XGNN: Boosting Multi-GPU GNN Training via Global GNN Memory Store

Dahai Tang
Hunan University
seatang@hnu.edu.cn

Jiali Wang
Shanghai Jiao Tong University
wangjiali@sjtu.edu.cn

Rong Chen
Shanghai Jiao Tong University
rongchen@sjtu.edu.cn

Lei Wang
Alibaba Group
jiede.wl@alibaba-inc.com

Wenyuan Yu
Alibaba Group
wenyuan.ywy@alibaba-inc.com

Jingren Zhou
Alibaba Group
jingren.zhou@alibaba-inc.com

Kenli Li
Hunan University
lkl@hnu.edu.cn

ABSTRACT

GPUs are commonly utilized to accelerate GNN training, particularly on a multi-GPU server with high-speed interconnects (e.g., NVLink and NVSwitch). However, the rapidly increasing scale of graphs poses a challenge to applying GNN to real-world applications, due to limited GPU memory. This paper presents XGNN, a multi-GPU GNN training system that fully utilizes system memory (e.g., GPU and host memory), as well as high-speed interconnects. The core design of XGNN is the Global GNN Memory Store (GGMS), which abstracts underlying resources to provide a unified memory store for GNN training. It partitions hybrid input data, including graph topological and feature data, across both GPU and host memory. GGMS also provides easy-to-use APIs for GNN applications to access data transparently, forwarding data access requests to the actual physical data partitions automatically. Evaluation on various multi-GPU platforms using three common GNN models with four large-scale datasets shows that XGNN outperforms DGL, Quiver and DGL+C by up to 7.9× (from 2.3×), 15.7× (from 3.3×) and 2.8× (from 1.3×), respectively.

PVLDB Reference Format:
Dahai Tang, Jiali Wang, Rong Chen, Lei Wang, Wenyuan Yu, Jingren Zhou, and Kenli Li. XGNN: Boosting Multi-GPU GNN Training via Global GNN Memory Store. PVLDB, 17(5): 1105 - 1118, 2024.
doi:10.14778/3641204.3641219

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lixiaobai09/xgnn.

1 INTRODUCTION

Graph Neural Networks (GNNs), as a family of deep learning, have recently achieved convincing performance on graph data and have been successfully used in recommendation systems on e-commerce platforms [52, 57], social network mining [14], drug discovery [40], fraud detection [31], amongst others. GNNs extract the node relationship in the graph into a low-dimension vector (i.e., embeddings), which can be useful in downstream tasks, such as node classification [18, 44] and link prediction [35, 54]. The learning process of GNNs can be described as *message passing* [16], where each node first aggregates information from its neighboring nodes before updating its embeddings through a traditional neural network.

Table 1: Comparison of execution time (in seconds) of different GNN systems on three GPU platforms for training GraphSAGE on Twitter. Sym. and Asym. represent symmetric and asymmetric GPU communication topologies, respectively. "×" means the system fails to train the model.

System	1-GPU	4-GPU (Sym.)	8-GPU (Asym.)
DGL	2.95	1.22	0.63
GNNLab	1.46	0.29	0.18
Quiver	6.45	0.92	0.54
WholeGraph	×	0.51	×
XGNN	0.92	0.18	0.08

To accelerate GNN training without sacrificing accuracy, GNN systems usually use a subset of neighbors for mini-batch training, known as sample-based GNN training [10, 12, 18, 52, 53]. Sample-based GNN training consists of two stages: sampling and training. In the sampling stage, for each training node, a set of k-hop neighbors is selected from the input graph topological data using various sampling algorithms. In the training stage, the selected nodes and their features are used as the input to train the GNN model with k layers.

GPUs are widely used to accelerate GNN training nowadays [19, 21], especially on a multi-GPU server with high-speed interconnects (e.g., NVLink and NVSwitch). However, the rapidly growing scale of graphs is an obstacle to applying GNN to real-world applications. There can be billion to trillion of edges in graphs in industry [7, 37]. This causes the total footprint of the graph topological and feature data to exceed GPU total memory, even cannot only store the graph topologies in the GPU. A common solution is to let each GPU store graph topological data to accelerate sampling and cache frequently accessed features in GPU memory [28, 47, 51]. However, this leads to inefficiency GPU memory usage due to redundant graph topological data and cached features across GPUs, and underutilization of GPU interconnects. Recent works [42, 49]

Proceedings of the VLDB Endowment, Vol. 17, No. 5 ISSN 2150-8097.
doi:10.14778/3641204.3641219

have noticed these issues and tried to improve GNN training for large-scale graphs by partitioning data into multiple GPUs and accessing data partitions through NVLinks. However, these systems (i.e., DGL, GNNLab, Quiver, WholeGraph) either have low performance or cannot run on diverse GPU communication topologies as shown in Table 1. Various communication topologies make it harder to utilize resources well on GPU platforms, both multi-GPU memory and interconnects. It is a challenge to efficiently utilize GPU memory and host memory through fast GPU interconnects and improve the system generality on diverse GPU communication topologies.

We therefore propose XGNN for large-scale GNN training with multiple GPUs on a single machine. The core design of XGNN is the Global GNN Memory Store (GGMS), which abstracts underlying resources to provide a unified memory store for both graph topological and feature data over multiple GPUs and CPUs. GGMS opens an opportunity to improve GPU memory and interconnects utilization efficiency systemically while shielding details of complex multi-GPU platforms. During system initialization, the graph topological and feature data are partitioned across multiple devices. During GNN training, users can access graph topological and feature data transparently by calling three exposed APIs. The GGMS automatically forwards the data access request to the actual physical data partitions.

GGMS consists of three modules, i.e., Data Access Manager (DAM), Placement Solver, and Data Partitioner. To shield off different GPU communication topologies, GGMS employs both *logical* and *physical* data partitions. After Data Partitioner partitions the graph topological and feature data into several logical data partitions, Placement Solver uses a pruning search algorithm to store the logical data partitions on GPUs (i.e., physical data partitions) based on the GPU network topology. Finally, Data Access Manager facilitates data access between multiple devices and forwards data access requests during training runtime.

GGMS is implemented as a function library for XGNN. To mitigate random remote memory access overhead during sampling, XGNN optimizes the KHop algorithm by leveraging GPU memory hardware features. This reduces the volume of remote memory access. To improve feature extracting performance, XGNN uses GPUs to extract features directly, which are stored in remote GPUs or host memory [32, 49]. We evaluate XGNN by training three common GNN models (i.e., GCN [24], GraphSAGE [18] and PinSAGE [52]) on four large-scale graphs. Experimental results show that XGNN outperforms DGL, Quiver and DGL+C by up to 7.9× (from 2.3×), 15.7× (from 3.3×) and 2.8× (from 1.3×), respectively. To show its generality, we further evaluate XGNN with large-scale graphs on various multi-GPU platforms.

Contributions. We summarize our contributions as follows.

(1) We provide a category and in-depth analysis of existing GNN training systems and identify issues with efficient GNN training on large-scale graphs (§2).

(2) We devise a global memory store abstraction to unify GPU memory and host memory for GNN training, which can be applied on various GPU platforms (§4).

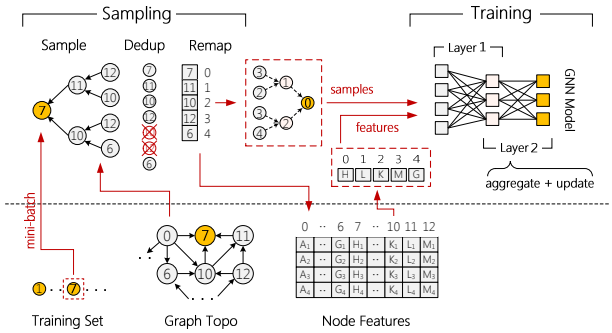

Figure 1: An example of sample-based training for a 2-layer GNN on a mini-batch with V_7.

(3) We implement XGNN, whose core is GGMS (§5), and comprehensively evaluate it on various GNNs, datasets, and GPU platforms to show the efficacy and generality of XGNN (§6).

2 BACKGROUND AND MOTIVATION

2.1 Graph Neural Networks (GNNs)

GNN is used to represent the node attributes (i.e., features) in a low-dimensional vector, which is implemented by aggregating features of neighboring nodes and learning the information through neural network (NN) layers. To train large-scale graphs with billions of nodes and edges, a common practice is to only use a part of neighbors to produce a subgraph for training [1, 2, 15, 28, 51, 57], which is also known as sample-based GNN training. This approach reduces both computation and memory pressure while still ensuring high accuracy [11, 29, 36].

Sample-based GNN training mainly contains two stages: sampling and training. In the sampling stage, for each training node, a set of k-hop neighbors is selected from the input graph following a sampling algorithm (e.g., k-hop neighborhood sampling [18, 20, 55] and random walks [17, 33, 52]); the output contains sampled nodes (referred to as *samples*). In the training stage, the samples and their features are used as the input to train the GNN model with k layers. Figure 1 is an example of a sample-based 2-layer GNN training. The training nodes are divided into mini-batches. The nodes in the mini-batch are used as the 1st-hop root nodes in the sampling stage. For each layer, two neighbors of each input node are sampled. As the sampled nodes are required to be unique, the nodes will be de-duplicated (e.g., V_{10}, V_{12}). Before feeding the samples to the NN model, the node features will be extracted and the node IDs will be remapped.

2.2 GPU Training Acceleration

Single machine with multiple GPUs is the common configuration for GNN training [13, 28, 42, 47, 49, 51], to accelerate both sampling and training stages. In the single-machine multi-GPU setup, the GPU and CPU are connected through PCIe, while GPUs are connected through a high bandwidth network, like NVLink (faster than the PCIe $2-5\times$, shown in Figure 2), AMD Infinity Fabric Link [4], and Huawei HCCS [27]. GNN systems usually adopt data parallelism to accelerate the training procedure by partitioning train nodes into multiple GPUs. Every GPU is dedicated to a training worker process (i.e., Worker), there is usually a Sampler responsible for the sampling stage and a Trainer in charge of the

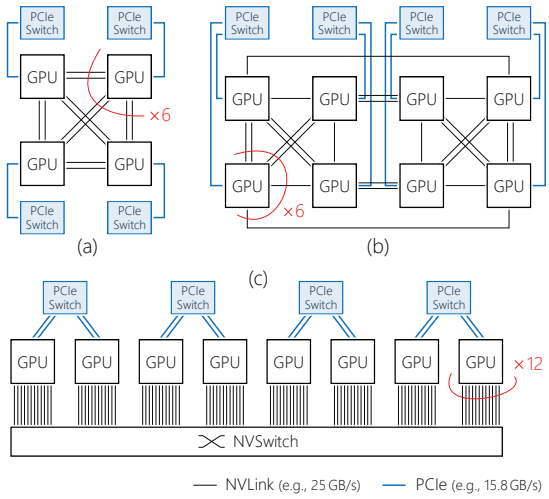

Figure 2: The cases of GPU platforms with different communication topologies. (a) and (c) are symmetric NVLink communication topologies, while (b) is an asymmetric topology.

training stage. For every mini-batch (i.e., a subset of train nodes), Sampler first performs sampling on the graph and sends the *sample* to Trainer. After Trainer gets sampled nodes from Sampler, then it extracts features from the host memory and trains the neural network. During the neural network backward, every Worker will synchronize gradients with each other through NVLinks.

For small graphs (the graph topology and feature can fit in GPU memory), GNN systems can store entire data in GPU memory to avoid slow PCIe transmission between the host and GPU. However, as the graphs get larger and larger, the entire data typically exceeds one GPU memory (e.g., the total size of PA in Table 3 exceeds 16GB for V100). This causes GPU memory contention and makes it more challenging to train GNN on large-scale graphs efficiently. GNN systems must choose to either sacrifice the feature cache by storing the graph topologies in GPU memory [28] or sacrifice sampling performance by storing the graph topologies in host memory to reserve GPU memory for the feature cache [42]. Recently, efficient large-scale GNN training gains increasing attention and many GNN systems are proposed to solve this challenge.

2.3 GNN Training Systems

There are three main kinds of systems in a machine with multiple GPUs. However, those systems still suffer from several issues to train on large-scale graphs efficiently.

GNN system with a replicated cache. This class of system (esp. DGL w/ cache in Figure 3(a)) optimizes GNN training by (a) sampling on GPU and storing the whole graph in each GPU to avoid accessing the host memory to accelerate sampling (b) using the rest of GPU memory to cache the same hottest features in each GPU to reduce data movement between GPU and the host during training (c) extracting feature data from the host memory with CUDA zero-copy-based method to reduce data movement [32].

This replicated caching method, however, has a redundant graph and feature data storage. Although the Sampler can utilize GPU high bandwidth memory to accelerate graph topology data access and have high sampling performance, the graph topology

Table 2: A comparison of XGNN systems. "○" means not supporting all GPU topologies.

Systems	Efficient GPU Mem.	NVLink full-utilized	Host Mem. full-utilized
DGL w/ cache	✗	✗	✓
GNNLab	✗	✗	✓
Quiver	✗	✗	✓
WholeGraph	✓	○	✗
XGNN	✓	✓	✓

(e.g., 6.4GB for OGB-Papers) is replicated in every GPU. Considering the limited GPU memory, the total footprint of the graph will be a large portion of total GPU memory. Thus, the left GPU memory reserved for features is relatively small (e.g., OGB-Papers 7%). Moreover, the cached feature is also replicated across GPUs (i.e., each Trainer only caches features for itself), which makes it worse and feature cache cannot optimize feature extracting well. More seriously, for some large graphs (e.g., Friendster) which exceed GPU memory, these systems are not a feasible design.

GNNLab [51] (Figure 3(b)) observed memory contention between graph topological data and feature data. To mitigate this, it assigns the Sampler and Trainer to different GPUs. However, this method still has the same problem of redundant graph topological data storage in Sampler GPUs and redundant feature data in Trainer GPUs.

Another problem with these systems is low bandwidth utilization. GPUs only communicate with each other for gradient synchronization. Besides synchronization, the modern high-bandwidth and low-latency inter-GPU network (e.g., NVLink) is not used in the whole progress. The system design is not optimal without considering such powerful links (e.g., 300GB/s bi-direction bandwidth per GPU with NVLink).

Partitioned feature cache GNN system. This class of GNN systems (Figure 3(c)) has observed inter-GPU high bandwidth on NVLink and adopts a partitioned feature cache optimization at the multi-GPU platform, like Quiver [42]. Quiver treats the feature cache on full-connected GPUs as a global cache pool. The features sorted by the hotness are cached across these GPUs. During feature extraction, features cached on every GPU will be transmitted to each other in parallel through NVLinks. Such a feature cache storage method can cache more features in GPU and reduce data movement from the host memory to GPU memory. Furthermore, to optimize data placement and reduce data movement between NVLink and PCIe, another work [38] (we refer to it as CliqueOpt) duplicates some of the most frequently accessed feature data in each GPU, striking a balance between the costs associated with data movement across NVLink and PCIe. To avoid out-of-GPU-memory when training on large-scale graphs and further increase feature cache space, these systems store the graph data in the host memory and execute sampling on the host CPU or with GPU direct access acceleration.

However, the Sampler performance is sacrificed for Trainer. The major problem of these systems is low sampling performance due to slow PCIe bandwidth (especially for random memory access during sampling, 10× slower than inter-GPU NVLink) between GPU and the host CPU. Another problem is they not using NVLinks

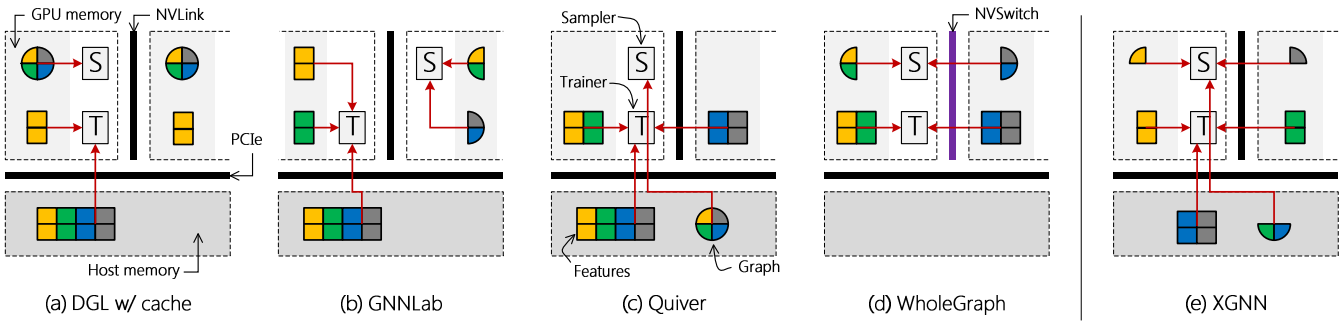

Figure 3: A summary of different designs of sample-based GNN training on multiple GPUs.

well. For asymmetric GPU topology like Figure 2(b), they simply split GPUs into two full-connected cliques, each of which works as a features cache pool independently. The trainer only extracts cached features from its GPU clique. Thus, links between cliques (e.g., 6 links at the V100 platform) are still wasted and the overall bandwidth of feature extracting is sub-optimal.

All-in-memory GNN system. Recently, another all-in-memory GNN system appears. In this all-in-memory system, the GNN system treats all GPU memory as a whole and partitions the whole graph topology and whole feature data into each GPU. WholeGraph [49] (Figure 3(d)) is such an all-in-memory GNN system. Though graph redundancy storage in replicated graph storage is resolved and sampling performance is not compromised for feature cache, there are still several problems for WholeGraph. First, this class of GNN system needs too much total GPU memory to store the whole graph and whole features in all GPU memory. If the total GPU memory space is less than the memory consumption of the graph and features, the system will have no idea to solve it. However, it is a trend that graph and feature data become larger and larger in the future, whereas GPU memory capacity grows slowly. Second, WholeGraph assumes that each GPU can direct access to the memory of all other GPUs. Therefore, it only works on a multi-GPU server with symmetric networks (e.g., NVSwitch 8xA100 in Figure 2(c)). Unfortunately, this assumption is invalid in more general cases, for example, asymmetric links (e.g., 8xV100 in Figure 2(b)).

Summary. From the above analysis of current GNN systems, we found that the existing systems have not performed well in the aspects listed in Table 2. This leads to poor performance when training GNN models on large-scale graphs.

- *GPU memory efficiency.* DGL w/ cache and GNNLab suffer from redundancy storage and waste GPU memory. Furthermore, Quiver partitions feature data into multiple GPUs to solve data redundancy, but Quiver stores the graph topology in the host memory and suffers from low sampling performance. These all cause low efficiency of GPU memory usage.
- *NVLink utilization.* DGL and GNNLab almost do not use NVLink. Quiver only uses a part of links. While WholeGraph fails to run on asymmetric links.
- *Host memory utilization.* WholeGraph only considers the GPU memory, which causes host memory underutilization. Moreover, it fails to run when the graph topological and feature data exceed the total GPU memory.

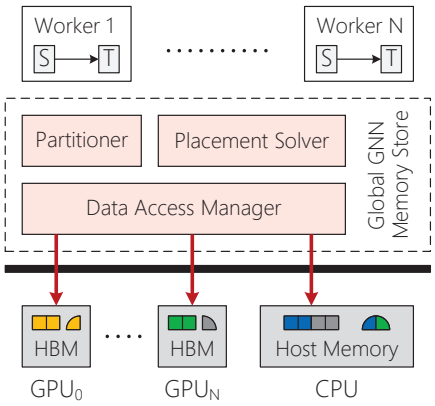

Figure 4: XGNN Architecture.

3 XGNN OVERVIEW

Goal. To solve the above three problems in present GNN systems, we propose a new GNN system, XGNN. The main goal of XGNN, as shown in Figure 3(e), is to store the graph and features across all available GPU memory and host memory to overcome the memory limitation of GPUs. Therefore, there are two storage levels for XGNN, the memory of all GPUs connected via NVLink and the host memory. XGNN is capable of distributing a portion of the hottest graph topological and feature data in GPU memory, whereas less frequently accessed data is stored in the host memory. Besides, XGNN should reduce data storage redundancy across GPUs by partitioning data into devices and ensure each Worker in XGNN can access their necessary data from other GPUs through NVLink and the host memory through PCIe. To achieve this goal and ensure high performance, XGNN not only needs to provide a unified GPU and CPU memory view and be easy to use but also utilize the fast link (e.g., NVLink) efficiently and deal with asymmetric GPU platforms. So we designed the Global GNN Memory Store (GGMS) to manage these memory resources in XGNN.

Global GNN Memory Store. GGMS is a storage engine designed for GNN systems to unify GPUs and CPU memory, whose architecture is shown in Figure 4. At the underlying storage level, GGMS manages memory between GPUs and CPUs through fast-connected networks for efficient utilization. While at the upper computation level, GGMS provides simple APIs for accessing data that allow the Sampler and Trainer components of the GNN system to focus solely on their respective computing tasks, without

```
▸ GGMS is mainly implemented as a class in library
class DistGraph
1   DistGraph(dataset, graph_ratio, feat_ratio, ⋯)
2     // ⋯ initialize GGMS at startup time
3   NumNeighbor(node_id)
4     // ⋯ get the number of neighbors of node_id
5   Neighbors(node_id)
6     // ⋯ get neighbors array of node_id
7   Features(node_id)
8     // ⋯ get features array of node_id

▸ sample_nodes: the nodes to sample
▸ fanout: the number of neighbor nodes to sample
KHopSampling(dist_graph, sample_nodes, fanout, ⋯)
9   for node_id in sample_nodes
10    num_neighbor = dist_graph.NumNeighbor(node_id)
11    neighbor_array = dist_graph.Neighbors(node_id)
▸ a parallel set in CUDA shared memory
12    idx_set = {}
13    while idx_set.cnt != fanout  ▸ parallel in warps
14      idx = random() % num_neighbor
15      idx_set.insert(idx)
▸ remote memory access with batched requests
16    for sample_idx in idx_set    ▸ parallel in warps
17      sampled_neighbor = neighbor_array[sample_idx]
18      save sampled_neighbor

▸ input_nodes: the nodes to extract features
FeatureExtracting(dist_graph, input_nodes, ⋯)
19  for node_id in input_nodes
20    features = dist_graph.Features(node_id)
21    save features
```

Figure 5: An example for DistGraph usage.

concerning for details of the underlying memory allocation and management.

GGMS includes three modules, Data Access Manager (DAM, §4.1), Placement Solver (§4.2) and Partitioner (§4.3). Firstly, the user will determine the ratio of the graph topology and features to store in GPUs and CPU. For data in GPUs, GGMS uses Partioner to divide the data into the same number of logical partitions as the number of GPUs. Ideally, each GPU has a physical data partition to map logical partition. However, in some asymmetric NVLink networks, such as illustrated in Figure 2(b), a GPU cannot access all physical partitions from connected GPUs and CPU. Therefore, Placement Solver will duplicate some physical partitions conditionally on other GPUs and give a partition deployment plan to ensure each GPU can access all physical partitions through NVLink or PCIe. Finally, DAM uses a CUDA technique, Unified Virtual Address (UVA), to manage the distributed physical data partitions on devices, and extend GPU memory to the host CPU. DAM employs UVA to allocate the data partitions on GPUs/CPUs and record their physical position information in the metadata. After GGMS startup, according to this metadata and UVA, DAM can redirect data requests for every node to their correct physical location and let CUDA kernel functions from the computation level access data in these partitions directly.

In summary, GGMS uses the DAM to solve the host memory underutilization problem by extending memory from one GPU to all GPUs and CPU. Moreover, the Placement Solver solves the NVLink underutilization problem and supports XGNN running on all types of NVLink networks. Finally, the Partitioner eliminates memory redundancy by partitioning data into multiple devices.

XGNN APIs. GGMS provides three main easy-to-use CUDA kernel APIs for convenient graph topology and features accessing. The Sampler can call `NumNeighbor(node_id)` to get the number of a node's neighbors and call `Neighbors(node_id)` to get the detailed neighbors array pointer for the searching node. The Trainer can call `Features(node_id)` to get the feature array pointer for the node. The API calls will forward the data access requests to the correct data partition transparently according to the placement result. The request forwarding only costs a modulo and division operation, which is almost no extra overhead introduced. Figure 5 gives the pseudocode of GGMS APIs and usage examples, there are only a few modifications (marked with blue) needed to use GGMS in the sampling and features extracting stages.

Challenge: Complex and various multiple device interconnects. GGMS not only needs to use inter-GPU links efficiently to get better performance but also maintain generality on different GPU platforms. However, the present multi-GPU interconnects are complex. At the physical level, there are various inter-GPU network topologies, such as asymmetric and symmetric topologies in Figure 2. Furthermore, the user may also specify a part of GPUs to train GNN, which makes link topologies more varied. Even in the same topology, the links between two GPUs have different bandwidths (e.g., 25GB/s and 50GB/s in Figure 2 (b)). The various topologies pose challenges in maintaining workload balance and efficiently accessing stored data across GPUs. At the software level, CUDA offers several program-level techniques to enable communication across GPUs (i.e., UVA, UM and VMM), which have different performances in different scenarios, we need to make clear their features and choose the best technique for our system. Hence, *how to shield complex link networks for applications and get optimal performance via communication techniques* poses a key challenge.

4 DESIGN OF GLOBAL GNN MEMORY STORE

4.1 Data Access Manager

Data Access Manager (DAM) provides underlying inter-GPU and GPU-to-CPU data access capability to support GGMS.

P2P Direct Access. When Workers cannot access the graph topology or features locally, Workers need to access them in remote GPU or the host memory. As far as our knowledge, there are two methods to access data in remote devices, remote procedure call (RPC) and P2P direct access. RPC is a kind of two-sided method. For example, if the graph topology is partitioned on GPU_0 and GPU_1, when the Sampler gets a 1-hop sampling task, the Sampler first divides input nodes into 2 parts and sends them to the corresponding GPU where the nodes are stored. Then sampling kernels will be called on two GPUs for each part of the input nodes. Finally, the Sampler gathers the sample results from remote GPU. Modern GPUs also enable a one-sided method, i.e., P2P direct access, GPUs can access their neighbor GPU memory in the kernel function. Back to the above example, with P2P direct access, the Sampler can access both local and remote data in the local sampling kernel function. We compare the two methods by sampling a few

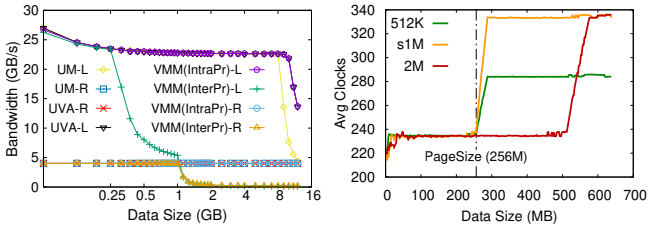

Figure 6: (a) A micro-benchmark to test UVA, UM and VMM performance on local and remote random data access. (b) Pointer chasing of VMM with shared handle.

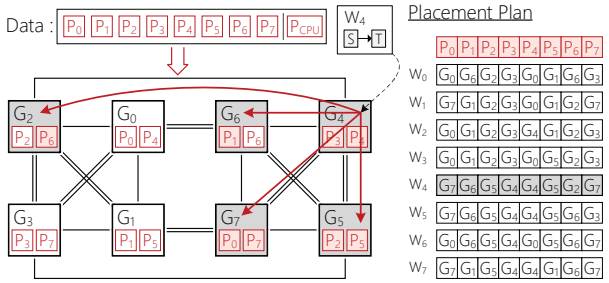

Figure 7: A case of graph partitioning and placement plan.

epochs in a micro-benchmark, and find P2P direct access method is faster than the RPC method by 2 – 3×. The first reason is that RPC methods incur more small kernel function callings (esp. the 1st and 2nd hop) and higher kernel launch latency [49]. Another reason is that direct access introduces an opportunity to overlap remote access in the kernel function.

P2P Method Comparison CUDA provides several techniques to implement P2P direct access, i.e., Unified Memory (UM), Virtual Memory Management (VMM) and Unified Virtual Address (UVA). However, UM, VMM and UVA have their own advantages and drawbacks.

UM is incompatible with multiple process training because UM cannot be shared between processes. It needs to fork a new process to use UM in XGNN, which will incur GPU memory overhead to store extra process context. Thus, we will not consider UM to implement P2P communication in XGNN. VMM, which allows managing memory allocation and mapping manually, however, has serious random memory access performance issues. UVA does not have the above issues but requires manually maintaining multiple pointers. Samplers in XGNN will access the graph topological data in remote GPU memory randomly during sampling. Figure 6(a) compares UM, VMM and UVA by accessing local and remote GPU memory in another process randomly w.r.t. different data sizes. Because UM cannot be shared between processes, we test the intra-process performance of UM. For random local access, UM and UVA have similar performance. Further, when the data size is larger than TLB coverage (i.e., 8GB), the performance degrades. However, VMM shows severe performance degradation when accessing memory randomly between processes when the data size is larger than 256 MB. For random remote access, the random memory access bandwidth is limited to a maximum of 4GB/s due to limited link bandwidth. VMM performance degrades after the data size is larger than 1 GB.

To analyze this problem, we also test VMM performance for an intra-process case and the performance problem does not occur. The reason is that exporting VMM to an inter-process sharable handle will let CUDA map memory with smaller page size due to some limitations. Thus, TLB will miss frequently when accessing remote memory randomly. There are two levels of TLB, L1 and L2. L2 TLB has a greater impact on accessing large data randomly than L1 TLB. To figure out the L2 TLB coverage and page size, we detect the TLB properties by pointer chasing [22]. Figure 6(b) shows the result. We find that VMM L2 TLB page size is 1 MB and L2 TLB can cover 256 MB. Compared to normal L2 TLB, which has a page size of 32 MB and can cover 8 GB, VMM L2 TLB's page size is much smaller.

Finally, we adopt UVA to implement DAM in XGNN. During initialization, each Worker synchronizes with its neighbors to exchange data pointers of allocated memory. DAM will use these multiple pointers to manage remote data access.

Metadata Management and Data Access. Placement Solver will pass the placement plan to DAM during XGNN initialization. DAM will use it to place logical graph topological and feature partitions on GPUs, which are partitioned by Partitioner. During XGNN runtime, DAM maintains an array of pointers, which point to data physical partitions allocated in local GPU, remote GPU or the host memory. When a data access request arrives, DAM will first decide which physical data partition to access by modulo operation according to the placement result and forward the request to it.

4.2 Placement Solver

The Placement Solver aims to provide a placement plan for placing logical data partitions to specific physical devices, allowing the Sampler and Trainer on each GPU to access all corresponding partitions directly.

However, due to the various NVLink topologies [26], direct access from one GPU to all other GPUs may not be always possible. Furthermore, users may selectively utilize a few GPUs on the machine, rather than all available GPUs, thereby potentially complicating the link topology. For an example of the 8-GPU topology shown in Figure 7, GPU_0 can not access GPU_4, GPU_5, GPU_7 directly.

A naive solution is to partition GPUs into fully connected cliques and replicate the entire data on each clique (as employed by Quiver). However, the clique placement policy results in low utilization of NVLinks between GPU cliques. Furthermore, it may have imbalanced storage across GPUs. Imbalanced storage will cause wasted memory on some GPUs and a decrease in cache ratio, ultimately increasing the end-to-end time. Because gradient synchronization in each mini-batch and end-to-end time is determined by the slowest one. Contrarily, given logical graph topology (feature) partitions, the Placement Solver will replicate them for each Sampler (Trainer) to balance data storage and maximize NVLink utilization based on the NVLink network topology, and accelerate Sampler (Trainer) data access. To achieve better performance, the partition placement needs to satisfy the following properties.

Firstly, the number of storage partitions on each GPU should be balanced. Imbalanced partition storage will cause an increase

in end-to-end time. Secondly, it is very intuitive that the number of redundant partitions should be minimized to store more data. Thirdly, it is important to maximize the utilization of NVLinks and ensure balanced access to remote data by Sampler (Trainer) over each NVLink. It is crucial to avoid idle NVLinks or bottlenecks on a single NVLink, as they can result in sub-optimal performance.

To obtain an optimal placement plan, usually, the Placement Solver needs to search all possible plans and chooses the best one. However, searching all possible placement plans is computationally intensive and time-consuming. Based on the analysis of partition placement, we present a pruning strategy with three prioritized rules to reduce search space. Figure 8 shows the searching algorithm progress. Initially, the logical data partitions (equal to the number of GPUs) will be placed into each GPU. For every GPU, the solver adopts the pruning strategy to duplicate data partitions, which it cannot access from itself and its neighbor GPUs. The first two rules below are used to choose possible GPUs (from itself and its neighbor GPUs) to store this partition during the pruning search. After finding a complete placement plan, the solver will use the third rule to filter the plan with a maximum of NVLink utilization as the final result. The three rules are defined as the following.

Balance rule chooses the GPU with the least number of storage partitions. This rule can balance the Trainers' cache space and try to achieve the minimum total partition storage, so this rule has the highest priority.

Less replication rule improves utilization of each partition and stores it to the neighbor GPUs where it is most needed. Besides, the GPU which needs to access that partition by itself has higher priority.

Bandwidth rule is used to filter the possible placement plans pruned by the first two rules and choose one with the highest minimal bandwidth weight. The minimal bandwidth weight of one placement plan is measured by

$$a_i = \min\left\{\frac{Bandwidth_{ij}}{AccessCnt_{ij}} \mid GPU_j \text{ is connected to } GPU_i\right\},$$
$$MinBandwidth = \min\{a_1, a_2, \cdots, a_n\},$$

where $Bandwidth_{ij}$ is the bandwidth from GPU_i to GPU_j, when GPU_i and GPU_j are the same GPU, it is local memory bandwidth. $AccessCnt_{ij}$ is the number of partitions which are accessed from GPU_i to GPU_j.

Figure 7 shows the final partition placement for the 8-GPU platform. As an example of $Worker_4$ (W_4), its logical partitions are placed in GPU_2, GPU_4, GPU_5, GPU_6, GPU_7 and the host memory, and W_4 will access these physical devices to get the whole data. The right table in Figure 7 shows the detailed placement result for each Worker. Finally, GGMS stores one extra partition in each GPU and makes all NVLinks be used evenly.

4.3 Data Partitioner

In this section, we introduce how GGMS partitions the graph topology and features into logical data partitions, which are then distributedly stored in GPUs and CPU through the Placement Solver.

When the total memory of GPUs cannot store all graph data, XGNN allows users to specify different ratios for the graph topology and features separately to store in GPU memory, due to their

```
1 results = []
PlacementSolver(parts, GPUs)
2   store_p = [][] ▸ place which partitions for each GPU
▸ place a part on each GPU
3   for p, g in (parts, GPUs)
4     store_p[g].insert(p)
5   PruningSearch(GPUs[0], GPUs.size(), 0, {}, store_p)
▸ R3 (Bandwidth)
6   sort results by their MinBandwidth
7   return results[-1]
PruningSearch(gpu, n_gpu, p_idx, p_list, store_p)
8   if (gpu == n_gpu) then
9     save store_p in results and return
10  if (p_list is null) then
11    save parts that gpu can not access to p_list
▸ try to store part in gpu.nbrs
12  if (p_idx < p_list.size()) then
13    part = p_list[p_idx]
14    for nb in ChooseNbrs(part, gpu)
15      store_p[nb].insert(part)
16      PruningSearch(gpu, n_gpu, p_idx+1, ...)
17      store_p[nb].remove(part)
18  else
19    PruningSearch(gpu+1, n_gpu, 0, {}, store_p)
▸ get the Nbrs. by rules with priority
▸ R1 (Balance), R2 (Redundancy)
ChooseNbrs(part, gpu) -> list
20  for nb in gpu.nbrs
21    nb.w1 = nb.stored_part_cnt
22    nb.w2 = part utilization if stored in nb
23  return the highest weighted Nbrs. in gpu.nbrs
```

Figure 8: Placement searching algorithm.

different access patterns. Given the graph topology or features, the Partitioner partitions them into logical partitions P_0–P_N and P_{CPU}, which belong to GPUs and CPU respectively. In experiments, prioritized graph topology storage usually achieves better end-to-end performance than prioritized feature storage in most cases. This is because the features of each node typically consist of a continuously stored high-dimension embedding vector (e.g., 128 floats for OGB-Papers). This vector is accessed as a complete unit, so feature extraction exhibits spatial locality and belongs to sequential memory access. However, the graph sampling belongs to random memory access as it randomly samples a subset from the neighbors of input nodes, which are unsigned integers. Considering the large bandwidth gap between PCIe and NVLink for random memory access, XGNN processes the graph topology and features separately and stores the graph topology in GPUs first by default. However, prioritized feature storage can outperform prioritized graph topology storage while GPU memory is more limited (see §6.8).

Graph Partition. As shown in Figure 9, the graph topology is stored in compressed sparse column (CSC) format. Each node's neighbors are stored contiguously in the "indices" array, while the "indptr" array stores the offset of neighbors' position for each node.

At first, the graph will be reordered by custom policies, like degree policy. The most frequently accessed nodes have a high priority to be stored in GPU memory. The hottest part of the graph is divided into N partitions for N GPUs based on the result of node ID modulo N. And the new node ID is equal to the old node ID divided by N. This approach results in a uniform node distribution across GPUs and ensures graph access load balance. By modulo

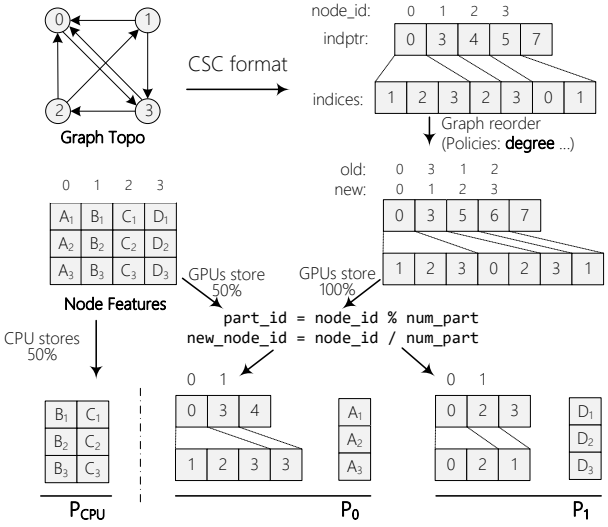

Figure 9: An example of the data partition workflow.

operation, it is not necessary to maintain the mapping data structure between old and new ID numbers and let memory requests for each partition be balanced.

Feature Partition. The rest of the GPU memory will be used to put features, which is a large 2D matrix. The features will be first sorted by hotness, the hottest features will be placed in the GPU memory. These features will be also partitioned into N partitions via modulo operations for the same reason.

5 IMPLEMENTATION

The core design GGMS in XGNN is implemented as a function library. We use Tsota [51], a state-of-the-art GNN system as our codebase, which extends DGL [1] with a GPU cache and a fast GPU sampler from scratch. We further attach Tsota with new optimizations like hashtable module optimization for sampling and features extracting on GPU [32]. XGNN extends Tsota with GGMS and graph accessing optimizations. In the following, we use DGL+C to represent XGNN without GGMS and sampling optimization. There are about 6,100 new lines of C++/CUDA codes for all above features.

Optimizations for graph accessing. In the XGNN architecture, the Sampler will randomly access data in the other device (i.e., GPU or CPU), which is more expensive than local GPU memory. XGNN implements a new KHop algorithm to reduce the number of remote access requests by leveraging underlying GPU memory access features (i.e., coalesced memory access).

In the present Reservoir-algorithm-based [45] sampling method, for each input sampling node, GPU should iterate all neighbor positions one by one and decide whether to choose it. XGNN optimizes it by batching the memory requests. XGNN first uses an on-chip memory (i.e., shared memory in CUDA) hashtable to de-duplicate the produced random neighbor positions of a sampling node. Then XGNN calculates the actual memory addresses of these positions, and send these memory addresses at one time. Because GPU can pack memory requests with continuous addresses into a memory transaction automatically, it reduces remote access requests efficiently. Figure 5 illustrates the pseudocode of our KHop algorithm (Lines 9–18).

Table 3: Datasets summary. Volume$_G$ and Volume$_F$ indicate the data volume of graph topology and features.

Dataset	#Vertex	#Edge	Dim.	Volume$_G$	Volume$_F$
TW [25]	41.7 M	1.5 B	256	5.6 GB	40 GB
PA [3]	111 M	1.6 B	128	6.4 GB	53 GB
UK [8]	77.7 M	3.0 B	256	11.3 GB	74 GB
CF [50]	65.6 M	3.6 B	140	13.7 GB	34 GB

6 EVALUATION

6.1 Experimental Setup

Environments. The experiments were conducted on a GPU server that consists of one Intel Xeon Platinum 8163 CPUs (total 32 cores), 256GB RAM, and eight NVIDIA Tesla V100 (16GB memory, SXM2) GPUs. The software environment of the server was configured with Ubuntu 20.04.3 LTS, Python v3.8, PyTorch v1.10, CUDA v11.7, DGL v0.9.1.

GNNs. We use three GNN models, i.e., GCN [24], GraphSAGE [18] and PinSAGE [52] for evaluation. The number of model layers of GCN and GraphSAGE is three and two respectively, and the dimension of the hidden layer is both 256, which is a common configuration. For each mini-batch, the Sampler will sample 3-hop (2-hop) for GCN (GraphSAGE) neighbors of the training nodes. The number of sampled neighbors of each layer is 15, 10, 5 (10, 25) for GCN (GraphSAGE). For PinSAGE, the number of layers is three, and each layer samples 5 neighbors by random walking from 4 paths of length 3. All above settings are following the details reported in the original papers [18, 24, 52]. Similar to previous work [28], we set the batch size to 6000.

Datasets. We use four large-scale graphs, i.e., Twitter [25] (TW), OGB-Papers [3] (PA), UK-2006-05 [8] (UK) and Friendster [50] (CF). Table 3 shows the detailed information of each dataset. TW is a network of Twitter user follow relationships. PA, a dataset from Open Graph Benchmark (OGB), represents the citations between papers. UK is a web graph of pages on the `.uk` domain. CF is an online game friendship network. Because TW, UK and CF do not have features originally, we use randomly generated features during training.

Baselines. We compared XGNN with four open-source and recent GNN systems: DGL [56], Quiver [42] , WholeGraph [49] and CliqueOpt [38]. The first three systems support GPU-based sampling to accelerate GNN training. DGL accelerates sampling by storing the graph topology in GPU, but it does not cache features in GPU to speed up Trainers. DGL also supports storing graph topological data in host memory for very large-scale graphs, allowing GPU access via UVA during sampling (referred to as DGL-Host, discussed in §6.8). However, DGL lacks support for feature data cache, leading to a longer feature extraction stage. To train on large-scale graphs, Quiver stores the graph topological data in the host memory and partitions feature data into GPUs. WholeGraph [49] is an all-in-memory GNN system that stores both graph topological and feature data across GPUs for better

Table 4: The runtime breakdown (in seconds) of one epoch for DGL, Quiver, DGL+C and XGNN. S, E, T and E2E represent sample stage, feature extraction and model training in train stage, and end-to-end time. R% and H% represent the *cache ratio* of features and the cache hit rate. GSG and PSG are short for GraphSAGE and PinSAGE. For sample stage time on XGNN, the default graph *cache ratio* is 100%.

GNN	DS	DGL				Quiver				DGL+C				XGNN			
		S	E	T	E2E	S	E (R%, H%)	T	E2E	S	E (R%, H%)	T	E2E	S	E (R%, H%)	T	E2E
GCN	TW	0.10	1.14	0.24	1.48	2.16	1.15(55, 90)	0.36	3.48	0.09	0.29(9, 72)	0.20	0.52	0.04	0.05(79, 99)	0.20	0.25
	PA	0.22	2.22	0.54	2.99	3.99	1.82(44, 73)	0.58	6.18	0.11	0.99(8, 44)	0.55	1.73	0.11	0.08(69,100)	0.54	0.62
	UK	OOM	OOM	OOM	OOM	3.39	10.2(20, 31)	0.57	13.8	OOM	OOM	OOM	OOM	0.07	0.47(42, 70)	0.43	0.88
	CF	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	0.24	7.75(10, 54)	1.14	8.67
GSG	TW	0.06	0.50	0.07	0.63	0.33	0.11(99,100)	0.09	0.54	0.05	0.11(16, 79)	0.07	0.17	0.03	0.02(100,100)	0.07	0.08
	PA	0.15	1.08	0.22	1.45	0.70	0.08(64, 98)	0.15	0.85	0.07	0.42(11, 56)	0.23	0.67	0.08	0.06(80,100)	0.23	0.26
	UK	OOM	OOM	OOM	OOM	0.72	3.00(42, 63)	0.20	3.55	0.07	0.64(1, 7)	0.22	0.85	0.06	0.19(51, 79)	0.16	0.33
	CF	OOM	OOM	OOM	OOM	0.78	0.28(100,100)	0.16	1.12	OOM	OOM	OOM	OOM	0.10	0.13(93,100)	0.17	0.33
PSG	TW	0.13	0.61	0.37	1.10	×	×	×	×	0.04	0.14(10, 76)	0.34	0.47	0.04	0.03(84, 99)	0.34	0.36
	PA	0.33	0.83	0.75	1.91	×	×	×	×	0.10	0.42(8, 46)	0.78	1.23	0.11	0.04(71,100)	0.81	0.83
	UK	OOM	OOM	OOM	OOM	×	×	×	×	OOM	OOM	OOM	OOM	0.10	0.48(35, 62)	0.90	1.37
	CF	OOM	OOM	OOM	OOM	×	×	×	×	OOM	OOM	OOM	OOM	0.17	0.94(28, 85)	1.43	2.35

performance. CliqueOpt [38] further optimizes the clique placement policy in Quiver. However, CliqueOpt [38] only supports the CPU-base sampling, which causes worse overall performance than Quiver, so we port its placement policy in XGNN codebase and compare their feature extracting performance separately in §6.6. We also compared XGNN with DGL+C, which shares the same codebase as XGNN but lacks sampling optimization, with graph data replicated in each Worker (DGL+C in Figure 3 (a)).

6.2 Overall Performance

We first compared XGNN with its competitors for end-to-end training time[1], and conducted a breakdown analysis[2] on 4 datasets. Since the total memory consumption (graph topological and feature data, runtime memory) of 4 datasets surpasses the capacity of fully connected GPUs (4 GPUs on our platform), WholeGraph cannot train these datasets. We compare it and XGNN separately in §6.10. Table 4 reports the detailed time on 8 GPUs. To present the performance details, we split the Train stage into Extract and Train stages, which include feature extracting and NN computing respectively. We mainly find the following.

(1) Overall, XGNN outperforms DGL, Quiver and DGL+C by up to 7.9× (from 2.3×), 15.7× (from 3.3×) and 2.8× (from 1.3×), respectively. The improvement mainly comes from GGMS, which improves resource utilization efficiency on the GPU platform. As XGNN does not change the neural network training behavior, XGNN has a similar training stage performance with other systems.

(2) Compared with DGL, XGNN stores the graph topological and feature data across GPUs by GGMS. As a result, XGNN outperforms DGL in feature extraction up to 27.8× (from 18×). Thanks to fast GPU interconnects and optimized graph access, XGNN does not introduce large overhead in the sampling stage and even beat DGL in the sampling stage. Furthermore, XGNN can still train on large-scale graphs (UK and CF) with high performance, which causes OOM in DGL.

(3) Compared with Quiver, XGNN has a better sampling performance and lower memory usage overhead. Quiver stores the graph topology in host memory and stores more feature data for better extracting performance. However, Quiver suffers from poor sampling performance (up to 54× slower than XGNN). Contrarily, XGNN stores the graph across GPUs by GGMS and enjoys high sampling performance thanks to high inter-GPU bandwidth. Additionally, Quiver also suffers from GPU memory efficiency issues, as (a) it spawns new training processes after the system initialization, which causes extra GPU memory consumption (about 0.8GB on each GPU). (b) Quiver uses PyG as its backend and consumes more GPU memory during model training.

(4) Compared with DGL+C, XGNN can train GNN on larger graph datasets (e.g., UK and CF) and store more feature data in GPU to speed up feature extracting (up to 12× faster). Similar to DGL, DGL+C stores the graph topological data in each GPU, which also causes GPU redundancy storage and training process termination for large-scale graphs. Thanks to GGMS, XGNN stores more feature data in GPUs and utilizes NVLinks well, which significantly improves feature extracting performance (e.g., 12× for GCN model on PA). The optimization for graph access also contributes to performance improvement (e.g., from 0.09s to 0.04s for GCN on TW).

6.3 Factor Analysis of Improvement

To study the impact of each design and how they affect the system performance, we cumulatively enable each optimization method and give the detailed running time of the Sampler and Trainer with respect to different models and datasets with 8 GPUs to test.

- **BASE** uses the same settings with DGL+C while storing the same graph topological and feature data in each GPU.

[1] We use the pipeline technique to overlap different stages for end-to-end training.
[2] We use the average time of 10 epochs after the first epoch for breakdown analysis.

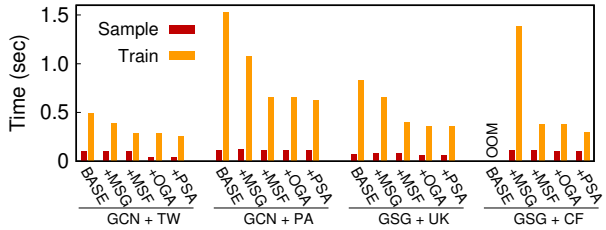

Figure 10: Factor analysis of improvement on an 8-GPU platform.

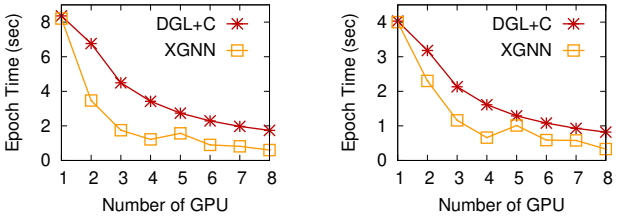

Figure 11: Scalability of DGL+C and XGNN training (a) GCN model on PA, (b) GraphSAGE on UK with different number of GPUs.

- **+MSG** uses GGMS to store the graph topological data.
- **+MSF** uses GGMS to store feature data.
- **+OGA** optimizing graph accessing for Sampler.
- **+PSA** placement solver algorithm supported.

As shown in Figure 10, enabling all optimizations (**+PSA**) beats the basic version (**BASE**) by up to 2.5× in training time and 2.5× in sampling time. Graph topological data with GGMS (**+MSG**) not only solves out-of-GPU-memory to train on large-scale graphs (e.g., GraphSAGE on CF), but also allows larger feature data cache due to graph topological storage de-duplication. Thanks to the high performance of NVLink, the sampling time does not increase obviously. Feature data with GGMS (**+MSF**) distributes the hottest feature data across multiple GPUs and improves Trainer performance significantly on GCN+PA, GraphSAGE+UK and GraphSAGE+CF (ranging from 1.6× to 3.6×). The improvement of (**+MSF**) on GCN+TW is smaller than others due to diminishing returns of feature data cache. Our optimization for graph accessing (**+OGA**) reduces sampling time on GCN+TW and GraphSAGE+UK obviously, the main reason is that we eliminate neighbor nodes iteration in the KHop algorithm. Finally, Placement Solver (**+PSA**) optimizes NVLink utilization for data across 8 GPUs and reduces training time on GraphSAGE+CF by 1.3×. Less improvement on other combinations of other models and datasets is due to feature extracting not dominating the training time.

6.4 Scalability

We evaluate the scalability of XGNN and DGL+C with respect to the number of GPUs. Figure 11 shows end-to-end times for training GCN on PA and GraphSAGE on UK. We find that XGNN outperforms DGL+C and has better scalability. The reasons are two folds. First, with the increase of the number of GPUs, XGNN split the graph topological and feature data on more GPUs. Hence, more

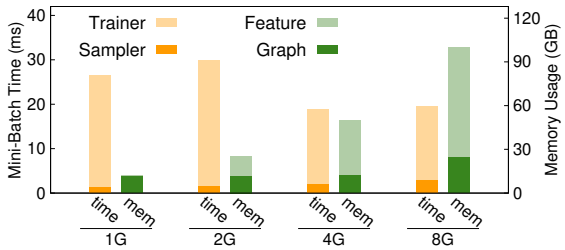

Figure 12: The memory usage of graph topological and feature data over all GPUs and Sampler, Trainer execution time breakdown for a mini-batch using different number of GPUs on GraphSAGE with UK.

data can be stored in GPU and reduce costly data movement from host memory. Second, XGNN leverages all NVLinks to accelerate remote data access during sampling and feature extracting. Higher NVLink utilization means less overhead in graph sampling and feature extracting.

XGNN can also train GNN on asymmetric GPU topologies, even for extremely irregular topologies. For example, training with five GPUs (GPU_0–GPU_4 in Figure 7). However, it is not possible to distribute the graph and features on GPUs equally while ensuring each GPU can directly access the whole data at the same time. Hence, there will exist unused memory among GPUs and degrade performance slightly.

6.5 GPU Memory Efficiency

We next evaluate GPU total memory usage in XGNN. Figure 12 displays the average execution time of the Sampler and Trainer in a mini-batch, along with the memory footprint of the graph and feature data across all GPUs. When there is only one GPU, however, the most of memory is used by the graph topology. XGNN trades sampling for faster feature extraction by partitioning the graph across GPUs. When the number of GPUs increases to two, the graph is partitioned on the two GPUs and more features are cached in GPU memory. However, the feature extraction performance has no improvement due to contention between two GPUs on a single PCIe. When the number of GPUs is four, XGNN only stores one copy of the graph cross GPUs. Moreover, the feature cache becomes larger and more NVLinks are available to facilitate feature extraction. Thus, Trainer performance improves. When the number of GPUs is eight, there exist two copies of the graph and features over all GPUs. Although the mini-batch execution time does not change, the number of per-epoch tasks of each GPU and the end-to-end training time halves.

6.6 Placement Solver Comparison

We train GraphSAGE on TW and CF datasets using different GPU NVLink topologies to evaluate Placement Solver. The evaluated topologies are a full-connected clique with 4 GPUs ($4G_f$), a circular topology with 4 GPUs ($4G_c$), a topology with 6 GPUs ($6G$), and a topology with 8 GPUs ($8G$). We use the same XGNN codebase to compare Placement Solver (Solver), the clique placement policy (Clique) [42], and the optimized clique policy (CliqueOpt) [38].

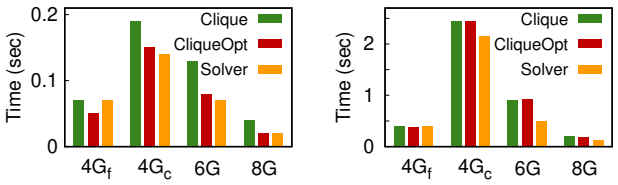

Figure 13: A comparison of partition placement for training GraphSAGE on (a) TW and (b) CF. We use G_0, G_1, G_2, G_3 in Figure 7 to form NVLink topology $4G_f$, use G_0, G_1, G_6, G_7 to form $4G_c$, use G_0, G_1, $\cdots$, G_5 to form $6G$, and use all GPUs to form $8G$.

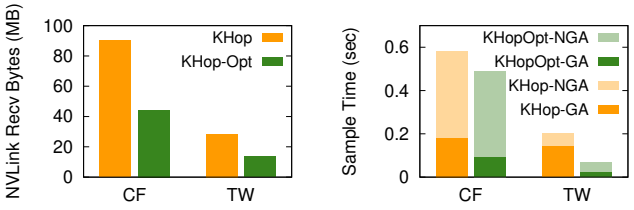

Figure 14: KHop optimization training GCN model with 4 GPUs on PA and UK datasets. (a) shows average NVLink received data bytes in the sampling kernel function. (b) shows sampling breakdown time during an epoch. The dark color represents the sampling kernel execution time.

The $4G_c$ and $6G$ topologies are split into several cliques of 2 full-connected GPUs for Clique and CliqueOpt policies. The $8G$ topology is split into 2 cliques of 4 full-connected GPUs. Figure 13 shows the impact of Placement Solver on feature extracting. The result shows that the XGNN Solver outperforms Clique and CliqueOpt policies by up to 1.8× in the above topologies. The improvement mainly comes from two aspects. First, the Solver allows Trainers to extract features from all neighbor GPUs and resolve the issue of low NVLink utilization between cliques. Second, the Solver eliminates GPU memory waste and improves the cache ratio, for example, from 41% in Clique and CliqueOpt policies to 59% on the CF dataset in the $6G$ topology. For the full-connected topology, which forms a GPU clique, Clique policy and the Solver achieve the same performance due to the same data placement result. CliqueOpt has the best performance due to intra-clique data movement optimization via replication. However, CliqueOpt policy benefits are marginal when cache space is insufficient for the large-scale graph (e.g., CF dataset).

Besides, we also record the search time for Placement Solver. Thanks to the pruning strategy, the time required in all cases is less than 15 ms, making it an efficient solution.

6.7 Sampling Optimization

We evaluate the impact of sampling optimization on sampling inter-GPU data transmission volume. Figure 14(a) shows the average data transmitted bytes between NVLinks during the sampling kernel function, which is collected by NSight Compute [6]. The result shows that the sampling optimization can reduce roughly half of the transmitted data. Figure 14(b) shows the sampling breakdown time during an epoch. For CF dataset, the decrease in the sampling kernel execution time (which we optimized) is mainly due to less remote transmission requests, constituting an average of 3/4 of the total requests. For TW dataset, the sampling kernel has a larger performance improvement due to the KHop sampling algorithm complexity optimization simultaneously.

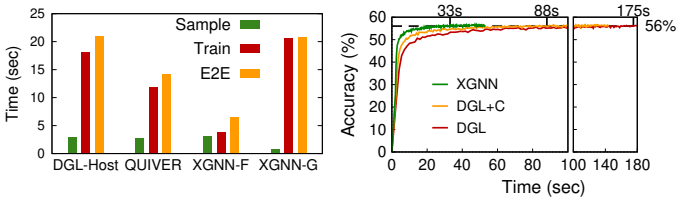

Figure 15: (a) A comparison of large-scale graph GNN training of different GNN systems. (b) A comparison of the end-to-end time between XGNN, DGL+C and DGL for 8-GPU training GraphSAGE on PA until convergence.

6.8 Large-scale Graph

We evaluate GNN training on large-scale graphs of different systems by only using one GPU, where the graph topological data exceeds the GPU memory. DGL-Host, Quiver and XGNN are evaluated to train GraphSAGE on CF dataset. XGNN is evaluated in two different configurations, i.e., prioritized feature storage (XGNN-F) and prioritized graph topological data storage (XGNN-G). The result is shown in Figure 15(a). Compared to training GNN with 8 GPUs (§6.2), all systems have a longer epoch training time. Besides more mini-batches per epoch, Quiver and XGNN cannot utilize multiple GPUs to store the graph topological or feature data. Compared to DGL-Host and Quiver, XGNN-G does not have advantages because sample time does not dominate the epoch time. Thus, storing the graph topological data in GPU memory only has little benefit. Contrarily, XGNN-F outperforms all other systems. The main reason is XGNN-F store the hottest features in GPU, which reduces Trainer feature extracting time significantly. Thus, with smaller GPU memory, prioritized feature storage can be better than prioritized graph topological data storage.

6.9 Training Convergence

We also evaluate the correctness of XGNN by comparing XGNN, DGL+C and DGL convergence. The three systems achieve comparable convergence accuracy of 56% after 120 epochs for training GCN on PA dataset, as shown in Figure 15(b). Remarkably, XGNN surpasses DGL+C and DGL by 2.7× and 5.3× respectively, indicating that novel GGMS design and other optimizations have significantly improved its performance.

6.10 Comparison with WholeGraph

We use a fully connected 8xA100 GPU platform to compare WholeGraph and XGNN. We compare two systems by training GCN on TW and PA, and GraphSAGE on UK. We let WholeGraph and XGNN use the same backend (i.e., DGL) to compare more fairly. Table 5 shows the epoch breakdown and end-to-end training time, and total GPU memory usage. We find that two systems have similar performance due to XGNN storing almost the same graph topology and features as WholeGraph in the symmetric GPU topology. Furthermore, upon further breakdown of the sampling stage, we

Table 5: Comparison of training performance on WholeGraph and XGNN for training GCN on PR and GraphSAGE on PR.

Model+DS	System	S	E	T	E2E	Mem.
GCN+TW	WholeGraph	0.11	0.07	0.10	0.23	138GB
	XGNN	0.02	0.02	0.10	0.12	102GB
GCN+PA	WholeGraph	0.33	0.06	0.27	0.55	130GB
	XGNN	0.05	0.03	0.27	0.29	105GB
GSC+UK	WholeGraph	0.24	0.02	0.10	0.35	143GB
	XGNN	0.03	0.02	0.10	0.10	120GB

find that PyTorch DataLoader in WholeGraph applications introduces higher latency, which is avoided in XGNN.

6.11 Performance on Other Platforms

We test XGNN, DGL+C and Quiver for training GraphSAGE on other platforms, a 2xPCIe-V100 platform and a 4xNVLink-V100 platform. The 2xPCIe-V100 platform has two NVIDIA V100 (32GB memory, PCIe) GPUs, two Intel Xeon E5-2650 v4 CPUs (total 48 cores), and 256GB RAM, representing the low-speed GPU interconnect platforms. The 4xNVLink-V100 platform has 4 full-NVLink-connected NVIDIA V100 (16GB memory, SXM2) GPUs, two Intel Xeon Gold 6138 CPUs (total 80 cores), and 378GB RAM, which represents the future fast-speed GPU interconnects.

The test result is shown in Figure 16. The XGNN can train models at the traditional PCIe platform. Besides, we can find that XGNN outperforms DGL+C and Quiver by $1.3-1.9\times$ and $1.9-6.5\times$ respectively, and achieves high performance at the 4xNVLink-V100 platform. From the evaluation in 6.10 and 6.11, we have reasons to believe that XGNN can still achieve good performance on the future fast inter-GPU networks (e.g., NVSwitch and CXL [5]). We will leave this part of the work for the future.

7 RELATED WORK

As the scale of graphs becomes larger, contrary to full-batch training [21, 30, 43, 46], recent GNN systems adopt sampling for scalability. Given that sampling and feature extracting usually dominate training time, various GNN systems strive to improve this. GPUs are de facto accelerators for sampling in GNN systems and are supported by the most popular GNN systems, e.g., DGL [1] and PyG [13]. To efficiently extract features stored in the host memory, another work [32] enables GPUs direct access to the features. GNNLab [51] notices the computation and data similarity between GPUs, and adopts a factored design to reduce resource contention. To mitigate the overhead of remote data access, MGG [48] overlaps remote data access with local data access and computation.

GNN systems additionally cache the most frequently accessed features to accelerate feature extraction. PaGraph [28] first proposes a degree-based cache policy to determine the hotness of each node. Then GNNLab [51] proposes a pre-sample policy to cache features to get a better cache hit rate than the degree-based policy. Both of them can be applied in XGNN.

As single machines with multiple GPUs become increasingly popular many GNN systems strive to accelerate GNN training by better memory efficiency and lower communication costs. Unlike

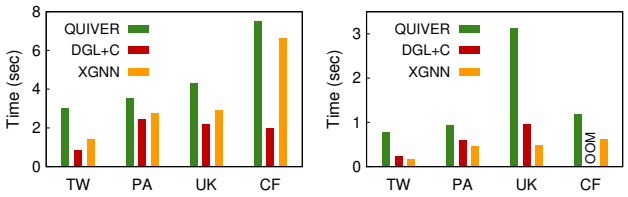

Figure 16: The performance of Quiver, DGL+C and XGNN on (a) a 2xPCIe-V100 platform and (b) a 4xNVLink-V100 platform for training GraphSAGE.

the traditional graph partitioning methods [23], GNN systems typically leverage GNN training characteristics to partition the graph topology and features across GPUs, examples of such systems include PaGraph [28] and P3 [15]. Quiver [42] partitions feature data across GPUs. Furthermore, similar to DeepSpeed in DNN training systems [34], Quiver extends the data storage to the host memory to store the rest of the feature data. Another work [38] improves the efficiency of data movement between NVLink and PCIe compared to the clique policy in Quiver [42]. WholeGraph [49] treats multi-GPU memory as a whole and stores the entire graph topological and feature data in GPUs. DSP [9] partitions the graph topology and features across GPUs and collectively executes sampling.

However, the mentioned systems overlook the complex GPU interconnects, causing inefficient utilization of GPU memory and bandwidth. XGNN addresses these issues by introducing a storage abstraction for GNN training. Furthermore, UGACHE [39] explores different GPU interconnects and cache policies, but it only focuses on graph features (embeddings). Legion [41] enhances GPU memory efficiency by trading off storage between graph topology and features. Both approaches are orthogonal to our work, and their integration is part of our future work.

8 CONCLUSION

This paper presents XGNN, a system fully utilizes system memory resources and GPU interconnects to improve GNN training on various GPU platforms. Through a comprehensive analysis of present systems, we figure out the issues in GPU memory efficiency, NVLink and host memory utilization. XGNN proposes a new global memory store abstraction to tackle the above issues. Specifically, it partitions hybrid input data across both GPU and host memory and further provides easy-to-use APIs for GNN applications to access data transparently. Evaluation on various multi-GPU platforms using typical GNN models with large-scale datasets confirms the efficacy and generality of XGNN.

ACKNOWLEDGMENTS

We sincerely thank the anonymous VLDB reviewers for their insightful comments and feedback. This work was supported in part by the National Key Research and Development Program of China (No. 2020AAA0108500), the National Natural Science Foundation of China (No. U23A20317, 62272291), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 22511106200), and a research grant from Alibaba Group through the Alibaba Innovative Research Program. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

REFERENCES

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

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

[3] 2021. Open Graph Benchmark: The ogbn-papers100M dataset. https://ogb.stanford.edu/docs/nodeprop/#ogbn-papers100M.

[4] 2023. AMD Instinct MI250X Accelerator. https://www.amd.com/en/products/server-accelerators/instinct-mi250x.

[5] 2023. Compute Express Link. https://www.computeexpresslink.org/.

[6] 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.

[7] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.

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

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

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

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

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

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

[14] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Benjamin Chamberlain, Michael Bronstein, and Federico Monti. 2020. SIGN: Scalable Inception Graph Neural Networks. In *ICML 2020 Workshop on Graph Representation Learning and Beyond*.

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

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

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

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

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

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

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

[22] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big data causing big (TLB) problems: Taming random memory accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*. 1–10.

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

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

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

[26] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[27] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 789–801. https://doi.org/10.1109/HPCA51647.2021.00071

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

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

[30] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC'19)*. 443–458.

[31] Xiaoxiao Ma, Jia Wu, Shan Xue, Jian Yang, Chuan Zhou, Quan Z Sheng, Hui Xiong, and Leman Akoglu. 2021. A comprehensive survey on graph anomaly detection with deep learning. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[32] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *Proc. VLDB Endow.* 14, 11 (oct 2021), 2087–2100. https://doi.org/10.14778/3476249.3476264

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

[34] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

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

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

[37] Yingxia Shao, Hongzheng Li, Xizhi Gu, Hongbo Yin, Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. 2022. Distributed Graph Neural Network Training: A Survey. *arXiv preprint arXiv:2211.00216* (2022).

[38] Shihui Song and Peng Jiang. 2022. Rethinking Graph Data Placement for Graph Neural Network Training on Multiple GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing*. Article 39, 10 pages.

[39] Xiaoniu Song, Yiwen Zhang, Rong Chen, and Haibo Chen. 2023. UGACHE: A Unified GPU Cache for Embedding-based Deep Learning. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)*.

[40] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. 2020. A deep learning approach to antibiotic discovery. *Cell* 180, 4 (2020), 688–702.

[41] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, et al. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. *arXiv preprint arXiv:2305.16588* (2023).

[42] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. *arXiv preprint arXiv:2305.10863* (2023).

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

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

[45] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (mar 1985), 37–57. https://doi.org/10.1145/3147.3165

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

[47] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George

Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[48] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. {MGG}: Accelerating Graph Neural Networks with {Fine-Grained}{Intra-Kernel}{Communication-Computation} Pipelining on {Multi-GPU} Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 779–795.

[49] Dongxu Yang, Junhong Liu, Jiaxing Qi, and Junjie Lai. 2022. WholeGraph: a fast graph neural network training framework with multi-GPU distributed shared memory architecture. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 767–780.

[50] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (Beijing, China) *(MDS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2350190.2350193

[51] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: A Factored System for Sample-Based GNN Training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 417–434. https://doi.org/10.1145/3492321.3519557

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

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

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

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

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

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