# EFFICIENT DATA LOADER FOR FAST SAMPLING-BASED GNN TRAINING ON LARGE GRAPHS

Youhui Bai<sup>1</sup> Cheng Li<sup>1</sup> Zhiqi Lin<sup>1</sup> Yufei Wu<sup>1</sup> Youshan Miao<sup>2</sup> Yunxin Liu<sup>2</sup> Yinlong Xu<sup>1</sup>

# ABSTRACT

By leveraging GPU accelerators, existing frameworks combine mini-batch and sampling for training graph neural networks (GNNs) on large graphs. However, this setup faces a scalability issue since loading rich vertex features from CPU to GPU through a limited bandwidth link usually dominates the training cycle. In this paper, we propose PaGraph, a novel, efficient data loader that supports general and efficient sampling-based GNN training on single-server with multi-GPU. PaGraph significantly reduces the data loading time by exploiting available GPU resources to cache frequently-accessed graph data. It also embodies a lightweight yet effective caching policy that takes into account graph structural information and data access patterns of sampling-based training simultaneously. Furthermore, to scale out on multiple GPUs, PaGraph develops a fast GNN-computation-aware partition algorithm to avoid cross-partition access during data-parallel training and achieves better cache efficiency. Finally, it overlaps data loading and GNN computation for further hiding loading costs. Evaluations on GCN and GraphSAGE models, using the Neighbor and Layer-wise sampling, show that PaGraph could eliminate the data loading time from the GNN training pipeline, and achieve up to  $4.8 \times$  performance speedup over the state-of-the-art baselines. Together with preprocessing optimization, PaGraph further delivers up to  $16.0 \times$  end-to-end speedup. The two papers related to this submission were published at ACM SoCC 2020 (Lin et al., 2020) and IEEE TPDS 2020 (Bai et al., 2021).

# **1 PROBLEM STATEMENT**

Recently, graph neural networks (GNNs) (Zhou et al., 2018; Kipf & Welling, 2017) have been gaining popularity. Under the time and resource constraints, it would be no longer efficient or even feasible to make a full giant graph train totally as a batch. So, a typical practice is sampling (Chen et al., 2018a), which repeatedly samples subgraphs from the original graph as the input of a mini-batch, reducing the single mini-batch computation while still converging to expected accuracy.

However, the sampling-based GNN training over GPU suffers from a severe data loading problem that needs to be resolved. To understand this, we train a 2-layer GCN (Kipf & Welling, 2017) as our walk-through example over 1 to 4 GPUs. We use the widely-used neighbor sampling (NS) (Hamilton et al., 2017) and layer-wise sampling (LS) (Chen et al., 2018a) to create mini-batches of vertices for each training iteration. LS is almost identical to NS except it considers a layer as a whole and constrains the total number of sampled vertices per layer rather than per



*Figure 1.* Data loading and computation time in an epoch of training a 2-layer GCN using two sampling methods.

vertex, therefore avoiding the number of sampled vertices growing exponentially with deeper layers. As suggested by existing work (Chen et al., 2018b;a), the neighbor sampling method here selects 2 neighbors for each vertex, while the layer-wise sampling method limits the number of vertices sampled per layer to 2400. As follows, we report our major observations of performance inefficiencies and reveal their root causes, which together motivate the design of our work. We begin with the single-GPU training with the popular DGL library (DGL Team, 2019) and a GTX-1080Ti GPU. The data loading dominates the training time. Figure 1 summarizes the training epoch time over different graphs and shows the time break down into data loading and computation, w.r.t 3 large real-world graphs, livejournal (Yang & Leskovec, 2015), lj-large (Mislove et al., 2007) and enwiki (KONECT, 2017). Note that we omit the sampling overhead since sampling runs faster than and is overlapped with data loading already. Clearly, across all three graph

<sup>&</sup>lt;sup>1</sup>School of Computer Science and Technology, University of Science and Technology of China <sup>2</sup>Microsoft Research. Correspondence to: Cheng Li <chengli7@ustc.edu.cn>.

Proceedings of the First MLSys Workshop on Graph Neural Networks and Systems (GNNSys'21), San Jose, CA, USA, 2021. Copyright 2021 by the author(s).



*Figure 2.* CDF (Cumulative Distribution Function) for the access frequencies of vertices when training GCN over the wiki-talk dataset using two sampling methods.

datasets, data loading takes much longer time than computation. For example, GCN on livejournal spends 74% and 56% of the end-to-end training time on data loading when using neighbor and layer-wise sampling, respectively. This situation will become worse, when multiple GPUs are used to train a shared model collectively, since the computation time will be reduced and the data loading overhead will become more dominating.

Redundant vertex access pattern. We continue to understand the total amount of data sent from CPU to GPU for completing a training epoch. Surprisingly, the loaded data volume can be up to more than  $4 \times$  the total number of vertices of the target graphs in our experiments. This indicates that some vertices are loaded multiple times. To validate this, we collect the vertex visiting trace across different training jobs and count the number of visits per-vertex basis. Figure 2 is a CDF graph for the access frequencies of loaded vertices for training GCN over the wiki-talk graph, when using both the NS and LS Sampling method. We observe more than 32.4% of vertices have been repeatedly used by up to 519 times. This redundant vertex access pattern exacerbates the data loading burden, and creates data loading of tens of gigabytes for each epoch. We further discover that those vertices have higher out-degree than other less frequently visited or non-visited vertices. This is because a vertex with a high out-degree in a graph is likely connected with multiple train vertices, making it have chances to be selected multiple times by different mini-batches.

Based on this observation, we draw an inspiration that caching in GPU memory the feature information of vertices with high out-degrees will reduce the data loading volumes shipped from CPU to GPU, and thus accelerate the sampling-based GNN training. However, this optimization would impose a few challenges, such as contending GPU memory with GNN computation, incurring overhead for maintaining such a cache, etc.

**GPU resource underutilized.** Surprisingly, regardless of the sampling method, only a small fraction of the computing and memory resources on GPU have been utilized. For instance, with neighbor sampling, only around 20% of the GPU computational resources are in-use, with even less memory consumption, e.g., less than 10%. This is because the mini-batch data that CPU sent to GPU is not sufficient to

fully explore the hardware parallelism in GPU. In the meanwhile, the GPU is idle waiting for training data samples to arrive at most time. This observation leads us to consider to leverage spare GPU resources for caching the feature information of frequently visited vertices as possible for cheap re-usage. Theoretically, this caching solution could eliminate the amount of feature information that should gathered and sent from CPU to GPU, which consequently eliminate the CPU feature collection and PCIe data transferring bottlenecks.

High CPU contention between sampling and feature col**lection.** We then break down the time spent in different stages of the data loading process and find that the feature collection phase is CPU-intensive and takes much longer than the CPU-GPU data movement. For instance, it accounts for 50.4, 55.1, and 56.3% of the total data loading time, for the three used datasets, respectively. This surprising result leads to: (1) with a single GPU, we achieved about 8GB/s PCIe bandwidth utilization at maximum (the capacity is 16GB/s), while the average utilization is lower; (2) with multiple GPUs, concurrent workers for collecting features would contend CPU resources with samplers, e.g., the time for sampling and feature collection increased by 88% and 59% over the 1-GPU case, respectively, while the GPU computation time remains unchanged. In the 4-GPU case, the maximum PCIe bandwidth utilization drops to half and the average is even worse. This indicates that the CPU capacity cannot cope with the GPU computation demands, given the large amount of data required by each iteration. As a consequence, to reduce the feature collection cost, we have to consider to reduce the amount of data it should be gathered in this phase, as well as to isolate the resource allocation for both sampling and feature collection.

Serialized data loading and GNN computation. With the dominating library DGL, though data loading consumes CPU and PCIe, and GNN computation is scheduled to GPU, they are still executed in a serial order. DGL does not leverage to overlap the two stages in the training pipeline, mainly because the data loading dominates the whole training space and the GNN computation runs faster. However, in our work, adopting caching significantly impacts the training pipeline by reducing the data loading cost while increasing the computation density as more data samples are fed. Therefore, the new situation provide us with opportunities to pipeline the data loading and GNN computation to hide one's cost into the other's, and vice versa. This pipeline design also introduces a positive effect to improve caching efficiency when facing large graphs as it can reduce the amount of data that need to be cached in GPU memory.

# 2 PAGRAPH

Motivated by the experimental results presented in §1, as shown in Figure 3, we propose PaGraph, a novel, efficient data loader to enable fast sampling-based GNN data paral-



Figure 3. Overall architecture of PaGraph. For simplicity, directions of edges in graphs are omitted.

lel training on large graphs. We introduce three key techniques to PaGraph: 1) a GNN computation-aware caching mechanism for reducing the data loaded from CPU to GPU, 2) a cache-friendly data parallel training method to scale GNN training on multiple GPUs, and 3) a two-stage training pipeline to overlap data loading and GNN computation.

#### 2.1 GNN Computation-aware Caching

**Caching policy.** To generate better models, for each epoch, most training algorithms require a randomly shuffled sequence of training samples, which makes it impossible to predicate the vertices in each mini-batch at runtime. The neighbors of a vertex are also randomly selected during training. Therefore, it is hard to foretell which vertex is most likely to be accessed at the next mini-batch. However, due to the unique access patterns of the neighbor-sampling method, the out-degree of a vertex indicates the probability of it being selected throughout the whole epoch. This says that with a higher out-degree, a vertex is more likely to be sampled in a mini-batch. Thus, it is sufficient to fill up the cache with high out-degree vertices.

However, a dynamic caching policy is not suitable for the on-GPU *cache*. This is because GPU cannot work stand-alone, and all computations performed at GPU must be assembled into GPU kernels and launched by CPU. Most current GNNs are lightweight (Zhou et al., 2018), and hence graph data swapping between CPU memory and GPU memory has intolerable overhead during training. Therefore, instead of making on-the-fly decisions on what to be cached, e.g., LRU (Chrobak & Noga, 1999), we use static caching to avoid the overhead of dynamic data swapping. To do so, we can pre-sort vertices by out-degree offline, and select top high out-degree vertices at runtime to fill up the GPU cache. Though it is simple, as shown in §3, this static caching policy effectively achieves a high cache hit ratio.

**Cache memory space.** To avoid resource contention with the high priority training computation, we need to estimate the maximum amount of available GPU memory for the



Figure 4. New data loading flow with caching.

cache allocation. To achieve this, we leverage the fact that memory consumption is similar across training iterations. This is because the sampling-based mini-batch training uses almost the same amount of data samples as input and performs almost the same amount of computation to train a shared GNN model for each iteration. As a result, it is sufficient to decide the right cache size via a one-time sampling of GPU memory usage. In more detail, right after the first mini-batch training, we check the size of free GPU memory during training and allocate the available GPU memory for caching graph data accordingly.

**Data management.** In the GPU *cache*, we manage the cached vertex features by maintaining two separate spaces. First, we allocate consecutive memory blocks for feature data. The cached feature data of vertices is organized as several large  $[N, K_i]$  matrices, where N denotes the number of the cached vertices, and  $K_i$  is the dimension of features under the *i*-th feature-name fields. Second, to enable fast lookup, we organize the vertex meta data into a hash table to answer whether the queried vertex is cached and where it locates for later retrieval. The meta data is far less than the cached feature data, e.g., no more than 50 MB for a partition with 10 million vertices.

## 2.2 Data Parallel Training and Partition

The current design of GNN systems such as DGL, balance computation across multiple GPUs but make them share a single copy of graph data (DGL Team, 2019). When directly applying the above GNN-aware caching method to this setting, we observe a cache inefficiency phenomena, i.e., the cache hit ratio keeps decreasing with the increasing number of GPUs. This is because the single graph serves the data visiting locality for parallel Trainers on multiple GPUs, and thus all GPU caches would keep similar vertices. To address this cache inefficiency, we introduce "data parallelism" to PaGraph, which has been widely applied to leverage multiple GPUs to train neural network models efficiently. In our system, rather than accessing a shared graph, a Trainer consumes its data partition (subgraph), performs the training computation to get local gradients, and then exchanges gradients among peers to update its model replica synchronously. Clearly, the benefits of data parallelism are that data locality can be improved and the number of cached vertices in total will be increased. To make this happen, although there exist numerous graph partition algorithms (Abbas et al., 2018), we still need to design a new one to meet the following two goals specific to data

parallel GNN training. First, it should keep computation balanced across different Trainers, as unbalanced computation may result in a different number of mini-batches per epoch for different Trainers. This will break gradient synchronization and get training stuck. Second, it needs to avoid cross-partition accesses from different Trainers as possible. **Computation balance.** To achieve computation balance across different Trainers, all the partitions should have a similar number of train vertices. Assume that we need Kpartitions. We scan the whole train vertex set, and iteratively assign the scanned vertex to one of K partitions. During every iteration t, a train vertex  $v_t$  is assigned with a Kdimension score vector, where the *i*-th element represents the feasibility for assigning the vertex to the *i*-th partition for  $i \in [1, K]$ . The score is computed by Eq.(1).

$$score_{v_t}^{(i)} = |TV_i \cap IN(V_t)| \cdot \frac{TV_{avg} - |TV_i|}{|PV_i|}, \quad (1)$$

 $TV_i$  represents the train vertex set already assigned to the *i*-th partition.  $IN(V_t)$  denotes the *L*-hop in-neighbor set of train vertex  $v_t$ .  $PV_i$  controls the workload balance, and denotes the total number of vertices in the *i*-th partition, including the replicated vertices.  $v_t$  is most likely to be assigned to a partition which has smallest PV.  $TV_{avg}$  is the expected number of train vertices in the final *i*-th partition. We set  $TV_{avg}$  as  $\frac{|TV|}{K}$  so that all partitions will get almost the same number of train vertices.

Self-reliance. For the queries including the edges across different partitions, they must be forwarded to the Graph Store Server to get a full set of neighbors. Inspired by (Gonzalez et al., 2012), PaGraph introduces minimum extra vertices and edges in each partition to deal with cross-partition edges. For each partition, PaGraph extends the sub-graph with redundant vertices and edges to include all the neighbor vertices of required hops during sampling. For GNN models with L GNN layers, we will include L-hops in-neighbor vertices for each train vertex, e.g., a one-layer GNN model only requires to include direct in-neighbors of each train vertex. PaGraph only brings in necessary edges for the extended vertices to satisfy the required message flow during training. Note that the extended vertices may include train vertices. These extended train vertices are regarded as mirrors (Gonzalez et al., 2012) and will not be trained. In this way, partitions are independent from each other. Each Trainer can sample mini-batches entirely from its own subgraph without accessing the global graph structure.

#### 2.3 Pipelining Data Loading and GNN Computation

Although most real-world graphs exhibit high skewness, given that GPU memory size is often limited to 10-30 GBs, the stand-alone caching mechanism may not be sufficient for supporting GNN computation over large graphs, most of whose vertices cannot be cached. Therefore, in those cases, data loading will remain as a bottleneck. To complement caching and partitioning, we further explore the opportunity



*Figure 5.* Overview of data-loading-GNN-computation pipeline to hide the data loading overhead into the computation time. This requires us to design a new pipeline to parallelize the current mini-batch computation and the prefetching graph

data for the next mini-batch. Figure 5a shows our two-stage training pipeline design, where we break the original sequential execution into two parallel streaming executions, namely, loading and computing. We use a message queue to coordinate the execution of two streams. By taking the input from Sampler, the loading streaming executor is responsible for organizing the required feature information for the selected mini-batch vertices from both graph store and GPU cache (see Figure 5b). When loading is done, it posts a ready message including the location of batched data in GPU memory to the shared message queue. On the other hand, the computing executor sits in a loop and periodically checks the arrival of new messages from the loading executor. It pops a message from the head of the shared queue and schedules the corresponding GNN computation, which will consume the already prefetched data.

The new pipeline design leads us to further partition GPU memory into three parts for GNN computation, graph data caching, and buffering prefetched data, respectively (See Figure 5b). However, we claim that the prefetching buffer will not exacerbate the GPU memory tension due to the following reasons. First, each mini-batch data consumes less than 900 MB memory space, taking only 8% of GPU memory on the one device used in our evaluation. Second, we put a limit on the maximal number of prefetching tasks to avoid memory contention. We also use the length of the message queue as feedback to guide the loading executor to adaptively slow down or speed up prefetching.

#### **3** EVALUATION

**Experimental Setup** We run experiments on a multi-GPU server, which consists of dual Intel Xeon E5-2620v4 CPUs, 512GB DDR4 DRAM and 4 NVIDIA GTX 1080Ti (11GB memory) GPUs with no NVLink connections. It runs CentOS 7.6, CUDA v10.1, DGL v0.4, and PyTorch v1.3.

Efficient Data Loader for Fast Sampling-based GNN Training on Large Graphs

Dataset	vertex#	edge#	feature	label
reddit	232.96K	114.61M	602	41
wiki-talk	2.39M	10.04M	600	60
livejournal	4.04M	69.46M	600	60
lj-link	5.20M	103.55M	600	60
lj-large	10.69M	224.61M	400	60
enwiki	12.15M	756.28M	400	60
friendster	64.19M	2.15B	400	60

Table 1. Statistics of datasets. (K: thousand, M: million, B: Billion)

We use seven real-world graph datasets listed in Table 1 for evaluation, and adopt Neighbor Sampling (NS) and Layer-wise Sampling (LS) combined with Skip Connection (Huang et al., 2018), to train two representative GNN models, Graph Convolutional Network (GCN) and Graph-SAGE (Hamilton et al., 2017). For NS, the neighbor size ranges from 2 to 16. We set the layer size as 2400 for layer-wise sampling as suggested by FastGCN (Chen et al., 2018a). We set the training batch size as 6000 for the whole of evaluation, following DGL's suggestions. Due to space limit, we report results corresponding to neighbor sampling at most time. We observe similar trends across the two sampling methods, proving that our solution is applicable to various sampling methods. We also omit the detailed results about convergence, impacts of the loaded data volume, etc. We deploy the original DGL as the first baseline. In addition, we run an advanced DGL where the preprocessing (Chen et al., 2018b) optimization is enabled as another baseline, denoted as "DGL+PP". The idea behind preprocessing is to remove the first layer of GNN model by aggregating the corresponding features offline.

#### 3.1 Single GPU Performance

Figure 6 shows the single-GPU training performance of GCN and GraphSAGE on different datasets. Overall, Pa-Graph achieves training performance speedups from  $2.4\times$ (lj-large) to  $3.9 \times$  (reddit) for GCN and from  $1.3 \times$  (lj-large) to  $4.9 \times$  (reddit) for GraphSAGE, compared to DGL. We observe that the combination of preprocessing and DGL (DGL+PP) behaves differently across the two GNN models, i.e., the performance speedup of GCN achieved by preprocessing is better than GraphSAGE. This is due to the different forwarding procedures used in GCN and Graph-SAGE, which leads GraphSAGE to more CPU-GPU data transfer compared with GCN. In contrast, we further observe that our optimizations in PaGraph can better exploit the potential of the preprocessing optimization, other than the vanilla DGL. For instance, PaGraph+PP improves the performance of DGL+PP by  $2.1 \times$  to  $3.0 \times$  and  $1.8 \times$  to  $6.2 \times$ for both GCN and GraphSAGE across the first 6 datasets.

**Training time breakdown.** To further explore the data loading overhead reduced by PaGraph, we break down the GCN training time on both DGL and PaGraph into GPU computation time and CPU-GPU data loading time. We collect such system statistics using nvprof (NVIDIA Corporation, 2007) and PyTorch Profiler (PyTorch Team, 2017).



*Figure 6.* Single-GPU training performance of GCN (top) and GraphSAGE (bottom). (PP: preprocessing optimization).

Figure 7 shows the breakdown results corresponding to experiments related to GCN in Figure 6 (We also observe the similar treads for GraphSAGE). In this clustered bar figure, from left to right, each bar cluster presents the result of DGL, DGL+PP, PaGraph and PaGraph+PP, respectively. Consistent with the results presented in §1, DGL using the built-in data loader faces a severe loading problem. Although preprocessing saves both computation and data loading, it still suffers from the data loading bottleneck, which occupies up to 61% of training time.

In contrast to the baseline data loader, thank to the joint effort of caching and pipelining, PaGraph completely overlaps the data loading phase and the GPU computation phase, thus removing the loading cost from the training pipeline, for the original GNN model and its variant with preprocessing enabled. We find that caching stand-alone is sufficient for training the target model over the first three datasets. This is because these datasets are small and can be fully cached into GPU memory, and the data loading time becomes negligible. Unlike this, although the data loading time has already been reduced from 65.7-78.7% for the second three more massive datasets by caching, it still accounts for up to 34.9% of the whole training time. In this case, pipelining data loading and computation could completely hide the cost of the former case into the latter one. Interestingly, the computation time is slightly reduced in some cases like "PaGraph+PP" for lj-large dataset, since we carefully avoid CPU contentions among parallel training jobs. However, the computation time is slightly increased in some cases like "PaGraph" for enwiki dataset. This is because the background pre-fetching process spends some CPU cycles, which may influence the GPU kernel launching.

#### 3.2 Effectiveness of Caching Policy and Pipelining

Next, we compare our static cache policy with the policy presented in AliGraph (Zhu et al., 2019), which supports GNN training across multiple CPU machines. It reduces communication costs between training tasks and the remote storage system by caching a vertex at local if its in-degree to out-degree ratio exceeds a threshold. Since the open-source



Figure 7. Breakdown of GCN training time on single GPU. Each bar cluster from left to right represents DGL, PaGraph, DGL+PP and PaGraph+PP. "Hidden Data Loading" points to the part of data loading time fully overlapped with computation in PaGraph.



Cached Data (%) Figure 8. Cache policy comparison with different cache capacity. "Optimal" represents the ideal cache hit ratio. (Dataset: livejournal)



*Figure 9.* Training performance of PaGraph at different cached percentage. (Dataset: livejournal)

version of AliGraph did not include the cache code, and it was built atop of TensorFlow other than PyTorch and designed for CPU machines, to make a fair comparison, we implement its caching policy in PaGraph. We also compare to the random strategy, which randomly keeps vertices inside Loader. We also derive the best cache hit ratio which can be obtained if all subsequent vertex visits can be absorbed in cache, denoted "optimal". The derivation is done by analyzing the visiting trace of training.

Figure 8 shows the cache hit ratio under different cached ratio using a single GPU, respectively. We observe that when only 20% of graph are cached, we can achieve more than 50% hit ratio, which is more than 200% of the performance of other polices. More interestingly, it shows that our caching policy is not complicated, but is incredibly effective, very close to the optimal case. We also achieve the similar close-optimal cache hit ratio on other datasets. Furthermore, we explore the performance implications of the different cache hit ratios achieved by both PaGraph and AliGraph. As shown in Figure 9, as the proportion of cached graph data increases, the per-epoch training time achieved by both DGL and PaGraph keeps declining and converges to 6.7 seconds when all required data is cached. However, when the cache size is constrained, PaGraph's caching policy achieves up



*Figure 10.* GCN (left) and GraphSAGE (right) scalability of Pa-Graph and DGL on enwiki dataset. (Preprocessing enabled)

to  $1.5 \times$  performance speedup than the AliGraph's. More interestingly, when the cached percentage reaches 40%, the training performance becomes stable, and no further improvements are observed when more cache space is used. This is because at that critical point, the data loading time is reduced to be shorter than the GPU computation time, and our pipeline mechanism could completely overlap the two phases. This further reveals an additional benefit of combining both caching and pipelining, which makes our solution applicable and deliver good performance under GPU memory constraints.

With our caching policy, PaGraph achieve massive data loading reductions on large graphs, compared to DGL. Overall, we achieve 91.8%, 80.9% and 81.0% loading reductions for lj-link, lj-large and enwiki datasets, respectively.

## 3.3 Multi-GPU Performance

Figure 10 shows the experimental results of training GCN and GraphSAGE against a real-world dataset enwiki with a different number of GPUs. Both DGL and PaGraph achieves higher throughput numbers when more GPUs are in-use. However, in general, PaGraph out-performs DGL and shows better scalability w.r.t different hardware configurations, e.g., PaGraph outperforms DGL by up to  $2.4 \times (4$ GPUs for GCN). We further evaluate the performance of PaGraph on multiple GPUs to show its effectiveness on different algorithm optimizations. For this purpose, we zoom in on the performance numbers when using 2 GPUs. Pa-Graph achieves the performance of  $2.1 \times$  to  $5.6 \times$  over DGL across all datasets. With preprocessing optimization, Pa-Graph can achieve speedups from  $2.8 \times$  to  $8.5 \times$  over DGL. The reasons for the notable speedups are two-fold. First, multiple GPUs provide more available memory for caching, thus can achieve a higher cache hit ratio and lower data loading cost. To confirm this, we also test the performance of GCN on enwiki with a total cache size fixed 6 GB on four GPUs. It shows that the speedup on 4-GPU is only  $3.7 \times$ , 23% lower than the speedup achieved without the limitation of cache size. Second, in all tested cases, data loading runs faster than computation, therefore, our pipeline mechanism completely hides it into the computation.

# **4 CONCLUSIONS**

To accelerate GNN training performance on large graphs, we present PaGraph, a general sampling-based training scheme that leverages the combination of GNN computation-aware caching and graph partition.

## REFERENCES

- Abbas, Z., Kalavri, V., Carbone, P., and Vlassov, V. Streaming graph partitioning: an experimental study. In *Proceedings of the VLDB Endowment*, volume 11, pp. 1590–1603. VLDB Endowment, 2018.
- Bai, Y., Li, C., Lin, Z., Wu, Y., Miao, Y., Liu, Y., and Xu, Y. Efficient data loader for fast sampling-based gnn training on large graphs. *IEEE Transactions on Parallel* & *Distributed Systems*, (01):1–1, 2021.
- Chen, J., Ma, T., and Xiao, C. FastGCN: fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations* (*ICLR*), 2018a.
- Chen, J., Zhu, J., and Song, L. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *International Conference on Machine Learning (ICML)*, pp. 941–949, 2018b.
- Chrobak, M. and Noga, J. LRU is better than FIFO. *Algorithmica*, 23(2):180–185, 1999.
- DGL Team. DGL Large-Scale Training Tutorial. https://docs.dgl.ai/tutorials/models/ 5\_giant\_graph/2\_giant.html, 2019. accessed, Jan-2020.
- Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 17–30, 2012.
- Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems (NeurIPS), pp. 1024–1034, 2017.
- Huang, W., Zhang, T., Rong, Y., and Huang, J. Adaptive sampling towards fast graph representation learning. In Advances in Neural Information Processing Systems (NeurIPS), pp. 4558–4567, 2018.
- Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.
- KONECT. Wikipedia links, English network dataset - KONECT, April 2017. URL http: //konect.uni-koblenz.de/networks/ wikipedia\_link\_en.
- Lin, Z., Li, C., Miao, Y., Liu, Y., and Xu, Y. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 401–415, 2020.

- Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., and Bhattacharjee, B. Measurement and Analysis of Online Social Networks. In *Proceedings of the Internet Measurement Conference (IMC)*, 2007.
- NVIDIA Corporation. Nvidia nvprof. https://docs. nvidia.com/cuda/profiler-users-guide/ index.html, 2007. accessed, Jan-2020.
- PyTorch Team. PyTorch Profiler. https: //pytorch.org/tutorials/recipes/ recipes/profiler.html, 2017. accessed, May-2020.
- Yang, J. and Leskovec, J. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems (KAIS)*, 42(1):181–213, 2015.
- Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., and Sun, M. Graph neural networks: A review of methods and applications. arXiv preprint arXiv:1812.08434, 2018.
- Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., and Zhou, J. AliGraph: A Comprehensive Graph Neural Network Platform. In *Proceedings of the VLDB Endowment*, volume 12, pp. 2094–2105. VLDB Endowment, August 2019. doi: 10.14778/3352063. 3352127. URL https://doi.org/10.14778/ 3352063.3352127.