A Unified CPU-GPU Protocol for GNN Training

Yi-Chien Lin*
yichienl@usc.edu
University of Southern California
Los Angeles, California, USA

Gangda Deng*
gangdade@usc.edu
University of Southern California
Los Angeles, California, USA

Viktor Prasanna
prasanna@usc.edu
University of Southern California
Los Angeles, California, USA

ABSTRACT

Training a Graph Neural Network (GNN) model on large-scale graphs involves a high volume of data communication and computations. While state-of-the-art CPUs and GPUs feature high computing power, the Standard GNN training protocol adopted in existing GNN frameworks cannot efficiently utilize the platform resources. To this end, we propose a novel Unified CPU-GPU protocol that can improve the resource utilization of GNN training on a CPU-GPU platform. The Unified CPU-GPU protocol instantiates multiple GNN training processes in parallel on both the CPU and the GPU. By allocating training processes on the CPU to perform GNN training collaboratively with the GPU, the proposed protocol improves the platform resource utilization and reduces the CPU-GPU data transfer overhead. Since the performance of a CPU and a GPU varies, we develop a novel load balancer that balances the workload dynamically between CPUs and GPUs during runtime. We evaluate our protocol using two representative GNN sampling algorithms, with two widely-used GNN models, on three datasets. Compared with the Standard training protocol adopted in the state-of-the-art GNN frameworks, our protocol effectively improves resource utilization and improves the overall training time. On a platform where the GPU moderately outperforms the CPU, our protocol speeds up GNN training by up to 1.41×. On a platform where the GPU significantly outperforms the CPU, our protocol speeds up GNN training by up to 1.26×. Our protocol is open-sourced and can be seamlessly integrated into state-of-the-art GNN frameworks and accelerate GNN training. Our protocol particularly benefits those with limited GPU access due to its high demand.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**.

KEYWORDS

GNN, Unified CPU-GPU protocol, GNN training

ACM Reference Format:
Yi-Chien Lin, Gangda Deng, and Viktor Prasanna. 2024. A Unified CPU-GPU Protocol for GNN Training. In *21st ACM International Conference on Computing Frontiers (CF '24), May 7–9, 2024, Ischia, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3649153.3649191

*Both authors contributed equally to this research.

CF '24, May 7–9, 2024, Ischia, Italy

ACM ISBN 979-8-4007-0597-7/24/05.
https://doi.org/10.1145/3649153.3649191

1 INTRODUCTION

Graph Neural Network (GNN) is an emerging type of Machine Learning model that can extract useful information from graph-structured data. GNNs are widely used in various applications such as Electronic Design Automation [22, 34], molecular property prediction [6, 36], and social recommendation system [20, 38, 45], where the input is often a large-scale graph with over billion edges. Training a GNN model on these real-world graphs involves a high volume of data communications and computations. While state-of-the-art CPUs and GPUs feature high compute power and memory bandwidth, existing GNN training protocols adopted in popular frameworks like PyTorch-Geometric (PyG) [3] and Deep Graph Library (DGL) [32] cannot efficiently utilize the available platform resources; therefore, large-scale GNN training is time-consuming, taking hours or even days [30]. We conduct a detailed analysis of state-of-the-art GNN frameworks and observe several inefficiencies in the existing Standard GNN training protocols. First, Standard GNN training protocols (shown in Figure 1) offload most of the workload to the GPUs, leaving the CPUs mostly idle. Such a task allocation is sub-optimal as state-of-the-art CPUs offer comparable GNN training performance with GPUs. For example, the epoch time of training a three-layer GCN [10] model on the ogbn-products [8] dataset takes 10 seconds using an Intel Xeon 8280 with only 28 cores [23], and takes 5 seconds using a high-end NVIDIA V100 GPU [37]. This suggests that CPUs should be incorporated in GNN training due to their potential to enhance overall performance. In addition, existing GNN training protocols adopt a coarse-grained task scheduling, alternating between memory-intensive and compute-intensive operations. This approach leads to sub-optimal resource utilization: the memory bandwidth is under-utilized during compute-intensive tasks, and the compute cores are under-utilized during memory-intensive tasks.

Motivated by the above challenges, we propose a novel Unified CPU-GPU training protocol that improves the resource utilization of GNN training on a CPU-GPU platform. The Unified CPU-GPU protocol defines the CPU-GPU interaction to perform GNN training collaboratively. Note that our protocol not only supports platforms with a single CPU and single GPU, but also platforms with multiple CPUs and multiple GPUs. Our protocol instantiates GNN training processes on both the CPUs and the GPUs, as opposed to offloading all the workload to the GPUs. Allocating GNN training processes on the CPUs to share a portion of the workload leads to several advantages: (1) It allows our protocol to exploit the compute cores of CPUs to execute GNN operations, leading to higher platform resource utilization than the Standard training protocol, and reduces the workload on the GPUs; (2) It reduces the data transferred to the GPU via the PCIe channel, which often causes significant overhead; (3) It reduces the memory usage on the GPU global memory, and the memory space that becomes available can be used for

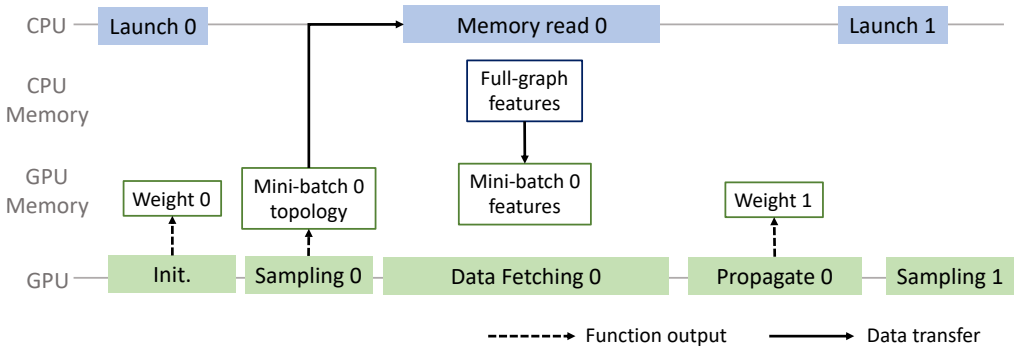

Figure 1: Standard training protocol used in state-of-the-art GNN frameworks

data caching, which further reduces the expensive PCIe data transfer. Furthermore, instantiating multiple GNN training processes improves memory bandwidth utilization because it overlaps the communications with computations across the processes. Since the performance of the CPU and GPU for training a GNN model varies, we develop a Dynamic Load Balancer to ensure the workload is balanced; otherwise, one platform can easily become the system bottleneck and lead to performance degradation. As GPUs often face high demand from multiple users, and the CPUs are becoming increasingly powerful, our protocol exploits the CPU resources, which are under-utilized by the Standard training protocol, to improve GNN training performance. Note that none of the proposed optimizations alter the GNN training semantics, meaning that training with the Unified CPU-GPU protocol leads to the same model accuracy and convergence rate as the Standard training protocol. Our key contributions are:

- We conduct a detailed analysis of the state-of-the-art GNN frameworks to identify the inefficiencies in the Standard GNN training protocol.
- We propose a novel Unified CPU-GPU protocol for GNN training that can effectively improve the utilization of both the compute resources and memory bandwidth.
- We develop a novel Dynamic Load Balancer that effectively balances the workload between the CPUs and the GPUs dynamically during runtime.
- We evaluate our work using various CPU-GPU platforms: on a platform where the GPU moderately outperforms the CPU, our protocol accelerates GNN training by up to 1.41×; on a platform where the GPU significantly outperforms the CPU, our protocol accelerates GNN training by up to 1.26×.
- Our protocol is open-sourced[1] and can seamlessly integrate into existing GNN frameworks such as PyG and DGL to improve GNN training performance.

2 BACKGROUND

2.1 Graph Neural Networks

Given a graph $G = (V, E)$ with $N = |V|$ nodes and $|E|$ edges. Let $A \in \mathbb{R}^{N \times N}$ be the adjacency matrix with an entry (i, j) equal to 1 if there exists an edge between node i and j, otherwise 0. For each node $v \in V$, $N(v)$ is the set of neighbors of v, and x_v is a F-dimensional feature vector associated with v. We use $H^0 \in \mathbb{R}^{N \times F}$ to denote the input feature matrix of V. A Graph Neural Network (GNN) is a neural network that operates on graph-structured data with node-related features as input. By aggregating feature information through graph structure and transforming the features into d-dimensional latent space, GNNs are able to output representations of nodes containing higher-order neighbor information. GNN models [6, 10, 31, 35] follow the above two operations (i.e., aggregation and transformation) and can be described using the Message-Passing paradgim [5]. We list two representative GNN models as examples:

- GCN [10] is one of the most widely used GNN models. The l-th layer of GCN can be defined as follows:

$$H^{(l)} = \sigma\left(\hat{A}H^{(l-1)}W^{(l)}\right), \tag{1}$$

  $W^{(l)}$ and $H^{(l)}$ indicates the weight matrix and feature matrix of layer l, respectively. $\hat{A}$ is the normalized and regularized adjacency matrix and $\sigma(\cdot)$ is the activation function. Multiplying $\hat{A}$ with $H^{(l-1)}$ aggregates the feature information, while multiplying $\hat{A}H^{(l-1)}$ with $W^{(l)}$ transforms the aggregated features into a d-dimensional latent space.
- GraphSAGE [6] added a self-concatenation upon the GCN layer. The l-th layer of GraphSAGE can be defined as follows:

$$H^{(l)} = \sigma\left(H^{(l-1)}W_1^{(l)} + \hat{A}H^{(l-1)}W_2^{(l)}\right). \tag{2}$$

Our Unified GNN training protocol supports all GNN models that follow the Message-Passing paradigm, including other widely used models such as GIN [35] and GAT [31].

2.2 Mini-batch SGD GNN Training

GNNs are initially trained using full gradient descent [10], which takes the whole adjacency matrix A and all nodes' features H^0 as input. However, full gradient descent suffers from unacceptable memory costs on large graphs and requires more epochs to converge since the model is updated only once per epoch. To address these issues, mini-batch SGD has been proposed for GNN training [6]. Unlike full gradient descent, mini-batch SGD does not require computing the gradient for all the nodes in the graph at once. Instead, it samples a subgraph based on a batch of node indices and

[1] https://github.com/jasonlin316/A-Unified-CPU-GPU-Protocol-for-GNN-Training

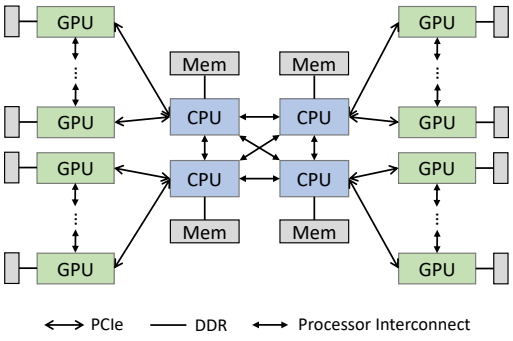

Figure 2: Target CPU-GPU platform

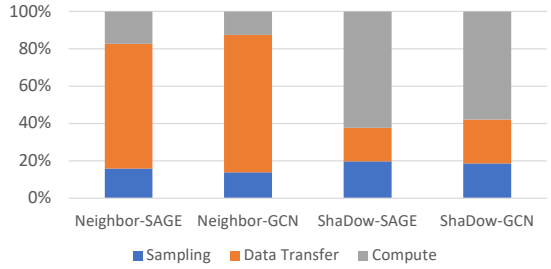

Figure 3: Training time breakdown of existing GNN library for various sampling algorithms and GNN models

applies the GNN model on top of the subgraph to calculate the gradient. Shown in Figure 1, to perform mini-batch training, the full graph can be stored in the CPU memory. After a mini-batch is sampled, the CPU transfers the node embeddings of the sampled nodes to the GPU memory for model propagations (i.e., data fetching). Let $\mathcal{B} \subseteq V$ with size $b = |\mathcal{B}|$ denote a batch of vertices. The update in every SGD step is based on the following gradient estimation:

$$\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla \mathcal{L} \left(y_i, h_i^{(L)} \right), \tag{3}$$

where $h_i^{(L)}$ is the i-th row of $H^{(L)}$ with ground truth label y_i. The mini-batch can be obtained via various approaches [2, 6, 40, 41]; we list two representative sampling algorithms as examples:

- Neighbor Sampling [6] randomly selects neighbors for each node and enforces a predefined budget on the sample size for each layer. For a GNN model with L layers, it samples a subgraph within L-hops for each root node.
- ShaDow K-Hop Sampling [40] is a variant of Neighbor Sampling. It applies an L-layer GNN (with arbitrary depth L) on top of a localized L'-hop subgraph associated with each root node, where L is often set to 5 and L' is set to 2. Each shallow subgraph is sampled using Neighbor Sampling.

We adopt synchronous SGD [1, 14] to train GNNs on multiple devices, as it is widely used for multi-GPU training [4, 21] and distributed training across multiple machines [23, 43]; most importantly, it is algorithmically equivalent to training with a larger mini-batch on a single device, as long as the sample gradient calculation $\nabla \mathcal{L} \left(y_i, h_i^{(L)} \right)$ is independent between different samples i. For synchronous SGD training, multiple sub-mini-batches are first sampled and assigned to different devices. Next, forward propagations are performed on each device based on the original GNN algorithm. Finally, gradients are gathered and averaged across devices to calculate the actual mini-batch gradient for model updates.

2.3 Target CPU-GPU Platform

Figure 2 shows the target CPU-GPU platform. The platform consists of multiple CPUs and multiple GPUs. The CPUs and GPUs are connected through PCIe. The CPUs are connected to other CPUs via processor interconnects such as UPI; similarly, the GPUs are connected to other GPUs via GPU-specific processor interconnects such as Nvidia NVLink [13]. Each CPU and GPU is connected to a memory via DDR memory channel. State-of-the-art libraries like CUDA and SYCL provide users with a unified view of the CPU and GPU memory system, accessing both the CPU main memory and GPU global memory as a shared memory space. However, it's important to note that the shared memory space actually consists of various memory channels, such as PCIe, DDR, and UPI, each with a different bandwidth. Thus, to achieve high performance, it is essential to exploit optimizations to mitigate the overhead of accessing channels with low memory bandwidth (Section 4.3).

3 UNIFIED CPU-GPU PROTOCOL FOR GNN TRAINING

To better understand the inefficiencies in the Standard GNN training protocols adopted by the state-of-the-art GNN frameworks, we use Deep Graph Library (DGL) [32] as an example and perform profiling on the MAG240M [7] dataset with several sampling algorithms and GNN models. As shown in Figure 3, the runtime of GNN training with the Neighbor Sampling algorithm is dominated by the data transfer overhead between CPU and GPU. On the other hand, under the ShaDow K-Hop Sampling algorithm, the system bottleneck shifts to computation. Both cases suggest that using CPU in GNN training can improve performance by handling a portion of the workload. By doing so, both the CPU-GPU data traffic and the workload on the GPU can be reduced. Furthermore, our profiling results show that the CPU is heavily under-utilized: the average CPU utilization for GNN training is below 3%. This suggests a potential for assigning some of the workload to the CPU platform.

We propose a novel Unified CPU-GPU Protocol, which defines the CPU-GPU interaction to perform GNN training, collaboratively. Our protocol supports platforms with multiple CPUs and GPUs. For simplicity, in Figure 4, we illustrate our protocol using a platform with one CPU and two GPUs as an example. We first explain the meaning of the suffix in each function block and data block: the first number indicates the number of the training iteration; the second alphabet in the parentheses indicates the device type (i.e., CPU or GPU) that holds the data or executes the function; the last number indicates the device number (e.g., GPU 1, GPU 2). For example, `Sampling 0(G.1)` indicates the sampling operation performed in training iteration 0 on GPU 1; Similarly, `Grad. 0(C.1)` indicates the local gradient generated in training iteration 0 on CPU 1.

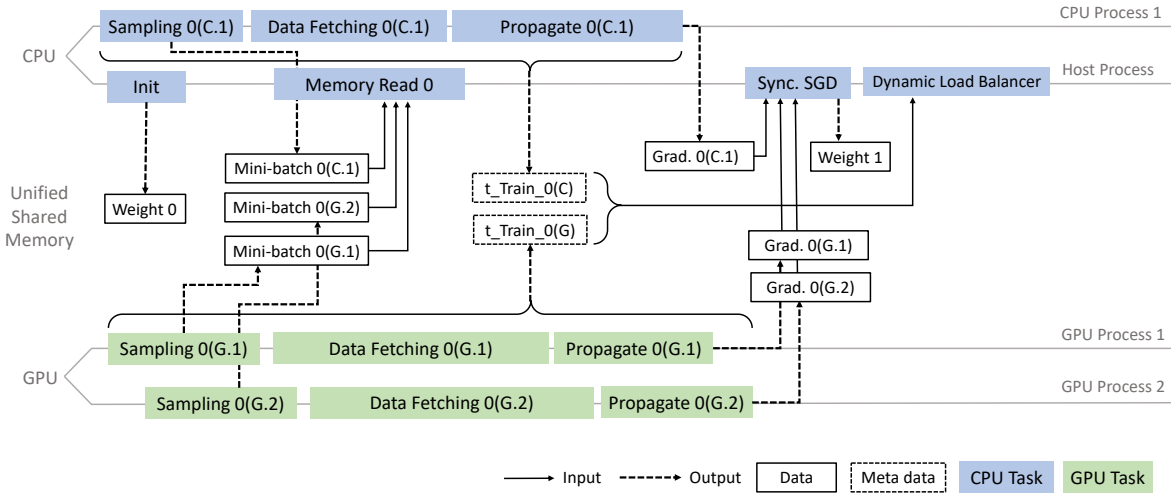

Figure 4: Unified CPU-GPU training protocol

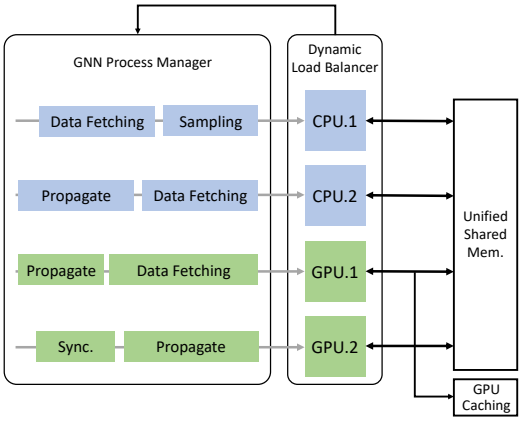

Figure 5: System Overview

Our protocol adopts the shared memory model for data exchange between the CPUs and the GPUs, i.e., it creates a unified shared memory space as mentioned in Section 2.3. The CPU platform consists of two types of processes: the Host Process and the CPU Process. The Host Process is responsible for launching GNN tasks, setting barriers for synchronization, and load balancing. The CPU Process is responsible for executing GNN operations, such as sampling and model propagation (i.e., forward and backward propagations), on the CPU platform. Similar to the Standard training protocol depicted in Figure 1, the sampling operation constructs subgraphs for each mini-batch. The node and edge sets of these subgraphs are then used for data fetching to load the corresponding feature vectors from the full graph for model propagation. We omit the data blocks of the full-graph features and mini-batch features in Figure 4 for simplicity. The CPU Processes allow our protocol to exploit CPU compute cores to perform GNN training, which reduces the workload on the GPU. Then, the GPU Processes execute the GNN operations on the GPU platform. After the GNN operations are performed, each CPU and GPU Process generates a local gradient; the local gradients are then gathered to perform a synchronous stochastic gradient descent (Sync. SGD) [1] to update the GNN model globally. In addition to model updates, the Host Process collects the runtime information for the Dynamic Load Balancer. The Dynamic Load Balancer performs workload-aware sub-batch assignments among the CPUs and the GPUs based on a certain ratio; if the ratio leads to workload imbalance, the Dynamic Load Balancer adjusts the ratio for the next epoch using the collected runtime information. We explain the mechanism of Dynamic Load Balancer in detail in Section 4.2.

4 SYSTEM DESIGN

We depict the overview of our Unified CPU-GPU training system in Figure 5. Our system consists of several building blocks to execute the Unified CPU-GPU protocol. It is worth noting that none of the optimization applied in our system alters the GNN training semantics, meaning that our system does not trade off model accuracy or convergence rate for speed.

4.1 GNN Process Manager

To perform Unified CPU-GPU training, we develop a GNN Process Manager that allocates GNN training processes on both the CPU and the GPU. On the CPU platform, two processes are running, one being the host process and the other being the CPU process, and on the GPU platform, n process is running simultaneously, where n is the number of GPUs on the platform. Compared with only allocating GNN training processes on the GPU, this leads to performance improvement because allocating GNN processes on the CPU platform reduces the workload on the GPU, and also reduces the amount of data that needs to be transferred to the GPU via the PCIe. Furthermore, instantiating multiple GNN training processes increases memory bandwidth utilization by overlapping the computations with communications. We show an example in Figure 5: while some processes perform data fetching (communications), others perform mini-batch sampling or model propagation (computations). Note that such scheduling does not alter the GNN training

semantic because synchronous SGD only requires each process to synchronize at the end of each iteration, so it is valid to have processes executing different tasks (e.g., sampling, propagation) during training.

4.2 Dynamic Load Balancer

To train GNN with multiple devices, such as multiple GPUs, the state-of-the-art runtime system [14] assigns the same number of mini-batches to each device by default. However, since the performance of CPU and GPU can be different, assigning the same number of mini-batches to the CPU and the GPU leads to workload imbalance; therefore, we develop a load balancer to distribute the mini-batches. A straightforward way to achieve load balancing on the CPU-GPU platform is to distribute the mini-batches to the CPU and GPU based on their relative performance; the relative performance can be derived by running one epoch of training on both platforms and measuring the execution time respectively. For example, assume the GPUs are 2× faster than the CPUs; to balance the workload, the system can assign 1/3 of the mini-batches to the CPUs and 2/3 of the mini-batches to the GPUs. We refer to this balancing mechanism as "Static Load Balancing," as the number of mini-batches is statically assigned to the CPUs and GPUs. We define the *workload ratio* as the workload assignment between the CPU and the GPU; the workload ratio of the above example is 1:2. If the current workload ratio leads to workload imbalance, the Load Balancer adjusts the workload ratio accordingly in the next iteration. Specifically, the Load Balancer collects the execution time of CPU and GPU, and recalculates the workload ratio to balance the workload. Static Load Balancing relies on the assumption that the workload of training on each mini-batch is similar (i.e., uniform workload distribution), so the number of mini-batches can be used to estimate the total workload; however, this is not true for some datasets, where the mini-batches have a highly skewed workload distribution. In such cases, workload balancing cannot be achieved merely by considering the number of mini-batches.

We develop a Dynamic Load Balancer to address the issue that the mini-batches can have a highly skewed workload distribution. Given a sampling algorithm and a dataset, the Dynamic Load Balancer estimates the workload of each mini-batch in advance during the pre-processing stage; the estimation is performed by running the sampling algorithm before the actual GNN model training and calculating the total number of aggregations that need to be performed using the computational graph of the mini-batches. The estimation process is a one-time cost that can amortized. Afterward, similar to Static Load Balancing, the Dynamic Load Balancer estimates the workload ratio that balances the workload between the CPU and the GPU. Assume the GPUs are 2× faster than the CPUs. Instead of statically assigning 1/3 of the mini-batches to the CPU, the Dynamic Load Balancer sorts the mini-batches by their estimated workload and assigns a certain number of mini-batches that account for 1/3 of the total workload to the CPU. This workload-aware sub-batch assignment leads to better load balancing than statically assigning a fixed number of mini-batches to the CPUs and GPUs. Since the number of mini-batches assigned to the CPUs and GPUs is adjusted dynamically during runtime, we refer to this mechanism as "Dynamic Load Balancing." After training for one epoch, the Dynamic Load Balancer collects the runtime information to evaluate the workload ratio. If the workload is imbalanced, the Dynamic Load Balancer recalculates the workload ratio using the collected runtime information, and adjusts the ratio accordingly.

4.3 GPU Feature Caching

By assigning a portion of the mini-batches to the CPU, the Unified CPU-GPU protocol reduces the amount of CPU-GPU data communication, as well as the usage of the GPU global memory; the additional memory capacity allows us to perform GPU feature caching, which stores a set of frequently accessed feature vectors in the GPU global memory. Such a mechanism reduces the expensive CPU-GPU communication overhead via PCIe during the data fetching operation (see Figure 4 for detail). Specifically, when loading the feature vectors of a mini-batch from the unified shared memory, if a vector resides in the GPU global memory, it eliminates the need for memory access over the PCIe. In this work, we integrate a GPU feature caching approach, as proposed by HugeCTR [33], into our system. This approach utilizes the least recently used cache replacement strategy (LRU) to retain the most recently accessed features on the GPU. Note that for large-scale datasets, most feature vectors are stored in the CPU main memory, which has a greater capacity (several terabytes) than the GPU global memory (several gigabytes). This storage disparity results in high communication overhead, as most data must be fetched from the CPU main memory. Such a characteristics makes GPU feature caching particularly useful, as it reduces the data traffic of fetching feature vectors from the CPU main memory.

5 EXPERIMENTS

5.1 Experiemental Setup

5.1.1 Environment. We conduct our experiments on two CPU-GPU platforms. Both platforms are equipped with the same CPU, which is a two-socket Intel Xeon Gold 6326; the CPU and GPU are connected via PCIe. Platform 1 consists of a high-end NVIDIA A100 GPU with the Multi-Instance GPU (MIG) [29] technology enabled. MIG is a technology that allows GPU resources to be shared efficiently on the cloud by partitioning the GPU into multiple instances. Such a setup is becoming increasingly important in real life as data center GPUs like A100 are often shared by multiple users. In addition, the MIG setup aligns with our protocol, which aims to improve performance under scenarios with limited GPU access. We adopt the 3g.20gb configuration, which partitions the A100 into two instances. Note that even with MIG enabled, A100 (3g.20gb) still offers superior peak performance compared with the CPU platform (Table 1). Platform 2 consists of a workstation NVIDIA RTX A5000 GPU. We list the detailed specifications of the CPUs and GPUs used in our experiments in Table 1. We implement our design using Python v3.8, PyTorch v2.0.1 with CUDA 11.7, and Deep Graph Library (DGL) v1.1.2. To profile the CPU utilization and memory bandwidth, we use the Intel VTune Profiler [28]; to profile the GPU utilization, we use the NVIDIA System Management Interface (nvidia-smi) [25].

5.1.2 Sampler, GNN Models, and Datasets. To evaluate our system, we use three large-scale datasets: Reddit [41], ogbn-products

Table 1: Specifications of the platforms

Platforms	CPU	Platform 1	Platform 2
Devices	Intel Xeon 6326	NVIDIA A100 (3g.20gb)	NVIDIA RTX A5000
Frequency	2.9 GHz	1405 MHz	2000 MHz
Peak Performance	537 GFLOPS	8.36 TFLOPS	27.8 TFLOPS
Last-level Cache	24 MB L3	20 MB L2	6 MB L2
Memory Bandwidth	171 GB/s	778 GB/s	768 GB/s

Table 2: Specifications of the datasets

Datasets	Vertices	Edges	f_0	f_L
Reddit	232,965	11,606,919	602	41
ogbn-products	2,449,029	61,859,140	100	47
MAG240M	244,160,499	1,729,762,391	768	153

```
def train(...):
  device = torch.device("cuda")
  model = GNN(...).to(device)
  loader = dgl.dataloading.DataLoader(
    g,
    train_idx.to(device),
    dgl.dataloading.NeighborSampler(...),
    device=device,
    use_uva=device.type == "cuda")
  opt = torch.optim.Adam(...)
  for epoch in range(runs):
    _train(loader, model, opt)
    ...
```

Listing 1: Single GPU training with DGL library

[8], and MAG240M [7]. We choose two widely-used GNN models: GCN [10] and GraphSAGE [6]. To generate the mini-batches, we choose two representative GNN sampling algorithms: the Neighbor Sampler [6] and the ShaDow Sampler [40]. The Neighbor Sampler constructs a three-layer model with a sampling size [15, 10, 5]; the ShaDow sampler first produces a localized three-hop subgraph with a sampling size [15, 10, 5] and then constructs a five-layer model on the subgraph (i.e., $L' = 3$, $L = 5$; details in Section 2.2). The hidden feature size is set to 128. We set the mini-batch size as 4096 for the Reddit and ogbn-products datasets, and set the mini-batch size as 1024 for MAG240M; this is because the MAG240M input feature size is larger than the other two datasets, and setting a batch size of 4096 leads to an out-of-memory error. We list the details of the three datasets in Table 2, which includes the number of nodes, edges, and the feature size of the input f_0 and output layer f_L.

5.2 System Implementation

Our protocol can be seamlessly incorporated into existing GNN frameworks. In Listing 1, we show an example program that performs GNN training on a single GPU platform using DGL. In Listing 2, we highlight the modifications to train with our Unified CPU-GPU training protocol. We only show an example using DGL because the modifications are similar to incorporating our protocol into PyG. The program modifications include (1) applying PyTorch

```
def train(...):
  torch.distributed.init_process_group("gloo", ...)
  device = get_device()
  model = GNN(...).to(device)
  model = torch.DistributedDataParallel(model)
  loader = dgl.dataloading.DataLoader(
    g,
    UnevenDDPIndices(train_idx.to(device)),
    dgl.dataloading.NeighborSampler(...),
    device=device,
    use_uva=device.type == "cuda")
  opt = torch.optim.Adam(...)
  balancer = DynamicLoadBalancer()
  for epoch in range(runs):
    profile = unified_train(balancer.config(),
        _train, args=(loader, model, opt))
    balancer.update(profile)
    ...
```

Listing 2: Unified CPU-GPU Training with our protocol ■: newly added lines. ■: modified lines

`DistirbutedDataParallel` wrapper to enable multi-process GNN training, (2) applying the `unified_train` wrapper to enable Unified CPU-GPU training, and (3) launching the Dynamic Load Balancer. The Dynamic Load Balancer takes the runtime information as input, and adjusts the workload distribution between CPU and GPU on the fly. However, the native `DataLoader` in PyTorch [27] does not support assigning different numbers of mini-batches to CPUs and GPUs, or dynamically adjusting the batch size during GNN training; thus, we further develop a `UnevenDDPIndices` wrapper to support the above features.

5.3 Overall Performance

In Table 3, we compare the performance of our work, which adopts the Unified CPU-GPU training protocol and the optimizations mentioned in Section 4, with the DGL baseline, which adopts the Standard GNN training protocol (details in Figure 1). The baseline designs are obtained from the official implementations of DGL [2,3]. We then make minor code changes to the baseline designs to enable the Unified CPU-GPU training; the code changes are shown in Section 5.2. We evaluate our protocol on two CPU-GPU platforms as described in Section 5.1.1. On Platform 1, our protocol leads to up to 1.41× speedup compared with the Standard GNN training protocol that offloads most of the workload to the GPU. On Platform 2, our protocol leads to up to 1.26× speedup compared with the Standard GNN training protocol.

To understand the performance improvement of our protocol, we show the training time breakdown in Figure 6 using MAG240M on Platform 2 as an example. For the Neighbor Sampling setup, our protocol leads to speedup primarily by reducing the CPU-GPU data transfer overhead. For the ShaDow Sampling setup, our protocol achieves speedup mainly by reducing the workload on the GPU to reduce computation time. The main difference between these two settings is that training with Neighbor Sampling is communication-intensive while training with ShaDow Sampling is computation-intensive. This discrepancy arises from the density of subgraphs constructed by the two samplers: The ShaDow Sampler constructs

[2]https://github.com/dmlc/dgl/tree/master/examples/pytorch/ogb/ogbn-products
[3]https://github.com/dmlc/dgl/tree/master/examples/pytorch/ogb_lsc/MAG240M

Table 3: Epoch Time (sec) Comparison

	Sampler	Model	Protocol	Reddit	Products	MAG240M
Platform 1 A100 (3g.20gb)	Neighbor	GCN	Standard	3.27 (1.00×)	5.20 (1.00×)	107.38 (1.00×)
			Unified	2.32 (1.41×)	3.95 (1.32×)	82.54 (1.30×)
		GraphSAGE	Standard	3.11 (1.00×)	4.94 (1.00×)	103.03 (1.00×)
			Unified	2.41 (1.29×)	3.85 (1.28×)	77.51 (1.33×)
	ShaDow	GCN	Standard	11.38 (1.00×)	41.50 (1.00×)	366.39 (1.00×)
			Unified	9.32 (1.22×)	34.30 (1.21×)	265.66 (1.38×)
		GraphSAGE	Standard	10.59 (1.00×)	40.62 (1.00×)	445.80 (1.00×)
			Unified	9.13 (1.16×)	35.76 (1.14×)	350.91 (1.27×)
Platform 2 A5000	Neighbor	GCN	Standard	3.06 (1.00×)	4.88 (1.00×)	100.32 (1.00×)
			Unified	2.43 (1.26×)	4.08 (1.20×)	84.03 (1.19×)
		GraphSAGE	Standard	2.82 (1.00×)	4.47 (1.00×)	93.12 (1.00×)
			Unified	2.37 (1.19×)	3.78 (1.18×)	78.33 (1.19×)
	ShaDow	GCN	Standard	8.96 (1.00×)	34.88 (1.00×)	375.21 (1.00×)
			Unified	8.29 (1.08×)	32.56 (1.07×)	306.80 (1.22×)
		GraphSAGE	Standard	9.71 (1.00×)	37.34 (1.00×)	409.83 (1.00×)
			Unified	8.92 (1.09×)	34.18 (1.09×)	337.07 (1.22×)

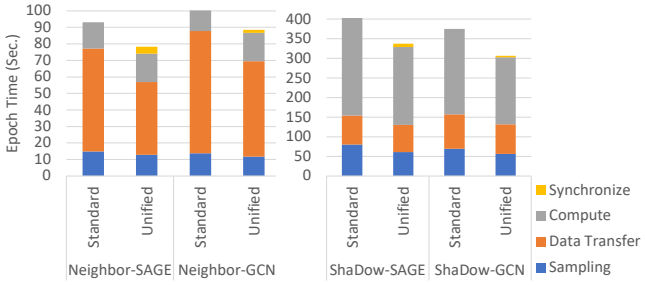

Figure 6: Epoch time breakdown

Table 4: Utilization of CPU, GPU, and memory bandwidth

Sampler-Model	Protocol	CPU	GPU	Mem. BW
Neighbor-SAGE	Standard	2.00%	25.50%	10.1 GB/sec
	Unified	25.10%	18.40%	21.2 GB/sec
Neighbor-GCN	Standard	1.70%	28.40%	9.0 GB/sec
	Unified	25.50%	24.10%	17.1 GB/sec
ShaDow-SAGE	Standard	2.54%	37.37%	12.2 GB/sec
	Unified	24.80%	31.05%	36.0 GB/sec
ShaDow-GCN	Standard	2.34%	35.30%	15.2 GB/sec
	Unified	23.90%	31.70%	38.4 GB/sec

a localized subgraph for each node in a given mini-batch by inducing (i.e., sampling all the edges between a node set) from K-Hop neighboring nodes, whereas the Neighbor Sampler only samples edges along the sampling path of each mini-batch. As a result, mini-batches constructed by the ShaDow Sampler hold significantly more edges than the mini-batches constructed by the Neighbor Sampler, which leads to higher computational complexity as the computation of GNNs is correlated with the number of edges in the mini-batch [42]. The experimental results show that our protocol can accelerate both computation-intensive and communication-intensive tasks, by reducing computation and data transfer time. respectively. Additionally, as shown in Table 4, the Unified CPU-GPU protocol effectively improves resource utilization and memory bandwidth compared with the Standard protocol. This is because our protocol exploits CPU compute cores to execute GNN operations, and also overlaps the computation with communications across multiple GNN training processes. To verify that our library works on platforms with multiple GPUs, we also experiment with 2 A5000 GPUs. For such a platform where the GPUs are much faster than the CPUs, the speedup of the unified CPU-GPU protocol is limited (1.00× – 1.07×). This is because, after Dynamic Load Balancing, the CPU is only assigned a small portion of the workload, limiting the achievable speedup. Similarly, using the Unified CPU-GPU training protocol on the A100 (no MIG enabled), the speedup is relatively limited compared with the Standard training protocol (1.04× – 1.19×). As mentioned in Section 1, our protocol is particularly useful in scenarios with constrained GPU resources; therefore, the limited speedup is expected when the GPU significantly outperforms the CPU.

5.4 Impact of Optimizations

We evaluate the effectiveness of the optimizations. We start from the baseline design and gradually apply our optimizations to observe the performance improvements. Figure 7 shows the speedup normalized with repect to the baseline design. We first use the Unified CPU-GPU protocol, which uses the GNN Process Manager (see Section 4.1) to assign workload to both the CPU and GPU platform, and adjust workload assignment with a Static Load Balancer (see Section 4.2). As mentioned in Section 4.2, static workload assignment only works if the workload of each mini-batch is similar, such as in the case of the ogbn-product dataset under the Neighbor Sampling algorithm. For Reddit and MAG240M, the workload of each mini-batch varies and so a static assignment cannot balance the workload. This causes the CPU platform to bottleneck the system,

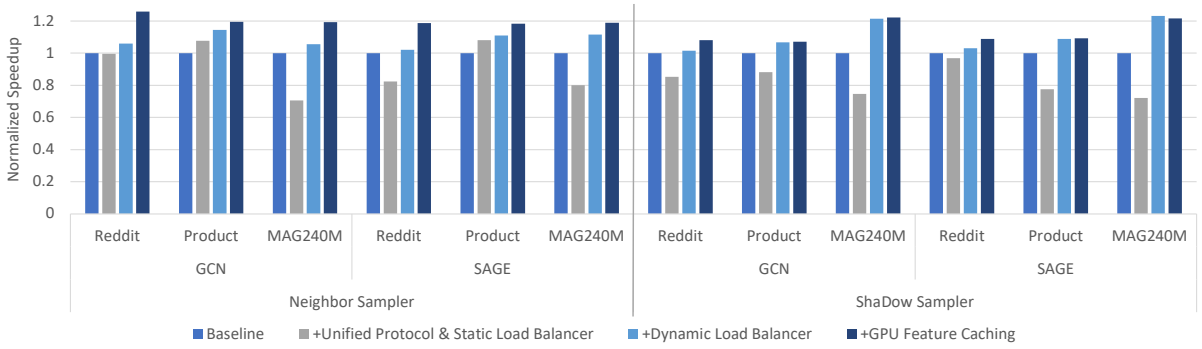

Figure 7: Impact of optimizations

resulting in performance degradation. After applying the Dynamic Load Balancer, which balances the workload based on the estimated workload for each mini-batch, our system is able to achieve performance improvement under various sampling algorithms, GNN models, and datasets. Finally, we apply GPU feature caching, which reduces the overhead of accessing the unified shared memory, especially if there exist some frequently accessed nodes and edges in the graph, for example, the Reddit and MAG240M datasets under the Neighbor Sampling algorithm.

6 RELATED WORK

6.1 Hybrid CPU-GPU Computation

Several works have proposed to utilize both CPU and GPU to perform parallel computing. CHC [11] exploits thread-level parallelism across the CPU and the GPU. CHC achieves 1.42× speedup on average compared with GPU-only baselines by utilizing the CPU cores to execute CUDA kernels. FluidiCL [26] is a runtime system that takes an OpenCL program as input, and executes the program with both the CPU and the GPU. SKMD [12] proposes a single kernel multiple device framework that exploits data parallelism across the CPUs and the GPUs. CHC, FluidCL, and SKMD are general frameworks that enable CPU-GPU computing for various applications. However, in order to be generic, these frameworks only exploit parallelism within a single kernel to ensure the semantics of the executed program are preserved. As discussed in Section 5.4, for GNN training, simply enabling CPU-GPU computing is not enough as it is non-trivial to achieve load balancing between the CPU and GPU. Therefore, domain-specific optimizations like Dynamic Load Balancer is required to achieve performance improvment in terms of the overall training time.

6.2 GNN Training on Heterogeneous Platforms

While most work on GNN training acceleration focus on using a homogeneous platform (e.g., GPU-only, CPU-only) [3, 15], several works have exploited [17–19, 39, 44] heterogeneous platforms such as CPU-GPU or CPU-FPGA platforms to accelerate GNN training. GraphACT [39] accelerates GNN training on a CPU-FPGA platform, and DistDGL_v2 [44] accelerates GNN training on a CPU-GPU platform. GraphACT and DistDGL_v2 only assign lightweight operations such as mini-batch sampling to the CPU. Thus, the CPU platform remains under-utilized. HyScale-GNN [16] accelerates GNN training on both CPU-GPU and CPU-FPGA platforms. It adopts the message-passing model for data communication among the devices, and the developer must explicitly handle numerous data transfers among the CPUs and GPUs or FPGAs during GNN training. In contrast, our library takes advantage of the shared memory feature provided by state-of-the-art programming libraries, which reduces programming efforts by leaving the data transfer to the runtime. In addition, HyScale-GNN is neither compatible with existing GNN libraries nor open-sourced; this limits its useability.

7 CONCLUSION

In this work, we proposed a Unified CPU-GPU training protocol for GNN training. The protocol explores the potential of incorporating CPU platform to improve GNN training performance. Such an approach holds practical value in real-world applications, as CPU platforms are becoming increasingly powerful, while GPU resources often face limited availability due to high demand from many users or other researchers. By exploiting the computation resources and balancing the workload dynamically on both the CPUs and the GPUs, our work achieves up to 1.41× speedup compared with the baseline design. While our protocol focused on CPU-GPU platforms in this work, it can be generalized to other accelerators. For example, state-of-the-art FPGAs also support Unified Shared Memory (USM) for data communication [9, 24], allowing it to be intergrated into the Unified training system. With USM, Dynamic Load Balancing can be supported by collecting the GNN training time on the CPU and the FPGA to determine the workload ratio, and using the USM for mini-batch assignemnt. In the future, we plan to generalize our work to support various types of accelerators, e.g., FPGA and TPU.

ACKNOWLEDGMENTS

This research was supported in part by a seed funding from Ershaghi Center for Energy Transitions, by the U.S. National Science Foundation (NSF) under grants CCF-1919289/SPX-2333009, CNS-2009057 and OAC-2209563, and by the DEVCOM Army Research Lab (ARL) under grant W911NF2220159.

REFERENCES

[1] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. In *International Conference on Learning Representations Workshop*.

[2] 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 ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[3] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

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

[5] 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.

[6] 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*.

[7] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. *arXiv preprint arXiv:2103.09430* (2021).

[8] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[9] Vinod Kathail. 2020. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 173–174.

[10] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[11] Changmin Lee, Won Woo Ro, and Jean-Luc Gaudiot. 2014. Boosting CUDA Applications with CPU–GPU Hybrid Computing. In *International Journal of Parallel Programming*.

[12] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. *ACM Trans. Comput. Syst.* (2015).

[13] 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 (TPDS)* (2020).

[14] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the VLDB Endowment* (2020).

[15] Yi-Chien Lin, Yuyang Chen, Sameh Gobriel, Nilesh Jain, Gopi Krishna Jha, and Viktor Prasanna. 2024. ARGO: An Auto-Tuning Runtime System for Scalable GNN Training on Multi-Core Processor. arXiv:2402.03671 [cs.DC]

[16] Yi-Chien Lin and Viktor Prasanna. 2023. HyScale-GNN: A Scalable Hybrid GNN Training System on Single-Node Heterogeneous Architecture. In *International Parallel and Distributed Processing Symposium*.

[17] Yi-Chien Lin, Bingyi Zhang, and Viktor Prasanna. 2022. Accelerating GNN Training on CPU+Multi-FPGA Heterogeneous Platform. In *High Performance Computing*. Springer International Publishing.

[18] Yi-Chien Lin, Bingyi Zhang, and Viktor Prasanna. 2022. HP-GNN: Generating High Throughput GNN Training Implementation on CPU-FPGA Heterogeneous Platform. In *International Symposium on Field-Programmable Gate Arrays*.

[19] Yi-Chien Lin, Bingyi Zhang, and Viktor Prasanna. 2023. HitGNN: High-throughput GNN Training Framework on CPU+Multi-FPGA Heterogeneous Platform. arXiv:2303.01568 [cs.DC]

[20] Zhongyi Lin, Louis Feng, Ehsan K. Ardestani, Jaewon Lee, John Lundell, Changkyu Kim, Arun Kejariwal, and John D. Owens. 2022. Building a Performance Model for Deep Learning Recommendation Model Training on GPUs. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[21] 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*.

[22] Daniela Sánchez Lopera, Lorenzo Servadei, Gamze Naz Kiprit, Souvik Hazra, Robert Wille, and Wolfgang Ecker. 2021. A survey of graph neural networks for electronic design automation. In *Workshop on Machine Learning for CAD (MLCAD)*. IEEE.

[23] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[24] Raúl Nozal and Jose Luis Bosque. 2021. Exploiting Co-execution with OneAPI: Heterogeneity from a Modern Perspective. In *Euro-Par 2021: Parallel Processing*.

[25] NVIDIA. [n. d.]. *System Management Interface [Online]*. https://developer.nvidia.com/nvidia-system-management-interface Accessed: 2023-09-05.

[26] Prasanna Pandit and R. Govindarajan. 2014. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. 273–283.

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

[28] James Reinders. 2005. *VTune performance analyzer essentials*. Vol. 9. Intel Press Santa Clara.

[29] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).

[30] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 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)*.

[31] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[32] 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).

[33] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. 2022. Merlin HugeCTR: GPU-Accelerated Recommender System Training and Inference. In *Proceedings of the 16th ACM Conference on Recommender Systems* (Seattle, WA, USA) *(RecSys '22)*. Association for Computing Machinery, New York, NY, USA, 534–537. https://doi.org/10.1145/3523227.3547405

[34] Nan Wu, Hang Yang, Yuan Xie, Pan Li, and Cong Hao. 2022. High-Level Synthesis Performance Prediction Using GNNs: Benchmarking, Modeling, and Advancing. In *Proceedings of the Design Automation Conference*.

[35] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[36] Chu-I Yang and Yi-Pei Li. 2023. Explainable uncertainty quantifications for deep learning-based molecular property prediction. *Journal of Cheminformatics* (2023).

[37] 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 17th European Conference on Computer Systems*.

[38] 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 International Conference on Knowledge Discovery & Data Mining*.

[39] Hanqing Zeng and Viktor Prasanna. 2020. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[40] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. 2021. Decoupling the Depth and Scope of Graph Neural Networks. In *Advances in Neural Information Processing Systems*.

[41] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[42] Z. Zhang, P. Cui, and W. Zhu. 2022. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge & Data Engineering* (jan 2022). https://doi.org/10.1109/TKDE.2020.2981333

[43] Da Zheng et al. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*.

[44] Da Zheng et al. 2022. Distributed Hybrid CPU and GPU Training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*.

[45] 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. *Proceedings of the VLDB Endowment* (2019).