CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-025-00224-3

REGULAR PAPER

$_{\nu}$GNN: Non-Uniformly partitioned full-graph GNN training on mixed GPUs

Hemeng Wang[1] · **Wenqing Lin**[1] · **Qingxiao Sun**[1] · **Weifeng Liu**[1]

Received: 1 August 2024 / Accepted: 5 March 2025

Abstract
Graph neural networks (GNNs) can be adapted to GPUs with high computing capability due to massive arithmetic operations. Compared with mini-batch training, full-graph training does not require sampling of the input graph and halo region, avoiding potential accuracy losses. Current deep learning frameworks evenly partition large graphs to scale GNN training to distributed multi-GPU platforms. On the other hand, the rapid revolution of hardware requires technology companies and research institutions to frequently update their equipment to cope with the latest tasks. This results in a large-scale cluster with a mixture of GPUs with various computational capabilities and hardware specifications. However, existing works fail to consider sub-graphs adapted to different GPU generations, leading to inefficient resource utilization and degraded training efficiency. Therefore, we propose *vGNN*, a Non-Uniformly partitioned full-graph GNN training framework on heterogeneous distributed platforms. *vGNN* first models the GNN processing ability of hardware based on various theoretical parameters. Then, *vGNN* automatically obtains a reasonable task partitioning scheme by combining hardware, model, and graph dataset information. Finally, *vGNN* implements an irregular graph partitioning mechanism that allows GNN training tasks to execute efficiently on distributed heterogeneous systems. The experimental results show that in real-world scenarios with a mixture of GPU generations, *vGNN* can outperform other static partitioning schemes based on hardware specifications.

Keywords Graph neural network · Distributed training · Graph partitioning · GPU

1 Introduction

Graph neural networks (GNNs) emerge as a paradigm for efficiently learning the relationship and interaction information in irregular graph structures. GNNs have achieved significant accuracy breakthroughs in tasks such as vertex classification (Kipf and Welling 2016a; Hamilton et al. 2017), link prediction (Zhang and Chen 2018; Kipf and Welling 2016b), and graph classification (Zhang et al. 2018; Ying et al. 2018) by combining graph operations and neural computation to understand the relationships among data objects. Due to their powerful performance, GNNs are widely used in areas such as knowledge graph (Bordes et al. 2013; Schlichtkrull et al. 2018), hardware design (Hakhamaneshi et al. 2022; Wu et al. 2022), and recommendation systems (Berg et al. 2017; Fan et al. 2019).

With the increasing size of graph data and vertex dimensions, the computational complexity of GNNs increases dramatically, and the demand for computational resources becomes more urgent (Shao et al. 2024; Dwivedi et al. 2023). To solve this problem, it has become a common practice to deploy GNN training tasks to high-performance graphics processing units (GPUs) and utilize their powerful parallel computing capabilities to accelerate the GNN training and inference process (Tripathy et al. 2020; Thorpe et al. 2021; Zhang et al. 2020; Mostafa 2022).

Hardware technology iterates at an extremely fast pace, with central processing units (CPUs), graphics processing units (GPUs), and other specialized acceleration units (Armeniakos et al. 2022; Xu et al. 2023) constantly making breakthroughs in performance, energy efficiency,

✉ Qingxiao Sun
qingxiao.sun@cup.edu.cn

Hemeng Wang
hemeng.wang@student.cup.edu.cn

Wenqing Lin
wenqing.lin@student.cup.edu.cn

Weifeng Liu
weifeng.liu@cup.edu.cn

[1] SSSLab, Department of CST, China University of Petroleum-Beijing, Beijing 102249, China

Published online: 15 May 2025

and other metrics. Due to the uncertainty of technological developments and commercial needs, researchers typically do not purchase large quantities of the same hardware all at once, but rather upgrade incrementally to ensure the stability of software and business services.

While this incremental upgrade strategy brings flexibility, it also incurs a significant problem: uneven computing capability in a distributed environment (Zhou et al. 2020; Xu et al. 2024). Since GPUs purchased from various batches have their own architectural characteristics, they may show obvious performance differences when performing the same tasks. In addition, the differences in support for mixed-precision further exacerbate the imbalance in computing capabilities. The mixed-precision technique can be well applied to large-scale tasks such as GNN training (Zheng et al. 2022a), which can reduce memory consumption and computational complexity while maintaining prediction accuracy.

Currently, mainstream deep learning frameworks, such as Deep Graph Library (DGL Wang et al. 2019) and PyTorch Geometric (PyG Fey and Lenssen 2019), usually adopt a uniform partitioning scheme when dealing with full-graph GNN training, where the vertices are evenly distributed to individual GPUs. However, this method does not consider the imbalance of computing capabilities during GNN training, which often leads to low resource utilization and makes gradient synchronization a bottleneck.

On the other hand, the parallel capability of GPUs is not always the key to improving the training speed of GNNs. In some cases, data pre-processing, memory access, and data communication may each become a performance bottleneck. Therefore, it is necessary to adjust the task partitioning according to the graph input to improve resource utilization and reduce the end-to-end training time. In a mixed heterogeneous distributed environment, GNN full-graph training faces unique challenges: *(1)* Uniform partitioning leads to under-utilization of some GPUs and overload of other GPUs, resulting in high synchronization latency; *(2)* Diverse subgraphs and GNN models have different resource requirements, and the partitioning scheme according to hardware specifications leads to suboptimal performance.

To address the above challenges, we propose *vGNN*, a **N**on-**U**niformly partitioned full-graph **GNN** training system on mixed GPUs. This paper makes the following contributions:

- We propose a novel performance model for GNN training on GPUs. To obtain the trained regression model, We comprehensively analyze the behavior of GNNs across GPU generations.
- We design an offline-online cooperative task assignment mechanism to fully utilize the mixed GPU resources. We search the scheme and adjust precision to balance the workloads.
- We implement a non-uniformed graph partition system on a cluster that equips different GPUs. The experimental results show that *vGNN* outperforms other partitioning methods by a factor of, on average, 1.33 on GCN (up to 2.75) and 1.23 on GAT (up to 2.41), respectively.

The rest of this paper is organized as follows. Sections 2 and 3 present the background and motivation. Section 4 presents the details of *vGNN* methodology. Sections 5 and 6 present the evaluation results of *vGNN* and the related work. Section 7 concludes this paper.

2 Background

2.1 GPU hardware architecture

GPUs are connected to the host system via the PCI-Express (PCIe) bus as a peripheral device that contains the GPU processor and onboard memory modules. The GPU processor consists of a number of streaming multiprocessor (SM) that share the main memory bus and the L2 cache but are independent of each other. Each SM contains multiple ALU (CUDA core), an instruction decoder, and local memory. CUDA cores share the SM resources, including the instruction decoder, and therefore execute the same instructions simultaneously.

GPUs differ significantly from CPU architectures (Zhang et al. 2017) because of their many-core structure and multiple types of memory; CPUs primarily use task parallelism, where each core executes a different piece of code, whereas GPUs are designed for data parallelism, where multiple cores execute the same code at the same time but work with different data (see Fig. 1).

Threads within a group are divided into subgroups (called warps or wavefronts) equal to the number of GPU cores in the SM. These subgroups run in true SIMT mode, with only one subgroup actually running. When one subgroup waits (e.g., for a memory transfer), SM performs a fast context switch to allow other subgroups to continue computation. SIMT execution suffers from branching problems, where all branches must be executed by all threads when threads in the group choose different branches, and branch-heavy code and loops with large differences in the number of iterations don't perform well on the GPU. However, SIMT simplifies intra-group synchronization by allowing threads to communicate and collaborate through the shared local memory of the SM. While CPUs typically employ task parallelism, GPUs focus on data parallelism and are suitable for handling large-scale graphical computations and data-parallel tasks.

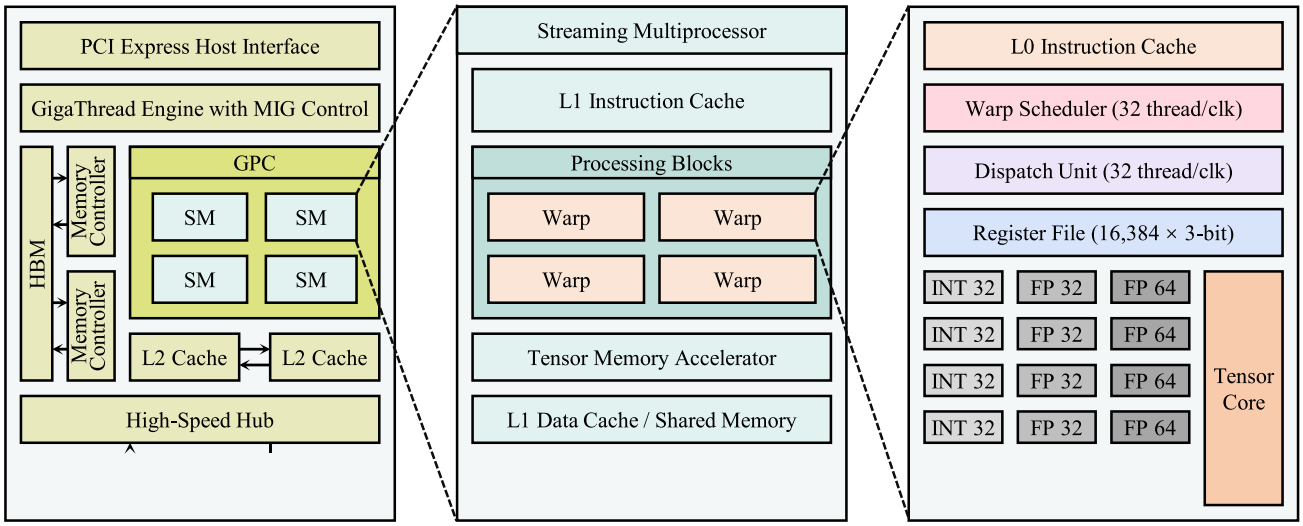


Fig. 1 Hardware architecture of general-purpose GPUs

Table 1 Important GNN notations

Notation	Definition
h_i^ℓ	The feature vector of a vertex i in ℓth layer
U^ℓ	Weight matrix of the ℓ layer
deg_i	Degree of vertex i
$\mathcal{N}_i$	The set of neighbors of vertex i
$e_{ij}^{k,\ell}$	Learnable attention factor
$\mathbb{R}$	Weight matrix space

GPUs have evolved significantly across generations, with each introducing architectural improvements to enhance performance and efficiency. For instance, the Pascal architecture introduced higher memory bandwidth and energy efficiency, while the Turing architecture added real-time ray tracing and tensor cores for AI acceleration. The Ampere generation further increased core counts and introduced second-generation RT cores and third-generation tensor cores. Those architectural differences lead to different computing capabilities and memory bandwidths among GPUs, thus resulting in different performance in terms of GNN training.

2.2 Graph neural network

Recently, there has been increasing interest in applying deep learning to unstructured data. Unlike the dense objects (e.g., images and text) processed by traditional deep neural networks, graphs represent sparse and irregularly connected links. Table 1 presents important GNN notations.

GNNs take graph-structured data as input (Wu et al. 2020), where each vertex is associated with a feature vector. Edges between vertices represent the topology of the graph,

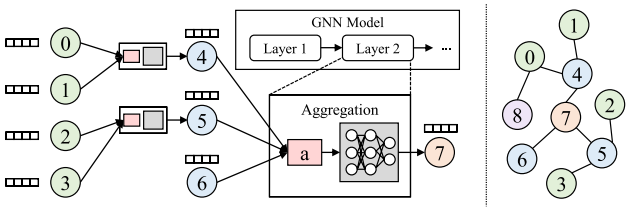


Fig. 2 A simple GNN illustration

quantified by the weights of the edges. GNN learns data relationships by combining graph structure and feature vectors. Figure 2 shows an example of GNN aggregation process.

2.2.1 GCN

Graph convolutional network (GCN) (Kipf and Welling 2016a) is one of the most successful networks for graph learning, which alleviates the problem of overfitting local neighborhood structures for graphs. It performs graph operation formulated as Eq. 1:

$$h_i^{\ell+1} = ReLU\left(U^\ell \frac{1}{\sqrt{deg_i}\sqrt{deg_j}} \sum_{j \in \mathcal{N}_i} h_j^\ell\right) \tag{1}$$

where $U^\ell \in \mathbb{R}^{d \times d}$, deg_i is the in-degree of vertex i.

2.2.2 GAT

The attention mechanism has been successfully used in many sequence-based tasks such as machine translation (Vaswani et al. 2017), machine reading (Cheng et al. 2016), and so on. Graph attention network (GAT) (Veličković et al. 2018)

adopts attention mechanisms to learn the relative weights between two connected vertices. GAT employs a multi-headed architecture to improve the learning capacity, formulated as Eq. 2:

$$h_i^{\ell+1} = Concat_{k=1}^{K}\left(ELU\left(\sum_{j\in\mathcal{N}_i} e_{ij}^{k,\ell} U^{k,\ell} h_k^{\ell}\right)\right) \tag{2}$$

where $U^{k,\ell} \in \mathbb{R}^{\frac{d}{k}\times d}$ are K linear projection heads, the attention coefficients for each head $e_{ij}^{k,\ell}$ are defined as:

$$e_{ij}^{k,\ell} = \frac{exp\left(\hat{e}_{ij}^{k,\ell}\right)}{\sum_{j'\in\mathcal{N}_i} exp\left(\hat{e}_{ij'}^{k,\ell}\right)} \tag{3}$$

2.3 Distributed GNN training

A typical GNN training process comprises forward propagation and backward propagation (Lin et al. 2023). In forward propagation, the input data traverses the layers of neural networks towards the output. Neural networks generate differences in the output of forward propagation by comparing it to the predefined labels. Then, in backward propagation, these differences are disseminated through the layers of neural networks in the opposite direction, generating gradients for updating model parameters.

As shown in Fig. 3, distributed GNN training can be classified into mini-batch training and full-graph training. Depending on whether the whole graph is involved in each model computation phase (forward, backward, and parameter update).

2.3.1 Mini-batch training

Mini-batch training utilizes a portion of the vertices and edges in the graph to update the model parameters in each forward and backward propagation. The aim is to reduce the number of vertices involved in a round of computation to minimize computational and memory resource requirements.

Before each round of training, a mini-batch v_s is sampled from the training dataset v_t. By replacing the full training dataset v_t in Eq. 6 with the sampled mini-batch v_s, we obtain the loss function for mini-batch training:

$$\mathcal{L} = \frac{1}{|\mathcal{V}_s|}\sum_{v_i\in\mathcal{V}_s} \nabla l\left(y_i, z_i\right) \tag{4}$$

It is shown that for mini-batch training, the model parameters are updated multiple times per epoch because a large number of mini-batches are needed to have the entire PASS of the training dataset, resulting in many rounds in one epoch.

Distributed mini-batch training is a distributed implementation of GNN mini-batch training. It also requires synchronization of gradients before model parameters are updated, so a round of distributed mini-batch training consists of three phases: sampling, model computation, and gradient synchronization. Model parameter updates are included in the gradient synchronization phase.

Distributed mini-batch training parallelizes the training process by processing several mini-batches at the same time, one mini-batch per node. Mini-batches can be sampled by the computing node itself or by other devices, such as another node dedicated to sampling. Each node performs forward propagation and backward propagation on its own mini-batch. The nodes then synchronize and accumulate the gradients and update the model parameters accordingly. Such a process can be performed by:

$$\mathcal{W}_{i+1} = \mathcal{W}_i + \sum_{j=1}^{n} \nabla g_{i,j} \tag{5}$$

where $\mathcal{W}_i$ is the weight parameter of the model in the i^{th} round of computation, $\nabla g_{i,j}$ is the gradient generated by the backward propagation of the computing nodes j in the ith round of computation, and n is the number of computing nodes.

Mini-batch GNN training currently has many limitations. The sampling stage introduces irregular calculations and requires traversing the entire graph to obtain neighbor

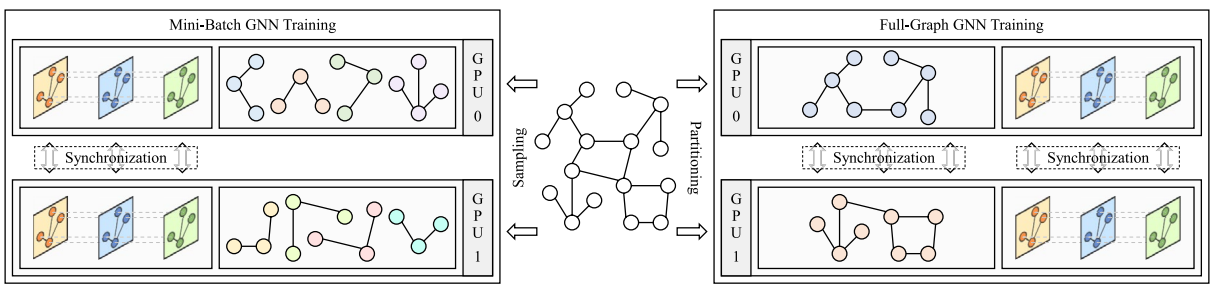


Fig. 3 Two implementations of distributed GNN Training

information (Wan et al. 2023). This causes subsequent stages to stall waiting for input, resulting in performance penalty (Lin et al. 2023). Also, mini-batch training may suffer from low accuracy due to information loss (Jia et al. 2020).

2.3.2 Full-graph training

Full-graph training utilizes the entire graph to update the model parameters in each round. Given a training set $\mathcal{V}_t \subset \mathcal{V}$, the loss function for full-batch training is:

$$\mathcal{L} = \frac{1}{|\mathcal{V}_t|} \sum_{v_i \in \mathcal{V}_t} \nabla l(y_i, z_i) \tag{6}$$

where $\nabla l()$ is the loss function, y_i is the known labeling of vertex v_i, and z_i is the output of the GNN model at feature x_i of input v_i. In each epoch, the GNN model needs to aggregate the representations of all neighboring vertices of each vertex in $\mathcal{V}_t$ once. Therefore, the model parameters are updated only once in each epoch.

Distributed full-graph training is a distributed implementation of GNN full-graph training. In addition to graph partitioning, a major difference is that multiple computing nodes need to synchronize the gradients before updating the model parameters so that the model across computing nodes remains uniform. Thus, a round of distributed full-graph training consists of two phases: model computation (forward propagation + backward propagation) and gradient synchronization. Model parameter updates are included in the gradient synchronization phase.

Since each round involves the entire raw graph data, each round requires a considerable amount of computation and a large memory footprint. To cope with it, distributed full-graph training mainly uses a workload partitioning approach: the graph is split to generate small workloads that are given to different computing nodes.

Such a workflow leads to a large number of irregular communications in each round, mainly to transfer features of vertices along the graph structure. This is due to the fact that the graph data is partitioned and therefore stored in a distributed manner, as well as irregular connection patterns in the graph, such as arbitrary numbers and locations of neighbors of vertices. As a result, there are many uncertainties in the communication of distributed full-graph training (Wan et al. 2022), including uncertainties in the content, target, time and delay of the communication, leading to challenges in the optimization of distributed full-graph training.

3 Motivation

3.1 Performance gap among GPUs

3.1.1 Throughput of SpMM kernel

There are significant differences in hardware specification among GPUs, such as architecture, number of compute units, core frequency, memory bandwidth, etc. Generally speaking, the key kernel during GNN training is SpMM, which corresponds to the aggregation of neighbor information. The throughput of SpMM kernel greatly impacts the training speed of GNNs (Huang et al. 2021). Therefore, we evaluation SpMM on three GPU generations, including A100, TITAN RTX, and TITAN X, to understand how GPU characteristics affect SpMM efficiency.

Figure 4 shows the GFlops of SpMM kernel on different GPUs. As seen, the performance difference among GPUs is not exactly equal to the constant ratios. Take ogbn-arxiv as an example, the achieved GFlops on A100 is 1.6× and 6.4× higher than that on TITAN RTX and TITAN X, respectively. While for ogbn-products, A100 achieves 2.3× and 6.7× higher GFlops than TITAN RTX and TITAN X. The results indicate that SpMM performance is not only determined by hardware but also affected by other factors, such as the graph dimensions and degrees.

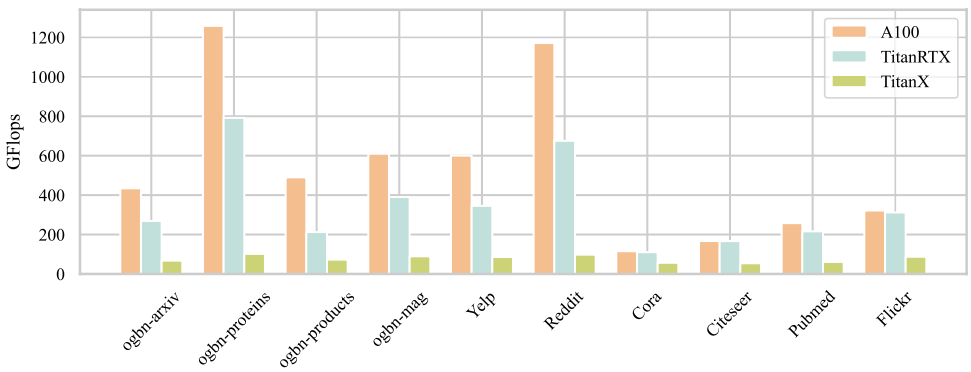


Fig. 4 SpMM throughput on three GPUs

3.1.2 Execution time of GNN training

The training process involves many steps, such as gradient calculation and parameter update. Therefore, we evaluate the end-to-end performance of GNN training on the same platform mentioned above. Figure 5 shows the execution time of GCN training with 100 epochs. We perform a $log2$ operation on the per-epoch execution time to better showcase the performance differences among GPUs.

It can be observed that the gap in training time is much greater than that in SpMM GFlops. Even for ogbn-arxiv, Cora and Pubmed, the A100 does not achieve the shortest training time. The results indicate that the graph input has a considerable impact on both the key kernels and the training process. To maximize the training performance of each GPU, it is necessary to deeply analyze the performance characteristics of GNNs to reasonably partition the training tasks.

3.1.3 Mixed-precision GNN training

Modern GPUs support computations with multiple precisions, and lower precision usually results in faster memory access and computation. However, low precision may lead to numerical instability. Therefore, mixed-precision (16-bit mixed with 32-bit) is usually used in deep learning to accelerate training (Micikevicius et al. 2018). PyTorch provide the interface `torch.cuda.amp` for mixed-precision training. The SOTA GNN framework DGL also support automatic mixed-precision (AMP). Table 2 shows the accuracy and per-epoch execution time of GCN trained with AMP and pure BF16 on A100 GPU with 100 epochs. Note that pure FP16 is not listed because it fails to converge.

It can be seen from the results that AMP does not show an advantage on GNN training. We also try the training of pure BF16, which achieves accelerations compared with FP32 on some datasets without loss of accuracy. Therefore, we believe that if BF16 can be taken into account due to its superior performance, the efficiency of full-graph training will be further improved.

3.2 Limitation of DL frameworks

3.2.1 Uniform graph partitioning

The graph partitioning function of popular GNN training frameworks is built on METIS (Karypis and Kumar 1998). The function focuses on partitioning the graph evenly so the workload of each process can be balanced. However, the function does not take into account the mixed heterogeneous

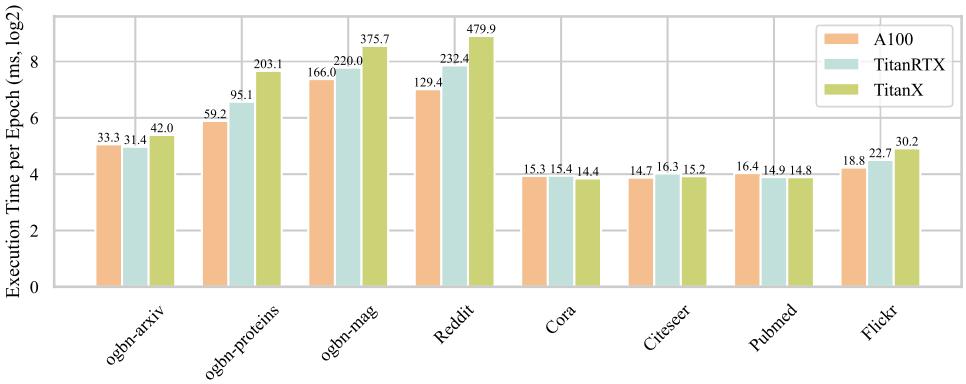


Fig. 5 GCN execution time on three GPUs

Table 2 Mixed-precision GCN training performance on A100

Graph Dataset	FP32		AMP BF16		AMP FP16		BF16	
	Accuracy	Time (s)	Accuracy	Time (s)	Accuracy	time(s)	Accuracy	Time (s)
ogbn-arxiv	43.44	**2.04**	44.39	2.47	44.88	2.40	43.96	2.39
ogbn-proteins	10.03	5.96	10.03	5.38	10.10	5.32	10.17	**4.75**
ogbn-products	72.65	25.13	72.53	23.22	72.30	23.18	72.43	**19.63**
ogbn-mag	10.08	**16.38**	9.99	34.00	9.99	33.75	9.98	49.49
Reddit	94.31	13.17	94.31	11.92	94.29	11.95	94.33	**10.65**
Cora	80.80	1.63	80.50	1.75	80.30	1.72	80.70	**1.50**
Citeseer	71.50	**1.49**	71.40	1.64	71.80	1.63	71.70	1.52
Pubmed	80.10	**1.51**	79.70	1.59	79.60	1.70	79.90	1.53
Flickr	52.97	**1.78**	53.04	2.10	52.78	2.04	52.74	1.92

The bold entries denote the fastest time recorded for each dataset across different precision

situation. The typical proof is that when compiling DGL, the architecture detection only extracts the information of the first GPU, and does not take into account the multi-GPU scenarios. Uniform graph partitioning can not fully exploit the heterogeneous platform with different generations of GPUs.

Figure 6 shows the vertex distribution partitioned by DGL's built-in function. The partitioned sub-graphs consist of two parts, including HALO vertices and inner vertices. We can see that the inner vertices are well-balanced. This is very important in a homogeneous distributed environment, as the number of inner vertices is one key factor determining the computational time required for GNN training. However, HALO vertices are not well balanced (for instance, in ogbn-arxiv, rank 1 has 2.65× HALO vertices compared with rank 0), which will result in a severe imbalance in sub-graph communication. Besides, if we deploy such evenly partitioned training tasks in a mixed heterogeneous environment, there will be a very obvious workload imbalance. This affects the overall training efficiency. Therefore, we need a new partitioning method to tackle the unbalanced processing capabilities among GPU generations.

3.2.2 Gradient synchronization cost

The uniform graph partitioning does not consider the differences in GNN processing capability among GPUs. In this way, GPUs with higher capability will complete the calculations quickly and waste time waiting for gradient synchronization. As shown in Fig. 7, we break down the computation and waiting time of different ranks during full-graph training. Each bar indicates a different rank, and

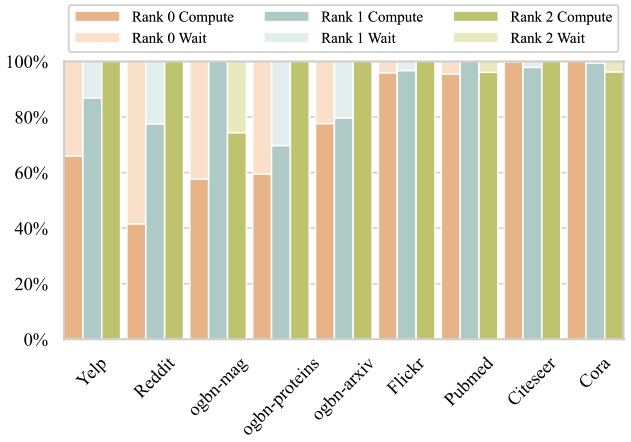


Fig. 7 The time comparison of computation and waiting time among three ranks

y-axis represents the proportion of time spent on computation and waiting.

It can be observed that Rank 2 (TITAN X) has the highest utilization and almost no waiting time, whereas Rank 0 (A100) wastes a lot of time waiting. On the Reddit dataset, Rank 0 even waits for half of the time to start computing, which causes significant resource idleness. This is because the A100 GPU quickly completes the training task and waits for data exchange with other GPUs. Therefore, we need a better graph partitioning method to adapt to the uneven GPU capabilities in heterogeneous distributed environments.

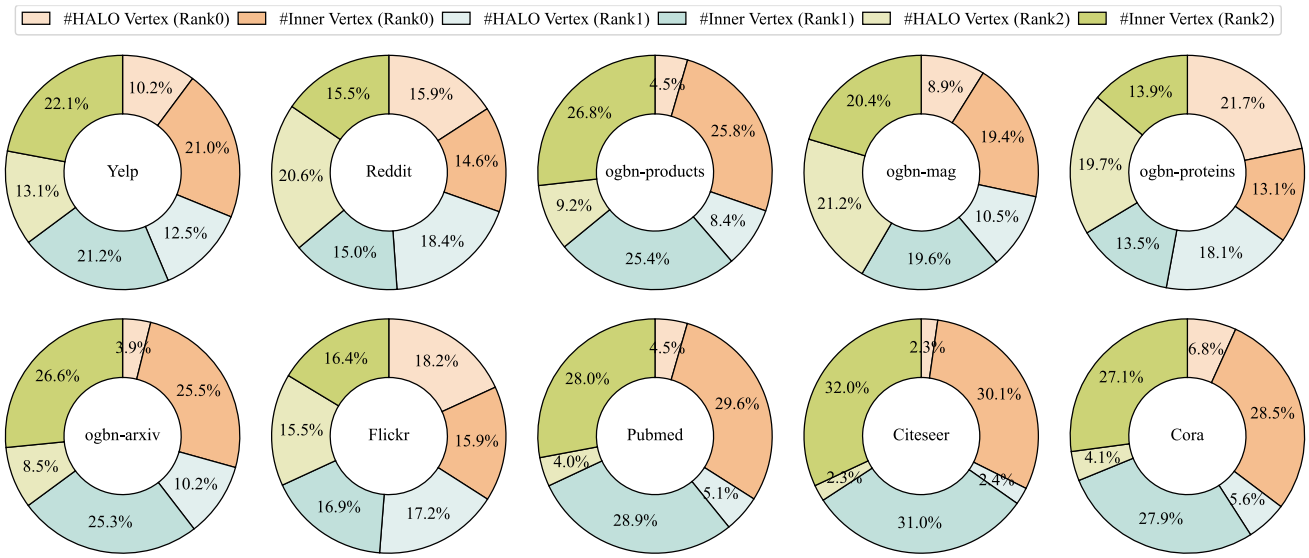


Fig. 6 The vertex distribution after DGL's built-in graph partitioning

4 Methodology

4.1 Design overview

In this section, we propose an efficient full-graph GNN training framework, *vGNN*, for heterogeneous distributed platforms. In order to balance the processing capability among different GPUs, *vGNN* first analyzes the input graph dataset. *vGNN* designs a GNN performance model by abstracting the graph and its matricized feature information. After obtaining the performance differences of GPUs, *vGNN* will find a partition scheme to perform a non-uniformed graph partition. By abstracting the information of GPUs, the key parameters affecting GNN training performance are deeply analyzed. In terms of underlying support, *vGNN* integrates a GPU-aware graph partitioning algorithm in DGL.

Figure 8 shows the design overview of *vGNN*. Before graph partitioning starts, *vGNN* first reads the hardware specification of each GPU, such as the number of SMs, memory bandwidth, Frequency, etc. *vGNN* uses this information to model the computing capability and obtains the initial graph partitioning scheme. Then, *vGNN* conducts a detailed performance analysis of training tasks by combining the GNN model and the input graph dataset information. *vGNN* further refines the initial partitioning scheme considering the performance modeling results. *vGNN* calls the underlying algorithm to partition the graph based on the final scheme, assigning non-uniform subgraphs to different GPUs. Finally, *vGNN* executes GNN training tasks in parallel on GPUs, and warm up several epochs to observe whether the workloads are balanced. If not balanced, *vGNN* will enable low-precision units on GPUs with larger workloads to reduce the communication latency time among GPUs as much as possible.

4.2 Performance modeling

It is necessary to accurately model the performance of GNN training tasks in order to fully utilize the resources of each GPU and reduce communication time. As shown in Fig. 5, when the input graph dataset changes, the execution time of GNN will fluctuate greatly. This is because the input graph and output tensor contain a larger amount of data compared to the model weights. As a result, the execution time of the GNN training task is mainly influenced by the input graph dataset. Therefore, to model the execution performance of GNN training tasks, it is necessary to combine the characteristics of the input graph dataset and the network structure.

Figure 9 shows the performance modeling process of distributed full-graph training. Due to the limited number of graph datasets in real scenarios, overfitting may occur if used directly as a training dataset. Therefore, we adopt a data augmentation method based on graph sampling. Specifically, we sample the input graph dataset by setting different sampling ratios to obtain a smaller sub-graph dataset. Then, we perform distributed GNN training on the sampled dataset to collect the training set of the regression models. During training, in addition to considering the number of nodes and edges of the graph, we also matricized the graph to obtain the adjacency matrix that reflects degree information. After

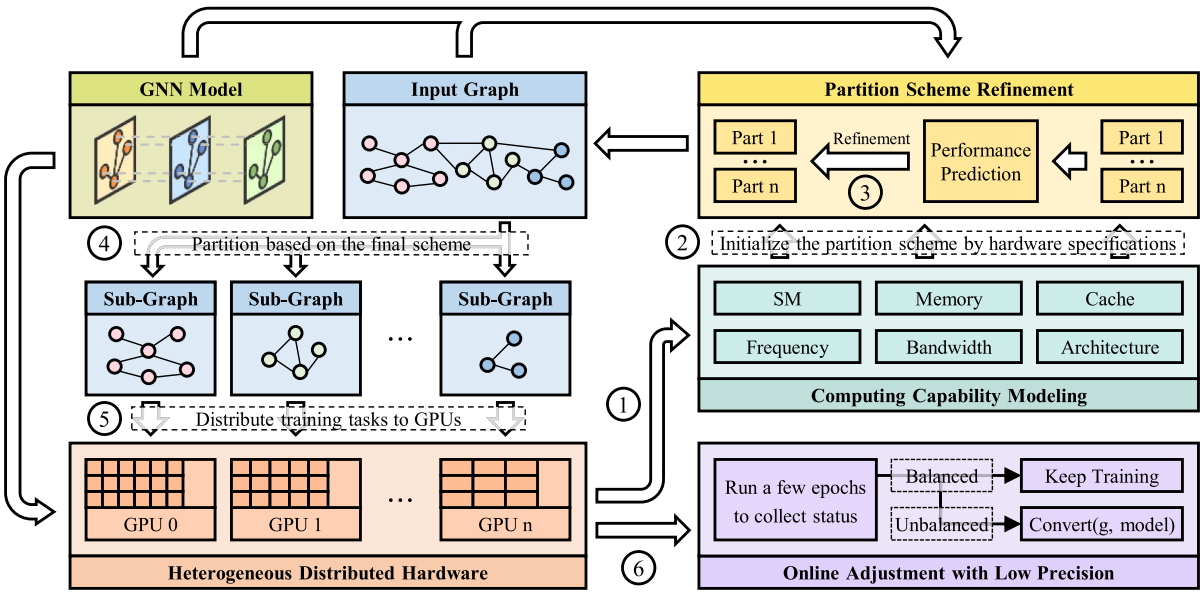


Fig. 8 The design overview of *vGNN*

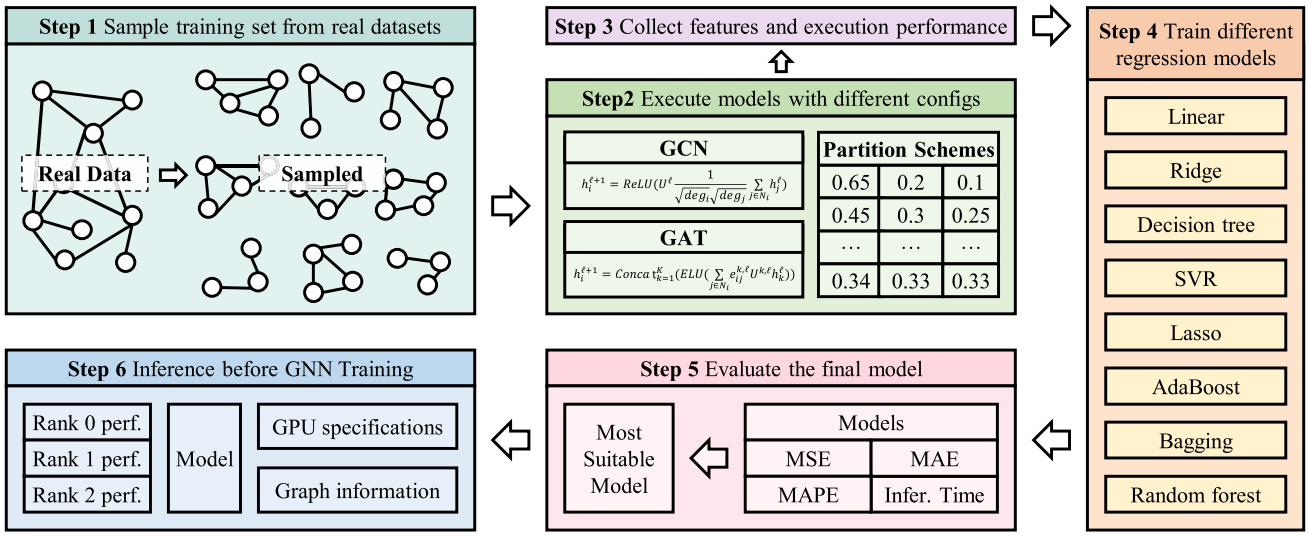


Fig. 9 The process of performance modeling of distributed full-graph training

that, we extract over 30 features from the adjacency matrix, such as the number of non-zero elements in each row and the coefficient of variation of non-zero elements among rows. *νGNN* concatenates these features together as feature vectors of the input data and passes them to multiple regression models for training.

These models will predict the computational time per epoch of a particular GNN structure for a specific GPU, providing guidance for subsequent non-uniformed graph partitioning. *νGNN* evaluates the prediction accuracy of the regression algorithms and selects the algorithm based on the trade-off between prediction accuracy and inference time. The trained regression model will be used for the subsequent graph partitioning.

4.3 Heterogeneity-aware graph partitioning

As shown in Fig. 5, the training tasks of the same dataset also vary greatly on different GPU hardware. This is due to the different processing capabilities of different GPU generations. Therefore, to model the performance of GNN training tasks, it is necessary to combine the characteristics of GPUs and model characteristics.

We consider hardware properties in two parts of *νGNN*. The first is the initial modeling of hardware computational capability. We obtain a series of key metrics from the GPUs, such as the number of SM, memory, frequency, bandwidth, etc. We model the computing performance of each GPU and generate a score for each GPU based on this model. With this model, we can obtain an initial graph partitioning scheme. Then, combined with the GNN model and the input graph dataset information, we will conduct a detailed performance analysis of the GNN training task, adjust and optimize the initial graph partitioning scheme. Based on the above analysis, we can further refine the graph partitioning scheme to ensure that each sub-graph can efficiently utilize the corresponding hardware resources, thereby improving the overall training throughput.

Once we have determined the partitioning scheme, we can partition the graph dataset according to that scheme. We deploy a weight-aware graph partitioning function in *νGNN*. The function can partition a graph into several sub-graphs according to the weight information. The ratio of the number of vertices among sub-graphs is as close as possible to the input partition weights. This way of partitioning ensures that the workload of each GPU is reasonable.

4.4 Algorithm implementation

The graph partitioning function provided by DGL is based on METIS. It divides the graph into multiple sub-graphs with minimal edge-cuts while keeping the number of vertices between the sub-graphs balanced. DGL provides settings for parameters such as the number of partitions, the number of hops, the type of nodes and edges, and the number of training nodes for each machine.

However, it does not take into account the computing capabilities of different GPUs in the case of heterogeneous computing nodes. If the graph is divided evenly, the hardware resources of some GPUs may not be fully utilized, thus affecting the overall training efficiency. Therefore, we have integrated an asymmetric heterogeneous GPU-aware graph partitioning algorithm in DGL.

We use a three-stage process to partition the graph: coarsening, initial-partition, and refinement. The coarsening merges vertices to reduce the size of the graph. In the

initial-partition, a greedy region growth algorithm divides the coarsened graph into partitions that satisfy the constraints. In the refinement, we project the partition information of the coarsened graph back to the original graph. During coarsening, multiple vertices of the original graph are combined into a single vertex in the coarsened graph. The partition labels assigned to these coarsened vertices are now projected back, so all the vertices in the original graph inherit the partition labels of their corresponding coarsened vertex. After projection, the initial partition of the original graph may not be optimal in terms of workload balance or edge-cut minimization. So, by moving vertices between partitions, we refine the partition to balance the final workload.

This algorithm divides the graph according to the partitioning scheme calculated by the performance model of *vGNN*, assigning different parts of the graph to different GPUs. This way, we can make full use of the capabilities of each GPU, reduce communication time, and reduce the overall training time.

4.5 Online adjustment considering BF16 precision

As can be seen from Table 2, when the graph dataset is relatively large, converting the model and graph data to BF16 format can significantly improve performance. The reason is that adopting BF16 format can reduce the number of data transfers during the training process, thereby alleviating bandwidth conflicts. However, for small graph datasets, the precision conversion overhead introduced by BF16 format will actually hurts the training efficiency. On the other hand, not all GPU generations support the BF16 format. For the experimental platform in this paper, only A100 supports the BF16 format from the hardware, while TITAN RTX and TITAN X do not.

In order to solve the above problems, we add the judgment of the precision supported by the hardware when partitioning the graph data. If any GPU supports the BF16 format and is capable of taking on more calculations, an exponential decay function is utilized to adjust the weight of graph partitioning (formulated as Eq. 7).

$$N(t) = N_0 e^{-\lambda t} + b \tag{7}$$

Equation 7 illustrates the exponential decay function, where t is the number of vertices, and N_0 is the initial quantity learned from hardware properties. By doing this, *vGNN* will increase the weight of the partitioning corresponding to the GPU that supports the BF16 format. As a result, more vertices will be allocated to the specified GPU, thus improving the overall training performance. Note that the graph re-partitioning is done online only once, which can further reduce the time required for model training on GPU without significant partitioning overhead.

Table 3 Hardware specifications

	A100	TITAN RTX	TITAN X
CUDA Cores	6912	4608	3072
GFLOPS (FP32)	19.49	16.31	6.691
Boost Clock (MHz)	1410	1770	1089
Memory Clock (MHz)	1215	1750	1753
L1 Cache (KB)	192	64	48
L2 Cache (MB)	40	6	3
Memory Size (GB)	40	24	12
Bandwidth (GB/s)	1560	672	337

Table 4 Graph datasets used for evaluation

Dataset	#Vertex	#Edge	#Feature	#Class
Yelp	716847	13954819	300	100
Reddit	232965	114615892	602	41
ogbn-products	2449029	61859140	100	47
ogbn-mag	1939743	21111007	128	349
ogbn-proteins	132534	39561252	128	10
ogbn-arxiv	169343	1166243	128	40
Flickr	89250	899756	500	7
Pubmed	19717	88651	500	3
Citeseer	3327	9228	3703	6
Cora	2708	10556	1433	7

5 Evaluation

5.1 Experiment Setup

5.1.1 Hardware and Software Configurations

The hardware specifications are presented in Table 3. The experiments are conducted on Ubuntu 22.04 with GCC v11.4 and NVCC v12.1. The *vGNN* is built on DGL v2.1 and PyTorch v2.2. In addition, DGL is modified to support asymmetric graph partitioning for *vGNN*.

5.1.2 Graph datasets and GNN models

The graph datasets used for experiments are presented in Table 4. As can be seen, the graph datasets have diverse dimensions and feature lengths. Such diversity indicates significant differences in computation and communication complexity.

For GCN, we set the number of hidden units in each GNN layer to 256 and the layer size to 4. For GAT, since the model is more complex, we shrink the number of hidden units in each GNN layer to 128 and the layer size to 3. We use the

Adam Optimizer with a learning rate of 0.01 for all models and trains each model 100 epochs.

5.1.3 Comparison methods and metrics

We compare *vGNN* with three different partitioning schemes. According to the core frequency, GFLOPS, and CUDA Core number of the three GPUs, graph datasets are partitioned following those ratios and compared with *vGNN*. We first compare the overall performance of the two GNN models with four methods on ten commonly used real-world datasets. Then, we perform a quantitative analysis of the models' computation distribution by standard deviation. After that, we conduct an in-depth analysis of the impact of different schemes on communication volume. Finally, we analyze the accuracy and inference overhead of regression models.

5.2 Overall comparison

To validate the effectiveness of *vGNN*, we conduct performance evaluation on the datasets presented in Table 4. The results of the GCN and GAT models are shown in Figs. 10 and 11, respectively. The three columns for each method indicates different ranks (GPUs). Due to the large training time gap among different datasets, we roughly divide the graph into upper and lower parts according to the number of vertices and edges, with the upper half being a larger dataset and the lower half being a smaller dataset. The different patterns in the figure represent the time breakdown of different parts, including computation, communication and reduction.

Compared with other partitioning methods on the GCN model, *vGNN* has an average speedup ratio of 1.39×, 1.32×, 1.27×, and the highest speedup ratio is 2.63×, 2.75×, and 2.62×, respectively. Compared with other partitioning methods on the GAT model, *vGNN* achieves 1.33×, 1.17×, 1.19× speedup ratios on average, and achieves the highest speedup ratio of 2.41×, 2.08×, and 2.19×.

It can be seen that *vGNN* outperforms the three partitioning schemes compared in most cases. This shows that *vGNN* can effectively accelerate the training of full-graph GNN training in distributed heterogeneous environments. We have also observed a performance degradation of *vGNN* in some cases. For example, on the Yelp dataset, the performance of *vGNN* is poor due to the high communication overhead. The experimental results show that although *vGNN* do balance the computational load well, the communication overhead between vertices increases compared with other methods, resulting in the performance degradation of *vGNN*. This shows that the balance of computing load is not the only factor, but also the communication overhead. This situation also occurs in the ogbn-mag dataset of the GAT model. Therefore, in future work, we would like to explore the relationship between graph partitioning and communication overhead.

We also compared the device memory consumption with different partitioning schemes in Table 5. The rank 0 is an A100 with 40 GB device memory, rank 1 is a TITAN RTX with 24 GB device memory and rank 2 is a TITAN X with only 12 GB device memory. It can be seen that compared with partition schemes based on hardware specifications, *v*

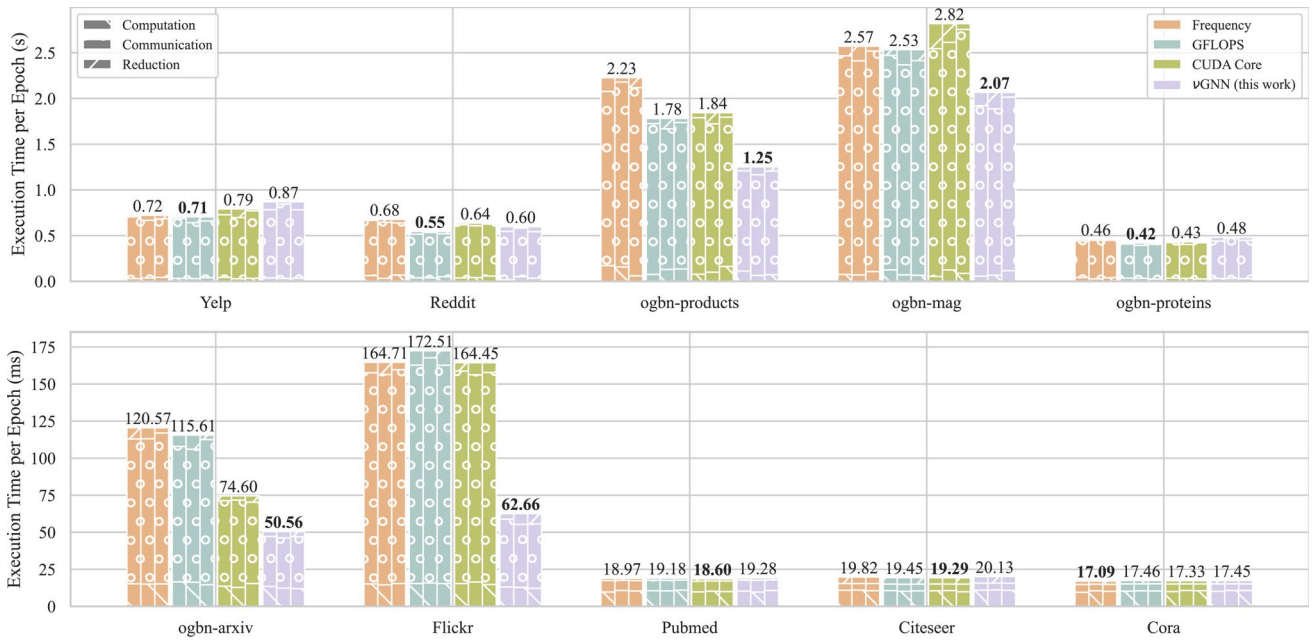


Fig. 10 The overall performance comparison of GCN

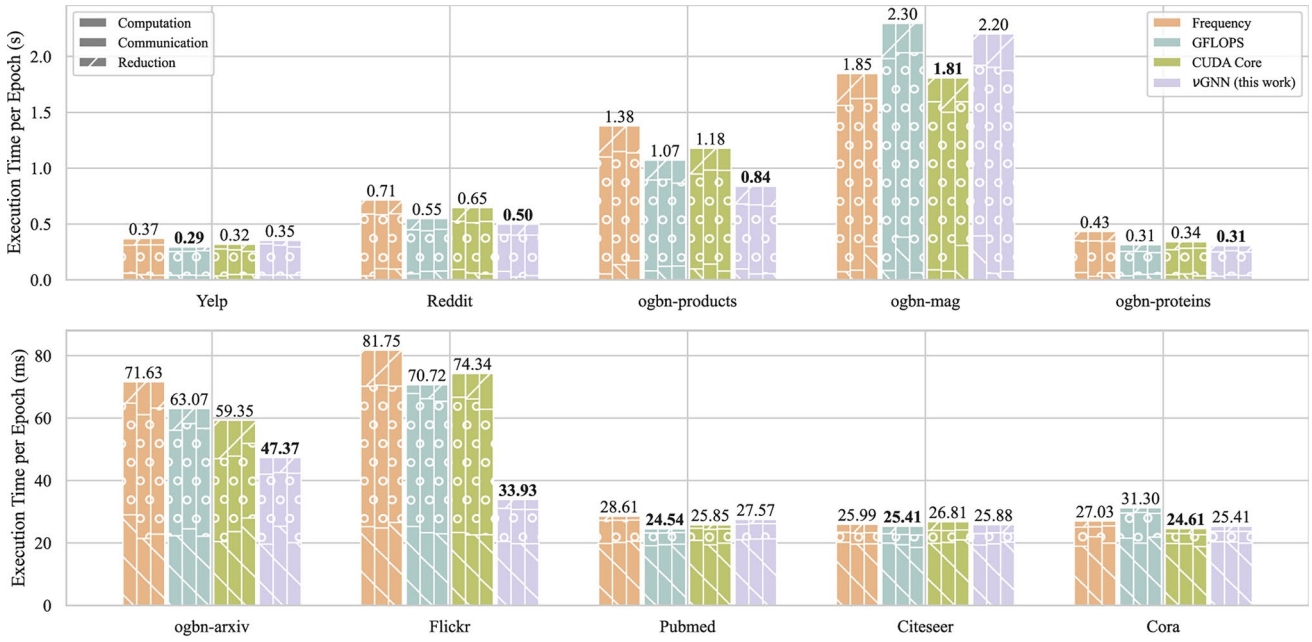


Fig. 11 The overall performance comparison of GAT

GNN is more suited to the device memory distribution. Also, the overall memory consumption of *vGNN* is lower than the other three partition schemes.

5.3 Load balancing comparison

To verify whether *vGNN* makes more efficient use of heterogeneous GPUs, it is necessary to analyze the load balance among GPUs during training. To compare the difference in computation time among GPUs, we obtain the standard deviation of rank time using four methods. The experimental results are shown in Table 6.

Overall, the mean standard deviation of *vGNN* is 30%, 4%, and 22% lower than that of other partitioning methods on GCN. For Yelp dataset, *vGNN* even achieves 2× lower standard deviation than other methods. The results indicate that *vGNN* ensures balanced task distribution among GPUs, reducing the synchronization latency in the mixed heterogeneous system.

For GAT, *vGNN* is more balanced on half of the datasets. However, from an average perspective, the standard deviation of *vGNN* is slightly worse. This is because the standard deviation of the computation time of ogbn-mag is large under all schemes. Since it is the only heterogeneous graph in our experiments, we believe that the dataset trained by the model is not universal enough, resulting in its low prediction accuracy. We will further try more heterogeneous graph datasets later to strengthen the model. If this is excluded, the standard deviations of the four schemes are 14.8, 5.8, 8.3, and 6.8, respectively.

5.4 Communication analysis

vGNN also has advantages in communication. Usually, the graph partitioning algorithm will consider reducing the number of edge cuts as much as possible, thereby reducing the communication volume between GPUs. However, in the uniformed partitioning algorithm, each partition has roughly the same number of vertices. We analyze the communication volume between *vGNN* and other three partitioning methods normalized to *vGNN*. The results of GCN and GAT models are shown in Fig. 12 and Fig. 13, respectively.

As can be seen, *vGNN*'s communication volume is significantly lower than the other three partitioning schemes. Overall, compared with the other three partitioning schemes on GCN, *vGNN* reduced communication volume by 97.59%, 99.78%, and 87.05%, respectively. This is because in *vGNN*, fewer vertices are allocated to GPUs with lower computational capability. When the number of vertices assigned to the GPU is small, it is highly likely that the number of edge cuts will decrease, thus reducing the total communication volume.

On GAT, *vGNN* reduced communication volume by 208.84%, 213.49%, and 174.77%, respectively. This is because the A100 GPU has an advantage over the other two GPUs in computing the GAT, which results in a more uneven partitioning. Therefore, on the GAT model, the other two partitions have fewer vertices than the A100, further reducing the communication volume between GPUs.

Table 5 The memory consumption of GAT (MB)

Dataset	Frequency				GFLOPS				CUDA Core				*v*GNN			
	rank 0	rank 1	rank 2	overall	rank 0	rank 1	rank 2	overall	rank 0	rank 1	rank 2	overall	rank 0	rank 1	rank 2	overall
ogbn-arxiv	475.06	657.62	431.37	1564.05	658.26	543.97	372.52	1574.75	703.90	536.14	274.71	1514.75	1255.05	76.95	39.53	1371.53
ogbn-proteins	2262.07	4403.32	1967.02	8632.41	3850.95	3306.15	1500.45	8657.55	3748.24	3678.96	1243.38	8670.58	6359.97	1506.76	848.61	8715.34
ogbn-products	8612.58	12926.38	8420.39	29,959.35	12,637.46	10,686.01	6538.76	29,862.23	12481.96	12410.49	4900.97	29,793.42	22411.28	3820.09	2515.06	28,746.43
ogbn-mag	6246.76	8982.24	4655.53	19884.53	10,135.03	5732.96	3976.57	19,844.56	8132.42	8515.14	3229.7	19,877.26	12,143.63	3586.63	2859.09	18,589.35
yelp	3485.45	4347.79	2703.86	10537.10	4574.35	3465.96	2186.31	10,226.62	4410.48	3990.63	1659.31	10,060.42	6626.90	2167.87	1802.61	10,597.38
reddit	3693.66	6751.88	4311.15	14756.69	6666.28	2503.13	5358.69	14,528.10	5804.15	6490.98	2242.09	14537.22	11,568.74	1139.30	834.76	13,542.80
cora	36.12	40.42	32.01	108.55	43.89	34.86	30.19	108.94	42.35	40.29	27.96	110.60	60.90	26.16	23.25	110.31
citeseer	63.77	72.76	54.59	191.12	80.57	63.11	49.91	193.59	80.40	68.32	43.98	192.70	121.12	37.62	34.44	193.18
pubmed	91.85	108.62	76.67	277.14	120.38	87.40	66.17	273.95	119.97	99.23	54.82	274.02	185.20	45.52	39.52	270.24
flickr	578.61	641.55	459.15	1679.31	720.82	534.60	464.41	1719.83	675.07	589.95	422.05	1687.07	871.29	117.31	82.04	1070.64

5.5 Regression accuracy analysis

Accurately predicting the performance of a model is important to manage the resources of GPU memory accurately. Eight lightweight regression models are selected for performance modeling in *vGNN*. We trained them on the generated dataset and then tested them on the real dataset, using several metrics to evaluate the models' performance. Table 7 shows the accuracy (MSE, MAE, MAPE) and inference cost comparison of regression models on GCN and GAT.

It can be seen that random forest regression usually achieves better accuracy, but its cost is the largest of all models. It is worth mentioning that, in the GCN, lasso regression has the lowest MSE. However, since MAPE takes into account the ratio of error to the true value, we choose random forest as the optimal model in the overall performance analysis. At the same time, if both the inference cost and prediction accuracy are valued, the linear and lasso regression model may be a good choice. When selecting a model, it is necessary to weigh accuracy and inference time according to the requirements of practical application scenarios.

5.6 Discussion and future work

At present, *vGNN* has considered and optimized the computation in heterogeneous distributed platforms. It can be seen from the experiments that on some graph datasets, the communication overhead cannot be ignored. Although *vGNN* also has certain advantages in communication, the advantage is generated by load balancing, not the optimization targeting at communication.

This advantage is generated by the optimization of computing, not the optimization of communication. For future work, we can consider further optimizing communication to improve the performance of *vGNN* on heterogeneous distributed platforms. Besides, the current experimental platform is only for GPUs from a single vendor. In a distributed scenario, there are also multiple clusters of devices from different vendors, such as AMD GPUs and Intel GPUs. How to model these devices and optimize computation is also a challenging issue. We would like to extend *vGNN* to support more diverse devices.

Since we use heuristics to estimate the workload of each partition, the accuracy of the performance model determines the performance of *vGNN*. As a result, if we change the model or use a unique graph dataset, the performance of *vGNN* may be affected. In the future, we will explore more accurate ways to estimate the workload of each partition to further improve the performance of *vGNN*.

Table 6 Standard deviation comparison of computation time among GPUs

Dataset	GCN				GAT			
	Frequency	GFLOPS	CUDA Core	*vGNN*	Frequency	GFLOPS	CUDA Core	*vGNN*
ogbn-arxiv	**0.12**	1.04	1.34	0.74	4.04	**1.31**	3.78	3.16
ogbn-proteins	7.20	4.88	4.17	**1.6**	18.81	8.44	8.77	**5.30**
ogbn-products	60.86	32.84	44.03	**25.89**	61.38	**24.21**	31.55	25.23
ogbn-mag	**19.81**	33.07	35.35	32.42	**127.51**	178.36	130.04	188.44
Yelp	6.48	5.01	6.96	**2.49**	9.78	1.67	7.39	**0.89**
Reddit	19.19	**7.37**	11.23	16.67	36.1	**13.18**	20.92	25.57
Cora	0.50	**0.11**	0.41	0.36	1.54	1.03	0.56	**0.26**
Citeseer	0.22	0.25	0.45	**0.10**	**0.33**	0.69	0.59	0.47
Pubmed	0.50	**0.23**	0.27	0.49	0.36	0.25	0.72	**0.12**
Flickr	0.47	**0.2**	0.83	0.47	0.93	1.28	0.44	**0.41**
Average	11.53	8.50	10.50	**8.12**	26.08	23.04	**20.48**	24.99

The bold entries denote the fastest time recorded for each dataset across different precision

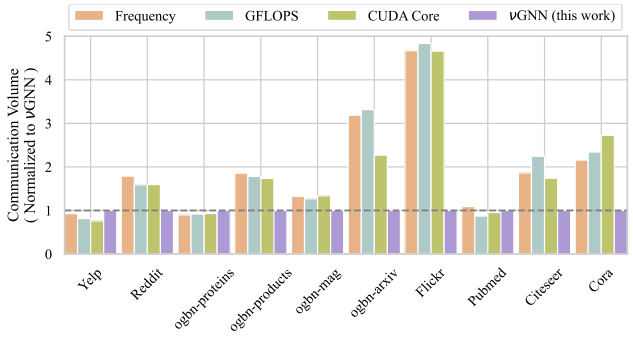


Fig. 12 The overall communication volume of different partitioning methods normalized to *vGNN* on GCN

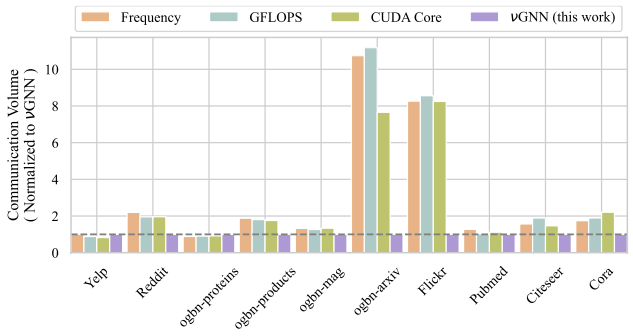


Fig. 13 The overall communication volume of different partitioning methods normalized to *vGNN* on GAT

6 Related work

6.1 GNN kernel optimization

In the research of GNN acceleration, one of the core challenges is how to improve the computational efficiency on graph processing. In recent years, kernel optimizations for GNN have emerged, aiming to improve performance.

GE-SpMM (Huang et al. 2020) improves access efficiency by utilizing shared memory to cache rows of the sparse matrix and by merging workloads across different warps. FeatGraph (Hu et al. 2020) optimizes cache utilization during GNN aggregation by integrating graph partitioning with feature dimension tiling. Huang et al. (2021) addresses the issue of load imbalance by clustering central vertices through locality-sensitive hashing and further partitioning the workload by grouping neighbors. GNNAdvisor (Wang et al. 2021b) reduces thread divergence by employing warp-aligned thread mapping along with neighbor and dimension partitioning. ES-SpMM (Lin et al. 2021) implements in-kernel edge sampling to downsize the graph for fitting into shared memory and eliminates the overhead associated with pre-processing. TC-GNN (Wang et al. 2021a) adapts the input graph to the dense computations of tensor cores by identifying non-zero tiles through sparse graph translation. QGTC (Wang et al. 2022) leverages Tensor Core to support arbitrary bit width computation for quantum graph neural networks (QGNNs) on GPUs. DA-SpMM (Dai et al. 2022) introduces a three-loop model to extract orthogonal design principles for SpMM on GPUs. HP-SpMM mixes node and edge computation tasks (Fan et al. 2023) and employs hierarchical vectorized memory access to enhance memory efficiency. TurboMGNN (Wu et al. 2023) fuses forward and backward propagation kernels, groups tasks, and matches their operators according to resource contention.

The above works can be combined with *vGNN* to improve the training and inferencing efficiency.

Table 7 Performance regression model accuracy and inference cost comparison

Model	GCN				GAT			
	MSE	MAE	MAPE	Time (ms)	MSE	MAE	MAPE	Time (ms)
Linear	8.20E–05	5.94E–03	2.57E–01	**2.60E+00**	1.31E–04	7.77E–03	2.51E–01	5.23E+01
Ridge	1.82E+00	9.46E–01	4.78E+01	4.42E+00	2.92E–01	4.34E–01	1.71E+01	6.89E+00
Decision tree	1.27E–04	5.23E–03	1.34E–01	3.88E+00	2.49E–04	5.54E–03	1.02E–01	4.00E+00
SVR	4.67E–03	6.64E–02	4.94E+00	3.00E+00	8.97E–03	9.32E–02	4.18E+00	**3.00E+00**
Lasso	**7.75E–05**	4.91E–03	1.71E–01	3.52E+00	1.23E–04	6.49E–03	1.80E–01	6.41E+00
AdaBoost	9.91E–05	4.85E–03	1.63E–01	2.77E+01	2.31E–04	1.08E–02	3.94E–01	7.64E+01
Bagging	1.03E–04	4.30E–03	1.04E–01	2.51E+01	1.12E–04	3.81E–03	7.67E–02	2.66E+01
Random forest	9.38E–05	**3.94E–03**	**1.01E–01**	2.14E+02	**7.05E–05**	**3.41E–03**	**6.92E–02**	1.91E+02

The bold entries denote the fastest time recorded for each dataset across different precision

6.2 GNN-customized accelerators

GNN accelerators employ various techniques such as dynamic distribution smoothing, ReRAM-based architectures, graph restructuring, near-memory processing, tensor decomposition, and hybrid precision quantization to address computation, memory, and I/O bottlenecks.

AWB-GCN (Geng et al. 2020) utilizes dynamic distribution smoothing, remote switching, and row remapping to accelerate GCN inference. ReGraphX (Arka et al. 2021) proposes a NoC-supported 3D heterogeneous ReRAM architecture for training GNN. Through the combination of heterogeneous ReRAM and 3D NoC, the performance and energy efficiency of GNN training are improved. I-GCN (Geng et al. 2021) introduces an online graph restructuring algorithm, islandization, to improve data locality and reduces unnecessary computation. MetaNMP (Li et al. 2022) is a DIMM-based near-memory processing HGNNs accelerator, which generates meta-path instances through Cartesian product paradigm, reduces redundant computation, and utilizes near-memory processing. Mao et al. (2023) propose a node-centered graph attention network based on ReRAM is proposed, which improves the computational efficiency and energy efficiency of GNNs through optimized hardware architecture and computing mode. TT-GNN (Qu et al. 2023) uses tensor-training decomposition to compress graph embedding matrices. Celeritas (Li et al. 2024 targets the memory and I/O bottlenecks in large-scale graph data processing through cross-layer computation and optimized I/O strategies. FlashGNN (Niu et al. 2024) implements critical computation of GNN training in SSDs, reducing data transmission and storage overhead and improving training efficiency. MEGA (Zhu et al. 2024) uses degree-aware hybrid precision quantization to reduce the memory footprint and improve the training efficiency.

Customized accelerators pose a challenge to the performance modeling part of *νGNN*, and new abstractions of hardware features are required to achieve accurate prediction results.

6.3 Distributed training system

In recent years, there has been many attempts to improve the efficiency of distributed GNN training with a variety of optimization techniques.

Roc (Jia et al. 2020) proposes an online-linear-regression-based strategy to achieve load balance, and coordinates optimized data transfers between GPU devices and host CPU memories with a dynamic programming algorithm. FlexGraph (Wang et al. 2021c) presents a GNN programming model, which utilizes hierarchical dependency graphs to express hierarchical dependencies among vertices. Dorylus (Thorpe et al. 2021) designs a computation separation mechanism and pipelines the different computation patterns in the Amazon EC2 machine and serverless Lambdas in the cloud environment. ByteGNN (Zheng et al. 2022b) abstracts the sampling phase of a mini-batch as a DAG of small tasks to support high parallelism, designs a two-level scheduling to improve resource utilization and reduce the end-to-end training time, tailors graph partitioning to reduce network I/O and balance the workload. GNNLab (Yang et al. 2022) proposes a space sharing design to tackle GPU memory contention and a pre-sampling based caching policy to accelerate sample-based GNN training. Legion (Sun et al. 2023) is a cache-based GNN system which proposes an NVLink-aware hierarchical partitioning technique and a novel hotness-aware unified cache mechanism with cache management. XGNN (Tang et al. 2024) utilizes global memory store abstraction to improve GPU memory efficiency, NVLink and host memory utilization.

However, previous works have not considered the mixed heterogeneous distributed scenarios, and *νGNN* filled the gap in such research.

7 Conclusion

In this paper, we propose *vGNN*, an efficient non-uniformly partitioned full-graph GNN training system on heterogeneous distributed platforms. The *vGNN* combines the performance modeling, the offline-online cooperative task assignment mechanism, and non-uniformed graph partitioning to accommodate the unbalanced computing capability of GPUs. The experimental results show that *vGNN* outperforms other partitioning schemes by a factor of, on average, 1.33 on GCN (up to 2.75) and 1.23 on GAT (up to 2.41), respectively.

Acknowledgements This work is supported by the National Natural Science Foundation of China (Grant No. 62402525), and the Fundamental Research Funds for the Central Universities (Grant No. 2462023YJRC023). Qingxiao Sun is the corresponding author.

Data availability The data that support the findings of this study are available from the first author (Hemeng Wang) upon reasonable request.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no Conflict of interest.

References

Arka, A.I., Doppa, J.R., Pande, P.P., Joardar, B.K., Chakrabarty, K.: Regraphx: Noc-enabled 3d heterogeneous reram architecture for training graph neural networks. In: Design, Automation & Test in Europe Conference & Exhibition, pp. 1667–1672 (2021)

Armeniakos, G., Zervakis, G., Soudris, D., Henkel, J.: Hardware approximate techniques for deep neural network accelerators: a survey. ACM Comput. Surv. **55**(4), 1–36 (2022)

Berg, R.v.d., Kipf, T.N., Welling, M.: Graph convolutional matrix completion. arXiv preprint arXiv:1706.02263, 1–9 (2017)

Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. In: Advances in neural information processing systems, 1–9. https://doi.org/10.5555/2999792.2999923 (2013)

Cheng, J., Dong, L., Lapata, M.: Long short-term memory-networks for machine reading. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, pp. 1–11 (2016)

Dai, G., Huang, G., Yang, S., Yu, Z., Zhang, H., Ding, Y., Xie, Y., Yang, H., Wang, Y.: Heuristic adaptability to input dynamics for spmm on gpus. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp. 595–600 (2022)

Dwivedi, V.P., Joshi, C.K., Luu, A.T., Laurent, T., Bengio, Y., Bresson, X.: Benchmarking graph neural networks. J. Mach. Learn. Res. **24**(43), 1–48 (2023)

Fan, W., Ma, Y., Li, Q., He, Y., Zhao, E., Tang, J., Yin, D.: Graph neural networks for social recommendation. In: The World Wide Web Conference, pp. 417–426 (2019)

Fan, R., Wang, W., Chu, X.: Fast sparse gpu kernels for accelerated training of graph neural networks. In: IEEE International Parallel and Distributed Processing Symposium, pp. 501–511 (2023)

Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds, pp. 1–9 (2019)

Geng, T., Li, A., Shi, R., Wu, C., Wang, T., Li, Y., Haghi, P., Tumeo, A., Che, S., Reinhardt, S., etal.: Awb-gcn: a graph convolutional network accelerator with runtime workload rebalancing. In: 53rd Annual IEEE/ACM International Symposium on Microarchitectur, pp. 922–936 (2020)

Geng, T., Wu, C., Zhang, Y., Tan, C., Xie, C., You, H., Herbordt, M., Lin, Y., Li, A.: I-gcn: a graph convolutional network accelerator with runtime locality enhancement through islandization. In: 54th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 1051–1063 (2021)

Hakhamaneshi, K., Nassar, M., Phielipp, M., Abbeel, P., Stojanovic, V.: Pretraining graph neural networks for few-shot analog circuit modeling and design. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **42**(7), 2163–2173 (2022)

Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Adv. Neural Inform. Process. Syst. **30**, 1–11 (2017)

Hu, Y., Ye, Z., Wang, M., Yu, J., Zheng, D., Li, M., Zhang, Z., Zhang, Z., Wang, Y.: Featgraph: a flexible and efficient backend for graph neural network systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–13 (2020)

Huang, G., Dai, G., Wang, Y., Yang, H.: Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2020)

Huang, K., Zhai, J., Zheng, Z., Yi, Y., Shen, X.: Understanding and bridging the gaps in current gnn performance optimizations. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 119–132 (2021)

Jia, Z., Lin, S., Gao, M., Zaharia, M., Aiken, A.: Improving the accuracy, scalability, and performance of graph neural networks with roc. In: Proceedings of Machine Learning and Systems, vol. 2, pp. 187–198 (2020)

Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)

Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 1–14 (2016a)

Kipf, T.N., Welling, M.: Variational graph auto-encoders. arXiv preprint arXiv:1611.07308, 1–3 (2016b)

Li, S., Niu, D., Wang, Y., Han, W., Zhang, Z., Guan, T., Guan, Y., Liu, H., Huang, L., Du, Z., et al.: Hyperscale fpga-as-a-service architecture for large-scale distributed graph neural network. In: Proceedings of the 49th Annual International Symposium on Computer Architecture, pp. 946–961 (2022)

Li, Y., Yang, T.-Y., Yang, M.-C., Shen, Z., Li, B.: Celeritas: Out-of-core based unsupervised graph neural network via cross-layer computing 2024. In: IEEE International Symposium on High-Performance Computer Architecture, pp. 91–107 (2024)

Lin, C.-Y., Luo, L., Ceze, L.: Accelerating spmm kernel with cache-first edge sampling for graph neural networks. arXiv preprint arXiv:2104.10716, 1–12 (2021)

Lin, H., Yan, M., Ye, X., Fan, D., Pan, S., Chen, W., Xie, Y.: A comprehensive survey on distributed training of graph neural networks. Proc. IEEE **111**(12), 1572–1606 (2023)

Mao, R., Sheng, X., Graves, C., Xu, C., Li, C.: Reram-based graph attention network with node-centric edge searching and hamming similarity. In: Proceedings of the 60th ACM/IEEE Design Automation Conference, pp. 1–6 (2023)

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al.: Mixed precision training. In: International Conference on Learning Representations, pp. 1–12 (2018)

Mostafa, H.: Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. In: Proceedings of Machine Learning and Systems, pp. 1–11 (2022)

Niu, F., Yue, J., Shen, J., Liao, X., Jin, H.: Flashgnn: An in-ssd accelerator for gnn training. In: IEEE International Symposium on High-Performance Computer Architecture, pp. 361–378 (2024)

Qu, Z., Niu, D., Li, S., Zheng, H., Xie, Y.: Tt-gnn: Efficient on-chip graph neural network training via embedding reformation and hardware optimization. In: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 452–464 (2023)

Schlichtkrull, M., Kipf, T.N., Bloem, P., Van Den Berg, R., Titov, I., Welling, M.: Modeling relational data with graph convolutional networks. In: The Semantic Web, pp. 593–607 (2018)

Shao, Y., Li, H., Gu, X., Yin, H., Li, Y., Miao, X., Zhang, W., Cui, B., Chen, L.: Distributed graph neural network training: a survey. ACM Comput. Surv. **56**(8), 1–39 (2024)

Sun, J., Su, L., Shi, Z., Shen, W., Wang, Z., Wang, L., Zhang, J., Li, Y., Yu, W., Zhou, J., et al.: Legion: Automatically pushing the envelope of multi-gpu system for billion-scale gnn training. In: USENIX Annual Technical Conference, pp. 165–179 (2023)

Tang, D., Wang, J., Chen, R., Wang, L., Yu, W., Zhou, J., Li, K.: Xgnn: boosting multi-gpu gnn training via global gnn memory store. Proc. VLDB Endow. **17**(5), 1105–1118 (2024)

Thorpe, J., Qiao, Y., Eyolfson, J., Teng, S., Hu, G., Jia, Z., Wei, J., Vora, K., Netravali, R., Kim, M., et al.: Dorylus: affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads. In: USENIX Symposium on Operating Systems Design and Implementation, pp. 495–514 (2021)

Tripathy, A., Yelick, K., Buluç, A.: Reducing communication in graph neural network training. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14 (2020)

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł, Polosukhin, I.: Attention is all you need. Adv. Neural. Inf. Process. Syst. **30**, 1–11 (2017)

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y.: Graph attention networks. In: International Conference on Learning Representations, pp. 1–12 (2018)

Wan, C., Li, Y., Li, A., Kim, N.S., Lin, Y.: Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In: Proceedings of Machine Learning and Systems, vol. 4, pp. 673–693 (2022)

Wan, X., Xu, K., Liao, X., Jin, Y., Chen, K., Jin, X.: Scalable and efficient full-graph gnn training for large graphs. Proc. ACM Manag. Data **1**(2), 1–23 (2023)

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., Zhang, Z.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315, 1–18 (2019)

Wang, Y., Feng, B., Ding, Y.: Tc-gnn: Accelerating sparse graph neural network computation via dense tensor core on gpus. arXiv preprint arXiv:2112.02052, 1–14 (2021a)

Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., Ding, Y.: Gnnadvisor: an adaptive and efficient runtime system for gnn acceleration on gpus. In: USENIX Symposium on Operating Systems Design and Implementation, pp. 515–531 (2021b)

Wang, L., Yin, Q., Tian, C., Yang, J., Chen, R., Yu, W., Yao, Z., Zhou, J.: Flexgraph: a flexible and efficient distributed framework for gnn training. In: Proceedings of Machine Learning and Systems, pp. 67–82 (2021c)

Wang, Y., Feng, B., Ding, Y.: Qgtc: accelerating quantized graph neural networks via gpu tensor core. In: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 107–119 (2022)

Wu, N., Yang, H., Xie, Y., Li, P., Hao, C.: High-level synthesis performance prediction using gnns: benchmarking, modeling, and advancing. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp. 49–54 (2022)

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y.: A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. **32**(1), 4–24 (2020)

Wu, W., Shi, X., He, L., Jin, H.: Turbomgnn: improving concurrent gnn training tasks on gpu with fine-grained kernel fusion. IEEE Trans. Parallel Distrib. Syst. **34**(6), 1968–1981 (2023)

Xu, R., Ma, S., Guo, Y., Li, D.: A survey of design and optimization for systolic array-based dnn accelerators. ACM Comput. Surv. **56**(1), 1–37 (2023)

Xu, S., Huang, Z., Zeng, Y., Yan, S., Ning, X., Ye, H., Gu, S., Shui, C., Lin, Z., Zhang, H., et al.: Hethub: a heterogeneous distributed hybrid training system for large-scale models. arXiv preprint arXiv:2405.16256, 1–13 (2024)

Yang, J., Tang, D., Song, X., Wang, L., Yin, Q., Chen, R., Yu, W., Zhou, J.: Gnnlab: a factored system for sample-based gnn training over gpus. In: Proceedings of the 17th European Conference on Computer Systems, pp. 417–434 (2022)

Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., Leskovec, J.: Hierarchical graph representation learning with differentiable pooling. In: Advances in neural information processing systems, 1–11. https://doi.org/10.5555/3327345.3327423 (2018)

Zhang, M., Chen, Y.: Link prediction based on graph neural networks. In: Advances in neural information processing systems, 1–11. https://doi.org/10.5555/3327345.3327389 (2018)

Zhang, M., Cui, Z., Neumann, M., Chen, Y.: An end-to-end deep learning architecture for graph classification. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 4438–4445 (2018)

Zhang, X., Tan, G., Xue, S., Li, J., Zhou, K., Chen, M.: Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 31–43 (2017)

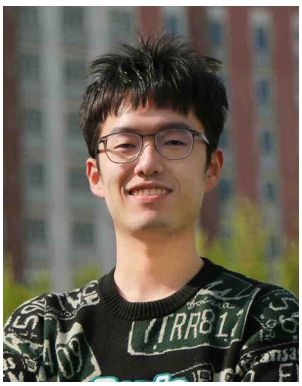
Zhang, D., Huang, X., Liu, Z., Zhou, J., Hu, Z., Song, X., Ge, Z., Wang, L., Zhang, Z., Qi, Y.: Agl: a scalable system for industrial-purpose graph machine learning. Proc. VLDB Endow. **13**(12), 3125–3137 (2020)

Zheng, D., Song, X., Yang, C., LaSalle, D., Karypis, G.: Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In: Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, vol. 32, pp. 1030–1043 (2022a)

Zheng, C., Chen, H., Cheng, Y., Song, Z., Wu, Y., Li, C., Cheng, J., Yang, H., Zhang, S.: Bytegnn: efficient graph neural network training at large scale. Proc. VLDB Endow. **15**(6), 1228–1242 (2022b)

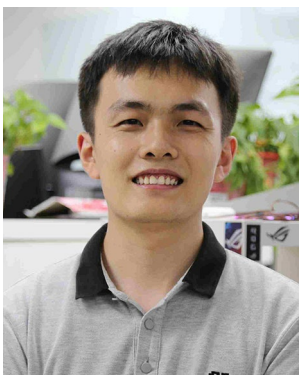
Zhou, Q., Guo, S., Qu, Z., Li, P., Li, L., Guo, M., Wang, K.: Petrel: heterogeneity-aware distributed deep learning via hybrid synchronization. IEEE Trans. Parallel Distrib. Syst. **32**(5), 1030–1043 (2020)

Zhu, Z., Li, F., Li, G., Liu, Z., Mo, Z., Hu, Q., Liang, X., Cheng, J.: Mega: a memory-efficient gnn accelerator exploiting degree-aware mixed-precision quantization. In: IEEE International Symposium on High-Performance Computer Architecture, pp. 124–138 (2024)

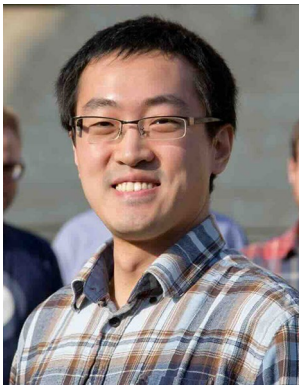


Hemeng Wang received his BE degree in Computer Science and Technology in China University of Petroleum-Beijing in 2021. He is studying for the full time Doctoral degree, majoring in Advanced Science and Engineering Computing in College of Artificial Intelligence in China University of Petroleum-Beijing. His research interests are in high performance computing, with a particular focus on parallel and distributed computing, numerical linear algebra and deep learning systems.

Wenqing Lin is studying for the full time Bachelor degree, majoring in Computer Science and Technology in College of Artificial Intelligence in China University of Petroleum-Beijing. Her research interests are in high performance computing, with a particular focus on sparse tensor computation, parallel computing and deep learning systems.



Qingxiao Sun is currently an associate professor at the China University of Petroleum-Beijing. He was awarded with ACM SIGHPC China Doctoral Disseration Award and CCF TCARCH Doctoral Disseration Award. He received his PhD in 2023 from Beihang University under supervision of Prof. Yi Liu and Asso. Prof. Hailong Yang. His research interests include high performance computing, computer architecture, deep learning system and parallel computing. His recent research invetigates performance auto-tuning, GPU architecture extension, runtime mechanism and graph neural network training. He has authored about 20 publications in the leading international journals and conferences. His papers have been selected as CLUSTER '21 Best Paper Nomination and IEEE Computer's Spotlight on Transactions. He currently serves as reviewers in the premier journals including TPDS, TC, TCC and THPC.



Weifeng Liu is currently a Full Professor at the China University of Petroleum–Beijing. Formerly, he was a Marie Curie Fellow at the Norwegian University of Science and Technology. He received his PhD in 2016 from the Niels Bohr Institute of the University of Copenhagen under advisor Brian Vinter. He has been shortly working as a Research Associate with Iain S. Duff at the STFC Rutherford Appleton Laboratory in 2016. He also has been working as a Senior Researcher in high performance computing technology at the SINOPEC Exploration and Production Research Institute for about six years (2006-2012). He received his BE and ME degrees in computer science, both from the China University of Petroleum–Beijing, in 2002 and 2006, respectively. He is a Senior Member of the IEEE and a Member of the ACM and the SIAM. His research interests are in high performance numerical linear algebra, in particular include domain specific architectures, data structures, parallel and distributed algorithms, linear solver mathematical software for sparse matrix computations.