04	0.10 (100%,100%)	0.24
	TW	0.38	4.65	0.44	3.38	4.70	0.34	0.16 = 0.11 + 0.01 + 0.04	0.44 (32%, 89%)	0.42
	PA	0.56	6.06	1.25	4.69	6.36	0.88	0.46 = 0.32 + 0.06 + 0.08	0.34 (25%, 99%)	1.12
	UK	OOM	OOM	OOM	4.01	8.45	0.84	0.27 = 0.18 + 0.02 + 0.06	1.44 (18%, 72%)	1.02
PSG	PR	0.16	1.56	1.75	×	×	×	0.20 = 0.15 + 0.01 + 0.04	0.10 (100%,100%)	1.72
	TW	0.23	4.97	2.57	×	×	×	0.28 = 0.22 + 0.02 + 0.05	0.55 (26%, 86%)	2.54
	PA	0.53	5.00	6.14	×	×	×	0.61 = 0.47 + 0.04 + 0.09	0.41 (22%, 97%)	5.97
	UK	OOM	OOM	OOM	×	×	×	0.65 = 0.48 + 0.03 + 0.13	3.39 (13%, 57%)	6.99

pling by CPUs. SGNN is built upon the same codebase of FGNN and supports both GPU-based graph sampling and degree-based caching adopted by PaGraph [33]. Both DGL and SGNN follow the time sharing design, i.e., each GPU conducts both graph sampling and model training. All results were computed by calculating the averages over 10 epochs.

7.2 Overall Performance

We first compared FGNN with its competitors. Table 4 reports the end-to-end training time of one epoch for each system. The number of GPUs allocated to Samplers n is determined by the method in §5.3. Note that with 8 GPUs, our flexible scheduling scheme already gave optimal GPU allocations for Samplers. Thus dynamic switching did not happen in this set of evaluations. We find the following.

(1) Overall, FGNN outperforms DGL and PyG by up to $9.1\times$ (from $2.2\times$) and $74.3\times$ (from $10.2\times$), respectively. For systems using GPUs for graph sampling, only FGNN can process dataset UK in all cases while others report OOM, due to the memory contention caused by the time sharing strategy. PyG performs the worst due to the high cost of sampling on CPUs and transferring features to GPUs (see §2). In general, the performance gain of FGNN over its competitors is from three aspects: (A1) the space sharing design that unleashes the power of both GPU-based sampling and caching, (A2) the robust PreSC caching policy, and (A3) a more efficient implementation of GPU-based graph sampling.

(2) Compared with SGNN, FGNN benefits from (A1) and (A2). Indeed, SGNN suffers from GPU memory contention and its inefficient degree-based caching policy. SGNN needs to load graph topology to all GPUs, leaving limited memory for caching. As shown in Fig. 11, our PreSC is more efficient than the degree-based policy when the cache ratio is small. Note that SGNN performs slightly better than FGNN on PR. This is because all topological and feature data of PR can be loaded into a single GPU. (A1) and (A2) cannot improve that case while our factored design introduces little overheads.

(3) The performance gain of FGNN over DGL comes from (A1)$\sim$(A3). In addition, DGL stores all feature data in the host memory and has no GPU-based caching mechanism. It transfers much more data from the host to GPUs than SGNN and FGNN. Note that DGL only uses CPUs to extract features of sampled vertices, which also incurs a large number of random memory accesses. Therefore, apart from the more severe GPU memory contention problem, the limited memory access bandwidth of CPUs is another major bottleneck.

7.3 Performance Breakdown

We next conducted a stage-level time breakdown analysis for the SET model on FGNN, DGL and PyG. The results are reported in Table 5. We find the following.

(1) For the Sample stage, FGNN and DGL outperform PyG with several orders of magnitudes, as PyG conducts graph sampling on CPUs. On GCN and GraphSAGE, FGNN beats DGL under most cases. We find that DGL adopts the Reservoir algorithm [46] for k-hop random neighborhood sampling on GPUs. The sampling complexity of each vertex is positively correlated with its in-degree, resulting in imbalanced workloads across GPU threads. Instead, FGNN implements a variant of the Fisher–Yates algorithm [15], which is GPU-friendly since each vertex has more balanced workloads. Note that the random walk algorithm is not available in DGL. Thus we make some adaptation efforts so that DGL can run on top of the random walk graph sampling module of FGNN. Therefore, on PinSAGE, FGNN and DGL have quite similar sampling performances. Our factored design also leads to some overheads in the Sample stage, i.e., M and C in Table 5. We can see that the overheads only account for a very small fraction, i.e., below 4% on average.

(2) For DGL and PyG, the Extract stage accounts for a large fraction of end-to-end time. In contrast, FGNN is not bogged down by feature extraction, thanks to our PreSC caching policy. On average, FGNN fetches 88% of required features di-

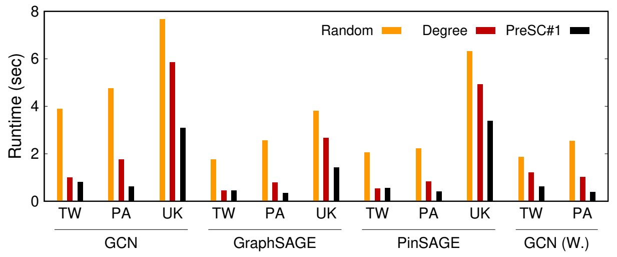

Fig. 12. The comparison of the runtime in the Extract stage. GCN (W.) stands for GCN with 3-hop weighted neighborhood sampling.

rectly from GPU cache. PreSC demonstrates surprising efficiency, e.g., on dataset PA, caching less than 25% of vertices reduces over 96% of data movement to GPUs.

(3) FGNN, DGL and PyG perform comparably in the Train stage as all GNN execution runtimes are highly optimized.

7.4 Impact of Caching Policy

We next explored how caching policies impact extracting time (including extracting features in CPUs/GPUs and transferring data to GPUs). We implemented the degree-based policy Degree [33] and the Random policy (i.e., randomly select and cache vertices) in FGNN, and compared them with PreSC#1. Fig. 12 reports the extracting time of one epoch for 4 GNN models and 3 datasets. We omit the results of PR dataset since all of its features can be cached in GPU memory. As we can see, PreSC#1 constantly beats its competitors. Compared with Degree and Random, PreSC#1 reduces 39% and 73% extracting time on average, up to 87%.

We also evaluated the impact of caching policies on end-to-end time. We find that the improvement of end-to-end time varies from model to model. For GNN models with low model training complexity (e.g., GCN and GraphSAGE), end-to-end time is largely reduced since model training only accounts for a small fraction. While for models with high training complexity (e.g., PinSAGE), the improvement is limited, as model training is even longer than feature extraction without caching.

7.5 Scalability

We next evaluated the scalability of FGNN w.r.t. the number of GPUs. Fixing the number of GPUs for Samplers as $1, 2$ and 3, Fig. 13 reports the time of one epoch of GCN and PinSAGE on dataset PA. We can see that when the number of Trainers increases, the training time of one epoch first decreases linearly, i.e., at this moment the capacity of consuming samples by Trainers is the major bottleneck. When the number of Trainers further increases, the epoch time decreases relatively slower, indicating that Trainers catch up with the sample generating speed. DGL has much larger epoch time than FGNN. In addition, the epoch time of DGL decreases slower than FGNN when the GPU number increases. This is because DGL stores features of all vertices in the host memory. When multiple GPUs are available, multiple workers for extracting and transferring features will contend the limited CPU resources and PCIe bandwidth.

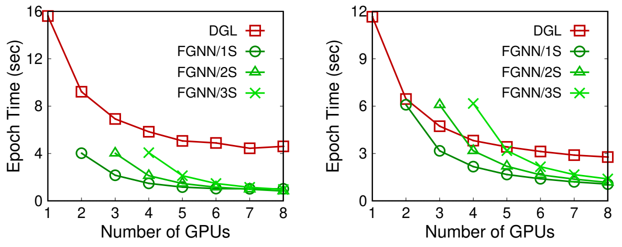

Fig. 13. The scalability in FGNN for training (a) GCN and (b) PinSAGE on dataset PA w.r.t. the number of GPUs. FGNN/kS indicates that FGNN uses k GPUs as Samplers, where $k = 1, 2, 3$.

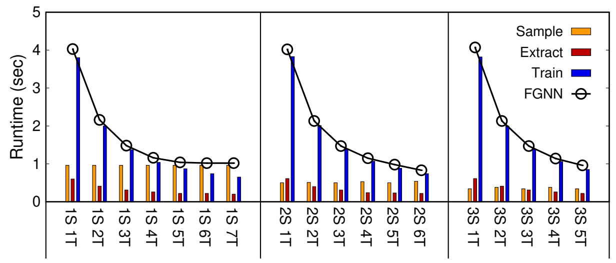

Fig. 14. The breakdown of an epoch in FGNN for GCN on PA w.r.t. the number of GPUs. mS nT indicates m Samplers and n Trainers.

We further plotted the breakdown of one epoch in FGNN for GCN on PA. As shown in Fig. 14, fixing the Sampler number m as 1, 2 and 3, the training time and the end-to-end time drop gracefully when the Trainer number n increases from 1 to 5, 6 and 5, respectively. In the case of one Sampler, the performance stops to improve when n reaches 5, i.e., the Sampler becomes the bottleneck. Instead, with 2 or 3 Samplers, the training time continues to decrease with more Trainers. Note that with the same number of GPUs, the end-to-end time of 2 Samplers drops faster than that of 3 Samplers (e.g., 1.5s for 2S3T vs. 2.1s for 3S2T). This is consistent with the Sampler number estimation as proposed in §5, i.e., 2 Samplers were used for GCN on PA (see Table 4).

Table 6: The preprocessing time (in seconds) for training GCN in FGNN. Here **G**: graph topology, **F**: feature data, $: cache data.

Preprocessing Time	**PR**	**TW**	**PA**	**UK**
Disk to DRAM (G & F)	1.19	30.56	48.62	58.24
DRAM to GPU-mem (G & $)	1.13	11.85	14.33	14.65
Load graph topology	0.27	2.65	3.21	5.32
Load feature cache	0.85	9.15	10.73	9.24
Pre-sampling for PreSC#1	0.42	0.70	1.81	1.14

7.6 Preprocessing Cost

We next evaluated the preprocessing cost of GNN training. Table 6 reports the preprocessing time of GCN on four datasets. The costs for other GNN models are consistent (not shown). Note that the preprocessing time includes (P1) loading graph topology G and feature data F from disk to

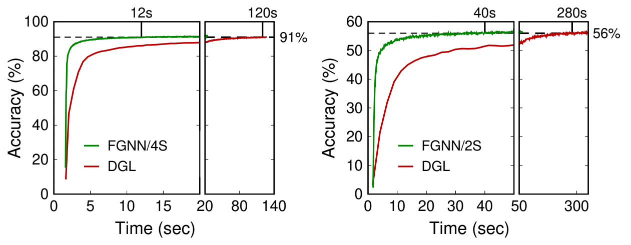

Fig. 15. A comparison of training time for GraphSAGE to the same accuracy target between FGNN and DGL on (a) PR and (b) PA.

DRAM, (P2) loading G and feature cache from DRAM to the GPU memory, and (P3) pre-sampling, i.e., a round of GPU-based sampling and hotness map construction.

The cost of (P1) dominates the overall cost. On average, it accounts for 67.0%. The costs of (P2) and (P3) are smaller. On average (P2) takes $13.9\times$ of one epoch time, $3.5\times$ for loading graph topology and $10.3\times$ for loading cache (see Table 4). (P3) is quite lightweight. It only takes about $1.4\times$ of one epoch time. Note that FGNN only conducts (P2) and (P3) once, and a GNN training task usually takes over 100 epochs which can benefit from both (P2) and (P3) (see §7.3 and 7.4). Therefore, the costs of (P2) and (P3) can be amortized.

7.7 Training Convergence

For a GNN model, we care more about the time of converging to the expected accuracy, instead of the execution time of one epoch. In addition, we need to verify the correctness of our implementation. To this end, given an accuracy target, we run GraphSAGE on PR and PA in both FGNN and DGL. As shown in Fig. 15, FGNN can converge to the same accuracy targets as DGL on both datasets, within fewer epochs (e.g., 100 vs. 120 for PA). Thanks to all the optimizations proposed in this work, FGNN outperforms DGL by $10.0\times$ and $7.0\times$ on PR and PA, respectively, for reaching the same accuracy.

7.8 Adaptive Switching

FGNN leverages dynamic executor switching mechanism to accelerate GNN training, which can temporarily switch Samplers to Trainers when there exists a capacity gap between Samplers and Trainers under some severe cases (see §5.3). In this experiment, we train PinSAGE on PA, and only one GPU is allocated to Sampler. As shown in Fig. 16(a), when the number of Trainers is small (e.g., less than 3), the improvement achieved by enabling dynamic switching is significant. When Trainers are sufficient, the workload becomes balanced, and there is no need to do dynamic switching. The results confirm the efficacy of dynamic switching.

7.9 Performance on a Single GPU

The dynamic executor switching mechanism also allows FGNN to work on a single GPU. FGNN first stores all samples of an epoch in the host memory; after that a standby Trainer is launched to conduct model training. We compared FGNN and DGL on a single GPU for various GNN models

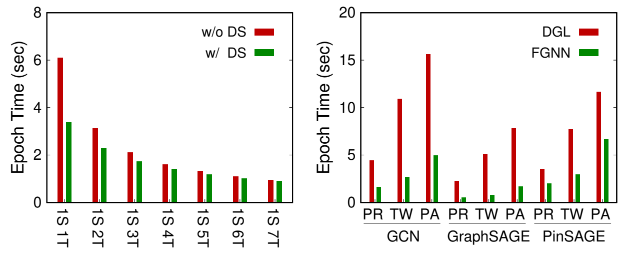

Fig. 16. (a) The runtime of one epoch in FGNN w/ and w/o dynamic switching (DS) for training PinSAGE on PA. (b) The end-to-end performance between DGL and FGNN over a single GPU.

and datasets. As shown in Fig. 16(b), FGNN still outperforms DGL by up to $6.8\times$ (from $1.7\times$). The performance gain in FGNN is mainly from reduced data movement to GPU memory due to our effective pre-sampling based caching policy.

8 Related Work

Recently, how to efficiently support GNN training at scale has attracted increasing attention [9, 49, 52, 57]. Existing solutions can be roughly divided into two categories, namely whole-graph training and sample-based training. In whole-graph training, a graph is divided into multiple partitions, and all vertices/edges are simultaneously processed on multiple GPUs/machines. NeuGraph [35], ROC [28], FlexGraph [47] and Dorylus [44] fall into this category. For sample-based training, it adopts various graph sampling algorithms to select certain neighbors for each training vertex. Then the training process deals with all training vertices in a mini-batch way. Typical GNN systems belonging to this category include AliGraph [61], DistDGL [60], PaGraph [33] and P^3 [17]. Our work focuses on the latter category.

GPUs have been widely adopted to improve different stages of sample-based GNN training, i.e., Sample, Extract and Train. For the Sample stage, GPUs are applied to accelerate graph sampling due to their much higher parallelism and memory access bandwidth than CPUs. Typical work includes NEXTDOOR [27], C-SAW [38] and DGL [4]. While PaGraph [34] adopts a degree-based caching policy to accelerate feature extraction, DGL [4] uses GPUs in the Extract Stage only if all feature data can be loaded in GPUs. For the Train stage, PyG [14], DGL [4] and AliGraph[61] implement highly optimized GNN runtimes. FGNN differs from them in the following. (i) FGNN is the first system that adopts a factored design and various optimizations to enable GPU acceleration for all three stages. (ii) FGNN employs an efficient GPU-based feature caching policy to improve the Extract stage, which is also more robust than PaGraph and DGL that also support GPU-based feature extraction.

9 Conclusion

This paper presents FGNN, a new factored system for sample-based GNN training over GPUs. It further embodies a new pre-sampling based caching policy. Our experimental results confirm the high performance of FGNN.

References

[1] DGL: Deep Graph Library. https://www.dgl.ai/.

[2] Open Graph Benchmark: The ogbn-papers100M dataset. https://ogb.stanford.edu/docs/nodeprop/#ogbn-papers100M.

[3] Open Graph Benchmark: The ogbn-products dataset. https://ogb.stanford.edu/docs/nodeprop/#ogbn-products.

[4] Using GPU for Neighborhood Sampling in DGL Data Loaders. https://docs.dgl.ai/guide/minibatch-gpu-sampling.html.

[5] Euler 2.0: A Distributed Graph Deep Learning Framework. https://github.com/alibaba/euler, 2020.

[6] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)* (2016), pp. 265–283.

[7] ADAMIC, L. A., HUBERMAN, B. A., BARABÁSI, A., ALBERT, R., JEONG, H., AND BIANCONI, G. Power-law distribution of the world wide web. *science 287*, 5461 (2000), 2115–2115.

[8] BOLDI, P., AND VIGNA, S. The WebGraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (2004), WWW'04, pp. 595–601.

[9] CAI, Z., YAN, X., WU, Y., MA, K., CHENG, J., AND YU, F. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the 16th European Conference on Computer Systems* (2021), EuroSys'21, pp. 130–144.

[10] CHEN, J., MA, T., AND XIAO, C. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *Proceedings of the 6th International Conference on Learning Representations* (2018), ICLR'18.

[11] CHEN, J., ZHU, J., AND SONG, L. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning* (2018), ICML'18, pp. 941–949.

[12] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 1–15.

[13] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems* (2015).

[14] FEY, M., AND LENSSEN, J. E. Fast graph representation learning with pytorch geometric.

[15] FISHER, R. A., YATES, F., ET AL. *Statistical tables for biological, agricultural and medical research, edited by ra fisher and f. yates*. Edinburgh: Oliver and Boyd, 1963.

[16] FOUT, A., BYRD, J., SHARIAT, B., AND BEN-HUR, A. Protein interface prediction using graph convolutional networks. In *Advances in neural information processing systems* (2017), pp. 6530–6539.

[17] GANDHI, S., AND IYER, A. P. P3: Distributed Deep Graph Learning at Scale. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation* (2021), OSDI'21.

[18] GILMER, J., SCHOENHOLZ, S. S., RILEY, P. F., VINYALS, O., AND DAHL, G. E. Neural message passing for quantum chemistry. In *International conference on machine learning* (2017), PMLR, pp. 1263–1272.

[19] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)* (2012), pp. 17–30.

[20] GROVER, A., AND LESKOVEC, J. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016), KDD'16, pp. 855–864.

[21] GUJARATI, A., KARIMI, R., ALZAYAT, S., HAO, W., KAUFMANN, A., VIGFUSSON, Y., AND MACE, J. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (2020), OSDI '20, pp. 443–462.

[22] HAMILTON, W. L., YING, R., AND LESKOVEC, J. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017), NeurIPS'17, p. 1025–1035.

[23] HU, W., FEY, M., ZITNIK, M., DONG, Y., REN, H., LIU, B., CATASTA, M., AND LESKOVEC, J. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (2020), NeurIPS'20.

[24] HUANG, K., ZHAI, J., ZHENG, Z., YI, Y., AND SHEN, X. 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* (2021), pp. 119–132.

[25] HUANG, W., ZHANG, T., RONG, Y., AND HUANG, J. Adaptive Sampling towards Fast Graph Representation Learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018), NeurIPS'18, p. 4563–4572.

[26] JAIN, A., LIU, I., SARDA, A., AND MOLINO, P. Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations. https://eng.uber.com/uber-eats-graph-learning/.

[27] Jangda, A., Polisetty, S., Guha, A., and Serafini, M. Accelerating Graph Sampling for Graph Machine Learning using GPUs. In *Proceedings of the 16th European Conference on Computer Systems* (2021), EuroSys'21, pp. 311–326.

[28] Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with ROC. In *Proceedings of the 3rd Machine Learning and Systems* (2020), MLSys'20, pp. 187–198.

[29] Kim, T., Park, K., Hwang, C., Cheng, P., Miao, Y., Ma, L., Lin, Z., and Xiong, Y. Accelerating gnn training with locality-aware partial execution. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (2021), APSys'21.

[30] Kipf, T. N., and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations* (2017), ICLR'17.

[31] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web* (2010), WWW'10, pp. 591–600.

[32] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment 13*, 12 (2020), 3005–3018.

[33] Lin, Z., Li, C., Miao, Y., Liu, Y., and Xu, Y. Pagraph: Scaling GNN Training on Large Graphs via Computation-aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), SoCC'20, pp. 401–415.

[34] Liu, X., Yan, M., Deng, L., Li, G., Ye, X., and Fan, D. Sampling methods for efficient training of graph convolutional networks: A survey. *arXiv preprint arXiv:2103.05872* (2021).

[35] Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proceedings of 2019 USENIX Annual Technical Conference* (2019), ATC'19, pp. 443–458.

[36] Mohoney, J., Waleffe, R., Xu, H., Rekatsinas, T., and Venkataraman, S. Marius: Learning Massive Graph Embeddings on a Single Machine. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation* (2021), OSDI'21.

[37] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), SOSP'19, pp. 1–15.

[38] Pandey, S., Li, L., Hoisie, A., Li, X. S., and Liu, H. C-saw: A framework for graph sampling and random walk on gpus. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), IEEE, pp. 1–15.

[39] Park, J. H., Yun, G., Chang, M. Y., Nguyen, N. T., Lee, S., Choi, J., Noh, S. H., and Choi, Y.-r. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *Proceedings of 2020 USENIX Annual Technical Conference* (2020), Usenix ATC'20, pp. 307–321.

[40] Perozzi, B., Al-Rfou, R., and Skiena, S. Deepwalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2014), KDD'14, pp. 701–710.

[41] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture* (2007), ISCA'07, pp. 13–24.

[42] Satorras, V. G., and Estrach, J. B. Few-Shot Learning with Graph Neural Networks. In *Proceedings of the 6th International Conference on Learning Representations* (2018), ICLR'18.

[43] Serafini, M., and Guan, H. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Operating Systems Review 55*, 1 (2021), 68–76.

[44] Thorpe, J., Qiao, Y., Eyolfson, J., Teng, S., Hu, G., Jia, Z., Wei, J., Vora, K., Netravali, R., Kim, M., 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)* (2021), pp. 495–514.

[45] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations* (2018), ICLR'18.

[46] Vitter, J. S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS) 11*, 1 (1985), 37–57.

[47] Wang, L., Yin, Q., Tian, C., Yang, J., Chen, R., Yu, W., Yao, Z., and Zhou, J. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the 16th European Conference on Computer Systems* (2021), EuroSys'21, pp. 67–82.

[48] Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[49] WANG, Y., FENG, B., LI, G., LI, S., DENG, L., XIE, Y., , AND DING, Y. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation* (2021), OSDI'21.

[50] WENTZLAFF, D., AND AGARWAL, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *ACM SIGOPS Operating Systems Review 43*, 2 (2009), 76–85.

[51] WU, W., LI, B., LUO, C., AND NEJDL, W. Hashing-accelerated graph neural networks for link prediction. In *Proceedings of the Web Conference 2021* (2021), pp. 2910–2920.

[52] WU, Y., MA, K., CAI, Z., JIN, T., LI, B., ZHENG, C., CHENG, J., AND YU, F. Seastar: Vertex-centric Programming for Graph Neural Networks. In *Proceedings of the 16th European Conference on Computer Systems* (2021), EuroSys'21, pp. 359–375.

[53] WU, Z., PAN, S., CHEN, F., LONG, G., ZHANG, C., AND PHILIP, S. Y. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[54] YANG, K., ZHANG, M., CHEN, K., MA, X., BAI, Y., AND JIANG, Y. Knightking: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 524–537.

[55] YING, R., HE, R., CHEN, K., EKSOMBATCHAI, P., HAMILTON, W. L., AND LESKOVEC, J. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2018), KDD'18, p. 974–983.

[56] ZENG, H., ZHOU, H., SRIVASTAVA, A., KANNAN, R., AND PRASANNA, V. Graphsaint: Graph sampling based inductive learning method. In *Proceedings of the 8th International Conference on Learning Representations* (2020), ICLR'20.

[57] ZHANG, D., HUANG, X., LIU, Z., ZHOU, J., HU, Z., SONG, X., GE, Z., WANG, L., ZHANG, Z., AND QI, Y. AGL: A Scalable System for Industrial-Purpose Graph Machine Learning. *Proc. VLDB Endow. 13*, 12 (2020), 3125–3137.

[58] ZHANG, M., AND CHEN, Y. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems 31* (2018), 5165–5175.

[59] ZHANG, Z., CUI, P., AND ZHU, W. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[60] ZHENG, D., MA, C., WANG, M., ZHOU, J., SU, Q., SONG, X., GAN, Q., ZHANG, Z., AND KARYPIS, G. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *Proceedings of the 10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms* (2020), IA3'20, pp. 36–44.

[61] ZHU, R., ZHAO, K., YANG, H., LIN, W., ZHOU, C., AI, B., LI, Y., AND ZHOU, J. AliGraph: A Comprehensive Graph Neural Network Platform. In *Proceedings of the VLDB Endowment* (2019), pp. 2094–2105.