Efficient GNN Training on Giant Graphs with Collective Batching and Scheduling

Xin Zhang
HKUST
xzhanggb@ust.hk

Yanyan Shen*
Shanghai Jiao Tong University
shenyy@sjtu.edu.cn

Yingxia Shao*
Beijing University of Posts and Telecommunications
shaoyx@bupt.edu.cn

Haoyang Li
The Hong Kong Polytechnic University
haoyang-comp.li@polyu.edu.hk

Lei Chen
HKUST (GZ) and HKUST
leichen@cse.ust.hk

ABSTRACT

Graph Neural Networks (GNNs) have achieved great success in many applications, and mini-batch training has become the de facto standard for training GNNs on large-scale graphs. When developing mini-batch GNN training systems on CPU-GPU platforms, existing *dedicated batching* systems adopt a *static* workload-processor binding strategy, where the costly mini-batch preparation workload is exclusively assigned to either the CPU or the GPU. This leads to low utilization of CPU cores, PCIe bandwidth, and GPU computing capability, resulting in suboptimal training efficiency. To address this problem, we develop `MorphGL`, a novel GNN training system featuring a *collective batching* design. `MorphGL` adaptively dispatches the mini-batch preparation workload to both the CPU and GPU, ensuring that the workload distribution aligns with the CPU-GPU setup of the running machine for optimal efficiency. To maximize resource utilization, `MorphGL` employs the `Dual-Buffer Scheduling` algorithm to collectively schedule training stages across the CPU, PCIe, and GPU. Extensive experiments on three large real-world graphs with billions of edges and four machines with representative CPU-GPU configurations demonstrate that `MorphGL` consistently outperforms state-of-the-art GNN training systems, achieving up to 2.76x and 2.2x speedup over SALIENT and DUCATI, respectively.

PVLDB Reference Format:
Xin Zhang, Yanyan Shen, Yingxia Shao, Haoyang Li, and Lei Chen. Efficient GNN Training on Giant Graphs with Collective Batching and Scheduling. PVLDB, 19(6): 1184 - 1197, 2026.
doi:10.14778/3797919.3797927

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

*Corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797927

1 INTRODUCTION

Graph Neural Networks (GNNs) [3, 14, 15, 21, 24, 26–28, 31, 33, 36, 41, 42, 45–48, 58, 65, 66, 70–73] have emerged recently as a prominent graph learning method. Their ability to effectively model graph-structured data has led to significant advancements in various downstream tasks, such as social media recommendation [10, 25, 30, 35, 37, 40, 61, 64, 68, 74], biological molecule property prediction [17, 52, 63], and weather forecasting [9, 29]. Similar to other deep learning models [7, 18, 53], GNNs typically employ the mini-batch training paradigm to enhance scalability on large graphs. The training process involves two major workloads: (1) *mini-batch preparation* which samples mini-batches from the complete graph dataset and transfers them to the GPU through PCIe; and (2) *GNN model training* which encompasses forward computation, backward propagation, and parameter updates. In the field of deep learning, the majority of computing systems are equipped with two types of processors: CPU and GPU. When developing GNN training systems, a crucial decision involves determining how to allocate the two major workloads over the available processors. There is a general agreement among existing systems to assign the GNN model training task to GPU, given GPU's expertise in accelerating neural network computation [6, 11, 22, 34, 38, 39, 44, 54, 56, 62, 69]. *However, there has been a considerable debate in the community regarding which processor should handle the mini-batch preparation task.* By this criterion, we divide existing mini-batch GNN training systems into the following two categories.

The Category I systems [12, 22, 56, 57] assign the mini-batch preparation task to the CPU. They employ various optimization techniques to enhance the preparation efficiency, e.g., increasing batching parallelism through multi-thread or multi-process sampling [22, 56, 57], optimizing node feature selection with improved data structures [22, 56], and pipelining data transfer with the asynchronous Direct Memory Access (DMA) technique [12, 22, 57]. Category II systems [6, 44, 54, 69] primarily allocate the mini-batch preparation task to the GPU. They employ the Unified Virtual Addressing (UVA) technique to enable GPU cores for rapid batching and data transfer when storing the graph dataset in main memory. Moreover, DUCATI [69], DSP [6], and Quiver [54] propose to cache frequently accessed graph data in device memory to reduce data transfer volume over PCIe. We refer to Category I and II systems

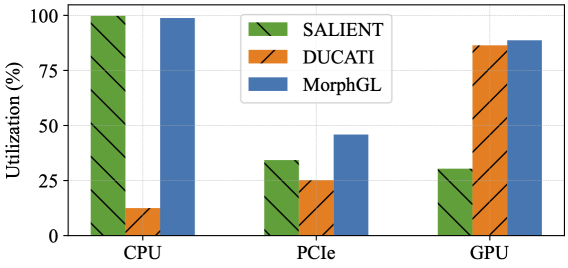


Figure 1: Comparison of CPU, PCIe, and GPU utilizations of SALIENT, DUCATI, and `MorphGL`.

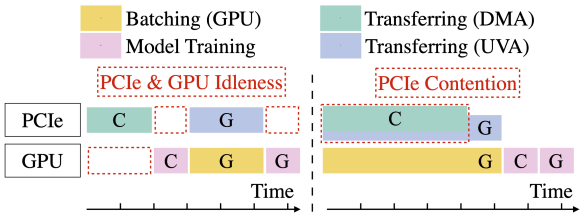


Figure 2: Hardware idleness and contention when arranging the execution of a CPU-produced and a GPU-produced mini-batch. Colored rectangles with `C`/`G` represent corresponding training stages of the CPU/GPU-produced mini-batch. Left: sequential execution. Right: pipelined execution.

as *dedicated batching* systems because the mini-batch preparation workload is assigned exclusively to the CPU or GPU for them.

Unfortunately, we find that dedicated batching systems suffer from low utilization of hardware resources, namely CPU cores, PCIe bandwidth, and GPU computing capability, when training on common hardware[1]. To demonstrate this, we collect and compare the hardware utilization[2] of SALIENT [22] and DUCATI [69], representing Category I and Category II systems, respectively, in a typical training setting[3], as shown in Figure 1. We find that SALIENT has busy CPUs generating mini-batches while GPU is idle for over 70% of training time. DUCATI saturates GPU with much workload but only uses one CPU core for kernel launching and training logic control, wasting 88% of CPU cores during training. The PCIe bandwidth is also underutilized in both systems.

The low hardware utilization of dedicated batching systems stems from their inflexible approach to distributing the mini-batch preparation workload. Existing systems adopt a *static* workload-processor binding strategy, assigning the whole mini-batch preparation task to either CPU or GPU exclusively. However, whenever a processor takes all the batching workload, (i) this processor becomes the bottleneck of the whole system and forces the other processor to wait, and (ii) the corresponding data transferring technique is also fixed as DMA/UVA, leading to the waste of PCIe bandwidth[4].

In this paper, we propose `MorphGL`, an efficient mini-batch GNN training system featuring the *collective batching* design to address the low hardware utilization problem. `MorphGL` adopts the *dynamic* workload-processor binding strategy, where the mini-batch preparation workload is distributed to both CPU and GPU adaptively. `MorphGL` also collectively schedules workloads across CPU, PCIe, and GPU, overlapping the execution to maximize hardware utilization. At a high level, `MorphGL` adaptively adjusts several operational aspects, including batching workload distribution, data transferring techniques, and scheduling patterns, therefore, efficiently training GNNs. However, it is challenging to design a collective batching and scheduling system that minimizes the training time.

First, it is non-trivial to determine an optimal distribution of the batching workload between CPU and GPU. The best allocation often depends on the specific hardware and training configuration, which can vary widely in practice. A misaligned distribution may cause bottlenecks or under-utilize available resources, ultimately degrading training efficiency. Second, naïve scheduling methods suffer from hardware idleness and contention when coordinating mini-batch execution. As illustrated on the left side of Figure 2, sequentially executing the transferring and model training stages–first for a CPU-generated mini-batch, then for a GPU-generated one–leads to idleness for both PCIe and GPU. Conversely, the right side of Figure 2 demonstrates that while pipelined execution attempts to overlap DMA transferring with GPU batching, it introduces PCIe contention. This arises from the UVA requirement, which mandates that UVA transferring occurs concurrently with GPU batching, as we will elaborate on in Section 2.2 and Section 3.

We formulate the collective batching and scheduling problem in Section 4.1, which optimizes workload distribution and scheduling to minimize training time under given hardware and user configurations, and prove it is NP-hard. `MorphGL` addresses this problem iteratively with two key components: the Dispatcher and the Scheduler. The Dispatcher uses offline profiling statistics to generate an initial workload distribution ratio based on relaxed constraints. Given the distribution ratio, the Scheduler then applies the `Dual-Buffer Scheduling` algorithm, which manages two buffers to coordinate data transfer and model training while avoiding hardware contention and idleness. Feedback from the Scheduler guides Dispatcher to refine distribution ratio, and the two components iterate until convergence. We further provide theoretical guarantees for the proposed solution.

We conduct extensive experiments on four machines equipped with consumer- and enterprise-grade GPUs and diverse CPUs. Results on three GNNs and three billion-scale graphs show that `MorphGL` adapts well to various settings and consistently outperforms state-of-the-art Category I and II systems (SALIENT and DUCATI), achieving up to 2.76 and 2.2 times speedup, respectively.

To summarize, the major contributions of the paper are:

- We propose `MorphGL`, a novel GNN training system featuring the collective batching and scheduling design, which adaptively distributes batching workload to CPU and GPU and judiciously schedules workloads on CPU, PCIe, and GPU to avoid both hardware contention and idleness.
- We formulate the collective batching and scheduling problem, prove the problem is NP-hard, and propose a tunable workload dispatching scheme and an asynchronous `Dual-Buffer Scheduling` algorithm to solve it. The resulting solution, which

[1]Definition of common hardware will be given in Section 3.
[2]Measurement of hardware utilization will be given in Section 5.1.
[3]Details will be given in Section 5. Setting: GraphSAGE model, UK dataset, Machine B.
[4]Detailed discussion will be given in Section 3.

has a theoretical bound, maximizes the hardware utilization and improves training throughput.
- We evaluate MorphGL on three billion-scale graphs, three mainstream GNN models, and four representative machines. Experimental results demonstrate that MorphGL can achieve universal speedup against the baselines in diverse settings.

2 BACKGROUND

In this section, we provide the background on mini-batch GNN training and review the existing works.

2.1 Mini-batch GNN Training

GNNs are powerful tools for modeling graph data by recursively aggregating the node features along the topology. GCN [60], the pioneering GNN model, initially uses a full-batch training paradigm which only involves model training on GPU. It requires loading the entire graph dataset into device memory before the training starts. Therefore, the full-batch training fails to scale to large graphs. For instance, the Ogbn-Papers100M dataset takes up over 80 GB of space, exceeding the capacity of most consumer-level GPUs, which typically have only about a dozen gigabytes of device memory.

For scalability, GraphSAGE [17] proposes a mini-batch training approach containing three major stages executed on the CPU, PCIe, and GPU. (1) In the **batching stage**, we prepare a mini-batch using CPU with random seed nodes from the labeled training set. Starting from the seed nodes, we recursively sample f_i neighbors for each selected node at the i-th hop, where $i \in [1, K]$ and K is GNN's depth. The seed nodes, sampled neighbors, and connecting edges form the topology component of the mini-batch. We then collect node features of selected nodes, deriving the feature component of the mini-batch. Since a node could have fewer neighbors than f_i and the common neighbors are included only once per-hop, mini-batches have non-constant total numbers of nodes, determined by intrinsic graph structures. (2) In the **transferring stage**, we transfer the prepared mini-batch from main memory to device memory through PCIe. (3) In the **model training stage**, we use the transferred mini-batch to train GNN models on GPU. The above process is repeated for all training samples to complete one epoch of training.

Since GNN models' weights, activations, gradients, and optimizer states typically occupy less than one gigabyte in total, mainstream GNN models [60, 67], including those leading the OGB leaderboard [19] and the key examples in DGL and PyG [12, 57] frameworks, employ single-GPU training. This paper also focuses on single-GPU mini-batch GNN training. Since most GNN datasets [5, 12, 19, 22, 23, 57, 60, 67, 69] fit within the main memory of modern commodity servers, this paper focuses on accelerating in-memory GNN training, leaving disk-based training for future work.

Some recent studies [2, 32] enhance the efficiency of mini-batch GNN training by modifying mini-batch composition and introducing staleness into the training process. Concretely, to reduce training cost, they reuse the historically computed node embeddings from previous mini-batches when calculating activations and gradients on future mini-batches. These approaches inevitably compromise model accuracy. In contrast, MorphGL and other GNN systems discussed in this paper avoid altering mini-batches or introducing staleness, thereby preserving accuracy and ensuring applicability.

2.2 Category I and Category II Systems

Category I systems [12, 22, 56, 57] advocate binding mini-batch preparation workload to the CPU. PyG (CPU) [12], which represents the CPU-batching mode of PyG, and DGL (CPU) [57], two of the most popular GNN libraries, provide multiprocessing-based batching and asynchronous mini-batch transferring. However, the overhead of data sharing between multiple processes harms the training efficiency. SALIENT [22] and MariusGNN [56] propose multithreading-based batching to avoid the overhead and improve efficiency. Generated mini-batches are pinned in page-locked main memory with CPU. Then, these systems leverage the **Direct Memory Access (DMA)** technique to transfer the pinned mini-batch to device memory through PCIe. Given the pinned mini-batch in the main memory and the destination in the device memory, the DMA engine independently executes the data transfer, leaving CPU and GPU free for other workloads. This allows the pipelining of workloads on CPU, PCIe, and GPU. However, these systems usually suffer from slow batching with insufficient CPU cores. For example, on a common machine with eight CPU cores and an RTX 3090 GPU, SALIENT cannot produce mini-batches fast enough and the GPU is hungrily waiting for mini-batches during training, yielding merely 21.6% of GPU utilization as we will discuss in Section 5.4.

In the presence of limited CPU resources, Category II Systems [12, 20, 44, 54, 57, 62, 69] propose to bind all or the majority of the batching workload to GPU instead. Specifically, PyG (GPU) [12], DGL (GPU) [57], GNNLab [62] and NextDoor [20] are systems that only use GPU for batching. However, they require the graph dataset, or at least the graph topology data, to be stored in the device memory. Such a requirement severely harms the scalability since they cannot handle giant graphs whose topology data is larger than the equipped GPU memory. These systems encounter the out-of-memory (OOM) problem with gigantic datasets like Ogbn-Papers100M that are too large to fit in the device memory.

To address the OOM problem, DSP [6], PyTorch-Direct [44], Quiver [54], and DUCATI [69] adopt the **Unified Virtual Addressing (UVA)** technique. UVA allows GPU to directly initiate data requests over the graph dataset stored in the main memory for fast subgraph sampling and feature selection. During UVA kernel execution, thread-level data requests are issued from streaming processors of the GPU, and other following computations in the same/later kernel are forced to wait until the data fetching is complete. Therefore, UVA operations occupy GPU and PCIe simultaneously but require no help from CPU in execution, which is great when users have limited CPU cores. DUCATI, DSP, and Quiver further cache the frequently accessed entries, including node features and adjacency lists, in spare GPU memory to save the data movement and accelerate the transfer on PCIe. These systems work well on machines with negligible CPU resources, however, cannot leverage the CPU computing power when applied to machines with non-negligible CPU resources. For example, when training on the same machine with eight CPU cores and an RTX 3090, DUCATI only utilizes one out of eight available CPU cores.

Summary. We summarize key designs of existing works in Table 1. Due to the *static* workload-processor binding strategy for mini-batch preparation, existing systems suffer from resource under-utilization on common hardware and suboptimal training efficiency.

Table 1: Comparison of existing works and `MorphGL`. *Binding* represents workload-processor binding strategy. *Hardware* stands for preferred machine types. *DBS* represents the `Dual-Buffer Scheduling` to be introduced in Section 4.

Systems	*Binding*	*Hardware*	*Scheduling*
Category I	static	CPU-abundant	Pipeline
Category II	static	CPU-scarce	Sequential
`MorphGL`	dynamic	All kinds	*DBS*

Table 2: The distribution of CPU:GPU ratio among 141 cloud GPU instances from AWS, Azure, and GCP.

CPU:GPU	12	6	8	4	Others
#Instances	48	25	17	11	40
Cum. Pct.	34.0%	51.7%	63.8%	71.6%	100%

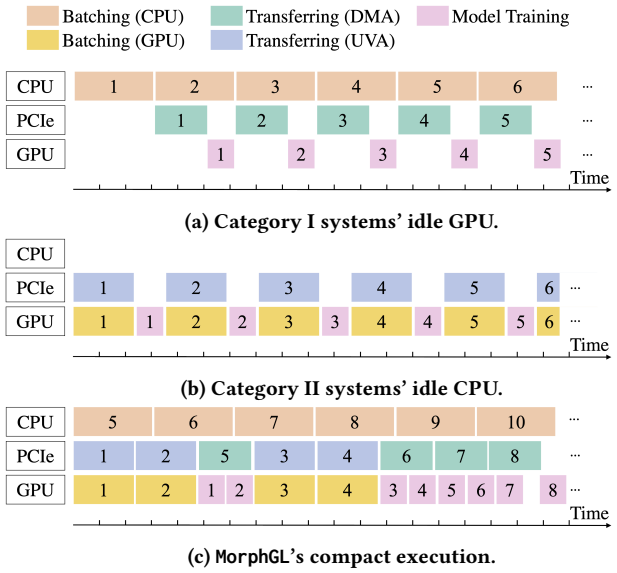
3 MOTIVATION

In this section, we provide empirical results that illustrate the limitations of existing systems and motivate the development of `MorphGL`.

Common hardware. In practice, it is desirable to ensure that a GNN training system aligns with common hardware. Since users always prefer shorter training times, the training system should exploit all available computing resources, including both CPU and GPU, allocated to the training task on common hardware. While there are many specifications, we find that the relative computing power between CPU and GPU largely determines where to bind the mini-batch preparation workload. Specifically, we collect the number of CPU cores per GPU, referred to as the CPU:GPU ratio, of all 141 GPU instances to date on the three largest cloud providers, namely Amazon Web Service (AWS) [4], Microsoft Azure (Azure) [43], and Google Cloud Platform (GCP) [16]. In Table 2, over 70% of machines have CPU:GPU ratios between 4 and 12, therefore, they are referred to as the *common hardware* in this work.

Performance Bottlenecks. Figure 3 presents simplified training profiles of Category I and II systems on Machine A, which features the most common CPU:GPU ratio (details in Section 5.1). We visualize only the major training stages on the CPU, PCIe, and GPU. Minor operations on CPU (e.g., kernel launches and training logic control) are omitted in the figure for brevity. The performance bottleneck lies in the processor, either the CPU or the GPU, whichever undertakes the batching stage. Two key inefficiency patterns emerge: (1) Category I systems suffer from GPU idleness; despite pipeline execution, slow CPU batching leads to substantial underutilization of both GPU and PCIe. (2) Category II systems keep the GPU busy with batching and training but leave the CPU largely idle, wasting potential parallelism. In both cases, PCIe bandwidth remains underutilized due to the batching bottleneck. Clearly, the root cause of these inefficiencies is the static workload-processor binding strategy employed by existing systems.

Opportunities. Given the status quo of *dedicated batching* systems and common hardware, there is an opportunity to build a *collective batching* system optimized for the common hardware: by breaking down the batching workload to both CPU and GPU, we can utilize computing resources better. The collective batching also allows us to leverage both the DMA and the UVA techniques for transferring, opening up a larger scheduling space to better utilize PCIe bandwidth. However, it is challenging to design a collective batching and scheduling mechanism that minimizes the training time.



(a) Category I systems' idle GPU.

(b) Category II systems' idle CPU.

(c) `MorphGL`'s compact execution.

Figure 3: Simplified execution timeline of the mini-batch training process. Each rectangle with number i represents the execution of one training stage of the i-th mini-batch. The length of each rectangle represents the duration of the training stage on the corresponding hardware.

First, distributing the batching workload is nontrivial. Users adopt diverse training configurations and run experiments on machines with varying CPU-GPU setups, leading to heterogeneous computational demands. For example, training the computation-intensive GAT model requires more time and GPU resources than the simpler GCN model. To prevent processor imbalance in such cases, the CPU must assume a larger share of the batching workload. This necessitates an adaptive dispatch mechanism that dynamically allocates tasks based on the training configuration.

Second, scheduling workloads across CPU, PCIe, and GPU presents additional challenges. In collective batching, both DMA and UVA transfers occupy the PCIe, while GPU batching and model training contend for GPU. Naïve pipelining thus causes PCIe and GPU contention, whereas sequential execution leads to hardware idleness by missing overlap opportunities. Therefore, training stages must be carefully arranged across CPU, PCIe, and GPU to avoid both resource contention and idleness, as illustrated in Figure 3c.

4 COLLECTIVE BATCHING AND SCHEDULING

In this section, we first formulate the *Collective Batching and Scheduling Problem* in Section 4.1 and then present the design and implementation of our `MorphGL` in Section 4.2.

4.1 Problem Formulation

In this section, we first introduce necessary notations and the constraints for the *Collective Batching and Scheduling Problem*. We then formally define the problem and provide its hardness proof.

We denote all mini-batches in one training epoch as $b_1, b_2, ..., b_n$. Each batch b_i ($i \in [1, n]$) undergoes three stages: batching (B_i), data transfer (T_i), and model training (M_i). These stages can be executed under two arrangements, **Route 0** and **Route 1**. In Route 0, CPU performs B_i, data transfer T_i occurs on PCIe via DMA, and GPU executes M_i, as in Category I systems. In Route 1, B_i and T_i use the UVA technique, occupying both GPU and PCIe, followed by M_i on the GPU, as in Category II systems. Let r_i denote the route of batch b_i, where $r_i = 0$ if b_i follows Route 0, and $r_i = 1$ otherwise. We denote the start time of stage X_i as $\Gamma(X_i)$, its duration as $Dur(X_i)$, and its hardware requirement as $\zeta(X_i)$, where $X_i \in \{B_i, T_i, M_i\}$.

$$\zeta(X_i) = \begin{cases} \{\text{CPU}\}, \text{ if } X_i = B_i \text{ and } r_i = 0, \\ \{\text{GPU, PCIe}\}, \text{ if } X_i = B_i \text{ and } r_i = 1, \\ \{\text{PCIe}\}, \text{ if } X_i = T_i \text{ and } r_i = 0, \\ \{\text{GPU, PCIe}\}, \text{ if } X_i = T_i \text{ and } r_i = 1, \\ \{\text{GPU}\}, \text{ if } X_i = M_i. \end{cases} \tag{1}$$

To prevent resource contention, we define **hardware occupation constraints** $\Psi^{hw}(\Gamma(X_i), r_i)$ in Constraints 2. These ensure that training stages sharing the same hardware cannot overlap, namely, one must start only before or after the other completes.

$$\begin{aligned} \Psi^{hw}(r_i, \Gamma(X_i)) \triangleq & \big(\forall i, j \in [1, n], \zeta(X_i) \cap \zeta(X_j) \neq \emptyset\big) \\ & \wedge \big((\Gamma(X_j) \geq \Gamma(X_i) + Dur(X_i)) \vee \\ & (\Gamma(X_i) \geq \Gamma(X_j) + Dur(X_j))\big). \end{aligned} \tag{2}$$

Mini-batch GNN training imposes **data dependency constraints** $\Psi^{dp}(\Gamma(X_i), r_i)$ in Constraints 3. Specifically, for each mini-batch b_i following Route 0, the transfer stage T_i begins only after the batching stage B_i completes. For Route 1, B_i and T_i execute concurrently on GPU and PCIe via UVA. In all cases, the model training stage M_i must start only after the corresponding transfer stage T_i finishes, ensuring data readiness on the GPU.

$$\begin{aligned} \Psi^{dp}(r_i, \Gamma(X_i)) \triangleq & \big(\Gamma(T_i) \geq (\neg r_i)(\Gamma(B_i) + Dur(B_i))\big) \wedge \\ & \big(\Gamma(M_i) \geq \Gamma(T_i) + Dur(T_i)\big) \wedge \\ & \big(\Gamma(B_i) \geq 0\big). \end{aligned} \tag{3}$$

To maximize hardware utilization, we introduce two buffers: a CPU buffer in main memory and a GPU buffer in device memory. Due to limited memory, buffer capacities are constrained by **buffer size constraints** $\Psi^{bf}(\Gamma(X_i), r_i)$ in Constraints 4. Since the mini-batch is the smallest processing unit, we measure buffer capacity by the number of storable mini-batches. Let *cbs* and *gbs* denote the CPU and GPU buffer sizes, respectively, and $\text{Cnt}(t, X_i)$ represent the total occurrences of X_i before timestamp t. At any time, the number of mini-batches in a buffer must not exceed its capacity.

$$\begin{aligned} \Psi^{bf}(r_i, \Gamma(X_i)) \triangleq & \ \big(\text{Cnt}(t, T_i) - \text{Cnt}(t, B_i) \leq cbs\big) \wedge \\ & \big(\text{Cnt}(t, M_i) - \text{Cnt}(t, T_i) \leq gbs\big). \end{aligned} \tag{4}$$

Definition 1 (Collective Batching and Scheduling Problem)**.** Given two sets of variables, namely (1) r_i that determines which processor undertakes the batching workload of mini-batch b_i and (2) $\Gamma(X_i)$ that determines the start timestamp of each training stage of all mini-batches. Our objective is to minimize the elapsed time of one training epoch, which is denoted as C, under the three constraints:

$$\begin{aligned} & \arg\min_{r_i, \Gamma(X_i)} C = \max\{\Gamma(X_i) + Dur(X_i)\}_{i \in [1,n]}, s.t., \\ & \Psi^{hw}(r_i, \Gamma(X_i)) \wedge \Psi^{dp}(r_i, \Gamma(X_i)) \wedge \Psi^{bf}(r_i, \Gamma(X_i)). \end{aligned} \tag{5}$$

Theorem 1. The optimization problem in Equation 5 is NP-hard.

Proof of Theorem 1. We prove that the optimization problem is NP-hard by reducing a variant of the job-shop scheduling (JSSP) problem with three machines [51], which is known to be NP-hard, to our problem. The JSSP seeks to minimize the makespan for n jobs processed on three machines. By fixing all mini-batch routes to Route 1, we can map CPU, PCIe, and GPU to the three machines, n mini-batches to n jobs, and training stages to job operations. A solution that minimizes the training epoch time thus yields the minimal makespan of the JSSP, establishing NP-hardness. □

Our optimization problem is more complex than the JSSP problem due to the additional route selection for each mini-batch. To conquer the problem, we use an iterative method that alternatively updates r_i and $\Gamma(X_i)$, namely the batching workload dispatching and training stages scheduling, to minimize C. We will illustrate the details in the next section.

4.2 Design and Implementation of `MorphGL`

The overall architecture of `MorphGL` is shown in Figure 4. `MorphGL` comprises four components. The Profiler captures details of hardware and user-defined training configurations. The Dispatcher distributes mini-batch workloads between the CPU and GPU. The Scheduler coordinates the execution of training stages across the CPU, PCIe, and GPU, while the Executor performs GNN model training according to the final scheduling plan. We next describe each component and our iterative solution in detail.

4.2.1 Profiler. The Profiler extracts essential information from the running hardware and training configurations. The basic execution unit in mini-batch GNN training is a training stage (CPU/GPU batching, DMA/UVA transferring, and GPU model training) of a mini-batch. It is the stage duration that directly influences the workload dispatching and scheduling decisions. While it is possible to build a complex cost model to predict each stage's duration based on numerous hardware specifications and training configurations, we find it is more convenient and also sufficiently accurate to leverage the profiling information thanks to the mini-batch's stability.

Concretely, in Figure 5, we visualize the distribution of the number of sampled nodes per mini-batch, which determines the execution duration of training stages, on the three datasets to be introduced in Section 5.1. Recall that the number of sampled nodes per mini-batch is influenced by both the actual number of neighbors of each seed node and the overlap between neighbors of different seed nodes, which are intrinsically determined by the graph structure (Section 2.1). In fact, we observe that for a given dataset, the total number of sampled nodes per mini-batch follows a narrow

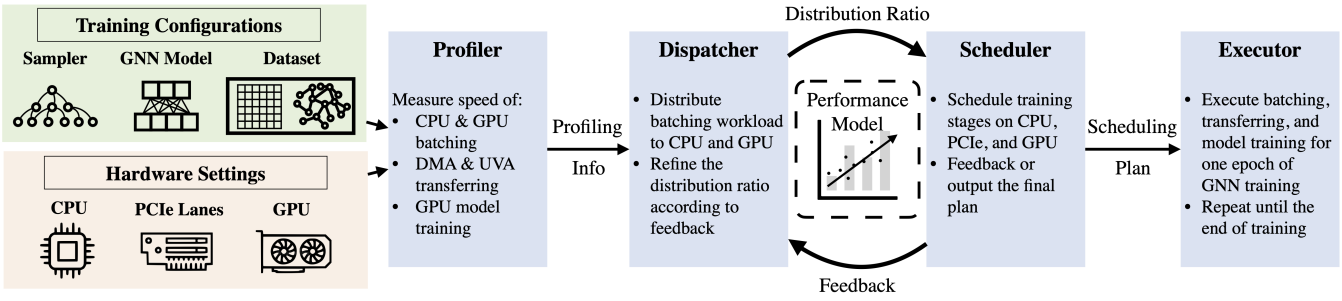


Figure 4: Overall architecture of MorphGL.

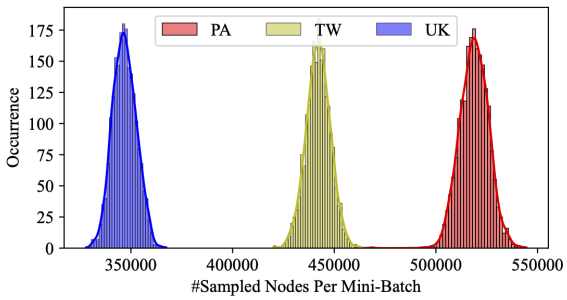
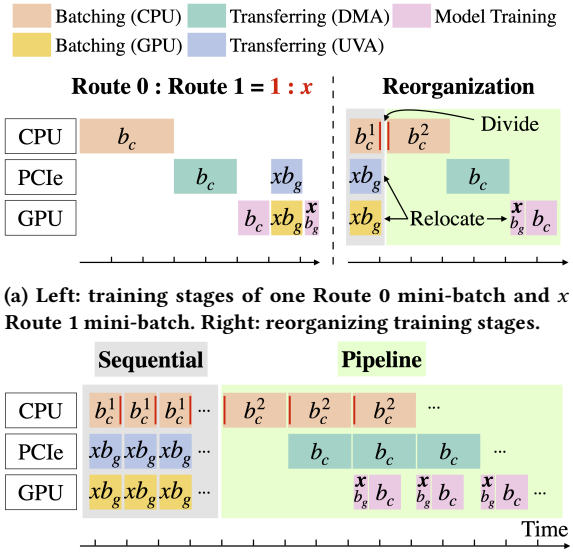


Figure 5: Distribution of the total number of sampled K-hop neighbor nodes per mini-batch for the three datasets.

normal distribution with a small relative standard deviation. Thanks to such stability, we can gather the mean duration of all five kinds of training stages quickly with several sample runs. We denote the mean duration of CPU batching, GPU batching, DMA transferring, UVA transferring, and model training as T_{CBT}, T_{GBT}, T_{DMA}, T_{UVA}, and T_{Model}, respectively. We profile each training stage for k mini-batches, where k is a hyperparameter, and calculate the mean duration as $T_X = \frac{1}{k}\sum_{i=1}^{k} Dur(X), X \in \{CBT, GBT, DMA, UVA, Model\}$. This information is sufficient for systems' efficiency tuning.

4.2.2 Dispatcher. The Dispatcher takes in the profiling information from the Profiler and outputs the workload distribution ratio $\boldsymbol{x}$, a positive number representing that the number of mini-batches prepared by CPU and GPU is $\mathbf{1} : \boldsymbol{x}$. The Dispatcher determines the distribution ratio through several iterations as detailed below.

Initial distribution ratio. We bootstrap the iteration and derive the first distribution ratio by relaxing the *buffer size constraints*, assuming (1) unlimited buffers in both main and device memory, and (2) infinitely divisible batching workloads. Let b_c denote a batch following Route 0 and b_g a batch following Route 1. To solve distribution ratio x, we optimize the sequential execution of one b_c and x b_g, as shown on the left of Figure 6a. In the CPU batching stage, b_c is split into b_c^1 and b_c^2 so that the CPU batching duration of b_c^1 equals the GPU batching duration of b_g. We then relocate the training stages of b_c and b_g, producing two phases (gray and green backgrounds) as shown on the right of Figure 6a. The gray phase consists of three equal-length workloads on CPU, PCIe, and GPU. We then arrange all mini-batches from both routes into an optimal execution comprising a sequential gray phase and a pipelined green



(a) Left: training stages of one Route 0 mini-batch and x Route 1 mini-batch. Right: reorganizing training stages.

(b) Optimal scheduling under relaxations.

Figure 6: Initial distribution ratio derivation. Training stage's duration is scaled by a factor of x for Route 1 mini-batches.

phase in Figure 6b, both fully utilizing CPU, PCIe, and GPU after warmup. The epoch training time of this schedule, denoted C', is a bounded piecewise function of x that is easy to minimize:

$$\arg\min_{x\geq 0} C' = n \cdot \Big(\frac{x \cdot T_{UVA}}{1+x} + \max\Big\{\frac{T_{DMA}}{1+x}, T_{Model}, \frac{T_{CBT} - x \cdot T_{UVA}}{1+x}\Big\}\Big) \tag{6}$$

Under the relaxations, we can simplify the NP-hard optimization on C to an easy optimization problem on C', which we solve and obtain a crude distribution ratio x. However, in practice, we actually have limited buffers, and the training stage is not divisible. Hence, we rectify the crude distribution ratio in the finetuning step.

Finetuning Step. We refine the initial distribution ratio by incorporating downstream feedback. As will be detailed in Section 4.2.3, the Scheduler converts x into a scheduling plan and reports whether the current allocation overloads the CPU or GPU. For instance, given the scheduling plan (cbs, gbs), namely the CPU and GPU buffer sizes, if the feedback indicates CPU overload (or GPU overload),

Algorithm 1 The `Dual-Buffer Scheduling` Algorithm.

Input: GPU/CPU batching iterator $iter\_g$/$iter\_c$; GPU/CPU buffer size gbs/cbs; initial GNN Model M; optimizer opt; loss function $loss\_func$; default CUDA stream s_0 for model training and GPU batching; CUDA stream s_1 for DMA transferring.
Output: trained GNN Model M';

```
1:  buf_c ← deque(size=cbs)                          ▷ CPU buffer
2:  buf_g ← deque(size=gbs)                          ▷ GPU buffer
3:  def train_batch(batch, label, event):
4:      s0.wait_event(event)          ▷ data transfer synchronization
5:      loss = loss_func(M(batch), label)
6:      loss, opt ···                 ▷ calculate gradients and update params
7:  while true do
8:      if iter_g.end() or iter_c.end() then
9:          ···                       ▷ dispose of remaining mini-batches and break
10:     end if
11:     if buf_c.not_full() and buf_g.not_full() then
12:         buffers_filling(buf_c, buf_g, iter_c, iter_g)
13:     else if buf_c.not_full() or buf_g.not_full() then
14:         buffers_blocking(buf_c, buf_g, iter_c, iter_g)
15:     else
16:         buffers_flushing(buf_c, buf_g)
17:     end if
18: end while
19: return M'
```

Dispatcher accordingly updates the x to $\frac{gbs}{cbs-1}$ (or $\frac{gbs}{cbs+1}$), which essentially decreases (or increases) cbs by one in the next loop. Historical scheduling plans and feedback are stored to prevent repetition. Since cbs is an integer, the feedback loop would become infinite only if we increase cbs by one forever, where x gradually approaches 0. This extreme case, which implies that the CPU should handle the entire batching workload, is explicitly managed as will be discussed in Section 4.2.3. When cbs equals 1 and the updated x equals positive infinity, `MorphGL` falls back to DUCATI's pure GPU training. Overall, the feedback loop is guaranteed to converge.

4.2.3 Scheduler. Given the distribution ratio, the Scheduler devises a schedule plan with the `Dual-Buffer Scheduling` algorithm to arrange the execution of all training stages across CPU, PCIe, and GPU. Depending on the internal evaluation, the Scheduler either (1) provides feedback to the Dispatcher to finetune the current distribution ratio, or (2) outputs the final scheduling plan to the Executor. Next, we present details of the `Dual-Buffer Scheduling` algorithm, whose key idea is to mimic the optimal scheduling in Figure 6b but adhere to all the constraints.

`Dual-Buffer Scheduling`. To maximize hardware utilization through asynchronous execution, we construct a **CPU buffer** in main memory to hold mini-batches generated by the CPU and a **GPU buffer** in device memory to hold mini-batches ready on the GPU. Mini-batches in the CPU buffer are prepared for DMA transfers, while those in the GPU buffer are ready for model training. The GPU buffer size gbs is a hyperparameter, while the CPU buffer size cbs depends on x with $cbs = \lfloor gbs/x \rfloor$. The pseudo code of the `Dual-Buffer Scheduling` algorithm is shown in Algorithm 1. `Dual-Buffer Scheduling` iteratively performs three phases: (1) the *buffer filling phase*, (2) the *buffer blocking phase*, and (3) the *buffer flushing phase*. Their pseudo codes are presented in Algorithm 2, 3, and 4, respectively. One complete cycle of these three phases is defined as an **overlap**. We present the detailed description of the `Dual-Buffer Scheduling` algorithm as follows.

We first initialize the CPU/GPU buffer buf_c/buf_g according to user configurations (Line 1-2 of Algorithm 1). The model training function on a mini-batch takes an event argument to ensure data transfer completes before training for CPU-generated mini-batches (Line 3-6 of Algorithm 1). In the main loop, unless either batching iterator $iter\_c$ or $iter\_g$ is exhausted (Line 8-10 of Algorithm 1), `Dual-Buffer Scheduling` executes one of the three phases.

Starting with empty buffers, we enter the *buffers filling phase* (Line 11-12 in Algorithm 1). Here, $iter\_c$ and $iter\_g$ independently generate mini-batches appended to buf_c and buf_g, respectively (Line 1-6 of Algorithm 2). During this phase, $iter\_c$ occupies CPU and $iter\_g$ occupies PCIe and GPU, resulting in no hardware idleness. If both buffers fill simultaneously, we proceed to the *buffers flushing phase*; otherwise, we enter the *buffers blocking phase*.

In the *buffers blocking phase*, one buffer is full while the other is not (Line 13-14 in Algorithm 1). We block mini-batch generation on the full buffer's hardware and continue filling the other (Line 1-7 of Algorithm 3). In implementation, an engineering optimization is adopted here. If buf_c is full but buf_g is not, blocking occurs; otherwise, we pop one mini-batch from buf_g, train the GNN model on the mini-batch, and append a newly generated mini-batch from $iter\_g$ to buf_g. This trick prevents GPU and PCIe idleness when $iter\_c$ is too slow. During this phase, we record which hardware causes blocking. At the end of scheduling, all records are aggregated to provide feedback to Dispatcher. If GPU blocking dominates, it indicates that distribution ratio x allocates excessive workload to the GPU. The Dispatcher then adjusts by increasing cbs. Through repeated feedback and tuning, CPU-GPU workload imbalance is progressively reduced. For instance, if GPU workload significantly exceeds CPU workload, CPU buffer size is increased by one repeatedly until the time gap becomes smaller than one GPU batching duration. At this point, further reallocation does not shorten the *buffers blocking phase*, marking convergence.

In the *buffers flushing phase*, both buf_c and buf_g are full (Line 15-16 of Algorithm 1). We first pop a mini-batch from the head of buf_g for later training (Line 2 of Algorithm 4). Then, we pop one mini-batch from the head of buf_c, transfer it via PCIe (DMA), and append it to buf_g (Line 3-8 of Algorithm 4). The DMA transfer runs asynchronously in a separate CUDA stream, with a CUDA event ce ensuring transfer completion before training (Line 6 of Algorithm 4, Line 4 of Algorithm 1). In parallel with DMA transfer, GPU training proceeds on the previously popped mini-batch (Line 9 of Algorithm 4). These steps repeat until we empty both buffers. Hardware is fully utilized during this phase, since $iter\_c$ prepares mini-batches with CPU for the next $overlap$ in the background, while PCIe and GPU perform data transfer and training in parallel.

We repeat multiple $overlap$s until we finish the training for the epoch. Among the three phases, hardware idleness occurs only in the *buffers blocking phase*. Using feedback, Dispatcher refines distribution ratio to minimize this phase. Consequently, most execution occurs in the *buffers filling* and *buffers flushing* phases, effectively avoiding hardware idleness during training.

Algorithm 2 The buffers_filling function.

Input: Buffers buf_c/buf_g and Batching Iterators $iter\_c$/$iter\_g$.

```
gb, gl = iter_g.next()        ▷ batch (gb) and label (gl) generated
buf_g.append((gb, gl, None))
cb, cl = iter_c.next(non_blocking=true)
if cb is not None then
    buf_c.append((cb, cl))
end if
```

Algorithm 3 The buffers_blocking function.

Input: Buffers buf_c/buf_g and Batching Iterators $iter\_c$/$iter\_g$.

```
if buf_c.not_full() and buf_g.full() then
    cb, cl = iter_c.next()
    buf_c.append((cb, cl))
else
    gb, gl = iter_g.next()
    buf_g.append((gb, gl, None))
end if
```

Algorithm 4 The buffers_flushing function.

Input: CPU buffer buf_c; GPU buffer buf_g.

```
while buf_g.not_empty() do
    gb, gl, ge = buf_g.popleft()
    if buf_c.not_empty() then
        cbcl = buf_c.popleft()
        cb, cl = cbcl.dma_transfer(s_1)   ▷ async transfer on s_1
        ce = s_1.record_event()
        buf_g.append((cb,cl,ce))
    end if
    train_batch(gb, gl, ge)               ▷ parallel train on s_0
endwhile
```

Performance Model While the above overlaps can be executed on hardware to collect feedback information, it is more efficient to perform them in a simulator that provides equivalent feedback. This simulation is enabled by the stable duration of X_i and our buffered execution. As discussed in Section 4.2.1, the number of sampled neighbors per mini-batch follows a normal distribution with a small coefficient of variation (CV), defined as the ratio of the standard deviation to the mean. A small CV indicates a narrow distribution, representing similar batching, transferring, and training workloads across mini-batches. Moreover, grouping the execution of X_i with buffering further reduces the CV, resulting in more stable execution times. Given this stability, we implement a simple yet accurate Python-based simulator for `Dual-Buffer Scheduling` that emulates scheduling without actual execution. The simulator serves as an efficient performance model, providing feedback to Dispatcher and predicting training time under the current distribution ratio and scheduling plan.

Extreme Cases We find that in extreme cases, a trivial optimal training plan already exists, corresponding to the naïve pipeline scheduling of batching on CPU, DMA transfer on PCIe, and model training on GPU. When model training takes longer than both CPU batching and PCIe transfer, the model computation becomes the bottleneck, making naïve pipelining optimal. Such cases are rare in GNN training, as GNN models typically have shallow depths and small sizes to prevent over-smoothing [50]. Although we did not observe such cases in our experiments, `MorphGL` still supports naïve pipeline training for completeness and generality.

Theoretical Analysis We provide the theoretical analysis of our proposed method. We rest our analysis upon the following two assumptions. First, we assume that the batching on CPU is longer than the data transferring on PCIe and model training on GPU, which holds for the majority of GNN training cases as we discussed earlier in this Section. Second, we assume that the training stage execution durations are the same for different mini-batches, which is supported by the stability observation as in Section 4.2.1.

Theorem 2. Given the bandwidth of PCIe as Bdw_p and the bandwidth of GPU memory as Bdw_g, the final converged solution of the Dispatcher and the Scheduler has an approximation rate of $3 + \frac{Bdw_p}{Bdw_g}$ for the problem in Definition 1.

Proof Sketch of Theorem 2. Due to page limit, we present the proof sketch here and put the full proof in our technical report [1]. We denote the optimal makespan as C^{opt}, the makespan of our method as C^0, the end time of the last operation on each device as $End^{opt}(X)$ and $End^0(X)$, $X \in \{GPU, PCIe, CPU\}$ for the optimal scheduling and our scheduling, respectively, the duration of the GPU model training/GPU batching/CPU batching/DMA transferring/UVA transferring stage as $t_{model}/t_{gb}/t_{cb}/t_{dma}/t_{uva}$, the total number of mini-batches as n, and the number of mini-batches assign to CPU/GPU as $n^{opt}_{CPU}/n^{opt}_{GPU}$ and n^0_{CPU}/n^0_{GPU}.

We first bound the optimal makespan C^{opt}. Specifically, the completion of one epoch of training is marked by the end of the execution of the last mini-batch's model training on GPU:

$$\begin{cases} C^{opt} = End^{opt}(GPU) > n \times t_{model} + n^{opt}_{GPU} \times t_{gb}, \\ C^{opt} > End^{opt}(PCIe) > n^{opt}_{GPU} \times t_{gb} + n^{opt}_{CPU} \times t_{dma}, \\ C^{opt} > End^{opt}(CPU) > n^{opt}_{CPU} \times t_{cb}. \end{cases} \tag{7}$$

Second, we bound the makespan C^0 of our method. We denote the size of CPU buffer and GPU buffer as p and q. We denote the blocking wait idleness duration in the *buffers blocking phase* as *idle*:

$$\begin{cases} C^0 = End^0(GPU) = idle + n \times t_{model} + n^0_{GPU} \times t_{gb}, \\ idle < t_{gb} \times (\frac{n}{p+q} + 1). \end{cases} \tag{8}$$

Third, we bound the execution duration of DMA- and UVA-based transferring with respect to hardware bandwidths. We denote the number of sampled neighbors/edges per mini-batch as N/E, node feature dimension as f, and data type of feature/topology as c_1/c_2:

$$\begin{cases} t_{dma} = \frac{Nfc_1 + 2Ec_2}{Bdw_p}, \\ t_{uva} < \frac{Nfc_1 + 2Ec_2}{Bdw_p} + \frac{2Ec_2}{Bdw_g}. \end{cases} \tag{9}$$

Finally, we combine the bounds in Equations 7, 8, and 9, and derive the approximation rate A:

$$A \triangleq \frac{C^0}{C^{opt}} < 3 + \frac{Bdw_p}{Bdw_g}. \tag{10}$$

□

Table 3: Datasets. The node features have float16 type. The bi-directed topology data has int64 type in the CSC format.

Dataset	#Nodes	#Edges	#Feat.	Size(topo)	Size(nfeat)
PA	111M	3.0B	100	26GB	27GB
TW	41.7M	2.1B	256	18GB	20GB
UK	77.7M	5.3B	256	41GB	37GB

Theorem 2 ensures that the output scheduling plan of our solution is close to the optimal scheduling plan.

4.2.4 Executor. The Executor absorbs the final scheduling plan and provides a one-epoch training interface, following existing works [12, 22, 57, 69]. This interface could be manually called multiple times by users or automatically managed with early stopping configurations, depending on the training progress and task-specific training requirements. MorphGL and existing systems are all developed to minimize the average training time of one epoch, which serves as a major and fair metric for training efficiency.

5 EXPERIMENTS

Experiments were conducted with various hardware setups, GNN models, and datasets. Notably, MorphGL preserves mini-batch content and model computations, ensuring identical accuracy and convergence to baselines. The only difference lies in epoch training time, which is the major metric we report and compare in this work.

5.1 Experimental Setup

Hardware. Real-world users usually have diverse hardware settings. In this work, we concentrate on improving GNN training efficiency on common hardware. Because NVIDIA takes 88% of GPU market [49], we mainly conduct experiments on three representative machines with NVIDIA GPU and the top-three CPU:GPU ratios, namely 12, 6, and 8. Since we observe that instances with stronger GPU usually have more CPU cores [4], we correspondingly choose enterprise-level A30, MIG-A30, and consumer-level RTX 3090 for each machine. Here MIG-A30 stands for A30 with the 2g.12gb multi-instance GPU setting which uses half of the total SMs and HBM, representing low-end GPUs. To verify the influence of the sizes of device memory on the training, we additionally test MorphGL and baselines on an NVIDIA V100 (32GB) machine. These machines are equipped with 128GB or more main memory. The CPU types for A30, RTX 3090, and V100 are Intel Xeon Gold 6248R, Silver 4210, and Gold 6240, respectively. PCIe 3.0x16 is equipped for all machines. Following the convention of cloud service providers [4, 16, 43], we correspondingly set the number of CPU cores available to each GPU. In summary, we conduct experiments on the following machines:

- **Machine A**: NVIDIA A30 (24G), twelve CPU cores.
- **Machine B**: NVIDIA RTX 3090 (24G), eight CPU cores.
- **Machine C**: NVIDIA MIG-A30 (12G), six CPU cores.
- **Machine D**: NVIDIA V100 (32G), eight CPU cores.

Datasets. We focus on scalable GNN training for billion-scale graphs using three widely adopted large-scale datasets (Table 3): a web graph (UK-2006-05, UK) [5], a social graph (Twitter, TW) [23], and an academic citation graph from the Open Graph Benchmark (Ogbn-Papers100M, PA) [19]. Following prior works [22, 62], we generate 256 random node features per vertex for UK and TW, which lack original features. We add bidirectional edges to all datasets, as is standard [19]. For PA, we use the official OGB training split; for UK and TW, we randomly sample 1% of vertices as the training set [34, 62]. Following [22], node features are stored in float16 and graph topology in int64 CSC format.

GNN Models. We conduct experiments with three mainstream GNN models, namely GCN [60], GraphSAGE [17], and GAT [55]. Following previous works [34, 44], we use a three-layer GNN model for experiments with (15, 10, 5) as fanouts and 1024 as training batch size. We set the hidden layer size of GCN, GraphSAGE, and GAT as 16, 256, and 64, respectively, which are given in their original papers [55, 60] and previous works [22, 69]. In terms of computing complexity and model training time on GPU, GraphSAGE is larger than GCN due to the larger hidden layer size, and GAT is larger than GraphSAGE due to the attention mechanism.

Baselines. We mainly compare MorphGL to the two strongest baselines, namely SALIENT and DUCATI, the state-of-the-art systems in Category I and Category II systems. SALIENT has advanced implementations for parallel CPU-based batching and DUCATI has dedicated dual-cache management and GPU-based batching kernels. For experiments of SALIENT and MorphGL, one CPU core is reserved for the main process logic and the rest of CPU cores are for batching workers. To demonstrate the necessity of MorphGL's design, we construct a baseline variant, DUCATI$^{\Delta}$, by integrating the CPU batching mechanism from SALIENT into DUCATI. DUCATI$^{\Delta}$ offloads part of the batching workload to the CPU, which asynchronously generates mini-batches in the background and transfers them to the GPU on demand. At each training step, a mini-batch is generated by the CPU with probability r ($0 \leq r \leq 1$) and by the GPU otherwise. r is the hyperparameter representing the fraction of the total batching workload handled by the CPU in DUCATI$^{\Delta}$.

Hyperparameters. For MorphGL, we set the number of mini-batches for profiling k as 1000 and set the size of the GPU buffer *gbs* as 10. DUCATI and SALIENT use the same hyperparameters as reported in their original works. For DUCATI$^{\Delta}$, we perform a grid search for the hyperparameter r from 0 to 1 with a stepsize of 0.05.

Metrics and Measurements. We measure and report (1) the training time per epoch of each system and (2) the hardware utilization of CPU, PCIe, and GPU in different settings. The training times are computed by calculating the averages over five training epochs after one epoch of warmup. For hardware utilization, we measure CPU utilization with `top`, measure GPU utilization with `nvidia-smi`, and measure PCIe bandwidth utilization with `perf-iostat`.

5.2 Main Results: Epoch Time Comparison

We compare the one-epoch training time of the three systems, as shown in Table 4. Overall, MorphGL outperforms SALIENT and DUCATI by up to 2.76× (1.55× on average) and 2.2× (1.41× on average), respectively. MorphGL consistently achieves speedups across all settings, demonstrating its adaptability and efficiency. Note that DUCATI (SALIENT) is inherently designed to perform batching solely on the GPU (CPU), without the ability to utilize both for batching. This leads to substantial underutilization of CPU (GPU) computing resources, explaining MorphGL's performance gain of

Table 4: Epoch training time comparison (unit: s).

	Machine A			Machine B			Machine C		
	SALIENT	DUCATI	MorphGL	SALIENT	DUCATI	MorphGL	SALIENT	DUCATI	MorphGL
					GCN				
PA	29.45 (1.74x)	19.91 (1.18x)	16.94 (1.00x)	54.08 (2.71x)	20.72 (1.04x)	19.92 (1.00x)	57.51 (2.33x)	28.65 (1.16x)	24.72 (1.00x)
TW	10.91 (1.08x)	15.42 (1.52x)	10.14 (1.00x)	19.80 (1.84x)	14.17 (1.32x)	10.76 (1.00x)	20.54 (1.38x)	21.90 (1.47x)	14.86 (1.00x)
UK	18.74 (1.22x)	32.17 (2.09x)	15.42 (1.00x)	36.14 (1.54x)	29.02 (1.24x)	23.40 (1.00x)	38.44 (1.25x)	41.58 (1.35x)	30.83 (1.00x)
					GraphSAGE				
PA	29.12 (1.71x)	22.75 (1.34x)	17.02 (1.00x)	53.63 (2.76x)	20.44 (1.05x)	19.40 (1.00x)	56.58 (1.79x)	38.30 (1.21x)	31.57 (1.00x)
TW	10.89 (1.05x)	16.69 (1.62x)	10.33 (1.00x)	19.34 (1.91x)	14.11 (1.39x)	10.14 (1.00x)	22.27 (1.16x)	26.04 (1.35x)	19.22 (1.00x)
UK	18.62 (1.12x)	35.59 (2.14x)	16.62 (1.00x)	33.40 (1.57x)	30.49 (1.44x)	21.24 (1.00x)	38.33 (1.11x)	51.47 (1.49x)	34.65 (1.00x)
					GAT				
PA	29.05 (1.35x)	28.30 (1.32x)	21.49 (1.00x)	53.50 (1.87x)	28.93 (1.01x)	28.57 (1.00x)	62.82 (1.85x)	38.42 (1.13x)	33.97 (1.00x)
TW	10.98 (1.03x)	17.85 (1.67x)	10.69 (1.00x)	20.56 (1.58x)	16.06 (1.24x)	13.00 (1.00x)	23.05 (1.31x)	24.40 (1.38x)	17.66 (1.00x)
UK	18.40 (1.09x)	37.12 (2.20x)	16.86 (1.00x)	33.43 (1.20x)	36.43 (1.30x)	27.92 (1.00x)	36.94 (1.21x)	46.73 (1.53x)	30.54 (1.00x)
Avg.	1.27 ×	1.67 ×	1.00 ×	1.89 ×	1.23 ×	1.00 ×	1.49 ×	1.34 ×	1.00 ×
Max	1.74 ×	2.20 ×	1.00 ×	2.76 ×	1.44 ×	1.00 ×	2.33 ×	1.53 ×	1.00 ×

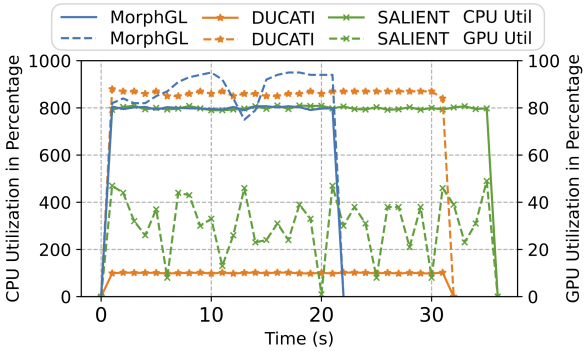


Figure 7: Detailed CPU and GPU utilization for the three systems in one training epoch (UK, GraphSAGE, Machine B).

over two times. We further analyze training efficiency across different machines, models, and datasets below.

First, comparing system performance across machines highlights the advantage of MorphGL's collective batching and scheduling design. SALIENT favors machines with abundant CPU cores relative to the GPU, performing reasonably on Machine A–only 1.27× slower than MorphGL, but degrading on Machine B, where fewer CPU cores make it 1.89 × slower. DUCATI shows similar variability across machines. In contrast, MorphGL maintains stable and consistently superior efficiency due to its collective batching strategy.

Because GNN training involves many repeated epochs, we profile CPU and GPU utilization for the three systems during a single stabilized epoch, as shown in Figure 7, aligning their start times for consistency. SALIENT maintains high CPU utilization but shows fluctuatingly low GPU usage, as GPU often waits for mini-batches. DUCATI achieves high GPU utilization but ignores 7 of 8 CPU cores. In contrast, MorphGL sustains high utilization on both CPU and GPU, resulting in shorter epoch time than baselines.

Second, different GNN models create distinct workload distributions and dispatching requirements. For instance, on Machine C with SALIENT, training GCN (compared to GAT) shortens GPU computation time, widening the gap between CPU batching and GPU training, and worsening GPU idleness. Thus, SALIENT's efficiency for GCN (1.65× slower than MorphGL on average) is lower than for GAT (1.46× slower). In contrast, MorphGL adapts to model-dependent workload changes by shifting more batching to the GPU, achieving balanced utilization and higher efficiency.

Third, baseline efficiency also varies across datasets for varying data locality. DUCATI performs best on PA but worst on UK, as its dual-cache design heavily depends on data locality. PA's labeled training nodes have high degrees and dense connections [19, 62, 69], offering strong locality, while UK's randomly sampled nodes provide little. MorphGL adapts to such locality differences, dynamically redistributing workloads. When detecting faster GPU batching on PA due to better locality, MorphGL allocates more batching to the GPU, maintaining high efficiency regardless of dataset locality.

5.3 Influence of CPU&GPU Setup

We examine how CPU-GPU configurations affect efficiency as shown in Figure 8. We vary the number of CPU cores paired with one RTX 3090 GPU from 2 to 16 in steps of 2. We report average training speed for the three systems using the UK dataset and GraphSAGE model; similar trends are observed in other settings.

The relative efficiency of SALIENT and DUCATI is highly sensitive to the CPU:GPU ratio. They fully utilize hardware only with either very few or many CPU cores. With fewer cores, DUCATI's GPU-based batching outperforms SALIENT. But with more than 10 CPU cores, SALIENT's CPU-based batching becomes faster. This

Table 5: CPU/PCIe/GPU utilization in percentage (%C/%P/%G).

	SALIENT			DUCATI			MorphGL		
	%C	%P	%G	%C	%P	%G	%C	%P	%G
PA	798	26.1	21.6	101	12.9	66.9	801	33.0	69.3
TW	797	39.4	39.3	100	23.5	87.7	799	51.5	87.4
UK	801	34.3	30.4	100	25.2	86.4	798	45.9	88.7

Table 6: Initial x, (CPU buffer size, GPU buffer size) examples.

	Machine A	Machine B	Machine C
PA	0.65, (15, 10)	2.00, (5, 10)	0.90, (11, 10)
TW	0.29, (20, 10)	1.12, (8, 10)	0.41, (24, 10)
UK	0.19, (30, 10)	0.83, (12, 10)	0.36, (28, 10)

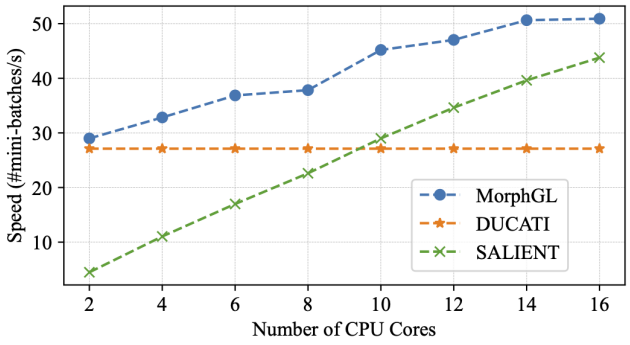


Figure 8: The training speed comparison of three systems with different numbers of CPU cores per RTX3090.

indicates that both CPU and GPU are effective for GNN training and should be jointly leveraged, as in MorphGL.

However, MorphGL consistently outperforms both baselines across all CPU:GPU ratios. As CPU cores increase, MorphGL shifts more batching from GPU to CPU, achieving higher speed. When CPU resources are extremely low or high, the gain diminishes, and MorphGL's efficiency approaches that of pure CPU or GPU batching.

5.4 Hardware Utilization

We compare CPU, PCIe, and GPU utilization across the three systems, as shown in Table 5. Training is profiled with GraphSAGE on Machine B, with similar trends in other settings. Overall, SALIENT achieves high CPU but low GPU utilization, while DUCATI keeps the GPU busy but idles 87.5% of CPU cores. Both underutilize PCIe bandwidth, as training is bottlenecked by CPU- or GPU-side batching. DUCATI's dual-cache design lowers PCIe utilization by reducing data transfers. In contrast, MorphGL maintains high utilization across CPU, PCIe, and GPU, effectively minimizing hardware idleness through balanced workload dispatching and scheduling.

Table 7: Epoch training time (unit: s) comparison on Machine D with different device memory sizes. T_{GBT}: Average GPU Batching stage duration per mini-batch (unit: ms).

Device (Mem.)	SALIENT	DUCATI	MorphGL	T_{GBT}
V100 (12 GB)	40.97 (1.29x)	54.26 (1.71x)	31.68 (1.00x)	38.80
V100 (16 GB)	40.76 (1.37x)	49.53 (1.67x)	29.67 (1.00x)	34.78
V100 (32 GB)	40.21 (1.38x)	48.61 (1.67x)	29.19 (1.00x)	32.99

Table 8: Preprocessing time of MorphGL (unit: s).

	Machine A		Machine B		Machine C	
	Prep	Epoch	Prep	Epoch	Prep	Epoch
PA	65.5	17.0	86.3	19.4	112.2	31.5
TW	41.0	10.3	**49.6**	**10.1**	63.4	19.2
UK	67.0	16.6	76.7	21.2	**110.6**	**34.7**

5.5 Scheduling Statistics and Examples

Across all of the 27 training settings, the average and the maximum number of rounds of the feedback loops in MorphGL are 17.6 and 53, respectively. We present the initial x used in the Dispatcher and the final configurations of the CPU buffer size and the GPU buffer size in Dual-Buffer Scheduling w.r.t. the GraphSAGE Model in Table 6. The final x determined is the ratio of *gbs* to *cbs*. Overall, the proportion of the CPU batching workload to total batching workload ranges from 33% to 75%, which demonstrates that MorphGL can assign batching workload to CPU and GPU adaptively. Compared to GPU batching speed, the CPU batching speed increases faster from Machine C to Machine A. Therefore, MorphGL relocates more batching workload from GPU to CPU correspondingly.

5.6 Influence of the Device Memory Size

We report epoch training time for the three systems on Machine D with varying device memory sizes, as shown in Table 7. Memory limits of 12GB, 16GB, and 32GB were set using PyTorch. Results are based on the UK dataset with the GAT model, with similar trends in other settings. First, larger device memory allows DUCATI to cache more topology and feature data, reducing epoch training time. However, as noted in [69], the benefit diminishes; the speedup from 16GB to 32GB is smaller than from 12GB to 16GB (T_{GBT} column). Second, CPU batching speed is unaffected by device memory, though larger memory slightly improves SALIENT by reducing device-side garbage collection [8]. Third, MorphGL adapts to batching speed changes in DUCATI and SALIENT caused by varying device memory, maintaining consistently superior performance.

5.7 Preprocessing Time

We define preprocessing time as the combined time of the Profiler, Dispatcher, and Scheduler, and compare it to epoch training time in Table 8, using the GraphSAGE model (similar trends hold for other settings). Preprocessing time is always under five epochs of training, ranging from a maximum ratio of 4.9 (Machine B, TW) to a minimum of 3.2 (Machine C, UK), with an average of 3.9. As

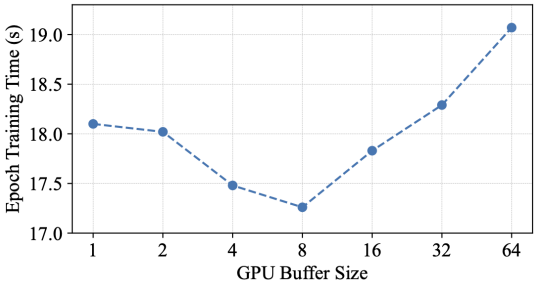


Figure 9: MorphGL with different GPU buffer sizes.

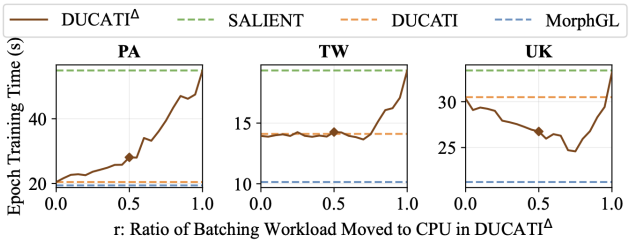


Figure 10: Evaluation of DUCATI$^{\Delta}$ against other systems.

prior works note [13, 59, 62], training GNNs on large graphs typically requires hundreds or thousands of epochs. Thus, MorphGL's preprocessing overhead is negligible relative to total training time.

5.8 Influence of the GPU Buffer Size

To study the impact of the hyperparameter GPU buffer size on MorphGL, we measure epoch training time (Figure 9) using GraphSAGE on the PA dataset on Machine A; similar trends occur in other settings. The mean and standard deviation of mini-batch sizes are 149.7 MB and 2.0 MB, respectively. Small GPU buffers (e.g., 1) limit the Dual-Buffer Scheduling algorithm's operational space, causing frequent transitions between filling, blocking, and flushing phases, and requiring repeated CUDA stream synchronization, which increases training time. Conversely, large buffers (e.g., 64) consume excessive device memory, reducing space for GPU topology and feature caches [69], prolonging GPU batching and overall training. We select a GPU buffer size of 10, occupying under 2 GB, which balances sufficient operational space for Dual-Buffer Scheduling with ample memory for GPU caches.

5.9 Comparison against DUCATI$^{\Delta}$

To evaluate the necessity of MorphGL's dedicated designs, we compare DUCATI$^{\Delta}$–a naïve hybrid CPU-GPU batching system without adaptive workload dispatching and buffered scheduling–against the three systems (Figure 10). Experiments use GraphSAGE on Machine B, with similar trends in other settings. We perform a grid search over DUCATI$^{\Delta}$'s CPU workload ratio $r \in [0, 1]$ with step 0.05.

Two key observations emerge. First, DUCATI$^{\Delta}$ does not consistently outperform DUCATI. With an even CPU-GPU split (brown diamond markers), DUCATI$^{\Delta}$ is 1.37× slower on PA, 1.02× slower on TW, and 1.18× faster on UK compared to DUCATI. This inconsistency reflects sensitivity of workload partitioning to training configurations; without adaptive dispatching, naïve workload partitioning often yields suboptimal performance. Second, MorphGL consistently outperforms DUCATI$^{\Delta}$ across all datasets. Even at DUCATI$^{\Delta}$'s best grid-search configuration, it remains 1.05×, 1.34×, and 1.16× slower than MorphGL on PA, TW, and UK, respectively. Naively enabling both the CPU and GPU to conduct batching introduces PCIe contention between DMA and UVA transfers in DUCATI$^{\Delta}$. Without dedicated scheduling, workloads on the CPU, PCIe, and GPU cannot be effectively overlapped, leading to low hardware utilization and suboptimal performance for DUCATI$^{\Delta}$.

6 CONCLUSION

In this paper, we present MorphGL, a novel GNN training system to speed up mini-batch GNN training on billion-scale graphs with collective batching and scheduling. We identify the performance bottleneck of dedicated batching systems on common hardware: the hardware resource under-utilization due to the static workload-processor binding. MorphGL addresses the problem by adaptively dispatching the batching workload and scheduling training stages across CPU, PCIe, and GPU to minimize the training time. We formulate the collective batching and scheduling problem, prove the problem is NP-hard, and propose an iterative method, which includes a tunable dispatching scheme and the Dual-Buffer Scheduling algorithm, with a theoretical bound to solve it. We evaluate the performance of MorphGL on four common machines, three sizes of GNN models, and three billion-scale datasets. Experimental results show that MorphGL has strong adaptivity and consistently outperforms the state-of-the-art GNN training systems, namely SALIENT and DUCATI, by up to 2.76x and 2.2x speedup on training time.

ACKNOWLEDGMENTS

This work is supported by National Key Research and Development Program of China Grant No. 2023YFF0725100. Yanyan Shen is supported by National Natural Science Foundation of China (No. 62522211) and Key Research and Development Program of Xinjiang Uygur Autonomous Region (Grant No. 2023B01027, 2023B01027-1). Yingxia Shao is supported by National Natural Science Foundation of China (Nos. 62272054, 62192784), Beijing Nova Program (No. 20230484319, 20250484968), CAAI-CANN Open Fund, developed on OpenI Community (No. CAAIXSJLJJ2025CANN10) and State Key Laboratory of Multimedia Information Processing Open Fund (No. SKLMIP-KF-2025-07). Lei Chen is supported by National Science Foundation of China under Grant No. U22B2060, Guangdong-Hong Kong Technology Innovation Joint Funding Scheme Project No. 2024A0505040012, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, CRF Project C2004-21G, Key Areas Special Project of Guangdong Provincial Universities 2024ZDZX1006, Guangdong Province Science and Technology Plan Project 2023A0505030011, HKUST(GZ) - CMCC(Guangzhou Branch) Metaverse Joint Innovation Lab under Grant No. P00659, Hong Kong ITC ITF grant PRP/004/22FX, Zhujiang scholar program 2021JC02X170, HKUST-Webank joint research lab.

REFERENCES

[1] [n.d.]. MorphGL technical report. https://github.com/initzhang/morphGL/blob/main/report.pdf

[2] Xin Ai, Qiange Wang, Chunyu Cao, Yanfeng Zhang, Chaoyi Chen, Hao Yuan, Yu Gu, and Ge Yu. 2024. NeutronOrch: Rethinking Sample-Based GNN Training under CPU-GPU Heterogeneous Environments. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1995–2008.

[3] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. *Proc. VLDB Endow.* 18, 2 (Oct. 2024), 173–186. https://doi.org/10.14778/3705829.3705837

[4] Amazon. [n.d.]. AWS GPU instances. https://aws.amazon.com/ec2/instance-types

[5] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th WWW*. 595–602.

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

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

[8] PyTorch Documents. [n.d.]. Memory management. https://pytorch.org/docs/stable/notes/cuda.html#memory-management/

[9] Joshua Fan, Junwen Bai, Zhiyun Li, Ariel Ortiz-Bobea, and Carla P Gomes. 2022. A GNN-RNN approach for harnessing geospatial and temporal information: application to crop yield prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 36. 11873–11881.

[10] Wenfei Fan, Lihang Fan, Dandan Lin, and Min Xie. 2024. Explaining GNN-Based Recommendations in Logic. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 715–728. https://doi.org/10.14778/3712221.3712237

[11] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2024. STile: Searching hybrid sparse formats for sparse deep learning operators automatically. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

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

[13] 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)*. 551–568.

[14] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. ETC: Efficient Training of Temporal Graph Neural Networks over Large-Scale Dynamic Graphs. *Proc. VLDB Endow.* 17, 5 (Jan. 2024), 1060–1072. https://doi.org/10.14778/3641204.3641215

[15] Shihong Gao, Yiming Li, Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. SIMPLE: Efficient Temporal Graph Neural Network Training at Scale with Dynamic Data Placement. *Proc. ACM Manag. Data* 2, 3, Article 174 (May 2024), 25 pages. https://doi.org/10.1145/3654977

[16] Google. [n.d.]. GCP GPU instances. https://cloud.google.com/compute/docs/gpus

[17] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

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

[19] 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. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[20] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 311–326.

[21] Yuli Jiang, Yu Rong, Hong Cheng, Xin Huang, Kangfei Zhao, and Junzhou Huang. 2022. Query driven-graph neural networks for community search: from non-attributed, attributed, to interactive attributed. *Proc. VLDB Endow.* 15, 6 (Feb. 2022), 1243–1255. https://doi.org/10.14778/3514061.3514070

[22] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. 2022. Accelerating training and inference of graph neural networks with fast sampling and pipelining. *Proceedings of Machine Learning and Systems* 4 (2022), 172–189.

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

[24] Haoyang Li and Lei Chen. 2023. EARLY: Efficient and Reliable Graph Neural Network for Dynamic Graphs. *Proc. ACM Manag. Data* 1, 2, Article 163 (June 2023), 28 pages. https://doi.org/10.1145/3589308

[25] Haoyang Li, Shimin Di, and Lei Chen. 2022. Revisiting injective attacks on recommender systems. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2174, 14 pages.

[26] Haoyang Li, Shimin Di, Lei Chen, and Xiaofang Zhou. 2024. E2GCL: Efficient and Expressive Contrastive Learning on Graph Neural Networks. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 859–873. https://doi.org/10.1109/ICDE60146.2024.00071

[27] Haoyang Li, Shimin Di, Calvin Hong Yi Li, Lei Chen, and Xiaofang Zhou. 2024. Fight Fire with Fire: Towards Robust Graph Neural Networks on Dynamic Graphs via Actively Defense. *Proc. VLDB Endow.* 17, 8 (April 2024), 2050–2063. https://doi.org/10.14778/3659437.3659457

[28] Haoyang Li, Shimin Di, Zijian Li, Lei Chen, and Jiannong Cao. 2022. Black-box Adversarial Attack and Defense on Graph Neural Networks. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1017–1030. https://doi.org/10.1109/ICDE53745.2022.00081

[29] Jia Li, Yanyan Shen, Lei Chen, and Charles Wang Wai Ng. 2023. SSIN: Self-Supervised Learning for Rainfall Spatial Interpolation. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–21.

[30] Xujia Li, Yanyan Shen, and Lei Chen. 2021. Mcore: Multi-Agent Collaborative Learning for Knowledge-Graph-Enhanced Recommendation. In *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 330–339.

[31] Zhiyuan Li, Xun Jian, Yue Wang, and Lei Chen. 2022. CC-GNN: A Community and Contraction-based Graph Neural Network. In *2022 IEEE International Conference on Data Mining (ICDM)*. 231–240. https://doi.org/10.1109/ICDM54844.2022.00033

[32] Zhiyuan Li, Xun Jian, Yue Wang, Yingxia Shao, and Lei Chen. 2024. DAHA: Accelerating GNN Training with Data and Hardware Aware Execution Planning. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1364–1376.

[33] Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. 2022. SCARA: scalable graph neural networks with feature-oriented optimization. *Proc. VLDB Endow.* 15, 11 (July 2022), 3240–3248. https://doi.org/10.14778/3551793.3551866

[34] 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*. 401–415.

[35] Hanmo Liu, Shimin Di, and Lei Chen. 2023. Incremental Tabular Learning on Heterogeneous Feature Space. *Proc. ACM Manag. Data* 1, 1, Article 18 (May 2023), 18 pages. https://doi.org/10.1145/3588698

[36] Hanmo Liu, Shimin Di, Haoyang Li, Xun Jian, Yue Wang, and Lei Chen. 2025. A Selective Learning Method for Temporal Graph Continual Learning. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (Eds.), Vol. 267. PMLR, 38332–38350. https://proceedings.mlr.press/v267/liu25l.html

[37] Hao Liu, Jindong Han, Yanjie Fu, Jingbo Zhou, Xinjiang Lu, and Hui Xiong. 2020. Multi-modal transportation recommendation with unified route representation learning. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 342–350. https://doi.org/10.14778/3430915.3430924

[38] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL: GPU-Efficient GNN training by optimizing graph data IO and preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 103–118.

[39] Tongxuan Liu, Tao Peng, Peijun Yang, Xiaoyang Zhao, Xiusheng Lu, Weizhe Huang, Zirui Liu, Xiaoyu Chen, Zhiwei Liang, Jun Xiong, et al. 2025. xLLM Technical Report. *arXiv preprint arXiv:2510.14686* (2025).

[40] Zirui Liu, Hailin Zhang, Boxuan Chen, Zihan Jiang, Yikai Zhao, Yangyu Tao, Tong Yang, and Bin Cui. 2025. CAFE+: Towards Compact, Adaptive, and Fast Embedding for Large-scale Online Recommendation Models. *ACM Trans. Inf. Syst.* 43, 3, Article 61 (Feb. 2025), 42 pages. https://doi.org/10.1145/3713072

[41] Ge Lv and Lei Chen. 2023. On Data-Aware Global Explainability of Graph Neural Networks. *Proc. VLDB Endow.* 16, 11 (July 2023), 3447–3460. https://doi.org/10.14778/3611479.3611538

[42] Ge Lv, Chen Jason Zhang, and Lei Chen. 2023. HENCE-X: Toward Heterogeneity-Agnostic Multi-Level Explainability for Deep Graph Networks. *Proc. VLDB Endow.* 16, 11 (July 2023), 2990–3003. https://doi.org/10.14778/3611479.3611503

[43] Microsoft. [n.d.]. Azure GPU instances. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview?#gpu-accelerated

[44] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. *arXiv preprint arXiv:2101.07956* (2021).

[45] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2024. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1227–1240. https://doi.org/10.14778/3648160.3648166

[46] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proc. VLDB Endow.* 15, 9

(May 2022), 1937–1950. https://doi.org/10.14778/3538598.3538614

[47] Jingshu Peng, Qiyu Liu, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2025. From Sancus to Sancusq: staleness and quantization-aware full-graph decentralized training in graph neural networks. *The VLDB Journal* 34, 2 (Jan. 2025), 26. https://doi.org/10.1007/s00778-024-00897-2

[48] Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. 2023. Computing Graph Edit Distance via Neural Graph Matching. *Proc. VLDB Endow.* 16, 8 (April 2023), 1817–1829. https://doi.org/10.14778/3594512.3594514

[49] Jon Peddie Research. [n.d.]. Shipments of graphics add-in boards decline in Q1 of 24 as the market experiences a return to seasonality. https://www.jonpeddie.com/news/shipments-of-graphics-add-in-boards-decline-in-q1-of-24-as-the-market-experiences-a-return-to-seasonality/

[50] T Konstantin Rusch, Michael M Bronstein, and Siddhartha Mishra. 2023. A survey on oversmoothing in graph neural networks. *arXiv preprint arXiv:2303.10993* (2023).

[51] Yu N Sotskov and Natalia V Shakhlevich. 1995. NP-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics* 59, 3 (1995), 237–266.

[52] Hannes Stärk, Dominique Beaini, Gabriele Corso, Prudencio Tossou, Christian Dallago, Stephan Günnemann, and Pietro Liò. 2022. 3d infomax improves gnns for molecular property prediction. In *International Conference on Machine Learning*. PMLR, 20479–20502.

[53] Yushi Sun, Jiachuan Wang, Peng Cheng, Libin Zheng, Lei Chen, and Jian Yin. 2024. Cross-domain-aware Worker Selection with Training for Crowdsourced Annotation. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 249–262.

[54] Zeyuan Tan, Xiulong Yuan, and Congjie He et. al. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. arXiv:2305.10863

[55] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[56] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 144–161.

[57] 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).

[58] Yubo Wang, Shimin Di, Zhili Wang, Haoyang Li, Fei Teng, Hao Xin, and Lei Chen. 2025. Understanding the Embedding Models on Hyper-relational Knowledge Graph. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management* (Seoul, Republic of Korea) *(CIKM '25)*. Association for Computing Machinery, New York, NY, USA, 3177–3187. https://doi.org/10.1145/3746252.3761370

[59] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 515–531.

[60] Max Welling and Thomas N Kipf. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

[61] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2020. Graph neural networks in recommender systems: a survey. *Comput. Surveys* (2020).

[62] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 417–434.

[63] Kevin Yang, Kyle Swanson, Wengong Jin, Connor Coley, Philipp Eiden, Hua Gao, Angel Guzman-Perez, Timothy Hopper, Brian Kelley, Miriam Mathea, et al. 2019. Analyzing learned molecular representations for property prediction. *Journal of chemical information and modeling* 59, 8 (2019), 3370–3388.

[64] Liangwei Yang, Zhiwei Liu, Yingtong Dou, Jing Ma, and Philip S Yu. 2021. Consisrec: Enhancing gnn for social recommendation via consistent neighbor aggregation. In *Proceedings of the 44th international ACM SIGIR conference on Research and development in information retrieval*. 2141–2145.

[65] Haitao Yuan, Gao Cong, and Guoliang Li. 2024. Nuhuo: An Effective Estimation Model for Traffic Speed Histogram Imputation on A Road Network. *Proc. VLDB Endow.* 17, 7 (March 2024), 1605–1617. https://doi.org/10.14778/3654621.3654628

[66] Hao Yuan, Yajiong Liu, Yanfeng Zhang, Xin Ai, Qiange Wang, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. Comprehensive Evaluation of GNN Training Systems: A Data Management Perspective. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1241–1254. https://doi.org/10.14778/3648160.3648167

[67] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[68] Hailin Zhang, Zirui Liu, Boxuan Chen, Yikai Zhao, Tong Zhao, Tong Yang, and Bin Cui. 2024. CAFE: Towards Compact, Adaptive, and Fast Embedding for Large-scale Recommendation Models. *Proc. ACM Manag. Data* 2, 1, Article 51 (March 2024), 28 pages. https://doi.org/10.1145/3639306

[69] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with the GPU. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[70] Yanping Zheng, Zhewei Wei, and Jiajun Liu. 2023. Decoupled Graph Neural Networks for Large Dynamic Graphs. *Proc. VLDB Endow.* 16, 9 (May 2023), 2239–2247. https://doi.org/10.14778/3598581.3598595

[71] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating large scale real-time GNN inference using channel pruning. *Proc. VLDB Endow.* 14, 9 (May 2021), 1597–1605. https://doi.org/10.14778/3461535.3461547

[72] Qiqi Zhou, Yanyan Shen, and Lei Chen. 2023. Narrow the Input Mismatch in Deep Graph Neural Network Distillation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 3581–3592. https://doi.org/10.1145/3580305.3599442

[73] Qiqi Zhou, Yanyan Shen, and Lei Chen. 2025. Faster Convergence in Mini-Batch Graph Neural Networks Training with Pseudo Full Neighborhood Compensation. *Proc. VLDB Endow.* 18, 11 (July 2025), 4309–4322. https://doi.org/10.14778/3749646.3749695

[74] 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. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2094–2105. https://doi.org/10.14778/3352063.3352127