BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing

Tianfeng Liu, *Tsinghua University, Zhongguancun Laboratory, ByteDance;* Yangrui Chen, *The University of Hong Kong, ByteDance;* Dan Li, *Tsinghua University, Zhongguancun Laboratory;* Chuan Wu, *The University of Hong Kong;* Yibo Zhu, Jun He, and Yanghua Peng, *ByteDance;* Hongzheng Chen, *ByteDance, Cornell University;* Hongzhi Chen and Chuanxiong Guo, *ByteDance*

https://www.usenix.org/conference/nsdi23/presentation/liu-tianfeng

This paper is included in the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing

Tianfeng Liu*[1,4,3], Yangrui Chen*[2,3], Dan Li[1,4], Chuan Wu[2], Yibo Zhu[3], Jun He[3], Yanghua Peng[3], Hongzheng Chen[3,5], Hongzhi Chen[3], Chuanxiong Guo[3]
[1]*Tsinghua University,* [2]*The University of Hong Kong,* [3]*ByteDance,*
[4]*Zhongguancun Laboratory,* [5]*Cornell University,*

Abstract

Graph neural networks (GNNs) have extended the success of deep neural networks (DNNs) to non-Euclidean graph data, achieving ground-breaking performance on various tasks such as node classification and graph property prediction. Nonetheless, existing systems are inefficient to train large graphs with billions of nodes and edges with GPUs. The main bottlenecks are the process of preparing data for GPUs – subgraph sampling and feature retrieving. This paper proposes BGL, a distributed GNN training system designed to address the bottlenecks with a few key ideas. First, we propose a dynamic cache engine to minimize feature retrieving traffic. By co-designing caching policy and the order of sampling, we find a sweet spot of low overhead and a high cache hit ratio. Second, we improve the graph partition algorithm to reduce cross-partition communication during subgraph sampling. Finally, careful resource isolation reduces contention between different data preprocessing stages. Extensive experiments on various GNN models and large graph datasets show that BGL significantly outperforms existing GNN training systems by 1.9x on average.

1 Introduction

Graphs, such as social networks [23, 36], molecular networks [19], knowledge graphs [21], and academic networks [47], provide a natural way to model a set of objects and their relationships. Recently, there is increasing interest in extending deep learning methods for graph data. Graph Neural Networks (GNNs) [22, 36, 46] have been proposed and shown to outperform traditional graph learning methods [50, 57, 59] in various applications such as node classification [36], link prediction [56] and graph property prediction [51].

Real-world graphs can be massive. For example, the user-to-item graph on Pinterest contains over 2 billion entities and 17 billion edges with 18 TB data size [53]. As a major online service provider, we also observe over 100 TB size of graph data, which consists of 2 billion nodes and 2 trillion edges. Such large sizes make it impossible to load the entire graph into GPU memory (at tens of GB) or CPU memory (at hundreds of GB), hence turning down proposals that adopt full graph training on GPUs [55]. Recent works [23, 28, 53] have resorted to mini-batch sampling-based GNN training, aggregating neighborhood information on sampled subgraphs.

Distributed systems [2, 17, 48] for this training typically include *distributed graph store servers* to store partitioned large-scale graphs and *worker machines* where each worker has one GPU for model training. Each training iteration contains three stages: (1) *sampling subgraphs* stored in distributed graph store servers, (2) *feature retrieving* for the subgraphs from graph store servers to workers, and (3) forward and backward *computation* of the GNN model.

The first two stages, which we refer to as *data I/O and preprocessing*, are often the performance bottlenecks in such sampling-based GNN training. After analyzing popular GNN training frameworks (*e.g.,* DGL [48], PyG [17], and Euler [2]), we made two key observations. (1) High data traffic for retrieving training samples: when the sampled subgraph is stored across multiple graph store servers, there can be frequent cross-partition communication for sampling; retrieving corresponding features from the storage to worker machines also incurs large network transmission workload. (2) Modern GPUs can perform the computation of state-of-the-art GNN models [22, 36, 46] quite fast, leading to high demand for data input. To mitigate these problems, Euler adopts parallel feature retrieval; DGL and PyG prefetch the sampling results. Unfortunately, none of them fully resolves the I/O bottleneck. For example, we observe only around 10% GPU utilization in a typical DGL training job on a large graph (§2 and §5), which means around 90% of GPU cycles are wasted.

In this paper, we propose BGL, a GPU-efficient GNN training system for large graph learning, to accelerate training and achieve high GPU utilization (near 100%). Focusing on eliminating data I/O and preprocessing bottlenecks, we identify three key challenges in the existing frameworks, namely: (1) very heavy network traffic for retrieving features, (2) large

*Tianfeng Liu and Yangrui Chen contributed equally to this work as first authors.

cross-partition communication overhead during sampling, and (3) resource contention between different training stages. We address those challenges, respectively.

The biggest bottleneck of distributed GNN training systems often lies in retrieving large features (§2.3). PaGraph [38], a state-of-the-art cache design for GNN training, uses a static cache (no replacement during training) and explicitly avoids dynamic caching policy (replacing some cached features at runtime) because of high overhead. However, we find that static cache has low hit ratios when the graphs are so large that only a small fraction of nodes can be cached. Hence, we co-design a dynamic cache policy and the sampling order of nodes. We show that a FIFO policy has acceptable overhead and high hit ratios combined with our *proximity-aware ordering*. The key idea is to leverage *temporal locality* – in nearby mini-batches, we always attempt to visit the neighboring training nodes in the graph. This approach largely increases the cache hit ratio of FIFO policy. We will further explain the details of how we ensure the consistency of our multi-GPU cache engine and GNN convergence in §3.2.

After optimizing feature retrieval, the cross-partition communication for subgraph sampling could become the major performance bottleneck. Existing algorithms either do not scale to large graphs or ignore *multi-hop neighbor* connectivity inside each partition. It leads to heavy cross-partition communication because, in GNN training, the sampling algorithm usually requests *multi-hop neighbors* from a given node. Hence, we design a graph partition algorithm tailored for the typical GNN sampling algorithms. Our algorithm (in §3.3.2) strives to maintain multi-hop connectivity in each partition, while maintaining load balance partitions and scaling to giant graphs.

Finally, data preprocessing in GNN training takes multiple stages and is much more complex than that in traditional DNN training. Execution of some stages may compete for CPU and bandwidth resources, throttling the performance. Existing frameworks largely ignore it and let the preprocessing stages freely compete with each other. Unfortunately, some stages do not scale well with more resources. They may acquire more resources than they need, leading to blocking other stages. Hence, we optimize the resource allocation of data preprocessing by profiling-based resource isolation. Our key idea is to formulate the resource allocation problem as an optimization problem, use profiling to find out the resource demands of each stage, and isolate resources for each stage.

We implement BGL, including the above design points, and replace the data I/O and preprocessing part of DGL with it. The design of BGL is generic – e.g., BGL can also be used with Euler's computation backend. However, our evaluation focuses on using BGL with the DGL GPU backend because it is more mature and performant. We conduct extensive experiments using multiple representative GNN models with various graph datasets, including the largest publicly available dataset and an internal billion-node dataset. We demonstrate that BGL outperforms existing frameworks, and the geometric mean of speedups over PaGraph, PyG, DGL, and Euler is 1.91x, 3.02x, 7.04x, and 20.68x, respectively. With the same GPU backend as DGL, BGL can push the V100 GPU utilization to 99% even when graphs are stored remotely and distributedly, higher than existing frameworks. It also scales well with the size of graphs and the number of GPUs.

2 Background and Motivation

2.1 Sampling-based GNN Training

We start by explaining sampling-based GNN training.

Graph. The most popular GNN tasks [1] are to train on graphs with node features, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$, where $\mathcal{V}$ and $\mathcal{E}$ denote the node set and edge set of the graph, and $\mathcal{F}$ denotes the set of feature vectors assigned to each node. For example, in the graph Ogbn-papers [47], each node (*i.e.*, paper) has a 128-dimensional feature vector representing the embeddings of the paper title and abstract. We assume graph structures and node features are *immutable* in this paper.

Graph neural networks (GNNs). Graph neural networks are neural networks learned from graphs. The basic idea is collectively aggregating information following the graph structure and performing various feature transformations. For instance, the Graph Convolution Network (GCN) [36] generalizes the convolution operation to graphs. For each node, GCN aggregates the features of its neighbors using a weighted average function and feeds the result into a neural network. For another example, GraphSAGE [23] is a graph learning model that uses neighbor sampling to learn different aggregation functions on different numbers of hops.

Real-world graphs, such as e-commerce and social networks [13, 53, 55], are often large. The Pinterest graph [53] consists of 2B nodes and 17B edges, and requires at least 18 TB memory during training. Even performing simple operations for all nodes would require significant computation power, not to mention the notoriously computation-intensive neural networks. Similar to other DNN training tasks, it is appealing to use GPUs to accelerate GNN training.

Sampling-based GNN training. There are two camps of training algorithms adopted in existing GNN systems: *full-batch training* and *mini-batch training*. Full-batch training loads the entire graph into GPUs for training [36], like NeuGraph [40] and ROC [31]. Unfortunately, for very large graphs like Pinterest's, such an approach would face the limitation of GPU memory capacity.

Thus, we focus on the other approach, *mini-batch training*, or often called *sampling-based GNN training*. In each iteration, this approach samples a subgraph from the large original graph to construct a mini-batch as the input to neural networks. Mini-batch training is more popular and adopted by literature [11, 23, 54] and popular GNN training frameworks like DGL [48], PyG [18] and Euler [2].

[1] We focus on node classification tasks in this work.

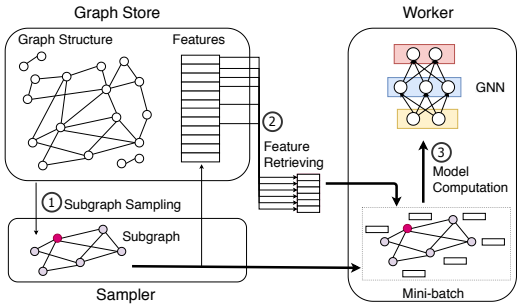


Figure 1: Sampling-based GNN training process.

The process of sampling-based GNN training is shown in Figure 1. The fixed graph data (including the graph structure and node features) are partitioned and stored in a distributed *graph store*. Multiple workers run on *worker machines*, with each worker equipped with one GPU. Each training iteration consists of three stages: ① **Subgraph sampling**: *Samplers* sample a subgraph from the original graph and send it to workers. ② **Feature retrieving:** After *workers* receive the subgraph, the features of its nodes are further retrieved from the graph store server and placed in GPU memory. ③ **Model computation:** Like typical DNN training, workers on GPU forward-propagate the prepared mini-batch through the GNN model, calculate the loss function, and then compute gradients in backward propagation. Then model parameters are updated using optimizers (*e.g.,* SGD [61], Adam [35]).

In the rest of this paper, we refer to the first two stages as *Data I/O and Preprocessing*.

2.2 Data I/O and Preprocessing Bottlenecks

Unfortunately, existing GNN training frameworks suffer from data I/O and preprocessing bottlenecks, especially when running model computation on GPUs. Here, we test two representative frameworks, DGL [48] and Euler [2]. We train GraphSAGE [23] model with one GPU worker. Using the partition algorithms of DGL and Euler, we split the Ogbn-papers graph [47] into four partitions and store them on four servers as a distributed graph store. More configuration details and the other framework results are in §5.

Figure 2 shows the training time of one mini-batch and the time breakdown of each stage. 87% and 82% of the training time were spent in data I/O and preprocessing by Euler and DGL, respectively. Long data preprocessing time leads to not only poor training performance but also low GPU utilization. The maximum GPU utilization of DGL and Euler is 15% and 5%, respectively, as shown in Figure 3.

In GNN training, such a bottleneck is much more severe than in DNN training like computer vision (CV) or natural language processing (NLP) for two main reasons.

First, due to the neighbor explosion problem [12, 54], the size of mini-batch data required by each training iteration is very large. For example, if we sample a three-hop subgraph from Ogbn-products with batch size 1,000 and fan out {15,10,5}, each mini-batch consists of 5MB subgraph structure (roughly 400,000 nodes) and 195 MB node features. Assuming that we use a common training GPU server like AWS p3dn.24xlarge [4] (8x NVIDIA V100 GPUs and 100Gbps NIC) as the worker, and that we could saturate the 100Gbps NIC pulling such data, we can only pull 60 mini-batches of data in every second.

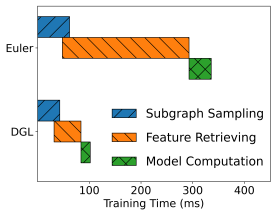


Figure 2: Training time per mini-batch of DGL and Euler.

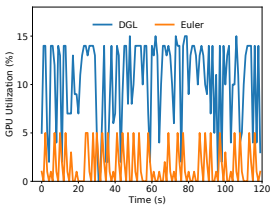


Figure 3: GPU utilization of DGL and Euler.

Second, the model sizes and required FLOPS of GNN are much smaller than classic DNN models like BERT [15] or ResNet [25]. V100 needs only 100MB and 20ms to compute a mini-batch of popular GNN models like GraphSAGE. P3dn.24xlarge can compute 400 mini-batches per second.

There is clearly a huge gap between the data I/O and preprocessing speed, and GPU computation speed. Consequently, though frameworks like DGL and Euler adopt pipelining, the data I/O and preprocessing bottlenecks can only be hidden by a small fraction and dominate the end-to-end training speed.

Some recent work [20, 29, 38] also observed this problem and made promising progress. Unfortunately, it still falls short in performance (§5) and cannot handle giant graphs well. Next, we will elaborate on the main challenges existing GNN training frameworks face.

2.3 Challenges in Removing the Bottlenecks

We identify three main challenges. Two are on large communication traffic for feature retrieving and subgraph sampling (as shown in Figure 1 and 2). The other is about resource contention when running all the stages together.

Challenge 1: Ineffective caching for node feature retrieving. As shown in Figure 2, due to the large volume of data being pulled to workers, node feature retrieval renders the biggest bottleneck. A natural idea to minimize such communication traffic is to leverage the power-law degree distribution [16] of real-life graphs. For example, PaGraph [38] adopted a static (no replacement at runtime) cache that stores the predicted hottest node features locally. Upon cache hit, the traffic of feature retrieving can be saved. Unfortunately, on giants graphs like Pinterest graph [53], such a static cache may only be able to store a small fraction of nodes due to memory constraints. We find, when only 10% of nodes can be cached, the static cache only yields <40% cache hit ratios.

Why not use dynamic (replacing some caches at runtime) cache policies? It is challenging because it would incur large searching and updating overhead, pointed out in [38]. Overheads become even larger when the cache is large (tens of GB) and stored on GPU. Our best-effort implementation

Table 1: Qualitative comparison of graph partition algorithms.

Partition Algorithms	Scalability to Giant Graphs	Balanced Training Nodes	Multi-hop Connectivity
Random [2, 30]	✓	✓	✗
METIS [32] & ParMETIS [33]	✗	✓	✓
GMiner [10]	✓	✗	✗
PaGraph [38]	✗	✓	✓

echos [42, 44] – we also find that popular policies like LRU and LFU lead to a near 80-millisecond overhead for updating.

Nevertheless, we will show in §3.2 that it is still possible to achieve a good trade-off between cache hit ratios and dynamic cache overhead by exploiting the characteristics of GNN training and carefully designing the cache engine.

Challenge 2: Need for a graph partition algorithm that is scalable and friendly to subgraph sampling. Beyond node feature retrieving, communication overhead of subgraph sampling renders another major bottleneck.

The partition algorithms affect the sampling overheads in two ways. First, they determine cross-partition communication overhead. GNN sampling algorithms construct a subgraph by sampling from a training node's *multi-hop* neighbors. If the neighbors are hosted on the same graph store server, the *sampler* colocated with graph store servers can finish sampling locally. Otherwise, it must request data from other servers, incurring a high communication overhead. Like random partitioning[2] [2, 30], naive algorithms are agnostic to the graph structure. Most state-of-the-art (SOTA) partition algorithms on graph processing and graph mining, like GMiner [10] and CuSP [26], only consider one-hop connectivity instead of multi-hop connectivity, which is suboptimal.

Second, partition algorithms determine the load balance across graph store servers and sampler processes. In a training epoch, one must iterate all *training nodes* and sample subgraphs based on them. For good load balance, one should balance the training nodes across partitions. However, SOTA graph partition algorithms only consider balancing all the nodes, of which only 10% [27, 47] are training nodes. Because they focus on maintaining neighborhood connectivity, they may produce less balanced partitions than the pure random algorithm, especially imbalanced for the training nodes.

Since we aim for GNN training on giant graphs, the partition algorithm must be scalable to giant graphs as well. Like the METIS [32, 33] used by DGL, some partition algorithms rely on maximal matching to coarsen the graph, which is not friendly to giant graphs due to high memory complexity [24]. Some other algorithms, such as PaGraph [38], have high time complexity and are not friendly to giant graphs.

Ideally, we need a partition algorithm that works on giant graphs and simultaneously minimizes the cross-partition communication and load imbalance during sampling. As shown in Table 1, none of the existing partition algorithms satisfies our needs, which motivates our algorithm (§3.3).

[2]Also including Lux [30], which is a random partition algorithm that frequently re-partitions the graph for load balancing.

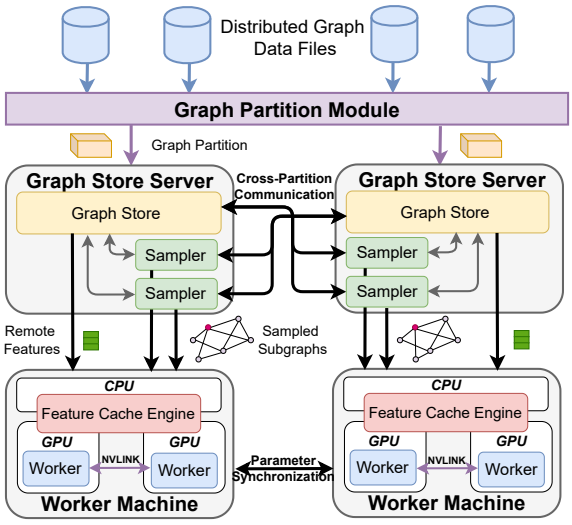


Figure 4: The architecture of BGL.

Challenge 3: Different data preprocessing stages contend for resources. When running all stages together, we further identify a unique problem of GNN training – the preprocessing is much more complex than traditional DNN training. The subgraph sampling, subgraph structure serialization and de-serialization, node feature retrieving, and cache engine all consume CPU and memory/PCIe/network bandwidth resources. We observe that if all the processes freely compete for resources, the resource contention may lead to poor performance. Some operations may try to acquire more resources than what they need and hence block other operations, while they do not scale well with more resources.

Existing GNN training frameworks largely ignore this problem. DGL, PyG, and Euler either blindly let all processes freely compete or leave the scheduling to underlying frameworks like TensorFlow and PyTorch. The low-level frameworks are agnostic to the specifics in GNN training, and thus are also naive and suboptimal. Our answer to this challenge is a carefully designed resource isolation scheme (§3.4).

3 Design

We design BGL to address the challenges presented in §2.3.

3.1 Architecture and Workflow

The overall architecture of BGL is shown in Figure 4. A training job has the following stages.

Pre-training preparation: graph partition. The *graph partition module* loads the graph data stored in the distributed storage system (*e.g.,* HDFS), and shards the whole graph into several partitions. Graph partitioning is a *one-time cost*, and the results can be saved in storage and used by other GNN training tasks later. Then, each partition is loaded into a graph store server's memory, ready for subgraph sampling.

To address Challenge 2 in §2.3, BGL's graph partition module first uses multi-source BFS to merge nodes into several blocks for reducing the graph size. Optimal graph partitioning is NP-hard [7]. Hence, we propose a partition heuristic considering both multi-hop connectivity of blocks and training workload balancing to maximize the partition locality, thus minimizing the cross-partition sampling time.

Subgraph sampling at each training step. Samplers run on the CPUs of graph store servers. They select several training nodes and sample their multi-hop neighbors by iteratively sampling next-hop neighbors several times. If all the next-hop neighbors are stored in the current graph store server, samplers can get the list locally; otherwise, they need to send network requests to other graph store servers.

Training GNN using the sampled subgraphs. Each worker in BGL runs on 1 GPU. It receives sampled subgraphs from samplers and retrieves features of subgraph nodes from graph store servers, with a local *feature cache engine* to improve the retrieving efficiency.

To address Challenge 1 in §2.3, BGL's feature cache engine adopts an algorithm-system co-design. We leverage the temporal locality — in nearby mini-batches, we always attempt to train nodes with close distance in the graph. Combined with a FIFO policy, BGL achieves high cache hit ratios and low cache overheads. Increasing the temporal locality of training nodes may influence the convergence of GNN models. We show BGL can preserve the SOTA training accuracy by carefully introducing randomness in ordering training nodes. On the system side, we exploit high-bandwidth GPU-to-GPU communication with NVLinks, and design a multi-GPU cache supporting dynamic caching strategies.

Finally, BGL uses a fine-grained pipeline, allowing parallel and asynchronous execution of each stage. To address Challenge 3 in §2.3, BGL adopts resource isolation when assigning resources to each pipeline stage. Specifically, BGL formulates an optimization problem and assigns isolated resources accordingly to minimize the execution time of each pipeline stage under resource constraints (§3.4).

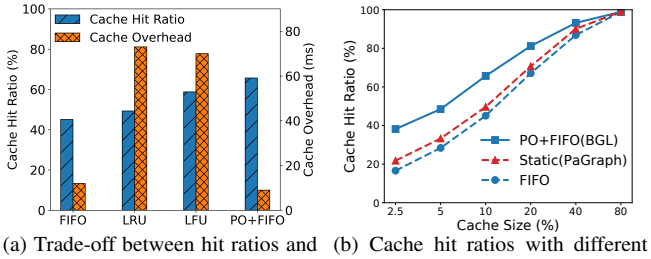
3.2 Feature Cache Engine

Feature retrieving contributes to the majority of communication overheads. We propose a feature cache engine, which uses system-algorithm co-design to minimize this overhead.

3.2.1 Dynamic Cache Policy

The first question is, which dynamic caching policy should we choose? PaGraph [38] indicates that dynamic policies have too high overheads. Based on our best-effort implementation[3], we compare popular caching policies, including LRU, LFU and FIFO in Figure 5a. Since cache queries arrive in batches, we define the cache hit ratio as the percentage of hit nodes in total number of nodes in a batch. The cache overhead is

[3]We implement LFU and LRU with $O(1)$ time complexity and use a contiguous 1D array as a HashMap to speed up key searching.



(a) Trade-off between hit ratios and overhead (10% cache size).

(b) Cache hit ratios with different cache sizes.

Figure 5: We test the cache hit ratios and overhead on Ogbn-papers with different cache sizes. PO is short for proximity-aware ordering, which is proposed in §3.2.2.

the *amortized* time, including cache lookup on all nodes *and* cache update upon cache misses. Hence, a higher cache hit ratio, representing less frequent cache updates for dynamic caching, can help reduce the amortized overhead.

LRU [42] and LFU [44] indeed have intolerable cache overhead. FIFO's overhead (<20ms per batch) meets the throughput requirement for GNN training – as mentioned in §2, an iteration of typical GNN model computation on GPU is around 20ms. In an asynchronous pipeline with cache as a part of data prefetching, FIFO cache will not become the bottleneck.

However, FIFO's cache hit ratio is unimpressive – it is even lower than static policy's (Figure 5b). The reason is that FIFO does not leverage the distribution of node features. Regardless of how hot the node feature is, it is evicted as frequently as other colder node features.

3.2.2 Proximity-Aware Ordering

To address the above problem, we propose *proximity-aware ordering* – in nearby mini-batches, we always attempt to visit the neighboring training nodes in the graph. Figure 5b shows that FIFO combined with proximity-aware ordering can achieve the highest cache hit ratio among all candidate cache policies while maintaining low cache overhead.

We observe that each node may appear more than once among different training batches (e.g., node ⑨ in Figure 6a appears three times in sampled subgraphs). This gives us an opportunity for data reuse by caching node features in nearby mini-batches (a.k.a., *temporal locality*). With random training nodes sampling, the chances of a node in nearby training batches are low. In order to increase the probability, we propose to select training nodes in a BFS order. BFS preserves the graph connectivity in terms of number of hops. Hence, nearby training nodes in graphs are more likely to be selected in consecutive batches. As a result, this ordering increases the probability that each node appears in consecutive batches and improves the cache hit ratio.

For example, in Figure 6a, starting from a BFS root node ⑰, we can generate a BFS sequence of training nodes. Random ordering (Figure 6b) results in no cache hits in the first three batches. On the contrary, with proximity-aware ordering (Figure 6c), the second batch and the third batch contain nodes that exist in the previous batches (*i.e.*, {⑰, ⑨, ③}

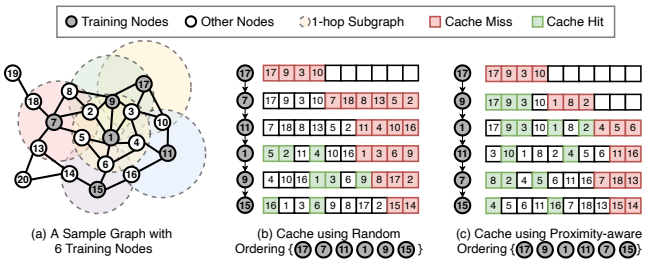


Figure 6: Compared to random ordering, using proximity-aware ordering improves hit ratios of FIFO cache.

in the second batch and {⑨, ③, ①, ②} in the third batch). Consequently, FIFO cache hits are improved from 8 to 14.

However, there is a trade-off between improving the temporal locality and ensuring model convergence. Traversal-based ordering improves the temporal locality but violates the i.i.d. requirement of SGD, leading to different label distributions of batches and slowing model convergence. On the other hand, random ordering, such as random shuffling, achieves state-of-the-art model accuracy by selecting random training nodes, with the cost of poor temporal locality.

Our proximity-aware ordering needs to balance the above trade-off. The key idea is that SGD is robust enough, and slightly relaxing the i.i.d. requirement does not influence the convergence rate. Theorem 3.15 in [41] shows that if there is *little difference* between the output distribution of one ordering algorithm A and the uniform distribution, A will not cause accuracy degradation. Hence, in BGL, samplers still select training nodes based on BFS traversal, while we carefully introduce randomness to reduce the difference.

We introduce the following randomness. First, we use several different BFS sequences, instead of only one, and each of them is generated by selecting random BFS roots. To form a training batch, we select training nodes from different sequences in a round-robin manner. Second, we circularly shift each BFS sequence by a random position. Since giant graphs have lots of small connected components [37], they are more likely to be traversed at last and appended at the end of each BFS sequence in our implementation. This deterministic behavior harms the model accuracy. Shifting by a random position minimizes its impact to the model, and circular shifting preserves the order of consecutive nodes in BFS sequences.

How many BFS sequences should we select? We find, as long as the model convergence is guaranteed, we should use the minimum number of sequences to maximize the temporal locality. Meng et al. [41] define the difference ε, named *shuffling error*, as the total variation distance between the two distributions, and proves that, if $\varepsilon \leqslant \sqrt{bM}/n$, the convergence is not influenced, where b is the batch size, M is the number of workers and n is the size of training data.

Based on the above theorem, we determine the number of sequences as follows. We use the label distribution to calculate the shuffling error. The label distribution of proximity-aware ordering is estimated as the probability of each label appearing in each mini-batch. Before training, BGL firstly generates

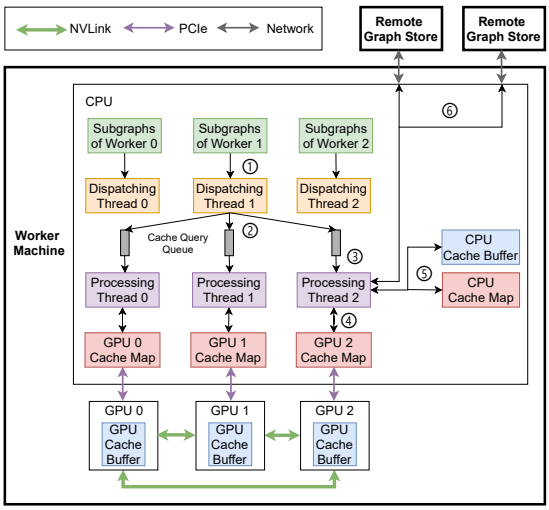


Figure 7: Structure and workflow of feature cache engine.

hundreds of BFS sequences. After that, it gradually increases the number of BFS sequences from one until the shuffling error is smaller than the requirement of convergence ($\sqrt{bM}/n$). During training, BGL constructs each training batch by introducing randomness and reusing generated sequences. This procedure incurs negligible overheads (<1% training time).

3.2.3 Maximizing Cache Size

Based on the observation that GNN models are typically small (§2.3) and large GPU and CPU memory are unused, in BGL, we jointly use the memory of multiple GPUs (if the training job uses multiple GPUs) and CPU memory to build a two-level cache, which can enlarge the cache size and increase the cache hit ratio. The detailed structure and cache workflow of our feature cache engine is shown in Figure 7.

Multi-GPU Cache. We create one *cache map* and one *cache buffer* for each GPU. Cache map is a HashMap with node IDs as keys and the pointers to buffer slots in cache buffer as values. Cache buffer contains buffer slots, storing node features. Each GPU cache map manages its own cache buffer.

To avoid wasting precious GPU memory, we ensure no duplicated entries among all GPU cache buffers by assigning different and disjoint node IDs to each GPU cache map (`mod` by the number of workers). A GPU can fetch node features from another GPU via P2P GPU memory copy using NVLinks. As mentioned in §2.2, transferring 60 mini-batches can saturate the 100Gbps NIC and PCIe 3.0 x16 bandwidth. Hence, using NVLinks not only provides high bandwidth and low latency for inter-GPU communication, but it also alleviates heavy communication in the network and PCIe links.

Since CPU memory is much larger than GPU memory, BGL also adds a CPU cache on top of the multi-GPU cache to further increase the cache size and reduce the communication traffic to graph store servers. The CPU cache uses the same cache policy as the GPU cache, so we omit the details.

Cache Workflow that Guarantees Consistency. As shown in Figure 7, the workflow of the cache engine goes as follows.

After receiving sampled subgraphs (①), dispatching threads split the subgraph nodes by `mod` operation into multiple *cache queries*[4] and send them to *cache query queues* (②). Each processing thread is assigned to one GPU cache buffer and processes all cache queries on this buffer (③). It first looks up the subgraph nodes in the GPU cache map and then gathers cached features of those nodes from GPU cache buffers (④). In case of GPU cache misses, it looks up the CPU cache map for uncached nodes, gathers cached feature tensors from CPU cache buffer, and sends them to the GPU (⑤). The remainders are requested from graph store servers and sent to GPUs once received (⑥). Finally, the cache map and the cache buffer are updated according to our FIFO caching policy.

Though node features are immutable (§2), cache buffers are still mutable. The cache buffer and the cache map may be *inconsistent* when some buffer slots are read and written by different GPU workers simultaneously (which occurs when different nodes are assigned to the same buffer slot). To ensure the consistency between the cache buffer and the cache map, a naive solution is to use locks for each buffer slot. But, this locking means synchronization in CUDA APIs for GPUs, leading to large overhead. Our solution is to queue all the operations towards a given GPU cache, including queries and updates. Only one processing thread polls the queue and then reads or writes the corresponding GPU cache buffer. This reduces the overhead by 8x compared with using locks while avoiding racing.

3.3 Graph Partition Module

3.3.1 Partition Workflow

Graph partitioning largely impacts the cross-partition communication when sampling subgraphs. As described in §2.3, a good partition algorithm should have the following properties: (1) scalability to billion-node graphs, ensuring (2) multi-hop connectivity, and (3) training load balance.

Our algorithm exploits two types of processes: *block generators* and *a block assigner*. Block generators generate blocks, each of which is a connected subgraph and treated as one node in the coarsened graph. The block assigner collects blocks of the coarsened graph from block generators and assigns each block to one partition. We outline the three major steps of our partition algorithm in Figure 8.

(1) Multi-level Coarsening: Each block generator loads disjoint graph data from HDFS and generates blocks on the loaded graph.

Different from merging procedures used in other partition algorithms (*e.g.*, maximal matching in METIS), we use multi-source BFS to generate blocks, which can preserve multi-hop connectivity in the original graph. The block generator randomly chooses a few nodes as the BFS source nodes. Each source node is assigned a unique block ID and broadcasts the block ID to its neighbors. Once the block size (*i.e.*, the number of nodes with the same block ID) exceeds a threshold (*e.g.*, 100K), or there are no unvisited neighbors in BFS, a block is generated. When all nodes are visited, the block generating procedure stops. At the same time, block generators maintain a mapping from the node ID to block ID, and synchronize it among them for uncoarsening.

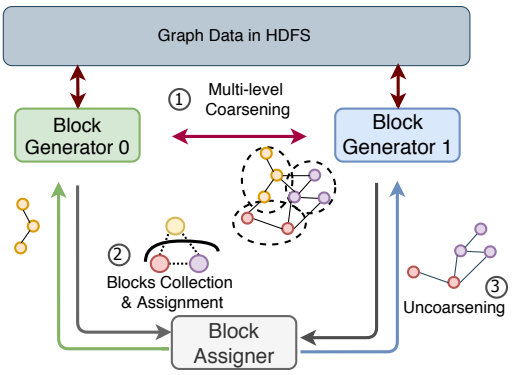


Figure 8: The partition workflow. Node colors denote different blocks in the coarsened graph (step ②), or the nodes belonging to different blocks (steps ① and ③).

However, we find billion-node graphs have numerous connected components [37]. After one round of coarsening, the coarsened graph still contains a large quantity of nodes, which results in large partition complexity. Hence, we further deploy a multi-level coarsening strategy. First, for small blocks connecting to large blocks[5], we merge them to their large block neighbors. Second, other small blocks without large block neighbors are randomly merged. By considering neighborhood relationship, this approach not only speeds up the partition process but also preserve the multi-hop connectivity.

(2) Block Collection & Assignment: The block assigner collects the blocks of the multi-level coarsened graph from block generators. It applies a greedy assignment heuristic for each block, targeting both multi-hop locality and training node balancing. The block assigner then broadcasts the block partitions to the generators. We leave the details of the assignment heuristics in §3.3.2.

(3) Uncoarsening: Upon receiving the block assignment results from the block assigner, the block generators start mapping back the blocks to the nodes in the original graph, *i.e.*, uncoarsening. The partition results are then saved to the HDFS file (step ③ of Figure 8).

As a result, our partition algorithm has low time complexity and is friendly to giant graphs. Let $\mathcal{E}_1$ be the set of edges in the coarsened block graph after BFS. $\mathcal{E}_2$ denotes the set of edges in the graph for assignment after multi-level block merging, and j denotes the number of hops to maintain connectivity. We reduce the time complexity of the assignment to $O(|\mathcal{E}_2|^j)$, much lower than SOTA $O(|\mathcal{E}|^j)$ [38], where $|\mathcal{E}_2| \ll |\mathcal{E}|$. The total partitioning complexity is $O(|\mathcal{E}| + |\mathcal{E}_1| + |\mathcal{E}_2|^j)$.

[4] A cache query contains all nodes which are assigned to one GPU cache buffer by `mod` operation in a sampled subgraph.

[5] Empirically, we set blocks with top 10% sizes as large blocks.

3.3.2 Assignment Heuristic

Since optimal graph partitioning is NP-hard [7], we propose a new heuristic for assigning blocks to partitions by considering the special requirements of GNN training.

Our heuristic is to derive the block assignments by solving the following maximization problem:

$$\max_{i\in[k]}\left\{\left(\sum_j |P(i)\cap\Gamma^j(B)|\right)\cdot\left(1-\frac{|T(i)|}{C_T}\right)\cdot\left(1-\frac{|P(i)|}{C}\right)\right\}$$

where k is the number of partitions; each partition is referred by its index $P(i)$. Based on this heuristic, each block B is assigned to the partition with the maximum value.

The first term in the heuristic is the *multi-hop block neighbor* term, $\sum_j |P(i)\cap\Gamma^j(B)|$, which counts the intersection between the set of j-hop neighbor blocks of B, $\Gamma^j(B)$, and the current partition $P(i)$. Using this term, we tend to assign the current block to a partition with the maximum number of neighbors and preserve the multi-hop connectivity. Second, we introduce the *training node penalty* term, $(1-|T(i)|/C_T)$, where $T(i)$ denotes the set of training nodes that have been assigned to the ith partition, and $C_T = |\mathcal{T}|/k$ denotes the training node capacity constraint on each partition. By maximizing this term, each partition is enforced with the same number of training nodes. Third, we introduce the *node penalty* term, $(1-|P(i)|/C)$, where $C = |\mathcal{V}|/k$ is the capacity constraint on each partition. This term is commonly used in existing partition algorithms to balance the number of nodes among the partitions. Finally, we multiply the three terms to maximize them simultaneously.

3.4 Resource Isolation For Contending Stages

To improve resource utilization and training speed, we divide GNN training into 8 asynchronous pipeline stages (see Figure 9) with careful consideration of data dependency and resource allocation. This is more complex than traditional DNN training. Some of the stages contend for CPU, Network, and PCIe bandwidth resources: (i) Processing sampling requests and constructing subgraphs compete for CPUs on graph store servers. (ii) Subgraph processing (e.g., converting graph format) and executing cache workflow compete for CPUs in the worker machine. (iii) Moving subgraphs and copying features to GPUs compete for PCIe bandwidth.

However, we find that if all the processes freely compete for resources, the resource contention may lead to poor performance. A key reason is that some operators may acquire more resources than what they actually need and block other stages, with which they do not scale well.

For example, we observe that for the executing cache workflow stage (Stage 4 in Figure 9), when the number of CPU cores exceeds a threshold (e.g., 40), the performance converges or even degrades with more CPU cores (e.g., more than 64). This is because of the memory bandwidth limit, synchronization and scheduling overhead in the multi-threading library like OpenMP [8].

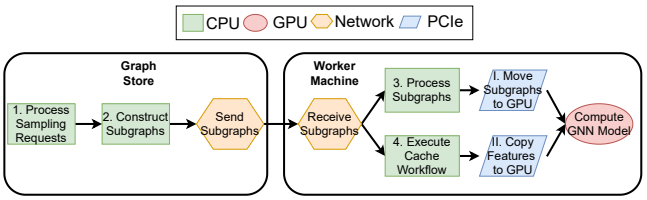


Figure 9: GNN training pipeline in BGL.

To solve the above problem, we propose a *profiling-based resource allocation* to assign isolated resources to different pipeline stages. We first profile the execution time of each stage and then adjust resource allocation to balance the execution time of each stage. We formulate the following optimization problem to compute the best resource allocation in a given GNN training task:

$$\min\ \max\left\{\frac{T_1}{c_1},\frac{T_2}{c_2},T_{net},\frac{T_3}{c_3},\frac{D_I}{b_I},f(c_4),\frac{D_{II}}{b_{II}},T_{gpu}\right\}$$
$$\text{s.t.}\quad c_1+c_2\leqslant C_{gs},\quad c_3+c_4\leqslant C_{wm},\quad b_I+b_{II}\leqslant B_{pcie}$$

The objective is to minimize the maximal completion time of all pipeline stages. The constraints are resource capacity constraints for CPU on graph store servers, CPU on worker machines, and PCIe bandwidth. The main decision variables are c_i ($i\in\{1,2,3,4\}$), the number of CPUs required for the ith stage; and b_i ($i\in\{I,II\}$), PCIe bandwidth of the ith stage. All the other quantities are profiled by our system, including the time of the ith stage T_i, the data size of processed subgraphs D_I, and the average data size of missed features D_{II} when the cache is stable. C_{gs} and C_{wm} denote the number of CPU cores on graph store servers and worker machines, respectively, and B_{pcie} is the PCIe bandwidth of the worker machines. We assume linear acceleration of CPU execution, except on processing caching operation (Stage 4 in Figure 9). We introduce a fitting function $f(c_4)=a/c_4+d$ to output the completion time of caching stage with a certain number of CPU cores c_4, where a and d are approximated by pre-running.

We use brute-force search to find the optimal resource allocation. To reduce the search space, we add integer assumptions on bandwidth variables b_I and b_{II}. The time complexity is $O(C_{gs}^2+C_{wm}^2+B_{pcie}^2)$ in the worst case. On average, our method spends less than 20ms on searching for the best resource allocation strategy for GNN training pipeline.

4 Implementation

We implement BGL with over 4,400 lines of C++ code and 3,300 lines of Python code. We reused the graph store module and GPU backend of the open-sourced Deep Graph Library (DGL v0.5 [1,48]), and utilized the graph processing module of GMiner [10] for partitioning. Our design can be applied to other GNN frameworks with little change. We are collaborating with the DGL team to upstream our implementation.

Requirement. BGL exploits NVLinks/NVSwitches for high-bandwidth low-latency cross-GPU communication for multi-GPU cache. Our measurement shows that without NVLinks, the feature cache engine retrieves cached features from other

Table 2: Datasets used in evaluation.

	Ogbn-products	Ogbn-papers	User-Item
Nodes	2.44M	111M	1.2B
Edges	123M	1.61B	13.7B
Feature Dimension	100	128	96
Classes	47	172	2
Training Set	196K	1.20M	200M
Validation Set	393K	125K	10M
Test Set	2.21M	214K	10M

GPUs via PCIe with much lower bandwidth, which could decrease throughput of BGL by 50%.

Feature Cache Engine. Cache workflow in feature cache engine contains several GPU operations, such as copying tensor from CPU memory to GPU memory and launching kernels to copy tensor from/to other GPUs. To make cache processing asynchronous, we enqueue all cache GPU operations into a *dedicated* CUDA stream, and pre-allocate dedicated CPU memory as buffers and pin these memory. Our cache engine uses CUDA Unified Virtual Addressing and enables fast GPU P2P communication on each cache processing thread. The cache processing thread enqueues a lightweight CUDA callback function into the CUDA stream, which counts the number of finished cache queries and notifies workers.

To further expedite FIFO performance, BGL uses multiple OpenMP threads to execute FIFO concurrently. We maintain an atomic `tail` shared by all threads to record the next column index of the GPU cache buffer for insertion or eviction. When inserting a new node, each thread finds the next position by atomically increasing `tail`, and the real position is `(tail+1)%buffer_size`. If this position has an old node, it evicts the old node from the GPU cache map. Since we assume node features are immutable during training, old node features are implicitly evicted by inserting new node features.

Inter-Process Communication. We use separate processes for sampling, feature retrieving, and GNN computation stages. To minimize the IPC overhead, we use shared memory to avoid unnecessary memory copy among different processes. Specifically, we use Linux Shared Memory and CUDA IPC to avoid unnecessary CPU and GPU memory copy, respectively.

5 Evaluation

5.1 Methodology

Testbed. We evaluate BGL on a heterogeneous cluster with 4 GPU servers and 32 CPU servers. The GPU server has 8 Tesla V100-SMX2-32GB GPUs (connected by NVLink v2), 96 vCPU cores, and 356GB memory. Each CPU server has 96 vCPU cores and 480GB memory. All servers are interconnected with 100Gbps Mellanox CX-5 NICs. The graph datasets are stored in HDFS.

Datasets. As shown in Table 2, we train GNNs on three datasets with different sizes, including two public graph datasets: Ogbn-products [27] and Ogbn-papers [47], as well as a proprietary web-scale graph dataset: User-Item.

GNN Models. We evaluate BGL with three representative GNN models: GCN (Graph Convolution Network) [36], GAT (Graph Attention Network) [46] and GraphSAGE [23]. We use the same model hyper-parameters as OGB leaderboards [3], e.g., 3 layers and 128 hidden neurons per layer.

Mini-batch Sampling Algorithms. In our experiments, we use Neighbor Sampling [23], which is shown to achieve comparable model performance with full-batch graph training.[6] Except for the experiment in §5.7, we set the mini-batch size to 1000, *i.e.,* each mini-batch contains 1000 sampled subgraphs and each subgraph contains one training node and its three-hop neighbors with fanout {15,10,5}.

Baselines. We use four open-sourced and widely-used GNN training frameworks as baselines for comparison[7].

- Euler [2]: Euler (v1.0) is a distributed graph learning system built atop TensorFlow [5]. We use TensorFlow's GPU backend for acceleration.
- DGL [1]: DGL is a deep learning library for graphs, compatible with multiple deep learning frameworks. We use the DGL v0.5 release (DistDGL [58]).
- PyG [17]: PyG (v1.6.0) extends PyTorch for deep learning on graphs. It contains a mini-batch loader for multi-GPU support in a single machine.
- PaGraph [38]: PaGraph is a sampling-based GNN framework with a static cache strategy on GPU, which supports multi-GPU in a single server.

Specifically, PyG co-locates graph store servers and workers and allows graph sampling on the same machine only, making it unable to process large graph datasets (i.e., Ogbn-papers and User-Item) due to memory limit. Hence, we only compare BGL with PyG on Ogbn-products dataset. When training on User-Item dataset with DGL and PaGraph, we separate the graph store servers from the workers since our GPU servers do not have enough memory to load the graph partitions. To evaluate the performance boundary, we use 4, 8 and 32 CPU-based graph store servers for all frameworks on Ogbn-products, Ogbn-papers and User-Item respectively.

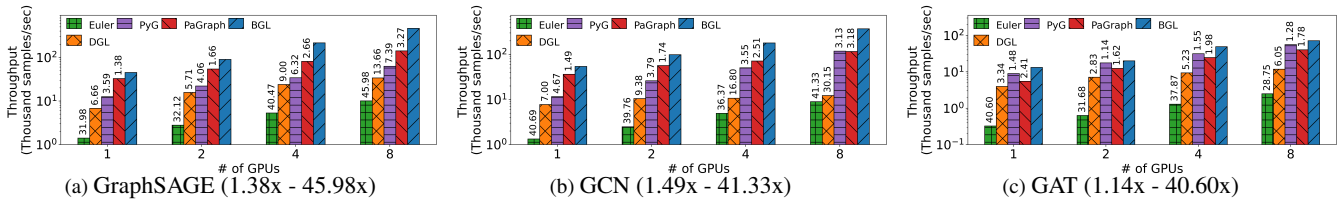
Graph Partitioning. DGL uses METIS partitioning for small graphs (*i.e.*, Ogbn-products), and Random partitioning for large graphs that cannot be fitted into a single machine (*i.e.*, Ogbn-papers and User-Item). Euler uses random partitioning for all graphs, and BGL uses the proposed algorithm in §3.3, where we set $j = 2$, *i.e.*, searching two-hop neighbors.

5.2 Overall Performance

Figure 10, 11 and 12 show the training speed of baselines and BGL in *log* scale when training the three GNN models

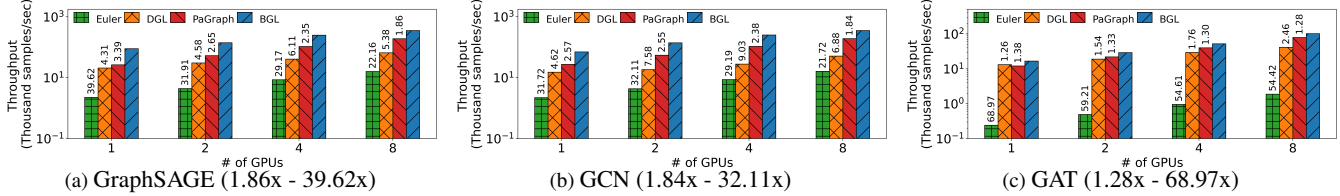
[6] BGL can also be applied to other vertex-centric GNN sampling algorithms, e.g., layer-wise sampling [11] and random walk sampling [53]. We omit the evaluation of other sampling algorithms since it is beyond our scope.

[7] We omit P^3 [20] because it is not open-sourced.



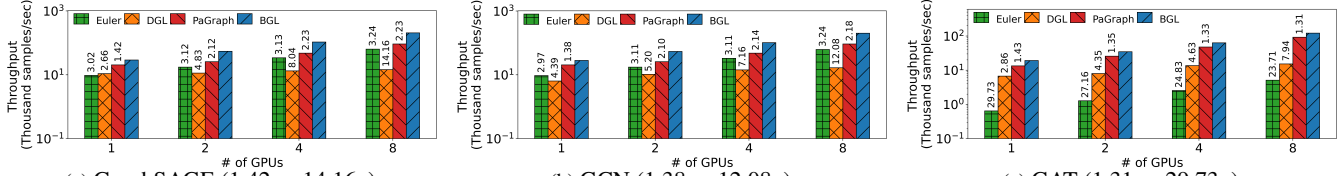
(a) GraphSAGE (1.38x - 45.98x) (b) GCN (1.49x - 41.33x) (c) GAT (1.14x - 40.60x)

Figure 10: Throughput of 3 GNN models on Ogbn-products in *log* scale. Numbers above bars are speedups of BGL over other systems.



(a) GraphSAGE (1.86x - 39.62x) (b) GCN (1.84x - 32.11x) (c) GAT (1.28x - 68.97x)

Figure 11: Training throughput of 3 GNN models on Ogbn-papers in *log* scale. Numbers above bars are speedups of BGL over other systems.



(a) GraphSAGE (1.42x - 14.16x) (b) GCN (1.38x - 12.08x) (c) GAT (1.31x - 29.73x)

Figure 12: Training throughput of 3 GNN models on User-Item in *log* scale. Numbers above bars are speedups of BGL over other systems.

on three graph datasets, with the number of workers ranging from 1 to 8, where each worker has one GPU. We use *samples/sec* as the metric to measure the training speed. A sample is a sampled subgraph of one training node.

Different Frameworks. BGL achieves 1.14x - 69x speedups over four baselines in all settings. BGL has 69x (the most) speedup over Euler. This is because Euler's random sharding in graph partition has very low data locality, resulting in frequent cross-partition communication in sampling. DGL does not cache features on GPU, introducing significant feature retrieving time. Thus, BGL outperforms DGL by up to 30x. PaGraph performs the best among baselines. It places graph structure data on each GPU with static caching on node features, leading to much faster data preprocessing. Even in this case, BGL still has up to 3.27x speedup, thanks to dynamic feature caching and resource isolation for contending pipeline stages. BGL outperforms all other systems, and the geometric mean of speedups over PaGraph, PyG, DGL and Euler is 1.91x, 3.02x, 7.04x and 20.68x, respectively.

Different GNN models. The training performance varies significantly across different GNN models. We see that BGL achieves significantly higher performance improvement with GraphSAGE and GCN models, by up to 30x as compared to DGL and PyG. With the computation-intensive GAT model, however, the training speed of PyG and DGL is closer to that of BGL. Hence the gain for BGL ranges from 14% to 8x. It is because the GAT model is computation-bound due to incorporating the attention mechanism into the propagation step, while its communication is less intensive than the other two GNN models; the higher ratio of computation over other stages results in a smaller improvement space for BGL. We see that Euler performs the worst in GAT, since it does not optimize the GPU kernels for irregular graph structures.

Scalability. BGL also outperforms other frameworks in terms of scalability. Without caching features on GPU, the throughput of baseline frameworks is bounded by PCIe bandwidth. For example, DGL has only 3x speedups when increasing the number of GPUs from 1 to 8. BGL reduces the transmitted data through PCIe bandwidth with efficient GPU cache, resulting in linear scalability in throughput. Multi-GPU systems often suffer poor scalability due to synchronization overhead or resource contention. However, our design and implementation of multi-GPU memory sharing scales well with the increased number of GPUs. With extra bandwidth brought by NVLink, accessing cache entries on other GPUs introduces negligible overhead. On the contrary, the increased cache capacity improved the cache hit ratio (Figure 5b) and reduced overall feature retrieving time (Figure 13).

We observe the relatively lower improvement with the User-Item dataset. On the billion-node graph dataset, the subgraph sampling and feature retrieving becomes more time consuming, due to the inconsistent sampling performance of DGL graph store server and sparse graph structure. Hence, BGL cannot produce the similar level of overlapping with the unchanging model computation time.

GPU Utilization. We compare the GPU utilization achieved by BGL and DGL with the same GPU backend. We run GraphSAGE and GAT models on Ogbn-products dataset with 8 GPU. BGL achieves 99% GPU utilization with the computation-intensive GAT model, while DGL's utilization is only 38%. For GraphSAGE model with shallow neural layers, BGL improves the GPU utilization from 10% to 65%.

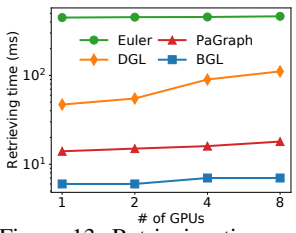


Figure 13: Retrieving time per mini-batch on Ogbn-papers.

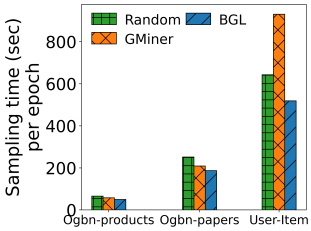


Figure 14: Graph sampling time per epoch during training.

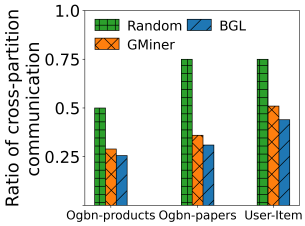


Figure 15: BGL reduces ratio of cross-partition communication.

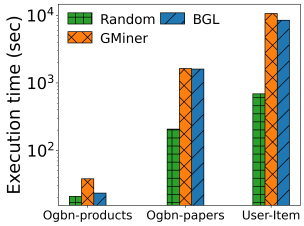


Figure 16: One-time partitioning execution time before training.

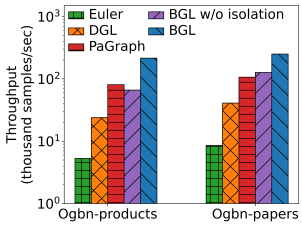


Figure 17: Resource isolation improves training throughput.

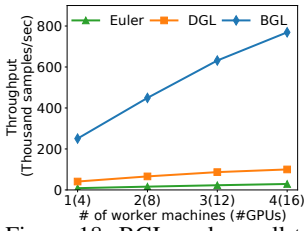
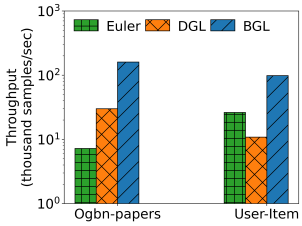
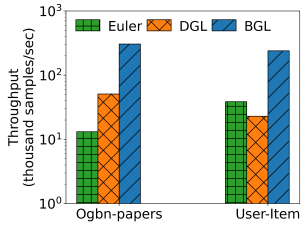


Figure 18: BGL scales well to multiple worker machines.



(a) BS 1000, 3 hops, FO {10,10,10}.



(b) BS 500, 2 hops, FO {10,25}.

Figure 19: Training throughput of GraphSAGE using different hyper parameters on 4 GPUs. BS and FO stand for 'batch size' and 'fanout'.

5.3 Impact of Feature Cache Engine

In §3.2, we have shown the cache hit ratio with different cache policies and cache sizes. The trend of them is similar on other datasets. Here, we present the amortized feature retrieving time with the feature cache engine.

We compare the feature retrieving time of one mini-batch using different GPUs on Ogbn-papers. We implement PaGraph static caching policy in BGL, which caches the features of high-degree nodes. Euler and DGL do not have cache, so the feature retrieving time is the elapsed time of transmitting features from graph store servers to GPU memory. As shown in Figure 13, due to high cache hit ratios and low cache overhead, the feature retrieving time of BGL is the shortest among all systems. Compared to other systems on 1 GPU worker, BGL reduces the feature retrieving time by 98%, 88% and 57% for Euler, DGL and PaGraph, respectively.

5.4 Impact of Graph Partition

We compare the graph partition algorithm in BGL with Random and GMiner partitioning, since only these two partition algorithms can scale to Ogbn-papers and User-Item. We evaluate the sampling time per epoch and the one-time partition time (counted from loading the graph data to saving the partition results to files) under different partition algorithms. Ogbn-products, Ogbn-papers and User-Item are divided into 2, 4 and 4 partitions, respectively.

Figure 14 shows the graph sampling time (per epoch) under different partition algorithms. BGL achieves the best performance across different graph datasets, reducing the sampling time by at least 20% over Random partition algorithm. Compared to GMiner, BGL manages to drop the sampling time by 14% and 10% for Ogbn-products and Ogbn-papers, respectively, thanks to its training node balancing and multi-hop connectivity of partitioning.

The reduction in sampling time mainly comes from the reduced cross-partition (inter-server) communication during distributed neighbor sampling. As shown in Figure 15, by including multi-hop locality when partitioning, BGL reduces the ratio of cross-partition communication by 25%, 44%, and 33% for Ogbn-products, Ogbn-papers and User-Item, respectively. The cross-communication traffic is only determined by the number of partitions, but not the number of graph store servers or worker machines.

Partitioning a large-scale graph is time consuming. Hence, BGL introduces multi-level coarsening to mitigate the extra complexity brought by computing multi-hop locality. Figure 16 shows BGL's partition algorithm runs as fast as the well-optimized original GMiner, and is even better than GMiner on graph User-Item with 20% reduction of time.

5.5 Impact of Resource Isolation

To evaluate the effectiveness of our resource isolation mechanism, we compare BGL with Euler, DGL, PaGraph, and BGL without resource isolation when training GraphSAGE with 4 GPUs on datasets Ogbn-products and Ogbn-papers. 'BGL w/o isolation' is a naive resource allocation method that shares all pipeline stages resources. It increases resource utilization but incurs larger contention and parallel overhead.

As shown in Figure 17 (in log scale), BGL achieves the highest throughput. Both BGL and 'BGL w/o isolation' outperform Euler and DGL. Due to the overhead of resource contention, the performance of 'BGL w/o isolation' on Ogbn-products is even lower than that of PaGraph. BGL uses resource isolation method, which mitigates the resource contention among different pipeline stages and incurs a lower parallel overhead of OpenMP. As a result, BGL speedups the throughput by up to 2.7x, compared to the naive resource allocation strategy without isolation and PaGraph.

5.6 Scalability to Multiple Worker Machines

To show the scalability of multiple worker machines, we vary the number of worker machines from 1 to 4, and each has 4 GPUs. We train GraphSAGE model on graph Ogbn-papers with Euler, DGL and BGL. The number of graph store servers remains the same as in §5.2.

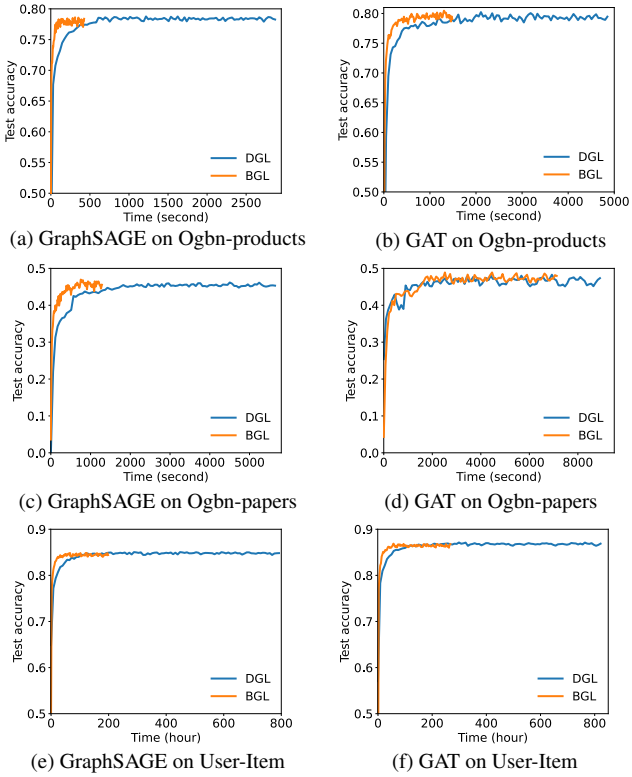
As shown in Figure 18, BGL improves throughput from 250K to 769K (76% of linear scalability) when the number of worker machine increases from 1 to 4. Due to no feature cache on GPU and bottleneck in PCIe and network bandwidth, throughput of Euler and DGL cannot scale well when increasing the number of worker machines. Since our GPU servers only use NVLink v2, the cache engine cannot share GPU memory across machines, and BGL's throughput increases slightly slower than linear scaling.

5.7 Impact of Hyper Parameters

To verify the robustness of BGL, we evaluate training speedup under different hyperparameters (batch size, number of layers and fanouts). As shown in Figure 19, we use another two widely-used training settings in OGB leaderboards [3]. We train GraphSAGE on graph Ogbn-papers and User-Item with 4 GPUs. BGL outperforms DGL and Euler as well. The geometric mean of speedup of BGL for Euler and DGL is 10.44x and 7.50x, respectively. The computation of 2-layer GraphSAGE is faster than that with 3 layers. Hence, throughput of three systems in Figure 19b is higher than in Figure 19a.

5.8 Model Accuracy

To verify the correctness of BGL, we evaluate the test accuracy on GAT and GraphSAGE with Ogbn-products, Ogbn-papers and User-Item. Each task is trained with 100 epochs for convergence. DGL uses RO while BGL uses PO. As shown in Figure 20, BGL converges to almost the same accuracy as the original DGL but the convergence of BGL is much faster.



(a) GraphSAGE on Ogbn-products (b) GAT on Ogbn-products

(c) GraphSAGE on Ogbn-papers (d) GAT on Ogbn-papers

(e) GraphSAGE on User-Item (f) GAT on User-Item

Figure 20: BGL achieves the same accuracy as DGL, using 1 GPU.

6 Related Work

Graph Partition Algorithms. Graph partitioning is widely adopted when processing large graphs. NeuGraph [40] leverages the Kernighan-Lin [34] algorithm to partition graphs into chunks with different sparsity levels. Cluster-GCN [12] constructs the training batches based on the METIS [32] algorithm, together with a stochastic multi-clustering framework to improve model convergence. When dealing with large graphs in distributed GNN training, partition algorithms, such as Random [2, 30, 39], Round-Robin, and Linear Deterministic Greedy [6], are often used [2, 48, 55, 60]. They incur low partitioning overhead while not ensuring partition locality.

GNN Training Frameworks. In recent years, new specialized frameworks have been proposed upon existing deep learning frameworks to provide convenient and efficient graph operation primitives for GNN training [2, 17, 40, 48, 60]. Other than DGL [48], Euler [2] and PyG [17], NeuGraph [40] translates graph-aware computation on dataflow and recasts graph optimizations to support parallel computation for GNN training. However, it can only train small graphs on multi-GPUs in a single machine. AliGraph [60] is a GNN system that consists of distributed graph storage, optimized sampling operators and runtime to support both existing GNNs and in-house developed ones for different scenarios. AGL [55] is a scalable and integrated GNN system implemented on MapReduce [14] that guarantees good system properties. However, neither Aligraph nor AGL exploits GPU acceleration.

GNN Training Acceleration. Various systems have been devoted to improving GNN training performance.

Some works [9, 31, 40, 45, 49] target full-batch training. GNNAdvisor [49] explores the GNN input properties and proposes a 2D workload management and specialized memory customization for system optimizations. DGCL [9] proposes a communication planning algorithm to optimize GNN communication among multiple GPUs with METIS partition. Both projects assume graphs are stored in a single machine.

Some works [20, 38, 60] target mini-batch training. PaGraph [38] adopts static GPU caching for high-degree nodes. GNNLab [52] proposes a pre-sampling-based static caching policy. They assume that a graph can be loaded in a single machine, making them infeasible for billion-node graphs.

P^3 [20] reduces retrieving feature traffic by combining model parallelism and data parallelism. However, hybrid parallelism in P^3 incurs extra synchronization overhead. Its performance suffers when hidden dimensions exceed 128 (a com-

mon practice in modern GNNs). Further, P^3 overlooked the subgraph sampling stage, where random hashing partitioning leads to extensive cross-partition communication.

Some works try to improve graph sampling performance on GPUs, such as NextDoor [29] and C-SAW [43]. However, their performance is limited by small GPU memory. Hence, they are not suitable for giant graphs.

7 Conclusion

We present BGL, a GPU-efficient GNN training system for large graph learning that focuses on removing the data I/O and preprocessing bottleneck to achieve high GPU utilization and accelerate training. To minimize feature retrieving traffic, we propose a dynamic feature cache engine with proximity-aware ordering, and find a sweet spot of low overhead and high cache hit ratio. BGL employs a novel graph partition algorithm tailored for sampling algorithms to minimize cross-partition communication during sampling. We further optimize the resource allocation of data preprocessing using profiling-based resource isolation. Our extensive experiments demonstrate that BGL significantly outperforms existing GNN training systems by 1.91x on average. We will open-source it in the future and hope to continue evolving it with the community.

Acknowledgement

We are thankful to the anonymous NSDI reviewers and our shepherd, Ying Zhang, for their constructive feedback. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800800, Hong Kong Innovation and Technology Commission's Innovation and Technology Fund (Partnership Research Programme with ByteDance Limited, Award No. PRP/082/20FX), the National Natural Science Foundation of China under Grant U21B2022, Tsinghua University-China Mobile Communications Group Co.,Ltd. Joint Institute, and grants from Hong Kong RGC under the contracts HKU 17204619, 17208920 and 17207621.

References

[1] Deep Graph Library (DGL). https://github.com/dmlc/dgl, 2020.

[2] Euler. https://github.com/alibaba/euler, 2020.

[3] OGB Leaderboards. https://ogb.stanford.edu/docs/leader_nodeprop/, 2020.

[4] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2021.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[6] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming Graph Partitioning: An Experimental Study. *VLDB Endow.*, 11(11):1590–1603, 2018.

[7] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006.

[8] J Mark Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proc of 1st European Workshop on OpenMP*, volume 8, page 49. Citeseer, 1999.

[9] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 130–144, 2021.

[10] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Proc. of the 13th ACM European Conference on Computer Systems (EuroSys)*. ACM, 2018.

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

[12] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proc. of the 25th ACM International Conference on Knowledge Discovery & Data Mining (KDD)*, 2019.

[13] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-Scale. *In Proc. of VLDB Endow.*, 2015.

[14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On Power-Law Relationships of the Internet Topology. *ACM SIGCOMM computer communication review*, 29(4):251–262, 1999.

[17] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. *CoRR*, abs/1903.02428, 2019.

[18] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[19] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein Interface Prediction using Graph Convolutional Networks. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

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

[21] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. Knowledge Transfer for Out-of-Knowledge-Base Entities : A Graph Neural Network Approach. In *Proc. of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[22] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

[23] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[24] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis S. Liu, Georgios Theodoropoulos, and Wentong Cai. Distributed edge partitioning for trillion-edge graphs. *Proc. VLDB Endow.*, 12(13):2379–2392, 2019.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[26] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 439–450. IEEE, 2019.

[27] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR*, abs/2005.00687, 2020.

[28] Wen-bing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive Sampling Towards Fast Graph Representation Learning. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[29] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating graph sampling for graph machine learning using gpus. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 311–326. ACM, 2021.

[30] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-gpu system for fast graph processing. *Proc. of the VLDB Endowment*, 11(3):297–310, 2017.

[31] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proc. of Machine Learning and Systems (MLSys)*, 2020.

[32] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[33] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distributed Comput.*, 48(1):71–95, 1998.

[34] Brian W Kernighan and Shen Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proc. of the 3rd International Conference on Learning Representations (ICLR)*, 2015.

[36] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proc. of the 5th International Conference on Learning Representations ICLR*, 2017.

[37] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[38] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. PaGraph: Scaling GNN Training on Large

Graphs via Computation-Aware Caching. In *Proc. of ACM Symposium on Cloud Computing (SOCC)*, 2020.

[39] Tianfeng Liu and Dan Li. Endgraph: An efficient distributed graph preprocessing system. In *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10 - 13, 2022*. IEEE, 2022.

[40] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proc. of USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[41] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 337:46–57, 2019.

[42] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[43] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. C-SAW: a framework for graph sampling and random walk on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 56. IEEE/ACM, 2020.

[44] Ketan Shah, Anirban Mitra, and Dhruv Matani. An o (1) algorithm for implementing the lfu cache eviction scheme. *no*, 1:1–8, 2010.

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

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

[47] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. Microsoft Academic Graph: When Experts Are Not Enough. *Quantitative Science Studies*, 1(1):396–413, 2020.

[48] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR*, abs/1909.01315, 2019.

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

[50] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A Comprehensive Survey on Graph Neural Networks. *CoRR*, abs/1901.00596, 2019.

[51] Sijie Yan, Yuanjun Xiong, and Dahua Lin. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. In *Proc. of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[52] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based GNN training over gpus. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 417–434. ACM, 2022.

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

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

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

[56] Muhan Zhang and Yixin Chen. Link Prediction Based on Graph Neural Networks. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[57] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep Learning on Graphs: A Survey. *CoRR*, abs/1812.04202, 2018.

[58] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv preprint arXiv:2010.05337*, 2020.

[59] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph Neural Networks: A Review of Methods and Applications. *arXiv preprint arXiv:1812.08434*, 2018.

[60] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. AliGraph: A Comprehensive Graph Neural Network Platform. *VLDB Endow.*, 12(12):2094–2105, 2019.

[61] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized Stochastic Gradient Descent. pages 2595–2603. Curran Associates, Inc., 2010.