# Efficient GNN Training on Giant Graphs with Collective Batching and Scheduling

Xin Zhang
The Hong Kong University of Science and Technology
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
The Hong Kong University of Science and Technology (Guangzhou)
leichen@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, 20(1): XXX-XXX, 2026.
doi:XX.XX/XXX.XX

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

## 1 INTRODUCTION

In recent years, Graph Neural Networks (GNNs) [2, 12, 18, 21, 25, 31–33, 48–50, 53–55] have emerged as a prominent and widely studied technique in graph learning. Their ability to effectively model and represent graph-structured data has led to significant advancements in various downstream tasks, such as social media recommendation [9, 23, 27, 44, 47, 56], biological molecule property prediction [14, 37, 46], and weather forecasting [8, 22]. Similar to other deep learning models [6, 15], 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 [5, 19, 26, 28, 30, 38, 40, 45, 52]. *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 [10, 19, 40, 41] 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 [19, 40, 41], optimizing node feature selection with improved data structures [19, 40], and pipelining data transfer with the asynchronized Direct Memory Access (DMA) technique [10, 19, 41]. The systems in Category II [5, 30, 38, 52] 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, even when the graph dataset resides in the main memory. Moreover, DUCATI [52], DSP [5], and Quiver [38] propose to cache frequently accessed graph data in the GPU memory to reduce data transfer volume over PCIe. We refer to Category I and II systems as *dedicated batching* systems because the mini-batch preparation workload is assigned exclusively to the CPU or the GPU in these systems.

Unfortunately, we find that dedicated batching systems suffer from low utilization of hardware resources, namely CPU cores, PCIe

Proceedings of the VLDB Endowment, Vol. 20, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

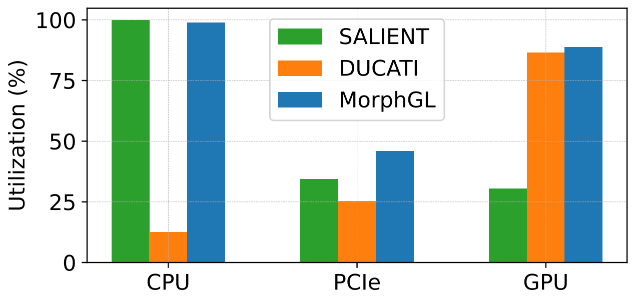


**Figure 1: Comparison of CPU, PCIe, and GPU utilizations of SALIENT, DUCATI, and `MorphGL`.**

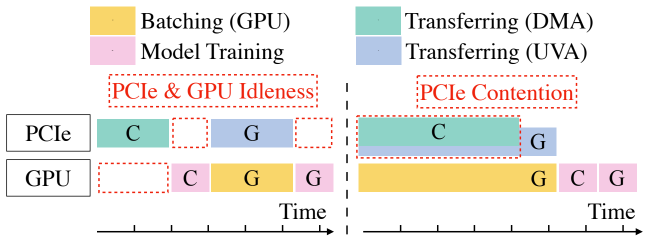


**Figure 2: Hardware idleness and contention when arranging the execution of a CPU-produced and a GPU-produced mini-batch. Colored rectangles with `C/G` represent corresponding training stages of the CPU/GPU-produced mini-batch. Left: sequential execution. Right: pipelined execution.**

bandwidth, and GPU computing capability, when training on common machines[1]. To demonstrate this, we collect and compare the hardware utilization of SALIENT [19] and DUCATI [52], representing Category I and Category II systems, respectively, in a typical training setting[2], 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 wastes 88% of CPU cores during training. The PCIe bandwidth is also underutilized in both systems.

The low hardware utilization problem 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, which usually leads to the waste of PCIe bandwidth[3].

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, which determines workload distribution and scheduling to minimize the mini-batch GNN training duration given the running machine and user-defined training configurations, in Section 4.1 and prove the problem is NP-hard. `MorphGL` contains two key components, namely the Dispatcher and the Scheduler, to solve the problem in an iterative way. To adaptively find the best workload distribution, the Dispatcher first takes in statistics collected with offline profiling on the given machine. Then, the Dispatcher calculates an initial workload distribution ratio based on necessary relaxations. Given the initial distribution ratio, the Scheduler uses the `Dual-Buffer Scheduling` algorithm which maintains two buffers, one residing in the main memory and the other in the device memory, and judiciously arranges the moment of transferring and model training to avoid both hardware contention and idleness. The Scheduler also provides downstream feedback on workload division that helps the Dispatcher adjust the old distribution ratio and output a new one. In this way, the Dispatcher and the Scheduler work together and iteratively optimize the problem until they converge on the final solution. We also establish the theoretical guarantee of the proposed solution.

We conduct extensive experiments on four machines with both consumer-level and enterprise-level GPUs and different CPU setups. Experimental results on three mainstream GNNs and three billion-scale graphs demonstrate that `MorphGL` adapts to diverse settings and consistently outperforms state-of-the-art Category I and Category II systems (i.e., SALIENT and DUCATI) by 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.

[1]Details and definitions will be given in Section 3.
[2]Details will be given in Section 5. Setting: GraphSAGE model, UK dataset, Machine B.
[3]See detailed discussion in Section 3.

- 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 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

Graph Neural Networks are powerful tools for modeling graph-structured data by recursively aggregating the node features according to the graph topology. GCN [43], the pioneering GNN model, initially uses a full-batch training method. It requires loading the entire graph dataset into device memory during model training. However, 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.

To cope with large-scale graphs, GraphSAGE [14] proposes a mini-batch training approach consisting of 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 the depth of the GNN model. The seed nodes, sampled neighbors, and their connecting edges form the topology component of the mini-batch. We then collect the node IDs of all selected nodes and fetch their corresponding node features, deriving the feature component of the mini-batch. (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.

The mini-batch training paradigm achieves scalability by limiting the data stored in the device memory. Only GNN model's weights, activations, gradients, and optimizer states, reside in the device memory. These components typically occupy less than one gigabyte in total. Hence, mainstream GNN models [43, 51], including those leading the OGB leaderboard [16] and the key examples in DGL and PyG [10, 41] frameworks, employ single-GPU training. In this paper, we also focus on single-GPU mini-batch GNN training.

Some recent studies [1, 24] have attempted to enhance the efficiency of mini-batch GNN training by modifying mini-batch composition and introducing staleness into the training process. However, 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. This paper focuses on improving training efficiency without harming model accuracy to ensure practical applicability.

### 2.2 Category I and Category II Systems

Category I systems [10, 19, 40, 41] advocate binding mini-batch preparation workload to the CPU. PyG (CPU) [10], which represents the CPU-batching mode of PyG, and DGL (CPU) [41], two of the most popular GNN libraries, provide multiprocessing-based batching and asynchronized mini-batch transferring. However, the overhead of data sharing between multiple processes harms the training efficiency. SALIENT [19] and MariusGNN [40] 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. All of them support the pipeline technique to hide the latency on CPU and PCIe as much as possible. However, these systems usually experience slow batching speeds when the number of CPU cores is insufficient. For example, on a common machine with eight CPU cores and a 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 [10, 17, 30, 38, 41, 45, 52] propose to bind all or the majority of the batching workload to GPU instead. Specifically, PyG (GPU) [10], DGL (GPU) [41], GNNLab [45] and NextDoor [17] 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 GPU OOM problem, DSP [5], PyTorch-Direct [30], Quiver [38], and DUCATI [52] adopt the Unified Virtual Addressing (UVA) technique. Storing the graph data in pinned (page-locked) main memory instead of GPU memory, the UVA technique allows GPU to directly access the graph data for fast subgraph sampling and feature selection. UVA operations occupy GPU and PCIe simultaneously but require no help from CPU, 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 transferring on PCIe. These systems work well on machines with negligible CPU resources, however, cannot leverage the CPU computing power when applied to machines with considerable CPU resources. For example, when training on the same machine with eight CPU cores and a 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* |

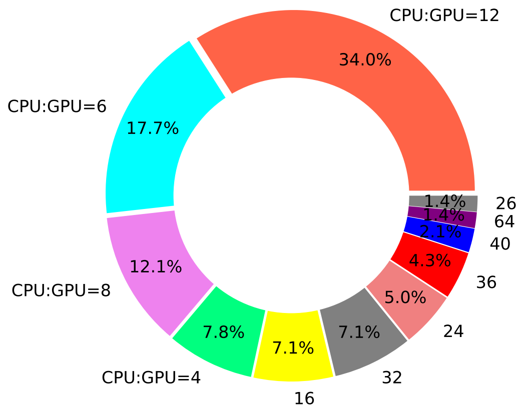


**Figure 3: The number of CPU cores per GPU (CPU:GPU ratio) distribution of 141 cloud GPU instances from AWS, Azure, and GCP.**

## 3 MOTIVATION

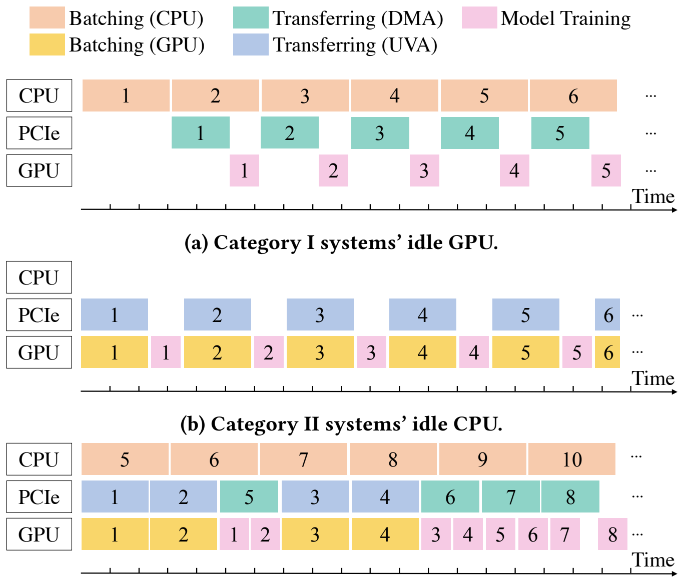
In this section, we provide relevant empirical results to illustrate the limitations of the existing systems and present the key observations that motivate the development of `MorphGL`.

**Common hardware.** In practice, it is desirable to ensure that a GNN training system aligns with common hardware. While there are many specifications to consider, 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,[4] of all 141 GPU instances to date on the three largest cloud providers, namely Amazon Web Service (AWS) [3], Microsoft Azure (Azure) [29], and Google Cloud Platform (GCP) [13]. In Figure 3, over 70% of machines have CPU:GPU ratio between 4 and 12, therefore, they are referred to as the *common hardware* in this work.

**Performance Bottlenecks.** In Figure 4, we provide the simplified training profiles [5] of Category I and Category II systems on Machine A, a server with the most common CPU:GPU ratio to be introduced in Section 5.1. We find that the performance bottleneck is the capability of the processor, either the CPU or the GPU, whichever undertakes the batching stage. Specifically, we find the following two resulting symptoms of suboptimal efficiency. (1) Category I systems are faced with the idle GPU problem. Even with the help

[4](1) While there are many other specifications such as the base/boost frequency affecting the CPU/GPU's performance, CPU:GPU ratio is the most influencing metric that reflects the relative CPU and GPU computing power. (2) Other specifications are considered by `MorphGL` through profiling as will be introduced in the next section.

[5]We visualize only the major training stages on the CPU, PCIe, and GPU. Minor operations, such as kernel launches on CPU, are omitted for brevity.



**(a) Category I systems' idle GPU.**

**(b) Category II systems' idle CPU.**

**(c) `MorphGL`'s compact execution.**

**Figure 4: 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.**

of the pipeline execution, the GPU and the PCIe are underutilized considerably due to slow CPU batching as shown in Figure 4a. (2) Category II systems have busy GPU for batching and model training, however, leaving almost no workload to the CPU. This wastes valuable CPU cores which could be used to share the intensive burden of batching. PCIe bandwidth is underutilized by both kinds of systems, as indicated by the blank space in the corresponding timeline, due to the bottleneck in the batching stage. Clearly, the root cause of these two symptoms is the *static* workload-processor binding strategy adopted 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 that is optimized for the common hardware. That is, by breaking down the mini-batch preparation workload to both CPU and GPU, we can utilize available 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.

First, the batching workload distribution is nontrivial. Users have different training configurations and conduct experiments on machines with diverse CPU-GPU setups, resulting in varying computational demands on each processor. For instance, training the computationally intensive GAT model requires more time and GPU resources compared to the simpler GCN model[6]. Consequently, the CPU must handle a larger portion of the batching workload in such

[6]See Section 5 for details.

cases to avoid imbalanced workloads on processors. This necessitates an adaptive workload dispatching mechanism to dynamically allocate tasks based on the specific training setting.

Second, the scheduling of workloads on the CPU, PCIe, and GPU is also challenging. With collective batching, both the DMA transferring and UVA transferring stages utilize the PCIe, and both the GPU batching and model training stages leverage the GPU. Due to such hardware dependency conflicts, naïve pipeline execution leads to resource contention on the PCIe and the GPU. On the other hand, sequentially executing CPU-related stages and GPU-related stages misses the opportunity of overlapped execution and easily results in hardware idleness. In fact, we should judiciously arrange the training stages across the CPU, PCIe, and GPU to avoid both resource contention and idleness, as shown in Figure 4c.

Next, we will illustrate in detail how we formulate and address the collective batching and scheduling problem in Section 4.

## 4 COLLECTIVE BATCHING AND SCHEDULING

In this section, we first formulate the *Collective Batching and Scheduling Problem* in Section 4.1 and prove the problem is NP-hard. Then we present the design and implementation of our `MorphGL` in Section 4.2. `MorphGL` consists of the Profiler to collect essential profiling information (Section 4.2.1), the Dispatcher for distributing the batching workload to the CPU and the GPU (Section 4.2.2), the Scheduler for scheduling the workload on the CPU, PCIe, and GPU (Section 4.2.3), and the Executor that performs the training given the final scheduling plan (Section 4.2.4).

### 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 the mini-batches to be processed in one training epoch as $b_1, b_2, ..., b_n$. Each batch $b_i (i \in [1, n])$ experiences the batching, data transferring, and model training stages, which are denoted as $B_i$, $T_i$, and $M_i$, respectively. There are two optional execution arrangements for the three training stages on CPU, PCIe, and GPU, which we denote as **Route 0** and **Route 1**. In Route 0, CPU executes $B_i$, then $T_i$ is performed on PCIe via DMA, and finally GPU executes $M_i$ as in Category I systems. In Route 1, $B_i$ and $T_i$ leverage the UVA technique, which simultaneously occupies GPU and PCIe, and then GPU executes $M_i$ as in Category II systems. Let $r_i$ denote the route for the mini-batch $b_i$. We have $r_i = 0$ if $b_i$ follows Route 0, otherwise $r_i = 1$. We denote the beginning timestamp of executing one stage $X_i$ as $\Gamma(X_i)$ and the duration of $X_i$ as $Dur(X_i)$. We denote the hardware requirement of each training stage $X_i$ 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 have **hardware occupation constraints** $\Psi^{hw}(\Gamma(X_i), r_i)$ given in Constraints 2. These constraints ensure that if two training stages require the same hardware, one stage must either start before the other begins or only after the other has completed:

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

The mini-batch GNN training itself poses **data dependency constraints** $\Psi^{dp}(\Gamma(X_i), r_i)$ given in Constraints 3. Concretely, for each mini-batch $b_i$ following Route 0, its transferring stage $T_i$ starts only after its batching stage $B_i$ is finished. For the mini-batch following Route 1, its transferring stage and batching stage are executed simultaneously with GPU and PCIe due to the mechanism of UVA. For all mini-batches, the model training stage $M_i$ cannot precede the end of the corresponding transferring stage $T_i$ since the model training cannot start before the training data is ready on 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 fully utilize the hardware, we construct two buffers, namely the CPU buffer residing in the main memory and the GPU buffer residing in the device memory. Due to limited main memory and device memory, we have limited sizes for the two buffers, namely the **buffer size constraints** $\Psi^{bf}(\Gamma(X_i), r_i)$ given in Constraints 4. Since the smallest processing unit in GNN training is a mini-batch, we use the number of mini-batches that can be stashed in the buffer as the metric of the buffer's capacity. We denote the sizes of the CPU buffer and the GPU buffer as $cbs$ and $gbs$, respectively. $\text{Cnt}(t, X_i)$ means the total number of appearances of $X_i$ before timestamp $t$. At any timestamp, the number of mini-batches stashed in a buffer must be smaller than the buffer's 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:

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

**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 job-shop scheduling (JSSP) problem with three machines [36], which is known to be NP-hard, to our problem. The JSSP problem aims to minimize the makespan given $n$ jobs on three machines. If there exists a solution to our optimization problem, we can fix the route of all mini-batches to Route 1 and apply the scheduling result to the JSSP problem by

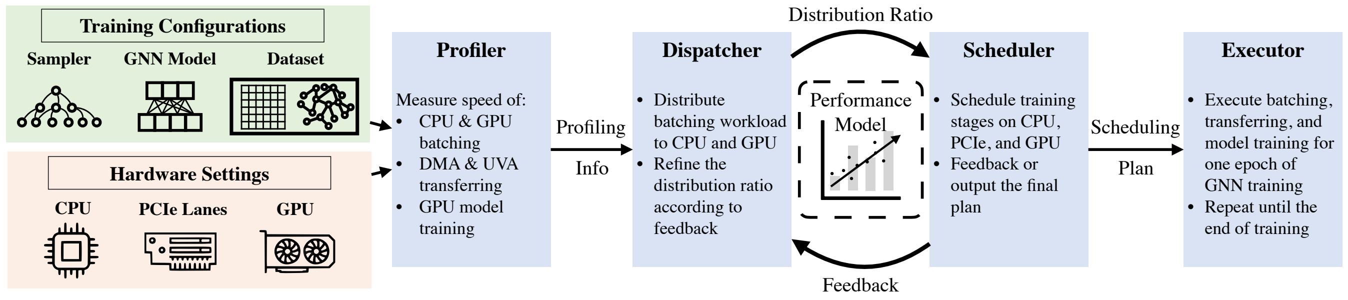

Training Configurations
Sampler
GNN Model
Dataset
Hardware Settings
CPU
PCIe Lanes
GPU
Profiler
Measure speed of:
• CPU & GPU batching
• DMA & UVA transferring
• GPU model training
Profiling Info
Dispatcher
• Distribute batching workload to CPU and GPU
• Refine the distribution ratio according to feedback
Distribution Ratio
Performance Model
Feedback
Scheduler
• Schedule training stages on CPU, PCIe, and GPU
• Feedback or output the final plan
Scheduling Plan
Executor
• Execute batching, transferring, and model training for one epoch of GNN training
• Repeat until the end of training

**Figure 5: Overall architecture of MorphGL.**

treating CPU, PCIe, and GPU as the three machines, $n$ mini-batches as $n$ jobs to complete, and different training stages as operations of the jobs. Since the resulting time of one training epoch is minimized, we can obtain the minimal makespan of the JSSP problem. This concludes the proof. □

Our optimization problem is more complex than the JSSP problem, which is known to be intractable and NP-hard [36], 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

We present the overall architecture of MorphGL in Figure 5. MorphGL contains four components, namely the Profiler, the Dispatcher, the Scheduler, and the Executor. The Profiler captures details of the running hardware and the user-defined training configurations with profiling. The Dispatcher is responsible for distributing the mini-batch preparation workload to the CPU and the GPU. The Scheduler arranges the execution of all training stages across the CPU, PCIe, and GPU. The Executor performs GNN model training with the final scheduling plan.

We next elaborate the details for each of the components, and present how to solve the *collective batching and scheduling problem* in an iterative way. Briefly, given the profiling results of the Profiler, the Dispatcher bootstraps the iteration by applying necessary relaxations to find a crude workload distribution ratio that dispatches mini-batches to the two routes. Then, the Scheduler devises a scheduling plan based on the current distribution ratio and evaluates the scheduling plan with a predictive model, which provides feedback to the Dispatcher about whether CPU or GPU has too much idleness. According to the feedback, the Dispatcher adjusts its old distribution ratio and sends the improved distribution ratio to the Scheduler for the next round of evaluation. In this way, the Dispatcher and the Scheduler negotiate and interact with each other for several rounds until they converge on the final distribution ratio and scheduling plan.

*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.

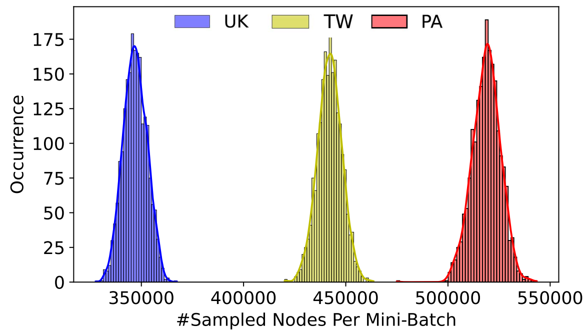
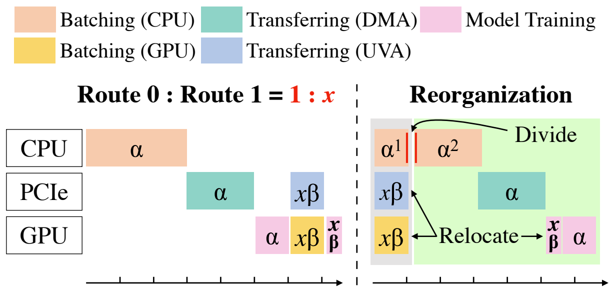


**Figure 6: Distribution of number of sampled $K$-hop neighbor nodes per mini-batch of the three datasets.**

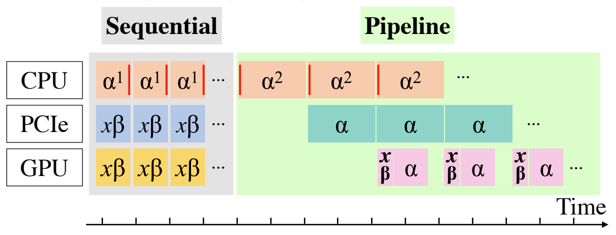
Concretely, as shown in Figure 6, the total number of sampled nodes in mini-batches, which determines the execution duration of training stages, follow narrow normal distributions with small relative standard deviations. Thanks to such stability, we can gather the mean duration of all five kinds of training 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 $k$ steps, 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 us to guide the systems' efficiency tuning.

*4.2.2 Dispatcher.* The Dispatcher takes in the profiling information from the Profiler and outputs the workload distribution ratio $\boldsymbol{x}$, representing that the number of mini-batches prepared by CPU and GPU is $\mathbf{1} : \boldsymbol{x}$. The Dispatcher determines the distribution ratio in a tunable way. The Dispatcher first obtains an initial guess by relaxing some constraints. Then the Dispatcher refines the initial guess by incorporating the downstream feedback provided by the Scheduler. We illustrate the initialization step and the finetuning step of the Dispatcher in detail as follows.

**Initial distribution ratio.** We bootstrap the iteration and derive the first distribution ratio by relaxing the *buffer size constraints* and



(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 given all mini-batches.

**Figure 7: Initial distribution ratio derivation. Training stage's duration (rectangle's length) is scaled by a factor of $x$ for the mini-batch that takes Route 1.**

assume that (1) we have unlimited buffers in both the main memory and the device memory and (2) the batching workload is infinitely divisible. We use $\alpha$ to represent a batch that takes Route 0 and $\beta$ to represent a batch that takes Route 1. To solve the distribution ratio $x$, we consider optimizing the sequential execution of one $\alpha$ and $x\ \beta$, as shown in the left of Figure 7a. For the CPU batching stage, we divide $\alpha$ into $\alpha^1$ and $\alpha^2$ so that the CPU batching duration of $\alpha^1$ equals the GPU batching duration of $\beta$. Next, we relocate the training stages of $\alpha$ and $\beta$ and the reorganized execution has two phases in the gray and green background, respectively, as shown in the right of Figure 7a. The gray phase has three equal-length workloads on CPU, PCIe, and GPU. Then, given all the mini-batches from both routes, we arrange an optimal execution for all the mini-batches which comprises a part of sequential execution with gray background and a part of pipeline execution with green background in Figure 7b, both of which make the most of CPU, PCIe, and GPU after warmup. The epoch training time of the scheduling in Figure 7b, which we denote as $C'$, is a bounded piecewise function of $x$ as in Equation 6 which is easy to minimize.

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

Under the relaxations, we can simplify the NP-hard optimization on $C$ to an easy optimization problem on $C'$, which we minimize 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 with downstream feedback in the finetuning step.

**Finetuning Step.** The above initial guess disobeys some constraints and inevitably leads to suboptimal scheduling. Therefore, we need to adjust the distribution ratio with more downstream information. As will be introduced later in Section 4.2.3, the Scheduler will provide feedback information about whether the current distribution ratio places too much batching workload on CPU or GPU side. With such feedback, the Dispatcher adjusts the assigning of workload between CPU and GPU accordingly. For example, when the Scheduler finds that GPU is too busy while CPU is idle during the scheduling, the feedback on reducing the number of mini-batches distributed to GPU will be provided to the Dispatcher. Then the Dispatcher adjusts the distribution ratio accordingly and provides the updated distribution ratio to the Scheduler for the next round of evaluation.

**Algorithm 1** The `Dual-Buffer Scheduling` Algorithm.

**Input:** GPU-based batching Iterator *iter_g*; CPU-based batching Iterator *iter_c*; CPU buffer size *cbs*; GPU buffer size *gbs*; initial GNN Model *M*; Optimizer class *optim_c*; Loss function *loss_func*; CUDA Stream for model training and GPU batching $s_0$; DMA Asynchronized Transferring CUDA Stream $s_1$

**Output:** trained GNN Model $M'$;

```
1:  buf_c ← deque(size=cbs)                                ▷ CPU buffer
2:  buf_g ← deque(size=gbs)                                ▷ GPU buffer
3:  opt ← optim_c(M.params(), ...)                         ▷ optimizer
4:  def train_batch(batch, label, event):
5:      $s_0$.wait_event(event)          ▷ ensure async transfer is finished before training
6:      loss = loss_func(M(batch), label)
7:      opt, loss ···               ▷ calculate gradients and update params
8:  while true do
9:      if iter_g.end() or iter_c.end() then
10:         ···                          ▷ dispose of remaining mini-batches
11:         break
12:     end if
13:     if buf_c.not_full() and buf_g.not_full() then
14:         buffers_filling(buf_c, buf_g, iter_c, iter_g)
15:     else if buf_c.not_full() or buf_g.not_full() then
16:         buffers_blocking(buf_c, buf_g, iter_c, iter_g)
17:     else
18:         buffers_flushing(buf_c, buf_g)
19:     end if
20: end while
21: return M'
```

4.2.3 *Scheduler.* Given the distribution ratio, the Scheduler devises a schedule plan that arranges the execution of all training stages on 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. The Scheduler employs the `Dual-Buffer Scheduling` algorithm to arrange CPU&GPU batching, DMA&UVA transferring, and GPU model training on CPU, PCIe, and GPU. The key idea of `Dual-Buffer Scheduling` is to mimic the optimal scheduling in Figure 7b but adhere to all the constraints. We describe the details as follows.

**`Dual-Buffer Scheduling.`** To maximize the hardware utilization with asynchronous execution, we construct the **CPU buffer** in the main memory holding mini-batches generated by the CPU

**Algorithm 2** The buffers_filling function.

```
Input: CPU buffer $buf\_c$; GPU buffer $buf\_g$; CPU batching Iterator $iter\_c$; GPU batching Iterator $iter\_g$.
1: gb, gl = $iter\_g$.next()
2: $buf\_g$.append((gb, gl, None))
3: cb, cl = $iter\_c$.next(non_blocking=true)
4: $buf\_c$.append((cb, cl))
```

**Algorithm 3** The buffers_blocking function.

```
Input: CPU buffer $buf\_c$; GPU buffer $buf\_g$; CPU batching Iterator $iter\_c$; GPU batching Iterator $iter\_g$.
1: if $buf\_c$.not_full() and $buf\_g$.full() then
2:     cb, cl = $iter\_c$.next(non_blocking=true)
3:     $buf\_c$.append((cb, cl))
4: else
5:     gb, gl = $iter\_g$.next()
6:     $buf\_g$.append((gb, gl, None))
7: end if
```

and the **GPU buffer** in the device memory holding mini-batches generated by the GPU. Mini-batches in the CPU buffer are ready for later DMA transferring. Mini-batches in the GPU buffer are ready for later GNN model training. Two buffers have fixed sizes while their contents are dynamically controlled during training by the three algorithms to be introduced later in this section. GPU buffer size *gbs* is a hyperparameter. CPU buffer size *cbs* is determined by $x$, which is the workload distribution ratio between CPU and GPU as we introduced earlier in the Initial distribution ratio part, and $gbs$: $cbs = \lfloor x \cdot gbs \rfloor$. The pseudo code of the `Dual-Buffer Scheduling` algorithm is given in Algorithm 1.

`Dual-Buffer Scheduling` is essentially a repeat of three phases, namely (1) the *buffers filling phase*, (2) the *buffers blocking phase*, and (3) the *buffers flushing phase*. The pseudo codes of the three phases are given in Algorithm 2, 3, and 4, respectively. We denote one pass of the three phases as one **overlap** and describe the details of each phase as follows.

First, in the *buffers filling phase*, we let CPU-batcher and GPU-batcher independently generate mini-batches and fill two buffers. In this phase, three devices are all fully utilized since CPU-batcher occupies CPU and GPU-batcher occupies PCIe and GPU simultaneously, therefore, we have no hardware idleness in this phase. Then, if two buffers are filled at the same time, we proceed to the *buffers flushing phase*. Otherwise, we enter the *buffers blocking phase*.

Second, in the *buffers blocking phase*, one of the buffers is full while the other is not. Then we block the mini-batch generation on the buffer-full hardware and wait for the other buffer[7]. In this phase, we make a record on which hardware causes the blocking. By the end of scheduling, we collect all the records and provide this feedback to the Dispatcher. If the majority of the blocking records are due to busy GPU and idle CPU, it means that the Dispatcher

[7]When the CPU buffer is full and the GPU buffer is not, we do the blocking. When the GPU buffer is full and CPU buffer is not, we repeatedly pop one mini-batch from the GPU buffer, train the GNN model with this mini-batch, and fill the GPU buffer with another mini-batch generated by GPU-batcher. This is a trick to avoid hardware idleness as much as possible even during the blocking phase.

**Algorithm 4** The buffers_flushing function.

```
Input: CPU buffer $buf\_c$; GPU buffer $buf\_g$.
1: while $buf\_g$.not_empty() do
2:     gb, gl, ge = $buf\_g$.popleft()
3:     if $buf\_c$.not_empty() then
4:         cbcl = $buf\_c$.popleft()
5:         cb, cl = cbcl.dma_transfer($s_1$)
6:         ce = $s_1$.record_event()          ▷ async transfer cpu batch on $s_1$
7:         $buf\_g$.append((cb,cl,ce))
8:     end if
9:     train_batch(gb, gl, ge)             ▷ async train on $s_0$
10: endwhile
```

assigns too much batching workload to GPU in the current distribution ratio. Therefore, we provide a feedback to the Dispatcher on decreasing the number of mini-batches distributed to GPU. With the repetition of such feedback and tuning, the mismatch between CPU workload and GPU workload is narrowed progressively. For example, if GPU batching is much longer than CPU batching in the current *overlap*, we will keep relocating one mini-batch's batching workload from GPU to CPU until the time mismatch is smaller than the execution time of one GPU-based batching. Then, the duration of the *buffers blocking phase* cannot be reduced further by re-dispatching the batching workload anymore, which marks the convergence of the iterative feedback and tuning process.

Third, in the *buffers flushing phase*, we control the buffer flushing of two buffers by repeating the following two procedures. First, we pop a mini-batch from GPU buffer and train the model on this mini-batch. Second, we pop a mini-batch from CPU buffer and transfer it through PCIe (DMA) into GPU buffer. The DMA transfer is launched asynchronously in a separate CUDA stream without blocking the main Python process. Hence, the PCIe DMA transfer is executed in parallel with the GPU model training as well as the CPU batching. We repeat the above two procedures until two buffers are empty. During the *buffers flushing phase*, CPU-batcher is preparing mini-batches for the next *overlap*, and PCIe and GPU keep transferring data and training the model thanks to the asynchronous execution with separated CUDA streams and two buffers.

We repeatedly run several *overlap*s until we finish the training of all mini-batches in the current epoch. For the three phases introduced above, hardware idleness is only present in the *buffers blocking phase* where either CPU or GPU is forced to wait. Therefore, with the help of feedback information, the Dispatcher can refine the distribution ratio and shorten the *buffers blocking phase* as much as possible. Then, the majority of scheduling is in the *buffers filling phase* and the *buffers flushing phase*, which effectively avoids the hardware idleness during training.

**Performance Model** While it is possible to execute the above overlaps on the hardware and accumulate the feedback information, we find that it is more efficient to run the overlaps in a simulator which can provide the same feedback information. Such simulation is facilitated by both the stability of the duration time of $X_i$ and our buffered execution. First, as we have mentioned in Section 4.2.1, the number of sampled neighbors per mini-batch follows a normal distribution with a small coefficient of variation (CV), which is the ratio of the standard deviation to the mean value. A small CV indicates a narrow normal distribution where all observed values are similar. This leads to similar batching/transferring/training workload for different mini-batches. In addition, the execution of $X_i$ is grouped with the buffer, which further reduces the coefficient of variation and leads to a more stable execution time. Therefore, given such stability of execution time, we can build a simple but accurate Python-based simulator for our `Dual-Buffer Scheduling` which simulates the scheduling without actually executing it. We use the simulator as a highly efficient performance model that provides feedback information to the Dispatcher and predicts the training time under the current distribution ratio and scheduling plan.

**Extreme Cases** We find that in some rare cases there already exists a trivial optimal training plan, namely the naïve pipeline scheduling of batching on CPU, data transferring on PCIe, and model training on GPU. Specifically, when the model training is longer than both the CPU batching time and the PCIe data transferring time, the real bottleneck is the computing capability of GPU and a naïve pipelining is the optimal training plan. However, such cases are rare in GNN training scenarios since GNN models usually have shallow depths and small sizes to avoid the over-smoothing problem on graphs [35]. Although we haven't seen such rare contexts in our experiments, `MorphGL` still supports the naïve pipeline training for better generality.

**Theoretical Analysis** We provide the theoretical analysis of our proposed method. As discussed earlier in this Section, for the majority of GNN training cases, the batching on CPU is longer than the data transferring on PCIe and model training on GPU. Therefore, we rest our analysis upon this assumption.

**Theorem 2.** Given the bandwidth of PCIe as $Bdw_p$ and the bandwidth of GPU memory as $Bdw_g$, 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.

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

Proof of Theorem 2. We denote the optimal scheduling result as $C^{opt}$, the scheduling result 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 one GPU model training as $t_{model}$, the duration of one GPU batching as $t_{gb}$, the duration of one CPU batching as $t_{cb}$, the duration of one DMA/UVA transferring as $t_{dma}/t_{uva}$, 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 $n^0_{CPU}/n^0_{GPU}$. One epoch of training time is marked by the end of the execution of the last mini-batch's model training. Therefore, the optimal scheduling result is larger than the last operation on each device:

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

We denote the size of CPU buffer and GPU buffer as $p$ and $q$. By the definition of the `Dual-Buffer Scheduling` algorithm, its output scheduling only has one possible source for GPU idleness, namely the blocking wait idleness *idle* in the *buffers blocking phase*:

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

Therefore, the approximation rate $A$ is :

$$\begin{aligned} A &= \frac{C^0}{C^{opt}} \\ &< \frac{idle + n \times t_{model} + n^0_{GPU} \times t_{gb}}{C^{opt}} \\ &< \frac{t_{gb} \times (\frac{n}{p+q} + 1) + n \times t_{model} + n^0_{GPU} \times t_{gb}}{C^{opt}} \\ &< 1 + \frac{t_{gb} \times (\frac{n}{p+q} + 1)}{n \times t_{dma}} + \frac{n^0_{GPU} \times t_{gb}}{C^{opt}} \\ &< 1 + \frac{t_{gb}}{t_{dma}}(\frac{1}{p+q} + \frac{1}{n}) + \frac{n^0_{GPU} \times t_{gb}}{n^{opt}_{GPU} \times t_{gb} + n \times t_{model}} \\ &< 2 + \frac{t_{uva}}{t_{dma}}. \end{aligned} \tag{9}$$

We denote the number of sampled neighbors per mini-batch as $N$, the number of features per node as $f$, and the number of sampled edges per mini-batch as $E$. While DMA-based transferring and UVA-based transferring have the same data transferring volume and can both saturate PCIe bandwidth, UVA-based transferring needs more time because it is executed simultaneously with UVA-based batching, which involves additional permutation for the topology data of the mini-batch:

$$\begin{cases} t_{dma} = \frac{Nfc_1 + 2Ec_2}{Bdw_p}, \\ t_{uva} < \frac{Nfc_1 + 2Ec_2}{Bdw_p} + \frac{2Ec_2}{Bdw_g}. \end{cases} \tag{10}$$

Here $c_1$ and $c_2$ are element sizes of the data type of features and topology data. Bringing Inequation 10 into Inequation 9, we obtain the final approximation rate:

$$A < 3 + \frac{Bdw_p}{Bdw_g}. \tag{11}$$

Typically, in modern hardware, the bandwidth of the GPU memory is at 1000GB/s level, while the PCIe bandwidth is at 10GB/s level. Therefore, the approximation of our algorithm is close to 3. □

*4.2.4 Executor.* The Executor absorbs the final scheduling plan and other training configurations and trains the GNN model for a user-defined number of epochs. The user-defined training configurations and hardware settings are processed by the `Profiler` API, which measures and stores the batching/transferring/training duration information. Then the `Dispatcher` and the `Scheduler` APIs are called in a loop until the iterative dispatching and scheduling converge on the final scheduling plan. Finally, we train with the `Executor` API for a number of epochs times.

## 5 EXPERIMENTS

We implemented `MorphGL` based on DGL (v0.8) and open source it on the github[8]. We run the experiments with different hardware, GNN models, and datasets in the following sections.

[8]https://github.com/DSLabor/morphGL

**Table 2: Dataset statistics. The node features are stored as float16 type. The bi-directed topology data are stored as 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 |

It is noteworthy that MorphGL does not change the content of mini-batches and model computations like forward pass, loss calculation, and backward propagation. Therefore, the accuracy and convergence of MorphGL are identical to those of baselines with the only difference in epoch training time. Consequently, we only measure and compare the epoch training time results in this work.

## 5.1 Experimental Setup

**Hardware.** As stated in Section 3, real-world users usually have diverse hardware settings. In this work, we concentrate on improving GNN training efficiency on the common hardware. Because NVIDIA takes 88% of GPU market [34], 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 [3], we correspondingly choose enterprise-level A30, consumer-level RTX 3090 and MIG-A30 for each machine. Here MIG-A30 stands for A30 with the 2g.12gb multi-instance GPU setting which uses half of total SMs and HBM. We use such MIG-A30 to represent low-end GPUs. To verify the the influence of the sizes of device memory on the training, we additionally test MorphGL and baseslines on a machine with NVIDIA V100 (32GB). 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. Following the convention of cloud service providers [3, 13, 29], we corresponding 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.** In this paper, we focus on scalable GNN training on billion-scale graphs. We use three large-scale datasets which are widely used in previous works for experiments as shown in Table 2. Specifically, we use a web graph: UK-2006-05 (UK) [4], a social graph: Twitter (TW) [20], and an academic citation graph in Open Graph Benchmark (OGB) [16]: Ogbn-Papers100M (PA). Following previous works [19, 45], we generate 256 random node features per vertice for UK and TW since they originally had no features. Following the common practice in the public leaderboard [16], we add bidirectional edges for the topology data of all datasets before training. We use the official training sets provided by the OGB when training on PA. For the other two datasets that do not provide a training set, we follow the common practice of previous works [26, 45] and randomly sample 1 percent of all vertices as the training set. Following the previous work [19], we store the node features in float16 and the topology data in int64 CSC format.

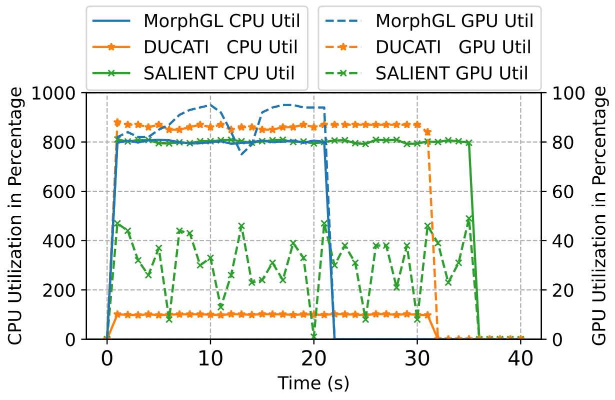


**Figure 8: Detailed CPU and GPU utilization for the three systems in one training epoch (UK, GraphSAGE, Machine B).**

**GNN Models.** We conduct experiments with three mainstream GNN models, namely GCN [43], GraphSAGE [14], and GAT [39]. Following previous works [26, 30], 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 [39, 43] and previous works [19, 52]. 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. As discussed in Section 2, 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, one CPU core is reserved for main process logic and the rest of CPU cores are batching workers.

We report the training time per epoch of each system under different settings. All results are computed by calculating the averages over five training epochs after one epoch of warmup.

## 5.2 Main Results: Epoch Time Comparison

We compare the training time of one epoch of the three systems and present the results in Table 3. Overall, MorphGL outperforms SALIENT and DUCATI by up to 2.76 times (on average 1.55 times) and 2.2 times (on average 1.41 times) respectively. MorphGL can achieve universal speedup against baselines in all settings, which verifies the adaptability and efficiency of MorphGL. We further analyze the training efficiency of the three systems regarding different machines, models, and datasets as follows.

First, by comparing the three systems' efficiency on different machines, we can observe the benefit of MorphGL's collective batching and scheduling design. SALIENT prefers machines with abundant CPU cores compared to the equipped GPU. Therefore, SALIENT performs relatively well on Machine A where it is only 1.27 times slower on average than MorphGL. However, SALIENT's efficiency

**Table 3: 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 × |

degrades considerably when applied to Machine B, which has fewer CPU cores, and SALIENT is 1.89 times slower than MorphGL on average. Similarly, the efficiency of DUCATI fluctuates significantly on different machines. By contrast, MorphGL's efficiency is stable and consistently better than baselines thanks to its collective batching design.

We present the detailed CPU and GPU utilization information during one epoch of training for the three systems in Figure 8. As shown, SALIENT achieves high CPU utilization throughout the training process but suffers from fluctuatingly low GPU utilization because GPU is always hungrily waiting for the mini-batches from CPU. DUCATI maintains high GPU utilization but has poor CPU utilization since it ignores 7 out of 8 CPU cores. MorphGL, on the other hand, achieves high CPU and GPU utilization, which leads to shorter epoch time compared to the two baselines.

Second, even on the same machine, different GNN models lead to different workload distributions and, thus, different requirements for workload dispatching. For example, for SALIENT experiments on Machine C, when training GCN instead of GAT, the model training on GPU is less time-consuming, which increases the mismatch between batching on CPU and model training on GPU and aggravates the idle GPU problem. As a result, the relative efficiency of SALIENT for GCN training (on average 1.65 times slower than MorphGL) is worse than that for GAT (on average 1.46 times slower than MorphGL). By contrast, MorphGL captures such change in model training time and correspondingly moves more batching workloads from CPU and GPU, which leads to balanced workloads and better efficiency.

Third, we observe that the baseline's training efficiency varies on different datasets due to different levels of data locality. For example, DUCATI achieves the best efficiency on the PA dataset but performs worst on the UK dataset. The reason behind is that the benefit of the dual-cache design of DUCATI is highly dependent on

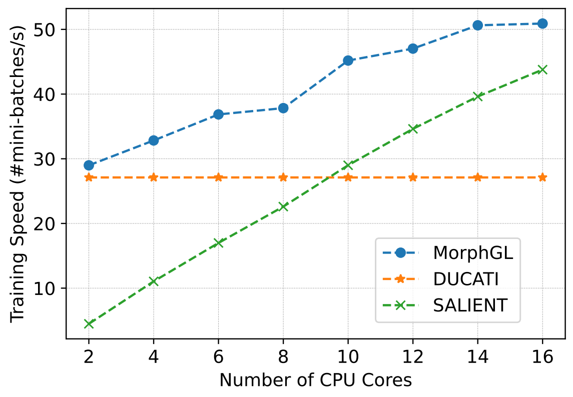


**Figure 9: The training speed comparison of three systems with different numbers of CPU cores per RTX3090.**

dataset's locality. PA has a manually labeled training set containing high-degree nodes with dense connection [16] and, thus, strong locality [45, 52]. But UK has randomly chosen training nodes that have limited locality. Therefore, DUCATI's efficiency is better on PA than on UK. MorphGL captures such locality differences and adaptively changes the workload on CPU and GPU. For example, when detecting faster GPU batching speed of DUCATI on PA due to better locality, MorphGL correspondingly allocates more batching workload to GPU, which ensures MorphGL's efficiency regardless of the locality of datasets.

## 5.3 Influence of CPU&GPU Setup

We further investigate how the CPU and GPU setups influence the systems' efficiency and present the results in Figure 9. To obtain fine-grained results, we vary the number of CPU cores equipped

**Table 4: Hardware utilization comparison**

| | SALIENT | | | DUCATI | | | `MorphGL` | | |
|---|---|---|---|---|---|---|---|---|---|
| | CPU | PCIe | GPU | CPU | PCIe | GPU | CPU | PCIe | GPU |
| 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 5: Scheduling plans of `MorphGL`.**

| | Machine A | Machine B | Machine C |
|---|---|---|---|
| | (CPU buffer size, GPU buffer size) | | |
| PA | (15, 10) | (5, 10) | (11, 10) |
| TW | (20, 10) | (8, 10) | (24, 10) |
| UK | (30, 10) | (12, 10) | (28, 10) |

to one RTX 3090 GPU from 2 to 16 with a stepsize of 2. We report the comparison of SALIENT, DUCATI, and `MorphGL` regarding the average training speed with the UK datatset on the GraphSAGE model. The results are similar in other settings.

First, the relative efficiency of SALIENT and DUCATI is highly influenced by the CPU:GPU ratio. Their solutions are sufficient to utilize the majority of hardware resources only in extreme settings where CPU cores are extremely abundant or scarce. When CPU cores are few, GPU-based batching of DUCATI is significantly better than SALIENT. However, when CPU cores are more than 10, CPU-based batching of SALIENT is faster.

Second, our `MorphGL` always performs better than the two baselines regardless of CPU:GPU ratio. When the number of CPU cores increases, `MorphGL` gradually relocates more batching workload from GPU to CPU, therefore, enjoying faster training speed. However, when the number of CPU cores is very small/large for some rare hardware, the improvement of `MorphGL` batching is less obvious. In more extreme cases where the CPU cores are fewer than 2 or more than 16, the efficiency of `MorphGL` will gradually fall back to pure CPU or pure GPU batching's efficiency.

## 5.4 Hardware Utilization

We compare the hardware utilization of CPU, PCIe, and GPU of the three systems and present results in Table 4. We profile the training with GraphSAGE on Machine B and the utilization in other settings is similar. We measure CPU utilization with `top`, measure GPU utilization with `nvidia-smi`, and measure PCIe bandwidth with `perf-iostat`. We report the average utilization or bandwidth during one epoch of training after two epochs of warm-up.

Overall, SALIENT utilizes CPU well but has poor GPU utilization. DUCATI has a busy GPU but wastes 87.5% of CPU cores. For these two systems, the PCIe bandwidth is also underutilized because the training is throttled by batching either on CPU or GPU. DUCATI has lower PCIe bandwidth utilization due to its dual-cache design, which reduces the PCIe traffic volume. By contrast, `MorphGL` achieves high CPU, PCIe, and GPU utilization simultaneously, which demonstrates that `MorphGL` can reduce hardware idleness with better workload dispatching and scheduling. In a nutshell, `MorphGL` improves the utilization of hardware, resulting in the speedup of GNN training as we observed in Section 5.2.

## 5.5 Scheduling Examples

We present the final configurations of the CPU buffer size and the GPU buffer size in `Dual-Buffer Scheduling` with respect to the GraphSAGE Model in Table 5. 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.

**Table 6: 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 |

## 5.6 Influence of the Device Memory Size

We present the epoch training time of the three systems with different sizes of device memory on Machine D with one V100 (32GB) and eight CPU cores as shown in Table 6. We limit the available device memory size to 12GB, 16GB, and 32GB using the `set_per_process_memory_fraction` utility provided by PyTorch. We obtain the results with the UK dataset and the GAT model and the results are similar in other settings.

First, with larger device memory, DUCATI benefits from larger topology and feature caches, which leads to reduced epoch training time. However, as pointed out by DUCATI [52], the benefit of larger caches shows a marginal effect. As shown in the $T_{GBT}$ column, the benefit of increasing device memory from 16GB to 32GB is less than that of increasing device memory from 12GB to 16GB. Second, the CPU batching speed is unrelated to the device memory size. However, a larger device memory size reduces the frequency of device-side garbage collection during training [7], which accounts for a slightly faster training speed for SALIENT. Third, `MorphGL` can capture and handle the batching speed variation of DUCATI and SALIENT due to the change of device memory, which leads to its consistently better performance compared to baselines.

## 5.7 Preprocessing Time

We put together the time of the Profiler, the Dispatcher, and the Scheduler as the preprocessing time and compare it to the epoch training time as in Table 7. We obtain the results with the GraphSAGE model and the results are similar in other settings.

As shown, the preprocessing time is always less than five epoch of training. The largest ratio of preprocessing time to epoch training time is 4.9 as highlighted for Machine B with the TW dataset. The

**Table 7: 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** |

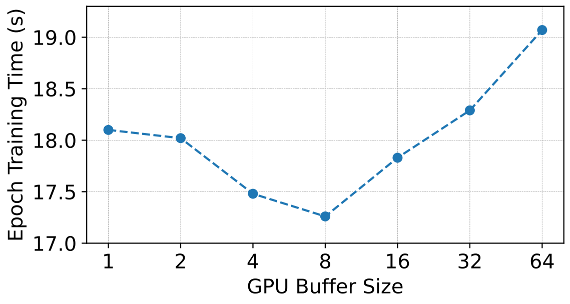


**Figure 10: Epoch training time of `MorphGL` with respect to different GPU buffer sizes.**

smallest ratio is 3.2 for Machine C and UK dataset. On average, the preprocessing is 3.9 times longer than one epoch training. As pointed out in previous works [11, 42, 45], training GNN models on giant graphs to desirable accuracy usually requires hundreds or even thousands of epochs of training. Therefore, the preprocessing time of `MorphGL` is negligible compared to the total training time.

## 5.8 Influence of the GPU Buffer Size

To investigate the impact of the hyperparameter GPU buffer size on the performance of `MorphGL`, we measure and visualize the epoch training time, as depicted in Figure 10. The results are obtained with GraphSAGE on PA dataset on Machine A. The results are similar in other settings. In this setting, the mean and standard deviation of mini-batch storage sizes in MB are 149.7 and 2.0, respectively.

First, when the GPU buffer size is too small (e.g., 1), the operational space is constrained for the `Dual-Buffer Scheduling` algorithm, resulting in frequent transitions between buffers filling, blocking, and flushing phases. This requires frequent CUDA stream synchronization, thereby increasing training time.

Second, an excessively large GPU buffer size (e.g., 64) consumes substantial device memory, diminishing the available space for GPU-based topology&feature caches [52]. This leads to prolonged GPU batching time, ultimately increasing the overall training duration.

In this paper, we select a GPU buffer size of 10, which occupies less than 2 GB of device memory. This configuration provides sufficient operational space for the `Dual-Buffer Scheduling` algorithm while preserving most of the device memory for GPU-based caches, thus avoiding delays in GPU batching.

## 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, namely 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.

## REFERENCES

[1] 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.

[2] 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

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

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

[5] 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.

[6] 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).

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

[8] 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.

[9] 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

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

[11] 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.

[12] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: lightweight interactive community search via graph neural network. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1006–1018. https://doi.org/10.14778/3447689.3447704

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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

[19] 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.

[20] 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.

[21] 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

[22] 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.

[23] 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.

[24] 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.

[25] 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

[26] 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.

[27] 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

[28] 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.

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

[30] 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).

[31] 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

[32] 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

[33] 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

[34] 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/

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

[36] 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.

[37] 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.

[38] 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

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

[40] 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.

[41] 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).

[42] 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.

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

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

[45] 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.

[46] 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.

[47] 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.

[48] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient Exact Subgraph Matching via GNN-Based Path Dominance Embedding. *Proc. VLDB Endow.* 17, 7 (March 2024), 1628–1641. https://doi.org/10.14778/3654621.3654630

[49] 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

[50] 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

[51] 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).

[52] 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.

[53] Yuhao Zhang and Arun Kumar. 2023. Lotan: Bridging the Gap between GNNs and Scalable Graph Analytics Engines. *Proc. VLDB Endow.* 16, 11 (July 2023), 2728–2741. https://doi.org/10.14778/3611479.3611483

[54] 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

[55] 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

[56] 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