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

Jianbang Yang[†1] Dahai Tang[†3,4] Xiaoniu Song[1,2] Lei Wang[4] Qiang Yin[5]
Rong Chen[‡1,2] Wenyuan Yu[4] Jingren Zhou[4]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]Shanghai AI Laboratory [3]Hunan University [4]Alibaba Group [5]BASICS, Shanghai Jiao Tong University

Abstract

We propose GNNLab, a sample-based GNN training system in a single machine multi-GPU setup. GNNLab 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, GNNLab applies a global queue to bridge GPUs asynchronously and adopts a simple yet effective method to adaptively allocate GPUs for different tasks. GNNLab further leverages temporarily switching to avoid idle waiting on GPUs. Furthermore, GNNLab proposes a new pre-sampling based caching policy 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 GNNLab outperforms the state-of-the-art GNN systems DGL and PyG by up to 9.1× (from 2.4×) and 74.3× (from 10.2×), respectively. In addition, our pre-sampling based caching policy achieves 90% – 99% of the optimal cache hit rate in all experiments.

CCS Concepts: • **Software and its engineering** → **Space-based architectures**; • **Computing methodologies** → **Concurrent computing methodologies**.

Keywords: graph neural networks, sample-based GNN training, caching policy

ACM Reference Format:
Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: A Factored System for Sample-based GNN Training over GPUs. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3492321.3519557

[†] Jianbang Yang and Dahai Tang contributed equally to this work.
[‡] Rong Chen is the corresponding author (rongchen@sjtu.edu.cn).

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519557

1 Introduction

Graph neural networks (GNNs) are an emerging family of neural networks that operate on graph-structured data [57], and have demonstrated convincing performance on many applications in recommendation systems [29, 58], molecule analysis [19], social network mining [33, 55, 63], fraud detection [16], to name a few. GNN models 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 [25, 49], link prediction [46, 63] and graph clustering [59]. 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 [21], posing both implementation and performance challenges for existing tensor-oriented frameworks, such as TensorFlow [7], PyTorch [42], and MXNet [14]. Many GNN systems [2, 17, 20, 31, 37, 38, 48, 51, 52, 56, 67] 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 [8, 13, 22]. 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 [35, 67]. A practical solution to this problem 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 [25, 58, 65]. 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 [20, 62, 66].

Recently, GPUs have been widely exploited to accelerate GNN training [27, 31]. A typical scenario is a single machine equipped with *multiple* GPUs. Since large-scale graphs (both topological and feature data) exceed limited

GPU memory capacity (e.g., 32 GB or less), most of existing GNN systems store graph topological and feature data in the host memory of a machine [66, 67]. Given a mini-batch size, CPUs repeatedly sample input graph and extract features of sampled vertices to produce *samples*; meanwhile, generated samples are *continuously* copied to GPUs to trigger model training. As also observed in §3, recent work [20, 35] has reported that under such a setting, conducting graph sampling and transferring feature data of sampled vertices 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 topological data itself, but the features of vertices. Based on this observation, some studies [30, 40] attempted to transfer graph topological data into the GPU memory and apply GPUs to accelerate graph sampling. In addition, prior work [35] found that some vertices are more frequently sampled than others, and thus a static caching strategy is introduced 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 that existing degree-based caching policy only works well on the k-hop random neighborhood sampling algorithm [25] and graphs with power-law degree distributions [13], and cannot cover the diversity of sampling algorithms and GNN datasets.

Based on the above analysis, we propose GNNLab, a factored system for sample-based GNN training in a single machine multi-GPU setup. GNNLab adopts a *space sharing* design for multiple GPUs, i.e., one GPU loads either graph topological data 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, GNNLab 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. GNNLab 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 sampling algorithms and GNN datasets, we propose a general caching scheme. It consists of two parameters, a *hotness metric* that estimates the frequency of a vertex being sampled during the graph sampling stage 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 [35], 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 sampling algorithms and GNN datasets.

We have implemented GNNLab that adopts all optimization strategies mentioned above. We evaluated GNNLab on three GNN models (i.e., GCN [33], GraphSAGE [25], and PinSAGE [58]) over four real-life graphs, and compared it with the state-of-the-art GNN systems DGL [52] and PyG [17]. Experimental results show that GNNLab outperforms DGL and PyG by up to 9.1× (from 2.4×) and 74.3× (from 10.2×), respectively. In addition, PreSC is able to achieve 90%–99% of the optimal cache hit rate in all tests.

Contributions. We make the following contributions.

(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 space sharing 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 a caching 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 shows the advantage and efficacy of GNNLab (§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 model 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*

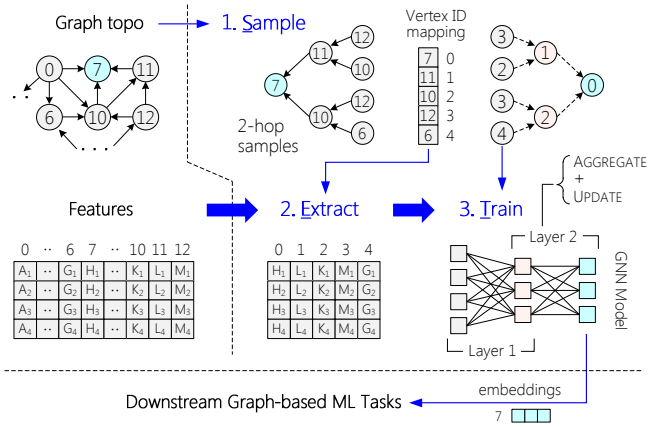

Figure 1. An example of the SET model for sample-based training in a 2-layer GNN on V_7.

training, i.e., each vertex considers its all neighbors when aggregating features. However, whole-graph training is hard to scale [47]. First, it is increasingly common for GNNs to encounter a large-scale graph with high-dimensional features [20, 35]. Second, many graphs follow a highly skewed degree distribution [13]. A well-connected vertex will aggregate features 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 GNN models [11, 25, 28, 61] adopt the sample-based training approach. 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 sampled vertices (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. Figure 1 illustrates an example of the SET model (left-right) for training a 2-layer GNN on V_7. The graph sampling algorithm uniformly selects two neighbors for each vertex. Note that the sampled vertices may be deduplicated (e.g., V_{12}) and re-assigned 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 [12, 36, 47]. Hence, it has been widely adopted by existing GNN systems [1, 2, 20, 35, 67]. 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 [17, 20, 35, 37, 38, 56]. Figure 2(a) shows a conventional design for sample-based

[1]There exist various sampling algorithms, such as k-hop random/weighted neighborhood sampling [25, 28, 64]. The sampling probability could be uniform [25] or non-uniform (e.g., in proportion to the edge weight [43]).

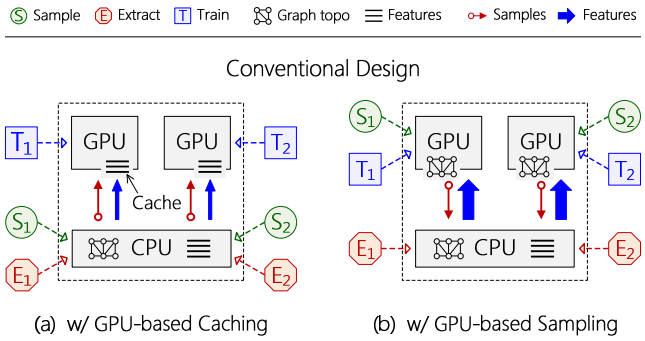

Figure 2. The conventional design for sample-based GNN training with two optimizations.

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 sampled vertices 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 [25, 33, 49]. Each GPU trains mini-batches independently and exchanges gradients among GPUs to update model parameters synchronously or asynchronously.

However, with the increase of input data size—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 [33] on OGB-Papers [4].[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 of sampled vertices from host memory to GPU memory due to the limited PCIe bandwidth (normally less than 16 GB/s) [32, 35]. Thus, prior work (e.g., PaGraph [35]) proposed to selectively cache the features associated with frequently sampled vertices in GPU memory (see Cache in Figure 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. Since DGL does not support GPU-based feature caching, we implemented T_{SOTA}, a state-of-the-art GNN system based on the conventional design, which extends DGL [1] with a static GPU-based cache [35] and a fast GPU-based sampler from scratch.

[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 [33] with random neighborhood sampling. **Dataset**: OGB-Papers [4]. **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
T_{SOTA}	2.93	5.55	4.00	12.50
w/ GPU-base Caching [35]	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

As shown in Table 1, by caching 21% features in GPU memory (11.4GB), T_{SOTA} reduces data transfer by 62.8% during one training epoch (from 25.3GB to 9.4GB), resulting in a 1.45× performance improvement in end-to-end GNN training time (12.5 s vs. 8.62 s).

GPU-based graph sampling. Graph sampling also takes a substantial portion of the end-to-end GNN training time [30, 47]. Thus, leveraging GPUs to accelerate graph sampling has appeared in both academic and open-sourced projects, like NextDoor [30] and DGL [6]. As shown in Figure 2(b), graph topological data is first loaded into GPU memory and then sampled on the GPU for each mini-batch. 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 sampled vertices and their features. Generally, the graph topological data is preloaded and kept in the GPU memory if possible [6]. For example, the end-to-end speedup by using one GPU for graph sampling reaches 1.28× in DGL and 1.22× in T_{SOTA}, 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, T_{SOTA} just achieves 1.49× performance speedup by enabling both of two optimizations.[3] 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 Figure 3. We observe that different GNN stages operate on different input data (e.g., graph topological data for sampling and feature data

[3]Prior work [30, 35] 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 [35].

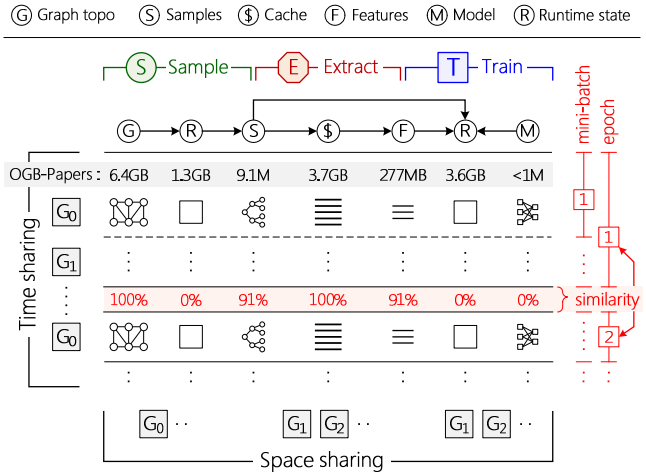

Figure 3. A breakdown of memory usage and data similarity for different stages of the SET model when training OGB-Papers over multiple GPUs (G_0, G_1, …) with 16GB of memory each.

for extracting), and only share a small amount of data (e.g., 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 (graph topological and feature data), especially for large-scale graphs with high-dimensional features. Thus, although GPU memory is usually able to keep all graph topological data, it will greatly limit the available memory capacity for feature cache (see $/Cache in Figure 3). As an example, OGB-Papers [4] dataset includes 6.4GB graph topological data and 53GB feature data. When sampling it on a GPU with 16GB memory, it will decrease the cache ratio of features from 21% to 7% (from 11.4GB to 3.7GB, see two vertical lines in Figure 4(a)), due to keeping graph topological data in GPU memory (6.4GB). Note that graph sampling and model training 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 Figure 4(a). As a result of degraded cache ratio, the extracting time significantly increases by 2.2× (from 13.1 ms to 28.7 ms for a mini-batch), even surpassing the training time (24.4 ms), 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 [3, 20, 25]. Figure 4(b) shows that the cache hit rate of a 5GB cache drops to 10% when the feature dimension of OGB-Papers increases to 768 (same as OGB-MAG [3]); it will take about 190 ms to load more than 1.5GB features into GPU memory.

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

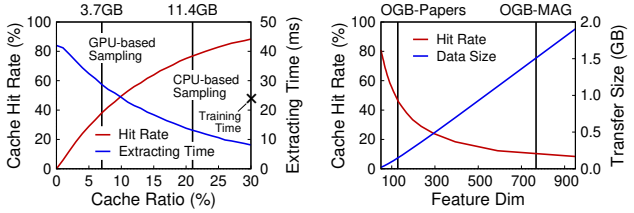

Figure 4. (a) The cache hit rate and the extracting time for OGB-Papers with the increase of cache ratio. (b) The cache hit rate and the size of transferred data with the increase of feature dimensions.

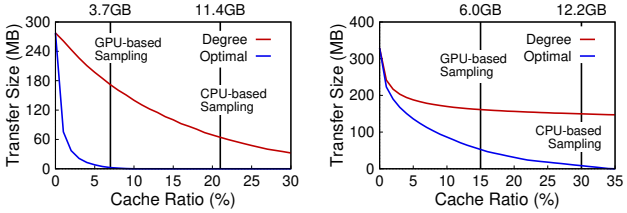

Figure 5. The size of transferred data of degree-based and optimal caching policies with the increase of cache ratio for (a) OGB-Papers with uniform sampling and (b) Twitter with weighted sampling.

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 [35]. It assumes that the input graph has a highly skewed out-degree distribution. 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 Figure 5, a significant gap exists in the size of transferred data to GPU memory between the existing (degree-based) policy [35] and the optimal results[4] 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 [4]) and web graphs (e.g., UK-2006 [9]), which violates the narrow assumption of graph structures in prior work [35]. Further, the sampling algorithm is only conducted on the training set—usually a small fraction of vertices—and just aggregates their neighbors within 2 or 3 hops [25, 33, 49]. However, the existing policy takes all vertices of a graph into consideration. For example, in Figure 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× data to GPUs (171.9MB vs. 2.5MB) 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 significantly[5], which prior work [28, 43, 58, 64] overlooks. As shown in Figure 5(b), even if Twitter [34] dataset has a skewed out-degree distribution, the amount of transferred data to GPUs is still far from optimal when using 3-hop weighted neighborhood sampling [28]. 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 sample-based GNN training has been proposed in AGL [62]. At the beginning of one epoch, all GPUs load graph topological data into memory and conduct graph sampling. After that, all GPUs swap graph topological data out, and load feature cache for effective feature extraction and model training. We find this design unsuitable for our setting, as graph topological data is swapped between the 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 topological data and large feature cache, while during the same time interval, tens of epochs can be finished. Hence, we omit such a 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 Figure 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 Figure 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) [54], the key idea behind GNNLab is to perform each stage of the SET model (e.g., Sample) on dedicated processors (GPUs and/or CPUs) for different mini-batches. GNNLab is a new factored system for sample-based GNN training over GPUs, in which space sharing replaces time sharing to improve performance significantly. Figure 6 illustrates a brief example of GNNLab that conducts two

[4]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.
[5]For example, k-hop weighted neighborhood sampling (e.g., ASGCN [28] and Thanos [64]), which is available in many GNN systems (e.g., DGL [52] and AliGraph [67]), selects neighbors of a target vertex based on the probability determined by weights of edges that connect to its neighbors.

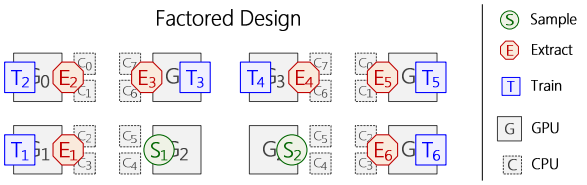

Figure 6. An example of the factored design for sample-based GNN training over 8 GPUs and two 8-core CPUs.

samplers, six extractors and six trainers on a machine with 8 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. More specifically, GNNLab keeps graph topological data and cached features in the memory of different GPUs. It brings two advantages by leaving more GPU memory space for both topological and feature data. First, GNNLab can sample larger graph data using a single GPU without additional data loading. Meanwhile, GNNLab can significantly reduce the extracting time by caching more features in the GPU memory (see Figure 4(a)).

Challenge: load imbalance. The above 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). Indeed, both Sample and Train stages perform on dedicated GPUs, while the execution time of two stages 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, GNNLab needs to flexibly assign GPUs to different stages and make them work together efficiently. In some cases, GNNLab 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 GNNLab Architecture

GNNLab is a new factored GNN system that performs different stages of the SET model on different processors (GPUs and CPUs). Hence GNNLab 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 section, we first describe the programming model in GNNLab and then introduce key designs to tackle load imbalance across GPUs, such as hybrid execution and flexible scheduling.

5.1 Programming Model

GNNLab provides a simple data-parallel programming model for various sampling algorithms and GNN models,

```
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
```

Figure 7. Example of GCN and k-hop sampling in GNNLab.

similar to existing GNN systems (e.g., DGL [1]). Figure 7 outlines the implementation of the GCN model [33] and 3-hop random neighborhood sampling in GNNLab. For sampling algorithms (see *class* KHOP), GNNLab'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 sampling [25, 28] and random walks [23, 58]. For GNN models (see *class* GCN), GNNLab's API defines a model by stacking multiple GNN layers.

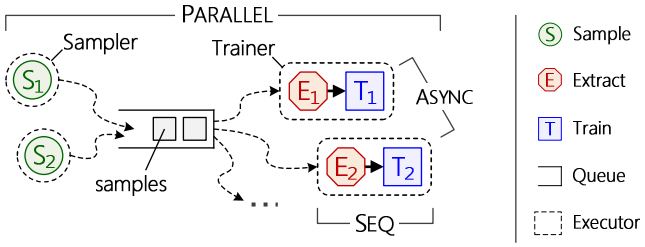

Figure 8. The execution flow of GNNLab.

5.2 Hybrid Execution

To adapt to diverse workloads, GNNLab 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 GNNLab divides the training pipeline of the SET model into two kinds of individual executors, named Sampler and Trainer respectively, as shown in Figure 8. GNNLab 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. Figure 9 outlines the implementation of executors in GNNLab.

```
class Sampler(...):
1   def run(self, dev, q, graph, ...):
2     load(dev, graph)  # load graph to GPU memory
3     khop3 = KHOP(graph, ...)  # define 3-hop sampling
4     ...
5     while (minibatch = get_minibatch())
6       samples = khop3(minibatch)
7       remap(dedup(samples))
8       q.enque(samples)  # (async) send task
class Trainer(...):
9   def run(self, dev, q, features, ...):
10    model = GCN(...)  # define a 3-layer GCN
11    loss_func = ...   # define a loss function
12    ...
13    while (samples = q.deque())  # (async) recv task
14      in_feats = extract(features, samples)
15      loss = loss_func(model(samples, in_feats), ...)
16      loss.backward()
```

Figure 9. Two kinds of executors in GNNLab.

GNNLab binds each Sampler to a GPU, and it will load graph topological data into 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 [45]. 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 [30]. We leave this as future work.

On the other hand, GNNLab 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 feature cache in GPU memory (if any). Note that GNNLab 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, GNNLab 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 GNNLab. For multiple Trainers, they do not interact with each other except for exchanging locally produced gradients to update GNN model parameters. To support pipelining, GNNLab updates model gradients with bounded staleness, similar to prior work [20, 38, 39, 41, 48], which effectively mitigates the convergence problem.

5.3 Flexible Scheduling

GNNLab 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 appropriate number of different executors, i.e., Samplers and Trainers, for a given GNN workload. Fortunately, we observe that the performance of executors in GNNLab 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 GPUs have negligible performance variability [24]. 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:

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

where N_g is the total number of available GPUs and K is the ratio of the training time (T_t) to the sampling time (T_s). GNNLab prefers to allocate GPUs to Samplers because temporarily switching from Samplers to Trainers can be fulfilled quickly, but not vice versa. More specifically, the Sampler may have to take a few seconds, which can sample hundreds of mini-batches, to load the whole graph topological data into GPU memory before execution (see §7.6). On the contrary, the Trainer can run on the GPU immediately, and the size of feature cache in GPU memory only affects the training 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, under unpredictable workload settings (e.g., a shared multi-tenant cluster), there may exist other workloads contending for GPUs, temporarily slowing down the Samplers and Trainers. Furthermore, GNNLab might encounter highly skewed workloads (e.g., Trainers are 10× slower than Samplers) 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 mentioned previously, GNNLab only needs to consider switching from Samplers to Trainers, and the switching is temporary. GNNLab 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, graph topological data should always be kept in GPU memory, even though it would limit the size of feature cache for Trainers. To enable fast switching, GNNLab 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, GNNLab 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 a standby Trainer (with limited feature cache) for a mini-batch. The profit metric ($\mathcal{P}$) measures whether a standby Trainer can complete one task before existing (normal) Trainers finish all remaining tasks. If the profit is greater than zero (i.e., $\mathcal{P}$>0), GNNLab will wake the standby Trainer to process training tasks. If there is no Trainer (i.e., N_t=0), the profit is infinity, and GNNLab should always switch Samplers to Trainers. Note that each standby Trainer should calculate the profit metric separately before each time it attempts to fetch a task from the global queue.

Although GNNLab adopts a space sharing strategy for the multi-GPU setting, 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 stage of all epochs. 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 [35] utilizes vertex out-degree as the hotness metric.

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

Sampling algorithms	PR [5]	TW [34]	PA [4]	UK [9]
3-hop random	73.97	78.89	91.29	77.46
Random walks	78.16	72.68	87.14	64.40
3-hop weighted	77.69	66.64	89.57	72.96

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 feature cache and the vertex feature dimension. To determine GPU memory capacity for feature cache, we adopt the method proposed in PaGraph [35], 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, GNNLab 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 GPUs. 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 GNNLab with this caching scheme. For example, to implement the degree-based caching policy adopted by PaGraph [35] in GNNLab, 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 sampled frequency of vertex v 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.

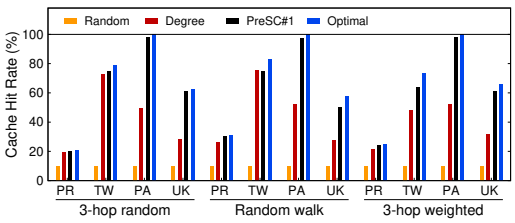

Figure 10. The comparison of the cache hit rate for various workloads by using different caching policies. The cache ratio of features is set to 10%.

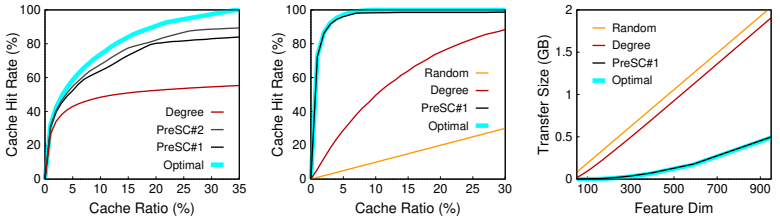

Figure 11. The comparison among different caching policies for (a) Twitter with weighted sampling, (b) OGB-Papers with 3-hop neighborhood, and (c) OGB-Papers with the increase of feature dimensions. PreSC#K conducts K sampling stages.

It is theoretically possible to compute $\mathbb{E}[\hat{h}_v]$ for each vertex v, but this would incur high preprocessing 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). Given a graph 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., $K \leq 2$, already produce a decent hotness estimation and suffice for most training tasks (see Figure 11(a)). Thus it is feasible to compute the h_v's of PreSC *online*, since (i) GPU-based graph pre-sampling is lightweight, e.g., on average it only takes 1.4× 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.

In general, the benefits of pre-sampling based caching policy are two-fold: *efficiency* and *robustness*.

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 Figure 10, fixing cache ratio $\alpha = 10\%$, the cache hit rate of PreSC is almost as good as the Optimal policy, and is on average 1.5× (up to 2.2×) 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 Figure 10, on four GNN datasets and three sampling algorithms, PreSC constantly beats other baselines, including the Random policy and Degree policy adopted by PaGraph [35]. This is because, as oppose to prior work, the hotness metric h_v of PreSC is computed by simultaneously taking the input graph G, the training set T and the sampling algorithm A into account. Observe that the performance of Degree policy is unstable. For example, for the 3-hop random neighborhood and random walks, the Degree policy has a similar cache hit rate as PreSC on the power-law graph TW [34]. However, if we either use a non-power-law graph, e.g., PA [4], 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 Figure 11(b) we first plot the cache hit rate to the cache ratio α on OGB-Papers dataset 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 and Degree policies 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, Figure 11(c) further shows the size of data moved to the GPU memory in one mini-batch 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 policies. 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 GNNLab 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 walks. Besides, it utilizes DGL as the GNN execution runtime (the Train stage).

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 v2.0.1.

GNNs. We used three representative models: GCN [33], GraphSAGE [25], and PinSAGE [58]. GCN (resp. GraphSAGE) adopts 3-hop (resp. 2-hop) random neighborhood sampling [66]. Starting from a training vertex, the number of sampled neighbors for different layers of GCN (resp. GraphSAGE) is 15, 10, and 5 (resp. 10 and 25). PinSAGE has 3 layers, and each layer uses random walks to select 5 neighbors from 4 paths of length 3. We determined the above settings (e.g., the size of sampled neighbors and the sampling algorithm) by comprehensively following the details reported in the original papers [25, 33, 58] together with official examples and guidelines from DGL. Considering that all three selected GNN models do not apply weighted sampling, we evaluated the 3-hop weighted neighborhood sampling algorithm [28] in §7.4 to further investigate how it affects the performance of GNNLab when using different caching policies. In addition, similar to prior work [31, 66], the dimension of the hidden layers of all models is set to 256, and the batch size is set to 8,000.

Table 3. Datasets and GNN systems used in evaluation. #TS denotes the size of training set. Vol_G (resp. Vol_F) is the data volume of graph topological (resp. feature) data in host memory. N/A represents the GPU is only used by a single stage (i.e., Train).

Dataset	#Vertex	#Edge	Dim.	#TS	Vol_G	Vol_F
PR [5]	2.4M	124M	100	197K	481MB	934MB
TW [34]	41.7M	1.5B	256	417K	5.6GB	40GB
PA [4]	111M	1.6B	128	1.2M	6.4GB	53GB
UK [9]	77.7M	3.0B	256	1.0M	11.3GB	74GB

System	Design	Sample	Extract	Train
PyG	N/A	CPU	No cache	GPU
DGL	Time S.	GPU	No cache	GPU
T_{SOTA}	Time S.	GPU w/ Opt.	Cache w/ Degree	GPU
GNNLab	Space S.	GPU w/ Opt.	Cache w/ PreSC	GPU

Datasets. We used four datasets as listed in Table 3 for evaluation, including a social graph Twitter [34] (TW), a web graph UK-2006 [9] (UK), and two GNN datasets from Open Graph Benchmark (OGB) [26]—a co-purchasing network OGB-Products (PR) and a citation network OGB-Papers (PA). Similar to prior work [20], we generated random features and labels for TW and UK since they originally had no features and labels. Both PR and PA from OGB provide an official training set. For TW and UK, which do not provide a training set, we followed a common practice [35] that randomly selects a small portion of vertices as the training set. Note that the training set is selected offline once and shared across each run. The overhead to select the training set is trivial, e.g., less than 150 ms for the largest graph (UK).

Baselines. We compared GNNLab with PyG [17], DGL [52] and T_{SOTA}.[6] As shown in Table 3, PyG conducts graph sampling on CPUs, while DGL enables GPU-based sampling to accelerate graph sampling. T_{SOTA} is built upon the same codebase of GNNLab, and supports both GPU-based graph sampling and feature caching. Different from GNNLab, T_{SOTA} follows a time sharing design, i.e., each GPU conducts both graph sampling and model training, and adopts the degree-based caching policy [35]. DGL also uses time sharing, but has no caching mechanism. Since DGL only supports synchronous gradient updates, for fair comparisons, GNNLab and other baselines employ synchronous gradient updates unless otherwise specified. All results were computed by calculating the averages over 10 epochs.

7.2 Overall Performance

We first compared GNNLab with its competitors. Table 4 reports the end-to-end training time of one training epoch for each GNN system. The number of GPUs allocated to Samplers (nS) is determined by the method in §5.3. Note that for

[6] Since PaGraph [35] is built with DGL v0.4.1 which only supports sampling by CPUs, T_{SOTA} clearly outperforms PaGraph and also uses degree-based caching policy. Therefore, we do not consider PaGraph in our experiments.

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

GNN Model	Dataset	PyG	DGL	T_{SOTA}	GNNLab
GCN	PR	11.91	1.33	0.22	0.33 (2S)
	TW	12.15	3.86	1.80	0.47 (2S)
	PA	14.82	4.56	2.46	0.84 (2S)
	UK	15.04	OOM	OOM	1.47 (2S)
GraphSAGE	PR	8.17	0.79	0.07	0.11 (4S)
	TW	8.18	1.81	0.35	0.20 (2S)
	PA	9.68	2.47	0.85	0.30 (2S)
	UK	9.86	OOM	2.01	0.61 (1S)
PinSAGE	PR	×	0.94	0.30	0.40 (1S)
	TW	×	2.50	0.98	0.58 (1S)
	PA	×	2.97	1.65	1.05 (1S)
	UK	×	OOM	OOM	1.81 (1S)

an 8-GPU machine, our flexible scheduling scheme already provides optimal GPU allocations for Samplers. Therefore, dynamic switching does not happen in this evaluation. We mainly find the following.

(1) Overall, GNNLab outperforms DGL and PyG by up to 9.1× (from 2.4×) and 74.3× (from 10.2×), respectively. For systems using GPUs for graph sampling, only GNNLab can process UK dataset in all cases, while other systems run out of memory (OOM) due to GPU memory contention. Note that PyG performs the worst in all experiments due to the high cost of graph sampling on CPUs and transferring features to GPUs (see §2). Due to space limitations, we do not report its experimental results in the rest of our evaluation. In general, the performance gain of GNNLab over its competitors mainly comes from three aspects: (A1) a new space sharing design that unleashes the power of GPU-based sampling and caching, (A2) an efficient and robust caching policy (PreSC), and (A3) an efficient implementation of GPU-based graph sampling. More details can be found in §7.3.

(2) Compared with T_{SOTA}, GNNLab benefits from (A1) and (A2). Indeed, T_{SOTA} suffers from GPU memory contention and its inefficient degree-based caching policy. Specifically, T_{SOTA} needs to load graph topological data into each GPU, leaving only limited memory for feature cache. As shown in Figure 11, PreSC is more efficient than the degree-based policy, especially when the cache ratio is small. Note that T_{SOTA} performs slightly better than GNNLab on PR. This is because all topological and feature data of PR can be loaded into a single GPU. Therefore, (A1) and (A2) cannot improve that case, while our factored design introduces little overhead in the Sample stage (see §7.3 for details). Note that T_{SOTA} is built upon the same codebase of GNNLab.

(3) The performance gain of GNNLab over DGL comes from (A1)~(A3). DGL stores all feature data in the host memory and has no GPU-based caching mechanism. Thus, it transfers much more data to GPUs than T_{SOTA} and GNNLab. In addition, 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 shared by CPUs is another major bottleneck.

7.3 Performance Breakdown

We next conducted a stage-level time breakdown analysis for the SET model on DGL, T_{SOTA}, and GNNLab with two GPUs (1S1T for GNNLab). The results are reported in Table 5. We mainly find the following.

(1) For the Sample stage (S), GNNLab and T_{SOTA} beat DGL on three models. We find that DGL adopts the Reservoir algorithm [50] for *k*-hop random neighborhood sampling on GPUs. The sampling complexity of each vertex is positively correlated with its in-degree, resulting in an unbalanced workload on GPU threads. Instead, GNNLab and T_{SOTA} implement a variant of the Fisher–Yates algorithm [18], which is GPU-friendly since the workload is more balanced for each vertex. Note that the performance gain of GNNLab and T_{SOTA} over DGL are larger on PinSAGE than on GCN and GraphSAGE. After profiling, we find that invoking CUDA code from Python code in DGL incurs considerable runtime overheads. Meanwhile, compared with *k*-hop random neighborhood sampling, random walks has more complex vertex access patterns, making the runtime overheads more significant. Furthermore, compared to T_{SOTA}, GNNLab incurs additional overheads (less than 0.1 ms on average) for copying samples to a global queue in the host memory (see §5.2).

(2) For the Extract stage (S), the performance largely depends on cache size and caching policy. The former determines the cache ratio of features (R%), and the latter determines the cache hit rate (H%). Without caching, DGL must transfer all features of sampled vertices from host memory to GPU memory, which accounts for up to 85.0% (from 38.8%) of end-to-end time. By enabling GPU-based caching and the degree-based policy, T_{SOTA} performs much better, especially for small datasets (e.g., PR). However, the time sharing design greatly limits the available memory capacity for the feature cache, resulting in relatively low cache ratios for large graphs (e.g., only 1% for GCN on TW). Further, the degree-based caching policy is still far from efficient for many graphs and sampling algorithms. For example, the cache hit rate for GCN on PA is only 37%, when caching 7% of features. In contrast, GNNLab never gets bogged down by extracting features in all experiments, thanks to our space sharing design and pre-sampling based caching policy (PreSC). First, the cache ratio is significantly improved in GNNLab, e.g., from 1% to 25% for GCN on TW. Second, PreSC demonstrates surprising efficiency. For example, on PA, caching less than 25% of features reduces

Table 5. The runtime breakdown (in seconds) of one epoch for DGL, T_{SOTA} and GNNLab. S, E, and T represent sample, extract, and train stages. G, M, and C represent graph sampling, marking cached vertices, and copying samples to the host memory in the Sample stage, respectively. R% and H% represent the *cache ratio* of features and the cache hit rate. GSG and PSG are short for GraphSAGE and PinSAGE.

GNN	Dataset	DGL			T_{SOTA}			GNNLab		
		S	E	T	S = G + M	E (R%, H%)	T	S = G + M + C	E (R%, H%)	T
GCN	PR	0.35	2.81	1.22	0.30 = 0.29 + 0.01	0.04 (100, 100)	1.18	0.39 = 0.29 + 0.01 + 0.09	0.15 (100, 100)	1.18
	TW	0.74	9.44	1.48	0.29 = 0.26 + 0.03	3.68 (1, 29)	1.53	0.37 = 0.26 + 0.03 + 0.08	0.76 (25, 89)	1.51
	PA	1.20	10.70	4.00	0.79 = 0.70 + 0.10	3.64 (7, 38)	4.00	0.96 = 0.68 + 0.10 + 0.18	0.49 (21, 99)	3.82
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.56 = 0.39 + 0.03 + 0.14	3.06 (14, 70)	3.09
GSG	PR	0.13	1.92	0.23	0.16 = 0.15 + 0.01	0.03 (100, 100)	0.25	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	0.24
	TW	0.38	4.65	0.44	0.12 = 0.11 + 0.01	0.62 (15, 77)	0.44	0.16 = 0.11 + 0.01 + 0.03	0.41 (32, 89)	0.43
	PA	0.56	6.06	1.25	0.38 = 0.33 + 0.06	1.42 (11, 56)	1.18	0.46 = 0.31 + 0.06 + 0.08	0.28 (25, 99)	1.15
	UK	OOM	OOM	OOM	0.19 = 0.19 + 0.00	4.49 (0, 0)	1.08	0.26 = 0.18 + 0.02 + 0.06	1.39 (18, 72)	1.01
PSG	PR	0.40	1.64	1.75	0.16 = 0.16 + 0.01	0.03 (100, 100)	1.74	0.20 = 0.15 + 0.01 + 0.04	0.08 (100, 100)	1.72
	TW	0.72	5.22	2.59	0.23 = 0.22 + 0.02	1.12 (4, 60)	2.60	0.28 = 0.21 + 0.02 + 0.05	0.51 (26, 86)	2.52
	PA	1.86	4.85	5.78	0.54 = 0.49 + 0.05	1.68 (6, 37)	6.09	0.61 = 0.47 + 0.04 + 0.09	0.33 (22, 97)	6.01
	UK	OOM	OOM	OOM	OOM	OOM	OOM	0.65 = 0.49 + 0.03 + 0.13	3.37 (13, 57)	7.00

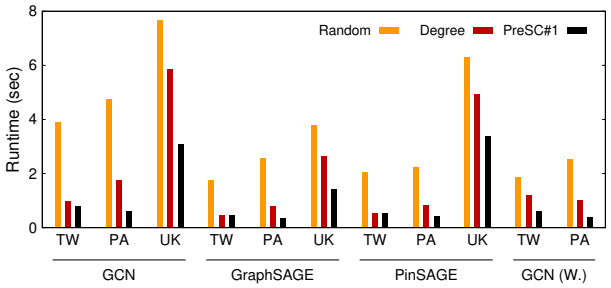

Figure 12. The runtime of the Extract stage on 1 GPU. GCN (W.) stands for GCN with 3-hop weighted neighborhood sampling.

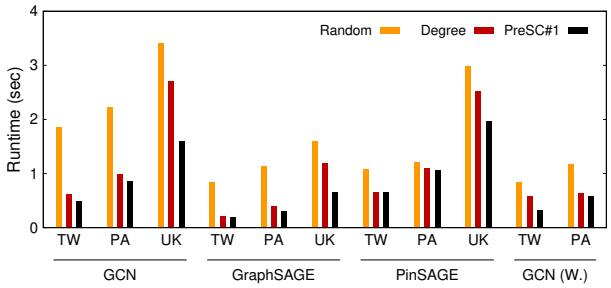

Figure 13. The runtime of one epoch in GNNLab using different caching policies. The GPU allocation scheme follows Table 4.

data movement by more than 97%. However, compared to T_{SOTA}, GNNLab incurs additional overheads (less than 0.1 ms on average) for loading samples into GPU memory from the global queue (see §5.2). Overall, GNNLab outperforms T_{SOTA} by 4.2× on average in the Extract stage (except for PR), due to fetching over 84% of required features directly from the GPU cache.

(3) For the Train stage (T), GNNLab, T_{SOTA} and DGL have similar performance, as all three systems employ the same GNN execution runtime (i.e., DGL) in this stage. For flexible scheduling (see §5.3), in most cases, the training time is used to calculate the number of GPUs allocated to Sampler (N_s). For GCN and GraphSAGE on UK, the extracting time dominates the processing time of Trainer (T_t), so that it replaces the training time to calculate N_s.

7.4 Impact of Caching Policy

We next explored how the caching policy impacts extracting time (including extracting features in CPUs/GPUs and transferring data to GPUs) and the end-to-end time of one epoch. We implemented the degree-based policy (Degree) [35] and the random policy (Random) (i.e., randomly select and cache vertices) in GNNLab, and compared them with our pre-sampling based policy (PreSC#1). Figure 12 reports extracting time of one epoch for four GNN models and three datasets. Note that we omit the results of PR dataset since all of its feature data can be loaded into GPU memory. As we can see, the degree-based policy only works well for TW dataset but not on other datasets. Conversely, PreSC#1 always beats its competitors, which further confirms our findings in §6. Compared to Degree and Random, PreSC#1 reduces extracting time by 39% and 73% on average, up to 87%.

We also investigated the impact of various caching policies on the end-to-end time of one training epoch. We find that the improvement of end-to-end time varies from model to model. For GNN models with low training complexity (e.g., GCN and GraphSAGE), the end-to-end time is largely reduced since model training only accounts for a small fraction. As shown in Figure 13, for GCN and GraphSAGE, PreSC#1 helps reduce up to 45% (from 5%) and 76% (from 50%) of end-to-end time compared to Degree and Random, respectively. Whereas for GNN models with high training complexity (e.g., PinSAGE), the improvement is relatively limited (1%–40% for PinSAGE), as the Train stage accounts

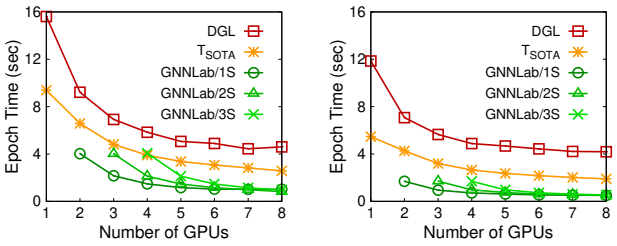

Figure 14. The scalability of DGL, T_{SOTA} and GNNLab for training GCN on (a) PA and (b) TW w.r.t. the number of GPUs. GNNLab/kS indicates that k GPUs is allocated to Samplers.

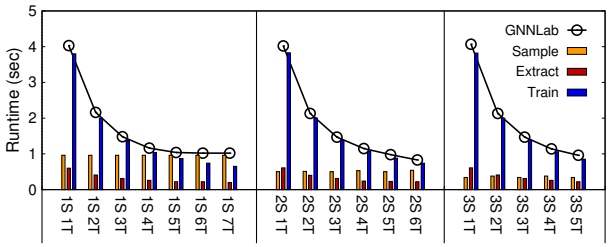

Figure 15. The runtime breakdown of an epoch in GNNLab for training GCN on PA w.r.t. the number of GPUs. mS and nT indicate m Samplers and n Trainers, respectively.

for a large fraction of the end-to-end time, especially when pipelining is enabled.

7.5 Scalability

We next evaluated the scalability of GNNLab w.r.t. the number of GPUs. Fixing the number of GPUs for Samplers as 1, 2 and 3, Figure 14 reports the end-to-end time of one epoch of GCN on PA and TW datasets. We can see that as the number of Trainers increases, the epoch time first decreases almost 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 gradually catch up with the sample generating speed. Compared to GNNLab, both DGL and T_{SOTA} have longer epoch times, and the epoch time decreases more slowly as the number of GPUs increases. This is because DGL and T_{SOTA} need to extract and transfer more feature data from the host memory in parallel, which consumes CPU resources and PCIe bandwidth. However, neither of them increase with the number of GPUs. Further, both DGL and T_{SOTA} follow a time sharing design, so that more GPUs are involved in extracting features, leading to more resource contention.

To demonstrate the efficacy of flexible scheduling in depth, we further plotted the execution time in stages of GNNLab for GCN on PA. As shown in Figure 15, fixing the number of Samplers (m) to 1, 2, and 3, the end-to-end time of one epoch (the black line) keep dropping as the number of Trainers (n) increases from 1 to 5, 6, and 5, respectively. In the case of one Sampler (1S), the overall performance stops improving when n reaches 5 because the Sampler becomes the new bottleneck. Instead, for the case of three Samplers (3S), although the overall perfromance continues to drop, it is always slower than the case of two Samplers (2S) due to fewer Trainers when using the same number of GPUs (e.g., 1.5s for 2S3T vs. 2.1s for 3S2T). Our flexible scheduling scheme can find the optimal GPU allocation, i.e., 2 Samplers for GCN on PA (see Table 4).

Finally, we also explored the scalability of the convergence time in GNNLab. Observe that, as the number of GPUs increases, the epoch time decreases but the number of epochs required to converge also increases. Overall, when using more GPUs, the convergence time still drops, but slightly slower than the epoch time. More details on convergence time can be found in §7.7.

Table 6. The preprocessing time (in seconds) for training GCN in GNNLab. Here **G**: graph topo. data, **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 topological data	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 estimated the preprocessing cost in GNNLab. Table 6 reports the results for training GCN on four datasets. The results for other GNN models are similar (not shown here for brevity). Note that the preprocessing time includes (P1) loading graph topological and feature data (*G* and *F*) from disk to host memory (DRAM), (P2) loading *G* and feature cache ($) from DRAM to the GPU memory (GPU-mem), and (P3) pre-sampling graph data, i.e., a round of GPU-based sampling, and constructing the hotness map.

As shown in Table 6, (P1) accounts for a major part of the overall preprocessing time, and (P2) and (P3) are relatively trivial. Note that optimizing (P1) is quite easy but not the focus of our work. On average, (P2) takes 13.9× of one epoch time (see Table 4), 3.5× for loading *G* and 10.3× for loading $. (P3) is very fast, only taking about 1.4× of one epoch time. GNNLab only needs to perform (P2) and (P3) once for one GNN training task that usually takes hundreds of epochs. Therefore, the overhead of (P2) and (P3) can be amortized.

7.7 Training Convergence

For training a GNN model, readers may be more concerned with the time to converge to the expected accuracy, rather than just the execution time of one epoch. In addition, we also need to verify the correctness of our implementation. To this end, given the accuracy target, we train GraphSAGE on PA dataset using DGL, T_{SOTA} and GNNLab. As shown in Figure 16(a), all three systems can converge to the same

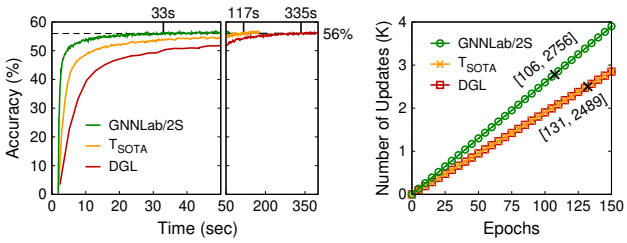

Figure 16. A comparison of (a) the end-to-end time and (b) the number of gradient updates using DGL, T_{SOTA} and GNNLab for training GraphSAGE on PA until convergence.

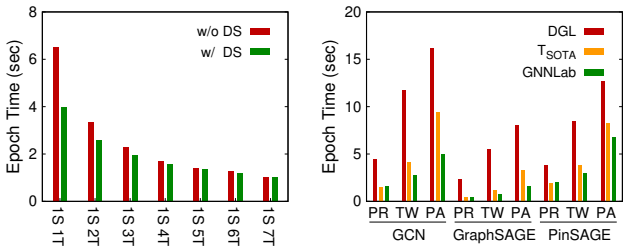

Figure 17. (a) The runtime of one epoch in GNNLab w/ and w/o dynamic switching (DS) for training PinSAGE on PA w.r.t. the number of GPUs. (b) The runtime of one epoch over a single GPU.

accuracy targets (i.e., about 56%), and GNNLab converges much faster than DGL and T_{SOTA}. The reasons are two-fold. First, for training one epoch, DGL outperforms DGL and T_{SOTA} by 8.2× and 2.8×, respectively, as shown in Table 4. Second, given a mini-batch size, the more GPUs allocated for model training, the fewer gradient updates per training epoch; therefore, more epochs are required to achieve the same expected accuracy. Compared with DGL and T_{SOTA}, GNNLab allocates fewer GPUs (i.e., 6 vs. 8) for model training due to our factored design. Thus, GNNLab achieves the expected accuracy in fewer epochs (i.e., 106 vs. 131) but more gradient updates (i.e., 2,756 vs. 2,489) than T_{SOTA} and DGL, as shown in Figure 16(b). Eventually, training GraphSAGE on PA to reach the same accuracy, GNNLab outperforms DGL and T_{SOTA} by 10.2× and 3.5×, respectively, thanks to the above two advantages.

7.8 Dynamic Switching

As mentioned previously (see §5.3), dynamic switching in GNNLab will greatly improve performance, if the number of GPUs are limited (e.g., two or even one) and/or the workload is highly skewed (e.g., Trainers consume samples much faster than Samplers produce samples). To show the benefits of dynamic switching, we train PinSAGE on PA and allocate only one GPU to Sampler. As shown in Table 5, the ratio of training time to sampling time (K) reaches 9.9, which means that if the number of GPUs allocated to Trainers less than 10. the only GPU allocated to Sampler will still be idle. As shown in Figure 17(a), for fewer Trainers (e.g., less than 3), the improvement achieved by enabling dynamic switching is significant. As Trainers increase, the workload tend to be balanced, and the improvements become limited. Note that asynchronous gradient updates are used in this experiment.

7.9 Performance on a Single GPU

The dynamic switching mechanism also allows GNNLab to work on a single GPU. GNNLab first stores all samples of an epoch in the host memory; after that the standby Trainer is launched to conduct model training. We compared GNNLab with T_{SOTA} and DGL on a single GPU. As shown in Figure 17(b), GNNLab outperforms DGL by up to 7.7× (from 1.9×) due to enabling GPU-based cache. Thanks to our caching policy (PreSC), even though transferring samples through a global queue in host memory incurs some overhead (less than 0.1 ms), GNNLab still outperforms T_{SOTA} by up to 2.0× (from 1.2×), except for PR where all graph topological and feature data can be loaded into a single GPU.

8 Discussion

We next discuss a design alternative for sample-based GNN training over multiple GPUs and some factors which may affect the efficiency of GNNLab.

Partitioning-based approach. Readers might be interested in an alternative that splits graph topological and feature data into multiple partitions and then loads them into different GPUs. We note that partitioning-based approach is orthogonal to our work and can benefit if applied to GNNLab. However, both two intuitive solutions suffer from severe problems, and we leave it as future work. One solution is to adopt cross-GPU memory access during graph sampling, which would incur considerable latency, as cross-GPU memory access is significantly slower than local memory access (e.g., 74× slower in our testbed). Another solution [35] is to ensure that each partition is *self-reliant* so that sampling on a partition can be done independently. However, it would inevitably introduce significant redundancy among partitions. Indeed, for a GNN model with L layers, a self-reliant partition is required to extend with the L-hops neighbors and edges. The redundancy largely reduces the GPU memory capacity for feature cache, especially for power-law graphs. For example, to run the 3-hop random neighborhood sampling on Twitter [34] dataset, each of eight partitions requires to include over 95% of total vertices to be self-reliant.

Mini-batch size. The mini-batch size will not affect the efficacy of our PreSC caching policy, since one epoch needs to process all training vertices, and the cached vertices are selected based on visited vertices of all training vertices. However, the mini-batch size will affect the end-to-end time of a training epoch and the time of converging to the expected accuracy. Based on our experience and recent literature [44], using a larger mini-batch can reduce the end-to-end time of one epoch, while the time to converge to the expected accuracy decreases first and then increases gradually.

Training set. The size of training set also affects the performance of GNNLab. The time of an epoch increases as the size of training set grows since more sampling and model training computations are needed. The Extract stage is more sensitive to the training set size. This is because, with a larger size, more feature data is transferred to GPU memory, exacerbating the GPU memory contention problem (see also §3). Thanks to the factor design that effectively eliminates resource contention and the PreSC caching policy with high cache hit rates, GNNLab can achieve more speed-ups over its competitors with a larger size of training set.

Other sampling algorithms. In addition to k-hop neighborhood sampling [25, 28, 64] and random walks [23, 43, 58], subgraph-based sampling algorithms [15, 60, 61] have recently gained more attention. Such algorithms become more lightweight and lead to highly skewed workloads, so that our dynamic switching scheme will be more helpful (see also §5.3). Further, some sampling algorithms may not follow the similarity of access footprint between epochs, which would limit the contribution of our PreSC caching policy. For example, ClusterGCN [15] samples all training vertices uniformly once in each epoch. However, GNNLab's factored design leaves a larger GPU-memory capacity for feature cache, which can still bring a substantial advantage.

9 Related Work

Recently, how to efficiently support GNN training at scale has attracted increasing attention [10, 31, 48, 62, 66]. Existing solutions can be roughly divided into two categories, namely whole-graph training and sample-based training.

Whole-graph GNN training. To retain scalability, whole-graph GNN training divides a large graph into multiple partitions, and trains GNN models simultaneously on all vertices/edges with multiple machines/GPUs. Typical GNN systems that fall into this category include NeuGraph [37], ROC [31], FlexGraph [51] and Dorylus [48]. In whole-graph training, each vertex needs to consider its all neighbors while different vertices may have different neighbor sizes. Thus, it is hard to use dense tensor operations to express neighborhood feature aggregation since dense tensor operations require a regular input form. To address this problem, existing systems proposed different techniques, e.g., kernel fusion in DGL [52], sparse tensor operations in PyG [17] and hybrid aggregation in FlexGraph [51], to efficiently perform neighborhood feature aggregation operations.

Sample-based GNN training. Sample-based training follows the SET model, and 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 fashion. Apart from GNNLab, typical GNN systems belonging to this category include AliGraph [67], DistDGL [66], PaGraph [35] and P^3 [20]. After the Sample stage, each vertex has a *fixed size* of neighbors, and thus neighborhood feature aggregation can be conducted efficiently by using dense tensor operations in existing tensor-based deep learning frameworks. However, most sample-based GNN systems perform the Sample and Extract stages on CPUs, and they suffer from underutilization of GPU resources. Thus, how to improve the GPU utilization has become a hot research topic in the recent past [6, 30, 40].

GPU acceleration in GNN training. GPUs have been adopted to improve both whole-graph GNN training and sample-based training. For whole-graph training, GPUs have mostly been applied to accelerate the NN and graph-related operations [31, 37, 48, 53]. For sample-based training, GPUs have been adopted to improve different stages of the SET model, 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 [30], C-SAW [40] and DGL [6]. While PaGraph [35] adopts a degree-based caching policy to accelerate feature extraction, DGL [6] uses GPUs in the Extract stage only if all feature data can be loaded in GPUs. For the Train stage, PyG [17] and DGL [6] implement highly optimized GNN runtimes on GPUs. GNNLab differs from them in the following. (i) GNNLab is the first system that adopts a factored design and various optimizations to enable GPU acceleration for all three stages. (ii) GNNLab employs an efficient GPU-based feature caching policy to improve the Extract stage, which is also more robust than PaGraph.

10 Conclusion

In this paper, we present GNNLab, a new factored system for sample-based GNN training over GPUs. Unlike existing GNN systems, which adopt the conventional time sharing design and the degree-based caching policy, GNNLab proposes a new space sharing design to eliminate GPU memory contention and a new pre-sampling based caching policy to achieve high efficiency for diverse workloads. Our experimental results confirm the advantage and efficacy of GNNLab. The source code of GNNLab is publicly available at https://github.com/SJTU-IPADS/gnnlab.

Acknowledgment

We sincerely thank our shepherd Thanumalayan Sankaranarayana Pillai and the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Key Research and Development Program of China (No. 2020AAA0108500), the National Natural Science Foundation of China (No. 61772335), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and research grants from Alibaba Group through Alibaba Innovative Research Program and Alibaba Research Intern Program.

References

[1] 2020. DGL: Deep Graph Library. https://www.dgl.ai/.

[2] 2020. Euler 2.0: A Distributed Graph Deep Learning Framework. https://github.com/alibaba/euler.

[3] 2021. Open Graph Benchmark: The MAG240M dataset. https://ogb.stanford.edu/docs/lsc/mag240m/.

[4] 2021. Open Graph Benchmark: The ogbn-papers100M dataset. https://ogb.stanford.edu/docs/nodeprop/#ogbn-papers100M.

[5] 2021. Open Graph Benchmark: The ogbn-products dataset. https://ogb.stanford.edu/docs/nodeprop/#ogbn-products.

[6] 2021. Using GPU for Neighborhood Sampling in DGL Data Loaders. https://docs.dgl.ai/guide/minibatch-gpu-sampling.html.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI'16)*. 265–283.

[8] Lada A Adamic and Bernardo A Huberman. 2000. Power-law distribution of the world wide web. *science* 287, 5461 (2000), 2115–2115.

[9] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*. 595–601.

[10] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. 130–144.

[11] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*.

[12] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*. 941–949.

[13] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–15.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*.

[15] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 257–266.

[16] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[17] Matthias Fey and Jan E. Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. (2019).

[18] Ronald Aylmer Fisher, Frank Yates, et al. 1963. *Statistical tables for biological, agricultural and medical research, edited by ra fisher and f. yates*. Edinburgh: Oliver and Boyd.

[19] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. In *Advances in neural information processing systems*. 6530–6539.

[20] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation (OSDI'21)*.

[21] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[22] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 17–30.

[23] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. 855–864.

[24] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 443–462.

[25] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS'17)*. 1025–1035.

[26] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS'20)*.

[27] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[28] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive sampling towards fast graph representation learning. *Advances in neural information processing systems* 31 (2018).

[29] Ankit Jain, Isaac Liu, Ankur Sarda, and Piero Molino. 2019. Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations. https://eng.uber.com/uber-eats-graph-learning/.

[30] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating Graph Sampling for Graph Machine Learning using GPUs. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. 311–326.

[31] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with ROC. In *Proceedings of the 3rd Machine Learning and Systems (MLSys'20)*. 187–198.

[32] Taehyun Kim, KyoungSoo Park, Changho Hwang, Peng Cheng, Youshan Miao, Lingxiao Ma, Zhiqi Lin, and Yongqiang Xiong. 2021. Accelerating GNN Training with Locality-Aware Partial Execution. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'21)*.

[33] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*.

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

[35] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. Pagraph: Scaling GNN Training on Large Graphs via Computation-aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*. 401–415.

[36] Xin Liu, Mingyu Yan, Lei Deng, Guoqi Li, Xiaochun Ye, and Dongrui Fan. 2021. Sampling methods for efficient training of graph convolutional networks: A survey. *arXiv preprint arXiv:2103.05872* (2021).

[37] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network

Computation on Large Graphs. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC'19)*. 443–458.

[38] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[39] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. 1–15.

[40] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[41] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. 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 (Usenix ATC'20)*. 307–321.

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

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

[44] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[45] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multicore and Multiprocessor Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (ISCA'07)*. 13–24.

[46] Victor Garcia Satorras and Joan Bruna Estrach. 2018. Few-Shot Learning with Graph Neural Networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*.

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

[48] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 495–514.

[49] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*.

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

[51] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. 67–82.

[52] 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. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[53] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, , and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation (OSDI'21)*.

[54] David Wentzlaff and Anant Agarwal. 2009. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 76–85.

[55] Wei Wu, Bin Li, Chuan Luo, and Wolfgang Nejdl. 2021. Hashing-Accelerated Graph Neural Networks for Link Prediction. In *Proceedings of the Web Conference 2021*. 2910–2920.

[56] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: Vertex-centric Programming for Graph Neural Networks. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. 359–375.

[57] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[58] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*. 974–983.

[59] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems* 31 (2018).

[60] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. 2020. Deep graph neural networks with shallow subgraph samplers. *arXiv preprint arXiv:2012.01380* (2020).

[61] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *Proceedings of the 8th International Conference on Learning Representations (ICLR'20)*.

[62] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-Purpose Graph Machine Learning. *Proc. VLDB Endow.* 13, 12 (2020), 3125–3137.

[63] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems* 31 (2018), 5165–5175.

[64] Qingru Zhang, David Wipf, Quan Gan, and Le Song. 2021. A biased graph neural network sampler with near-optimal regret. *Advances in Neural Information Processing Systems* 34 (2021).

[65] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2020. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[66] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *Proceedings of the 10th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3'20)*. 36–44.

[67] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. In *Proceedings of the VLDB Endowment*. 2094–2105.

A Artifact Appendix

This artifact provides the source code of GNNLab and scripts to reproduce the main experimental results from the EuroSys 2022 paper—"GNNLab: A Factored System for Sample-based GNN Training over GPUs" by J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yuan, and J. Zhou. GNNLab is a sample-based GNN training system in a single machine multi-GPU setup. To reproduce the results easily, we also provide instructions to build the software package and run all experiments in §7. Our artifact obtained the "Artifacts Available", "Artifacts Evaluated & Functional" and "Results Reproduced" badges from the Artifact Evaluation process of EuroSys 2022. The DOI of our artifact is https://doi.org/10.5281/zenodo.6347456.

Artifact repository. All the project source code including the instructions on how to build and run experiments on GNNLab and baselines is available in the following git repository: https://github.com/SJTU-IPADS/fgnn-artifacts.